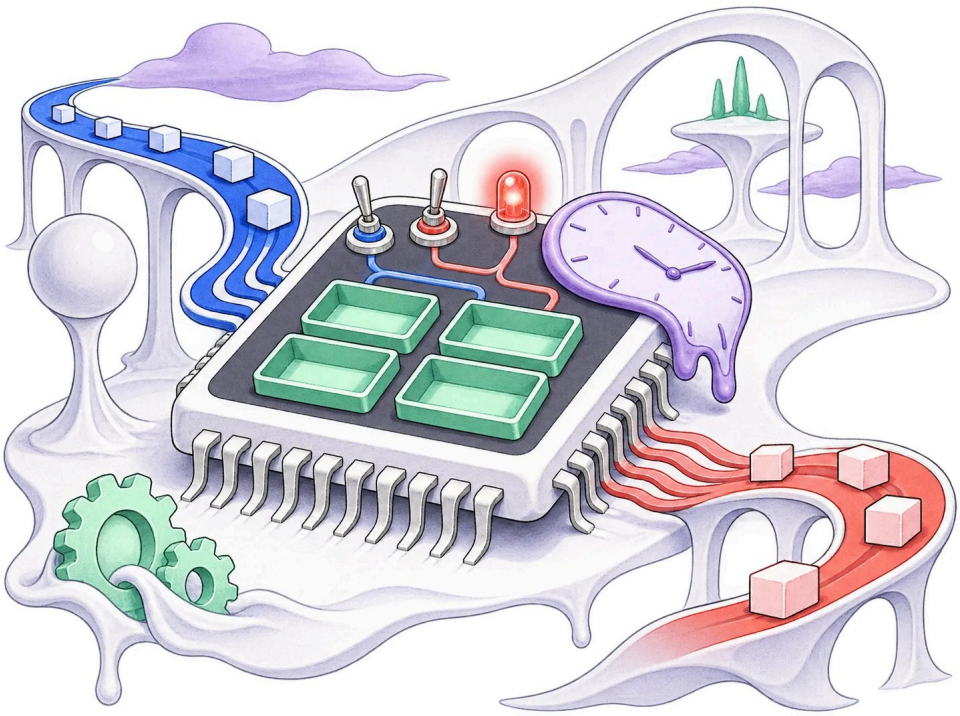


COURSE TEXTBOOK

Microprocessors

Sercan Külcü

Giresun University · Department of Computer Engineering

Microprocessors

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/microprocessors/`

Slides, study questions and simulators are on that page. The programs are in the
`github.com/sercankulcu/microprocessors-8086-en` repository.

© Sercan Külcü. All rights reserved.

Generated from the notes of 25 November 2026.

Preface

Microprocessors execute instructions, process data and coordinate every operation of a computer system. This book studies processor organisation, registers, memory access, instruction execution and low-level programming through the 8086 architecture. Today's x86-64 processors are far more complex, but the 8086 is still a good foundation for learning registers, segmentation, addressing modes, interrupts and assembly programming. This book is made up of the notes for the Microprocessors course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course.

The book has four parts. In order, they are **Foundations** (Chapters 1–3), **Memory and System Services** (Chapters 4–5), **Instructions and Programming** (Chapters 6–9), and **Hardware and Instruction Encoding** (Chapters 10–11).

Some chapters are made up of more than one set of lecture notes. These chapters open with a short title page, and the notes follow as numbered sections such as 4.1 and 4.2.

Every chapter follows the same layout. It opens with learning outcomes. Formal definitions and worked examples are set in named boxes. *Common mistake* boxes point out the misconceptions students fall into most often. Code traces show, step by step, how a program runs and how the registers and flags change. Each chapter closes with a summary, a list of key terms, *Test Yourself* questions with their answers, suggestions for further reading and a bibliography. The glossary at the end of the book defines each key term in a sentence and names the chapter that explains it.

The examples run in emu8086. Every program in the book is also in the companion repository `github.com/sercankulcu/microprocessors-8086-en`, where each one is assembled and run automatically and checked against its expected output. Slides, study questions, the CPU simulator and the other simulators are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · FOUNDATIONS

1 Introduction to Microprocessors and Number Systems 4

1.1 Introduction 5

1.2 Number Systems 60

2 8086 Architecture and Registers 115

2.1 8086 Architecture 116

2.2 8086 Registers 163

3 Assembly Language 209

3.1 Programming Languages 210

3.2 Assembly Language 268

PART II · MEMORY AND SYSTEM SERVICES

4 Memory Access and Variables 308

4.1 Memory Access 309

4.2 Variables 363

5 Interrupts and BIOS/DOS Services 413

5.1 Interrupts 414

5.2 Interrupt Functions 461

PART III · INSTRUCTIONS AND PROGRAMMING

6 Arithmetic, Logic and Shift Instructions 525

6.1 Flags and the result of an operation 526

6.2 Instruction groups and operand rules 532

6.3 Addition, subtraction and comparison 535

6.4 Logical operations 539

6.5 Multiplication and division 543

6.6 Single-operand instructions 554

6.7 Shifts and rotates 556

6.8 Example: array summation 562

6.9 BCD arithmetic 565

6.10 Example: temperature conversion 572

6.11 Which instruction affects which flag? 574

6.12 Common mistakes 575

6.13 Exercises and solutions 575

6.14 Summary 578

6.15 Key terms 578

6.16 Test Yourself 579
6.17 Further reading 581
6.18 References 581

7 Flow Control 582

7.1 Sequential execution and flow control 583
7.2 The unconditional jump: JMP 585
7.3 Conditional jumps 592
7.4 Comparison: CMP 596
7.5 Counterparts of high-level constructs 606
7.6 Loop instructions 611
7.7 Example programs 620
7.8 Common mistakes 627
7.9 Exercises and solutions 629
7.10 Summary 630
7.11 Key terms 631
7.12 Test Yourself 631
7.13 Further reading 634
7.14 References 634

8 The Stack and Procedures 635

8.1 The Stack 636
8.2 Procedures 674

9 Macros and the emu8086.inc Library 732

9.1 Macros 733
9.2 The emu8086.inc Library 786

PART IV · HARDWARE AND INSTRUCTION ENCODING

10 8086 Hardware and Device Control 833

10.1 8086 Pins 834
10.2 Device Control 891

11 Instruction Encoding and String Instructions 949

11.1 Overview of the instruction set 950
11.2 Instruction format 952
11.3 Encoding step by step 957
11.4 Decoding machine code 964
11.5 Data-transfer instructions 968
11.6 Arithmetic, logic, and shift instructions: summary 974
11.7 String instructions 975
11.8 Repeat prefixes 981
11.9 Control transfer: the encoding of JMP 987
11.10 Processor-control instructions 989
11.11 Flag effects 992
11.12 Common mistakes 994

11.13 Exercises and solutions 995
11.14 Summary 996
11.15 Key terms 997
11.16 Test Yourself 998
11.17 Further reading 1000
11.18 References 1000

APPENDICES

A Glossary of Terms **1001**

B Prerequisites for Microprocessors **1003**

C Important People in Microprocessors **1011**

MICROPROCESSORS

Course Map

This page puts every chapter of the course on one map. Each chapter is a node, and an arrow shows which idea from an earlier topic the next one relies on. Read it before the course starts to see where the topics are heading, or during the course to see where you are.

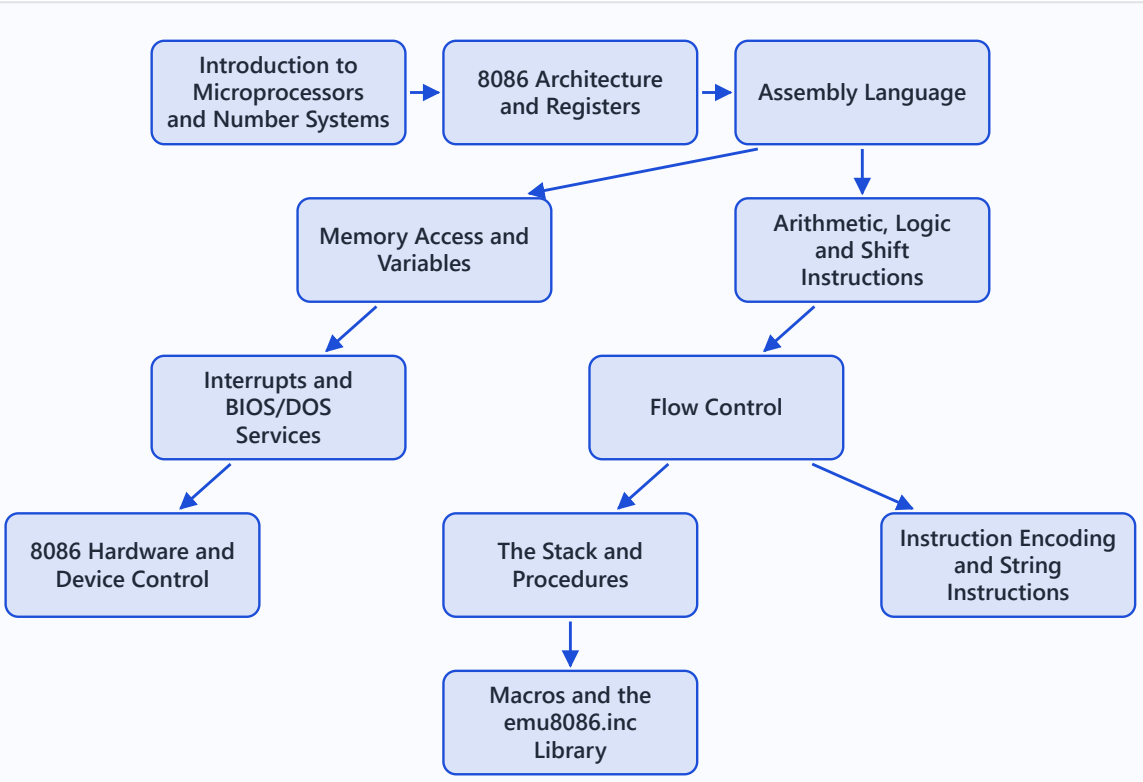


Figure 1: The course map. Each arrow points from a topic to one that builds on it. The path runs from architecture to assembly language, then on to instructions, system services and hardware.

Figure 1 shows the connections between the topics. Each node is a chapter, and an arrow indicates which idea of the earlier topic the later one relies on. Only the closest connections are drawn on the map; every chapter that a given chapter depends on is listed in Table 1.

Topic (chapter)	Builds on	Connection
Introduction to Microprocessors and Number Systems (1)	—	The fetch, decode, execute cycle; binary and hexadecimal numbers, two's complement representation.

Topic (chapter)	Builds on	Connection
8086 Architecture and Registers (2)	Introduction to Microprocessors and Number Systems	The internal structure of the 8086: BIU and EU, registers, flags and the segment:offset address.
Assembly Language (3)	8086 Architecture and Registers	From machine language to assembly: the assembler, the linker and the first program.
Memory Access and Variables (4)	8086 Architecture and Registers, Assembly Language	Addressing modes, effective and physical addresses; variables with DB, DW and DUP.
Interrupts and BIOS/DOS Services (5)	8086 Architecture and Registers, Memory Access and Variables	During INT, FLAGS, CS and IP are pushed onto the stack; the interrupt vector table is in memory.
Arithmetic, Logic and Shift Instructions (6)	Introduction to Microprocessors and Number Systems, 8086 Architecture and Registers, Assembly Language	The result of a computation sets the flags: CF, ZF, SF, OF; MUL, DIV and shifts.
Flow Control (7)	Arithmetic, Logic and Shift Instructions	Conditional jumps that read the flags; deciding after CMP and looping with LOOP.
The Stack and Procedures (8)	Memory Access and Variables, Interrupts and BIOS/DOS Services, Flow Control	The stack through SS:SP; CALL and RET keep the return address on the stack; the stack frame with BP.
Macros and the emu8086.inc Library (9)	The Stack and Procedures	The textual counterpart of a procedure: defining and expanding a MACRO, emu8086.inc routines.
8086 Hardware and Device Control (10)	8086 Architecture and Registers, Interrupts and BIOS/DOS Services	The 40-pin 8086, bus cycles and the 8259A; device control with IN and OUT.
Instruction Encoding and String Instructions (11)	Memory Access and Variables, Arithmetic, Logic and Shift Instructions, Flow Control	The machine code of instructions: opcode, D and W bits, ModR/M; string instructions with the REP prefix.

Table 1: The connections on the map.

The backbone of the map can be read as **number → architecture → assembly → instruction → hardware**. The first three chapters set up the processor and the language used to program it. Two branches grow from there: the memory branch starts with addressing and variables and reaches interrupts and device control; the instruction branch starts with arithmetic and flags and reaches flow control and procedures. The stack is where the two branches meet: CALL with RET and INT with IRET both use it. The instruction-encoding chapter shows how the instructions learned along the way look in machine code.

PART I

Foundations

Chapters 1–3

PART I · CHAPTER 1

Introduction to Microprocessors and Number Systems

The microprocessor is the heart of a computer system: it fetches instructions from memory, decodes them and executes them. The chapter begins with the difference between microprocessors and microcontrollers, the fetch–decode–execute cycle, the internal structure of a processor and clock speed, then compares CISC, RISC and EPIC designs, special-purpose processors and the evolution of the Intel x86 family from the 4004 to today. The number-systems part covers decimal, binary and hexadecimal bases, base conversion, signed numbers in ones' and two's complement, and binary arithmetic. It ends with encodings such as IEEE 754 floating point, BCD, Gray code and byte order.

NOTES IN THIS CHAPTER

1.1 Introduction

1.2 Number Systems

PART I · CHAPTER 1.1

Introduction

The same idea runs inside the control board of a washing machine, the engine controller of a car, and a laptop: a programmable chip that reads instructions from memory one after another and carries them out. This chapter introduces that chip, the microprocessor, from the outside in. It first places the microprocessor in a computer system, shows the buses that connect it to memory and to input/output devices, and explains how it differs from a microcontroller. It then follows the fetch–decode–execute cycle that repeats for every instruction, step by step, on 8086 machine code. The internal structure of the processor, the clock signal, and the three quantities that determine a program's running time (instruction count, cycles per instruction, clock frequency) are worked out with numerical examples. The later sections cover instruction set architectures (CISC, RISC, EPIC), special-purpose processor types, and the evolution of the Intel x86 family from the 4004 to the Core i7. The chapter ends with the features that determine performance in modern processors, such as caches, multiple cores, and power management. All code examples are written in 8086 assembly, and each one is traced in a table that shows how the registers, flags, and memory change.

LEARNING OUTCOMES

Define the basic concepts

Define the microprocessor, the microcontroller, and the address, data, and control buses of a computer system; compute how many bytes an *n*-line address bus can address.

Trace the instruction cycle

Trace the fetch–decode–execute cycle step by step on machine code; show how the instruction pointer (IP) changes in sequential flow and at a jump.

Compute performance

Use the relationship among clock frequency, clock period, bus cycle, cycles per instruction (CPI), and execution time; count how many clock cycles a small 8086 loop takes.

Compare architecture types
Compare the CISC, RISC, and EPIC approaches and the vector, SIMD, DSP, GPU, and symbolic processor types by their core ideas, strengths, and examples; compute the cycle in which a pipeline finishes.

Describe the x86 family
Describe the data and address bus widths of Intel processors from the 4004 to the Core i7, the innovations each one brought, and the place of the 8086 in this family.

Recognize modern features
Recognize concepts such as caches and the memory hierarchy, multiple cores, virtualization, and power management; show with a numerical example that clock speed alone is not a measure of performance.

1.1.1 The microprocessor and the computer system

The shortest way to understand the microprocessor is to look at the job it does in a computer. This section defines the microprocessor, explains the buses that join it to memory and to input/output devices, and shows how the same processor takes a different form in a microcontroller.

1.1.1.1 What is a microprocessor?

At the center of every computer sits a unit that executes instructions. It is called the **central processing unit** (CPU). In the first computers the CPU was built from cabinets full of vacuum tubes, relays, or discrete transistors. As integrated-circuit technology advanced, the same circuits first fit onto a few boards and then onto a single piece of silicon. The microprocessor is the product of that last step.

Definition 1.1.1

A **microprocessor** is the central processing unit of a computer implemented on a single **chip**. It is a programmable device: the **machine-language** instructions stored in memory determine what it does. It reads these instructions in order and carries them out.

The word “programmable” in the definition separates the microprocessor from the circuits built in digital design courses. An adder circuit only adds; a counter only counts. What such a circuit does is fixed by its wiring, and changing its function means rewiring it. The wiring of a microprocessor, by contrast, never changes. What changes is the instructions in memory. The same chip edits text under one program and drives a motor under another. This flexibility has made the microprocessor a part of almost every electronic product today.

The microprocessor's work can be summarized in three steps. First the processor takes **input**: a key pressed on a keyboard, a temperature measured by a sensor, or a file read from a disk. Then it performs **arithmetic and logic operations** on that data: it adds, subtracts, compares, and masks bits. Finally it produces the desired **output**: it writes a character on the screen, turns a motor, or lights an LED. Throughout these three steps the processor talks to memory constantly. It reads both the instructions it will execute and the data it will work on from memory, and it writes intermediate results back to memory. Figure 1.1.1 shows this relationship.

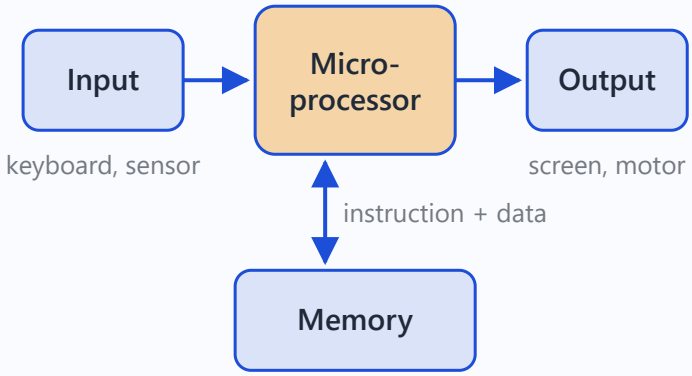


Figure 1.1.1: The microprocessor's workflow. Input (keyboard, sensor) enters the processor, is processed, and leaves as output (screen, motor). The processor takes instructions and data from memory and writes results back to memory.

A simple calculator is an everyday example of this flow. The keys are the input; addition is the processing; the result that appears on the display is the output. The small processor inside the calculator runs a program that reads the keys, adds the numbers, and writes the digits to the display. Changing that program means making the same chip do a different job.

The only language a microprocessor understands is **machine language**. In machine language every instruction is a sequence of bytes. On the 8086, for example, the bytes `B0 05` mean "load 5 into register AL." Because people find it hard to memorize such numbers, programs are mostly written in **assembly** language. Assembly gives the machine instructions short names such as `MOV`, `ADD`, and `JMP`, called **mnemonics**. A translator program, the assembler, converts this text into machine code. In this chapter instructions are written in assembly, and the corresponding machine-code bytes are given where needed.

1.1.1.2 The computer system and its buses

A microprocessor cannot work alone. It needs a **memory** that holds instructions and data, and **input/output** (I/O) devices that connect it to the outside world. These three parts, the CPU, memory, and I/O devices, are joined by a common bundle of wires. This bundle is called a **bus**. The bus that connects all parts of the system is called the **system bus**.

Definition 1.1.2

A **bus** is a group of parallel conducting lines that carry the same kind of information between units. A computer system has three buses:

- **Address bus**: selects the memory cell or I/O port the processor will access. It is unidirectional; only the processor generates the address.
- **Data bus**: carries the data being read or written. It is bidirectional; on a read, data flows from memory to the processor, and on a write, from the processor to memory.
- **Control bus**: carries timing and control signals such as read, write, memory or I/O select, clock, and interrupt.

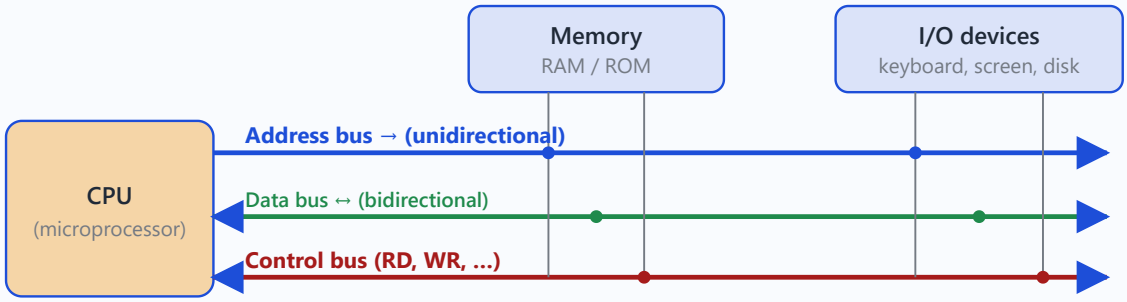


Figure 1.1.2: The CPU, memory, and I/O devices are connected by three buses. The address bus is unidirectional (out of the CPU), the data bus is bidirectional, and the control bus carries signals such as RD (read) and WR (write).

How a memory read proceeds shows clearly how the three buses share the work. The processor puts the address of the cell it wants to read on the address bus. Then it activates the read signal on the control bus (the `RD` pin on the 8086). The memory decodes the pattern on the address lines, selects the corresponding cell, and drives its contents onto the data bus. The processor captures the value on the data bus and releases the read signal. A write proceeds in a similar way; this time the processor drives the data onto the data bus, and the write signal (`WR`) is active on the control bus. Each of these steps happens in a fixed order tied to the clock signal. The time a read or write takes from start to finish is called a **bus cycle**.

The width of the buses, that is, the number of lines each has, sets the limits of the system. Each line of the address bus carries a 0 or a 1. With n lines, 2^n different patterns can be formed, and each pattern selects one memory cell. An n-line address bus therefore addresses 2^n bytes of memory. The 8086 address bus has 20 lines: $2^{20} = 1{,}048{,}576$ bytes, that is, 1 MB of memory. The width of the data bus is the number of bits transferred in one bus cycle. The 8086 data bus is 16 bits wide; in one bus cycle it carries 2 bytes, that is, one **word**.

Example 1.1.1

How many bytes of memory do address buses with 16, 20, and 24 lines address?

- 16 lines: $2^{16} = 65{,}536$ bytes = 64 KB (1 KB = 1024 bytes; 65,536 / 1024 = 64). The 8080 and 8085 are like this.
- 20 lines: $2^{20} = 1{,}048{,}576$ bytes = 1024 KB = 1 MB. The 8086 and 8088 are like this.
- 24 lines: $2^{24} = 16{,}777{,}216$ bytes = 16 MB. The 80286 is like this.

Looking at it the other way, selecting every byte of 1 MB individually takes $log_2(2^{20}) = 20$ lines. Each additional line doubles the addressable memory.

The 8086 has an interesting detail in its design: the processor is 16-bit, but its address bus is 20 bits. A 20-bit address cannot be held directly in 16-bit registers. The 8086 solves this by combining two 16-bit values, the **segment** and the **offset**. The physical address is found by multiplying the segment value by 16 and adding the offset. Throughout this chapter, addresses are mostly written with the offset alone, for example `0100h`, and the segment value is taken as fixed.

Common mistake

Confusing the width of the data bus with the amount of addressable memory. The conclusion "the 8086 is 16-bit, so it addresses 64 KB of memory" is wrong. It is the address bus, not the data bus, that determines addressable memory: the 8086's 20 address lines reach 1 MB. A 16-bit register can directly point only to a 64 KB window within a segment; the segment register slides this window around within the 1 MB.

1.1.1.3 Microprocessor and microcontroller

In a computer built around a microprocessor, memory and I/O devices are separate chips. On an 8086 board, the processor sits next to RAM chips, the ROM chip that holds the program, an interface chip for I/O, and a timer chip. The chips are joined by buses on the board. This arrangement is flexible: the amount of memory and the peripherals are chosen to fit the need. The price is board area, cost, and power consumption.

Many applications do not need this flexibility. The board of a washing machine runs a single program for years, makes do with a few kilobytes of memory, and manages a few buttons and a few motors. For jobs like this, devices were developed that put the CPU, memory, and peripherals on a single chip.

Definition 1.1.3

A **microcontroller** (MCU) puts the CPU together with RAM, Flash/ROM program memory, I/O ports, timers, and peripherals such as the analog-to-digital converter (ADC) on the same chip. For this reason it is also called a "computer on a chip." A microprocessor (MPU), by contrast, contains only the CPU; memory and peripherals are added externally.

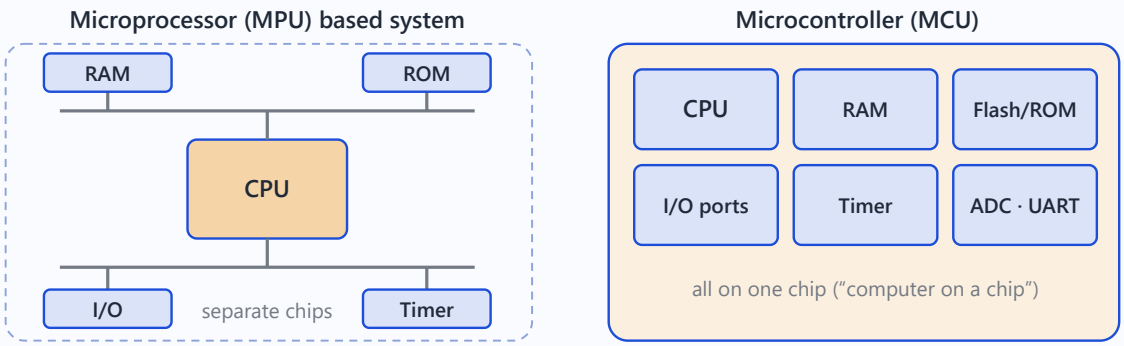


Figure 1.1.3: On the left, a microprocessor-based system: the CPU, RAM, ROM, I/O, and timer are separate chips. On the right, a microcontroller: the CPU, RAM, Flash/ROM, I/O ports, timer, ADC, and UART are on one chip.

Examples of microprocessors include the 8086, the Intel Core i7, and the AMD Ryzen. Examples of microcontrollers are the 8051, the PIC, the AVR in Arduino boards, and the STM32 family. Table 1.1.1 compares the two types on eight criteria.

Criterion	Microprocessor (MPU)	Microcontroller (MCU)
Contents	CPU (ALU, control unit, registers)	CPU + RAM + Flash + I/O + timer + ADC…
Memory	external, large (GB)	on chip, small (KB–MB)
Clock rate	high (GHz)	low to medium (MHz)
Power	high (W to 100 W)	very low (mW, µW in sleep mode)
Cost	high; extra chips are needed	low; a single chip is enough
Software	usually an operating system + many applications	a single, fixed program (embedded software)
Use	PC, server, laptop	washing machine, remote control, car ECU, IoT

Table 1.1.1: Comparison of the microprocessor and the microcontroller.

Most of the differences in the table come from a single design decision: where the memory sits. In an MPU the memory is external, so gigabytes of RAM can be attached. This makes an operating system and many simultaneous applications possible. In an MCU the memory is inside the chip, and chip area is limited. Memory measured in kilobytes is enough for a fixed, small program. Clock rate and power

follow the same split. An MPU running at a high frequency draws tens of watts and needs a heat sink. A low-frequency MCU stays at the milliwatt level and drops to microwatts in sleep mode; a battery can power it for years.

Example 1.1.2

Which type is chosen for a washing-machine controller and for a laptop?

The washing machine has a single, unchanging job: read the buttons, monitor the water level and temperature, and drive the motor and valves. The program is small; cost and power matter. A microcontroller is chosen for this job. A laptop, on the other hand, runs an operating system, wants gigabytes of memory, and the user may open a new application at any moment. Because it needs high performance and large memory, a microprocessor is chosen.

The difference between a microprocessor and a microcontroller is about what is packed onto the chip; the operating principle of the CPU inside is the same. Both fetch, decode, and execute instructions from memory. Both have registers, an arithmetic logic unit, and flags. This course studies the 8086 microprocessor. The concepts learned on the 8086's buses, registers, and instructions carry over with the same logic when you move to a microcontroller.

1.1.2 The instruction cycle

Everything a microprocessor does is nothing more than the repetition of a single cycle. The processor takes an instruction from memory, works out what it asks for, does the work, and puts the result in place. Then it repeats the same thing for the next instruction. This section defines the stages of the cycle, traces the cycle on the machine code of a three-instruction 8086 program, and shows how the program counter directs the flow.

1.1.2.1 Fetch, decode, execute

Definition 1.1.4

The **instruction cycle** is the sequence of stages the processor repeats for every instruction:

1. **Fetch**: reads the instruction from memory at the address the program counter points to.
2. **Decode**: decodes the **opcode**; determines which operation is to be performed on which **operands**.
3. **Execute**: carries out the decoded instruction; computes in the ALU, moves data, or jumps.
4. **Result**: writes the result to a register or to memory and updates the flags.

Some books count the fourth stage as part of execution and call the cycle simply “fetch–decode–execute.” Whichever naming is chosen, the order is the same. As long as the processor is on, this cycle runs without ever stopping. The arrow in the figure returns from the last stage to the first; the only situation in which the cycle ends is when the processor is halted.

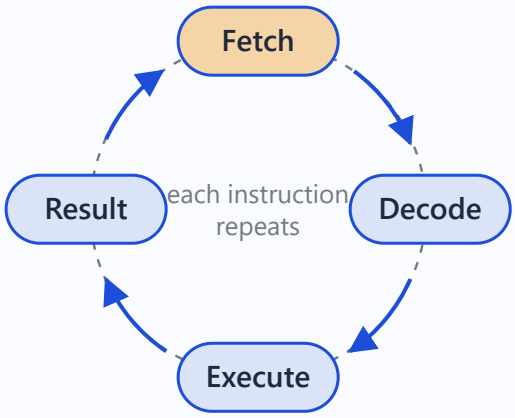


Figure 1.1.4: The fetch, decode, execute, and result stages form a cycle. The cycle starts over for every instruction.

A different part of the processor handles each stage. The bus interface does the fetching: it puts the address in the program counter on the address bus, issues the read signal, and takes in the bytes arriving on the data bus. The control unit does the decoding: it interprets the opcode byte as if looking it up in a table and generates the control signals that determine which units will work. In execution the ALU comes into play, or data is moved from one register to another. In the result stage the value is written to the register file or to memory.

Because the stages run in sequence, the number of instructions executed per unit time is limited: while fetching goes on the ALU waits idle, and while execution goes on the bus waits idle. Fast processors close this gap by overlapping the stages. While one instruction is executing, the next is being decoded and the one after that is being fetched. This arrangement is called a **pipeline**; it is the basic technique of RISC processors. The 8086 uses a simple form of this idea: while the execution unit works, the bus interface unit prefetches the upcoming instruction bytes and holds them in a 6-byte queue.

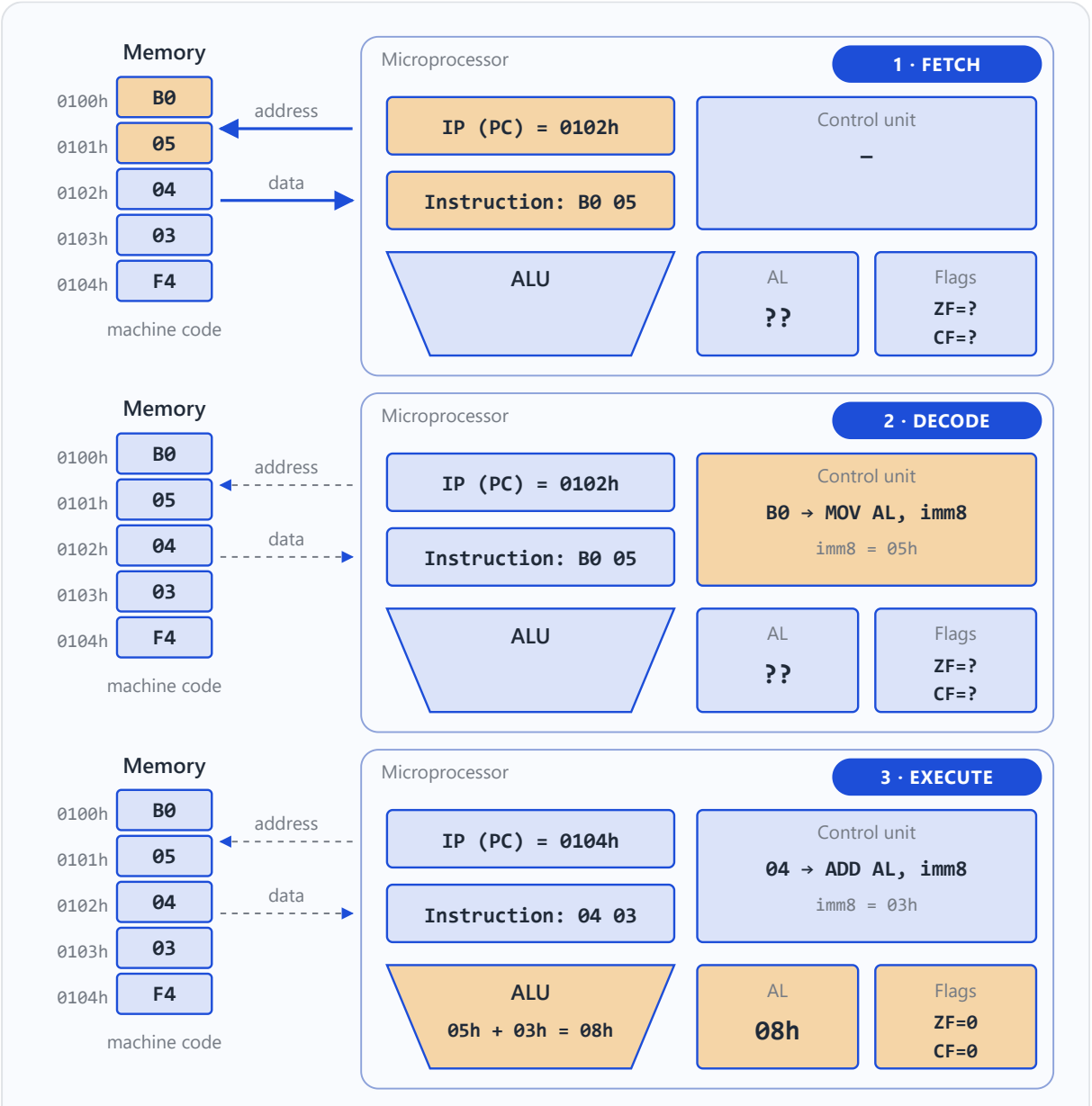
1.1.2.2 Tracing machine code

To see the cycle concretely, take a program of three instructions:

```
mov al, 5      ; B0 05
add al, 3      ; 04 03
hlt            ; F4
```

The assembler translates this text into five bytes of machine code. The program is loaded into memory starting at address `0100h`: `B0` is at `0100h`, `05` at `0101h`, `04` at `0102h`, `03` at `0103h`, and `F4` at `0104h`. The first byte is the opcode. `B0` means "load an 8-bit constant into register AL," and the constant is in the byte right after the opcode. A constant embedded in the instruction like this is called an **immediate value**. The opcode `04` means "add an 8-bit immediate to AL." `F4` is the `HLT` instruction; it takes no operand and is a single byte.

On the 8086 the program counter is called the **instruction pointer** (IP). IP always holds the offset of the next byte to be fetched. Initially IP = `0100h`. Figure 1.1.5 shows four moments of the cycle.



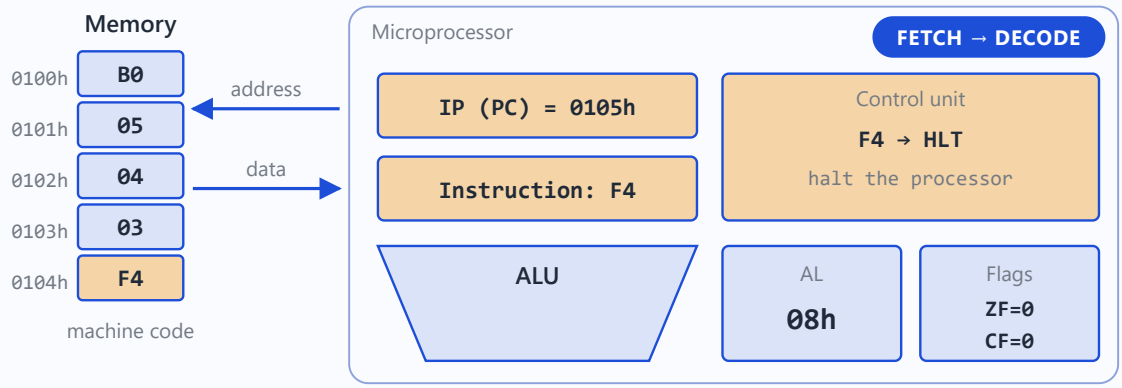


Figure 1.1.5: The instruction cycle in the program `MOV AL,5 · ADD AL,3 · HLT`. From top to bottom: the first instruction has been fetched (IP = 0102h); `B0` has been decoded; the second instruction has been executed (AL = 08h, ZF = 0, CF = 0); `F4` has been fetched and decoded (IP = 0105h).

The steps of the trace are as follows:

1. **Fetch.** The `0100h` in IP is put on the address bus. Memory returns the byte `B0`. From the opcode the control unit learns that the instruction is 2 bytes long and fetches the byte `05` as well. IP grows by the instruction length: `0100h + 2 = 0102h`.
2. **Decode.** The control unit decodes `B0` as "load an 8-bit constant into AL"; the constant is `05h`.
3. **Execute and result.** `05h` is written to register AL. The ALU is not used. The `MOV` instruction changes no flags; the flags keep their previous values.
4. **Fetch.** The cycle starts again. The `04 03` at `0102h` is fetched; IP becomes `0104h`.
5. **Decode.** `04` = "add an 8-bit constant to AL." The operands are the `05h` in AL and the constant `03h`.
6. **Execute and result.** The ALU computes `05h + 03h = 08h` and writes the result to AL. Because the result is not zero, the zero flag is ZF = 0. Because there is no carry out of the top bit, the carry flag is CF = 0.
7. **Fetch and decode.** The `F4` at `0104h` is fetched; IP becomes `0105h`. `F4` is decoded as `HLT`, and the processor stops until an interrupt or a reset arrives.

Address	Machine code	Instruction	IP (after)	AL	ZF	CF
—	—	initial	0100h	??	?	?
0100h	`B0 05`	`mov al, 5`	0102h	05h	?	?
0102h	`04 03`	`add al, 3`	0104h	08h	0	0
0104h	`F4`	`hlt`	0105h	08h	0	0

Table 1.1.2: Trace of the program `MOV AL,5 · ADD AL,3 · HLT`. Each row gives the state after that instruction has executed.

`ADD` updates not only ZF and CF but all six of the 8086's status flags. In the result `08h = 0000 1000b` the top bit is 0, so the sign flag is SF = 0. Because the sum of two positive numbers stays positive, the overflow flag is OF = 0. In the low four bits `5 + 3 = 8` fits in 4 bits, so the auxiliary carry flag is AF = 0. The low byte of the result has an odd number (one) of 1 bits, so the parity flag is PF = 0. Only ZF and CF are shown in the table.

Two details of this trace matter. First, IP is incremented as soon as an instruction is fetched; while the instruction executes, IP already points to the next one. Jump instructions steer the flow by changing this value. Second, the processor does not know whether bytes are "instruction" or "data." The `05` at `0101h` is read as a constant, not as an opcode, because when the control unit decodes `B0` it knows a constant follows. Had IP gone to `0101h` by mistake, the processor would have treated `05` as an opcode and executed an entirely different instruction.

On a real 8086, fetching works a little differently from this description. While the execution unit (EU) is busy with one instruction, the bus interface unit (BIU) prefetches the next bytes and stores them in a 6-byte prefetch queue. When the EU finishes an instruction, the bytes of the next one are usually already waiting in the queue, so there is no need to wait for memory. From the programmer's point of view nothing changes: instructions still run in order with the same effects.

> **Common mistake**
>
> It is often assumed that IP grows by 1 with every instruction. IP holds a byte address, not an instruction count, and it grows by the length of the instruction. 8086 instructions are 1 to 6 bytes long: `hlt` is 1 byte, `mov al, 5` is 2, and `mov cx, 3` is 3. To find the address of an instruction, add up the lengths of the instructions before it.

1.1.2.3 Three basic functions

Although a processor's instruction set may contain hundreds of instructions, every one of them serves one of three functions.

Arithmetic and logic operations. Addition, subtraction, comparison, AND, OR, XOR, and shifts belong to this group. These operations are performed by the **arithmetic logic unit** (ALU). Newer processors also work with floating-point numbers; a separate **floating-point unit** (FPU) handles that work. Examples on the 8086: `ADD AX, BX` (add BX to AX), `CMP AL, 10` (compare AL with 10), `AND AL, 0Fh` (clear the upper four bits of AL).

Data movement. Data is copied from one place to another among registers, memory, and I/O. This is the most common operation in programs; most of the code compilers generate consists of move instructions. Examples: `MOV AX, [1234h]` (read the word in memory into AX), `MOV [BX], AL` (write AL to the address BX points to), `IN AL, 60h` (read a byte from I/O port 60h).

Changing the program counter. The program counter holds the address of the next instruction. Writing a new value into it makes the flow jump elsewhere. Conditional jumps look at the flags and decide whether to jump; every `if` and every loop in a program is built this way. Examples: `JMP label` (jump unconditionally), `JZ is_zero` (jump if ZF = 1), `CALL subroutine` (save the return address and jump to the subroutine).

Every program is a combination of these three functions: computing, moving, and changing the order. A sorting program reads numbers (movement), compares them (computation), and swaps or skips them depending on the result (changing the order).

1.1.2.4 The program counter and loops

In sequential flow IP increases by itself. To build a loop, a backward address must be written into IP. The following program computes the sum 3 + 2 + 1. The numbers on the left are the offsets of the instructions in memory (in hexadecimal).

```
0100  mov  cx, 3     ; count = 3
0103  mov  ax, 0     ; sum = 0
0106  again: add ax, cx
0108  loop again     ; CX--, jump if CX≠0
010A  hlt            ; halt
```

`mov cx, 3` puts the count into CX; this instruction, which loads a 16-bit constant into a 16-bit register, is 3 bytes (`B9 03 00`). `mov ax, 0` clears the sum and is also 3 bytes. `add ax, cx` adds two registers and is 2 bytes. `loop again` first decrements CX by 1; if CX is not zero it jumps to the label `again` , and if it is zero it moves on to the next instruction. `loop` is 2 bytes and changes no flags. `hlt` stops the processor.

Step	Executed	IP	CX	AX	Note
1	`mov cx, 3`	0103h	0003h	????	advances 3 bytes
2	`mov ax, 0`	0106h	0003h	0000h	sum cleared
3	`add ax, cx`	0108h	0003h	0003h	0 + 3
4	`loop again`	0106h	0002h	0003h	CX = 2 ≠ 0, jump
5	`add ax, cx`	0108h	0002h	0005h	3 + 2
6	`loop again`	0106h	0001h	0005h	CX = 1 ≠ 0, jump
7	`add ax, cx`	0108h	0001h	0006h	5 + 1
8	`loop again`	010Ah	0000h	0006h	CX = 0, no jump
9	`hlt`	010Bh	0000h	0006h	processor stops

Table 1.1.3: IP trace. Each row gives the state after that instruction has executed.

The fourth row of the table shows how sequential flow is broken. When `loop` was fetched, IP had become `010Ah`. Because CX is not zero, `loop` writes `0106h` into IP, and the next fetch is made from `add ax, cx`. In machine code the target address is not written directly. The second byte of the `loop` instruction is a signed displacement added to the address of the next instruction: `0106h − 010Ah = −4`, that is, `FCh`. The instruction is therefore encoded as `E2 FC`, and the program still works correctly if it is moved to another address in memory.

The loop body runs three times, because CX starts at 3 and each `loop` decrements it by one. Nine instructions are executed in total, and the result is AX = 3 + 2 + 1 = 6. Using CX as the loop counter is not a habit but a necessity: the `loop` instruction always uses CX as its counter.

> **Common mistake**
>
> It is often assumed that `loop` first tests CX and then decrements it. The order is the reverse: CX is decremented first, and then the new value is tested. So if a loop is entered with CX = 0, the first `loop` makes CX `FFFFh` and the body runs 65,536 times.

1.1.3 Structure of the microprocessor

Different units of the processor take care of the stages of the instruction cycle. This section introduces those units, explains the clock signal that drives them all in step, and shows how to compute the running time of a program. It ends with what the bit width of a processor means and how a 16-bit processor adds 32-bit numbers.

1.1.3.1 Structural components

The inside of a microprocessor consists of four main units: the control unit, the registers, the ALU, and the bus interface. These units are joined by an **internal data bus** on the chip. The bus interface connects the internal world to the external buses, that is, to memory and I/O devices. Figure 1.1.6 shows this arrangement.

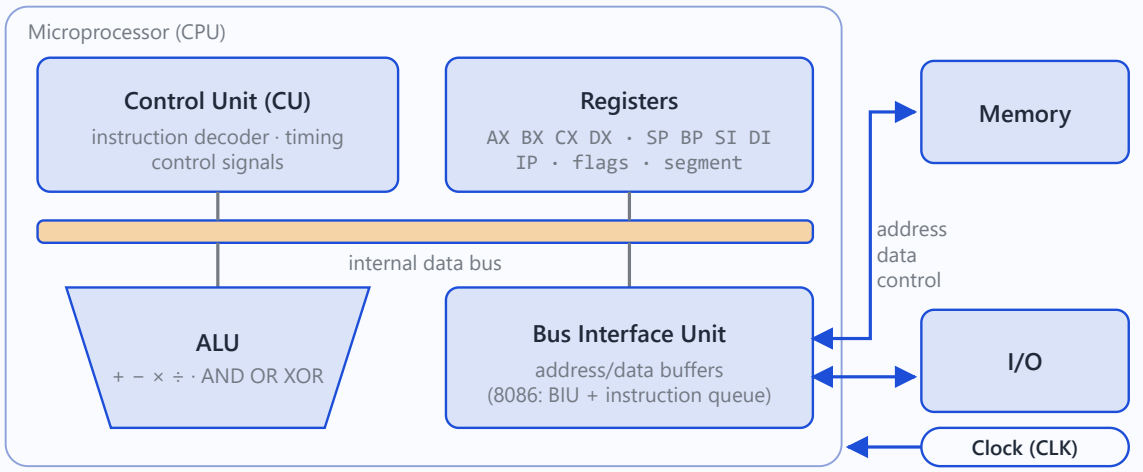


Figure 1.1.6: The internal structure of the microprocessor. The control unit, registers, ALU, and bus interface unit are joined by the internal data bus. The control unit sends control signals to the other units. The bus interface drives the external address, data, and control buses. All units work in step with the clock signal (CLK).

Arithmetic logic unit (ALU). It performs arithmetic (+, −) and logic (AND, OR, NOT, XOR) operations on binary numbers, as well as comparison and shifting. The full adders, multiplexers, and shifters built in digital design are the building blocks of the ALU. The ALU takes its two inputs, produces a result according to the operation-select signals from the control unit, and writes properties of the result into the **flags**. If the result is zero, the zero flag (ZF) becomes 1. If a carry came out of the top bit, the carry flag (CF) is set; if the top bit of the result is 1, the sign flag (SF) is set. If the signed result did not fit, the overflow flag (OF) is set. Conditional jumps look at these flags.

Control unit (CU). It is the conductor of the processor. It decodes the instruction; it determines which register is connected to the internal bus, which operation the ALU performs, and when the memory read and write signals are issued. A control unit can be built in two ways. In a **hardwired** control unit the signals for each instruction are generated directly by logic gates; this is fast but gets hard to design for complex instruction sets. In a microprogrammed control unit each instruction is translated into a sequence of steps stored in a small ROM on the chip, called **microcode**. The 8086 is a microprogrammed processor.

Registers. These are small, very fast storage cells inside the processor. Each register is a latch built from flip-flops and can be read and written in one clock cycle. Going to memory costs a bus cycle, that is, several clock cycles. For this reason programs keep frequently used values in registers. Registers are grouped by purpose: general-purpose (data) registers, address and pointer registers, the status (flags) register, and the program counter. On the 8086 the general-purpose registers are AX, BX, CX, and DX. SP, BP, SI, and DI are pointer and index registers. IP is the program counter and FLAGS is the flags register; CS, DS, SS, and ES are the segment registers.

Bus interface. It drives the address, data, and control buses. Every byte to be read from or written to memory passes through this unit. On the 8086 this unit is called the **bus interface unit** (BIU). The BIU computes the physical address, runs the bus cycles, and prefetches instruction bytes and holds

them in a 6-byte queue. The part that decodes and executes the instructions is the **execution unit** (EU). Because the two units work at the same time, fetching and execution overlap.

Clock and timing. All units of the processor advance in step with the pulses of the clock signal. A register takes its new value only on a clock edge; each step of a bus cycle falls on a specific clock cycle. The duration of one pulse is $T = 1/f$.

Cache, FPU, and cores. On the 8086 these units are not on the chip. Modern processors, however, have a cache, a floating-point unit, several cores, and a graphics unit on the same chip. These units are among the main features added to processors since the 8086.

Example 1.1.3

While the `add ax, bx` instruction executes, what does each unit do?

The BIU fetches the bytes `03 C3` from the address IP points to and puts them in the queue (most likely while the previous instruction was executing). The EU takes the bytes from the queue; the control unit decodes the opcode `03` as "add a word to a register" and the byte `C3` as "destination AX, source BX." The control unit connects AX and BX through the internal bus to the ALU inputs and selects addition in the ALU. The ALU computes the sum; the result is written to AX and the flags are updated. Because there is no memory access, the bus interface is idle during this time and can fetch the bytes of the next instruction.

1.1.3.2 The clock signal and the period

The clock signal is a square wave that alternates between 0 and 1 at a given frequency. On the 8086 this signal is produced by the 8284A clock generator chip and fed to the processor's CLK pin. The clock signal does the same job as the clock of the synchronous circuits in digital design: it makes all flip-flops change state at the same moment.

Definition 1.1.5

The **clock frequency** f is the number of pulses of the clock signal per second, measured in hertz (Hz). One pulse is called a **clock cycle**, or simply a **cycle**. The duration of a cycle is the **period**:

$$T = 1 / f$$

An instruction usually takes several clock cycles. On the 8086 a memory access, that is, one bus cycle, is at least four clock cycles. These cycles are called **T states** and are named T1, T2, T3, and T4. Roughly, the address is put on the bus in T1 and the read or write signal becomes active in T2. In T3

the data becomes valid on the bus, and in T4 the bus cycle ends. If the memory is slow, **wait states** (Tw) are inserted between T3 and T4.

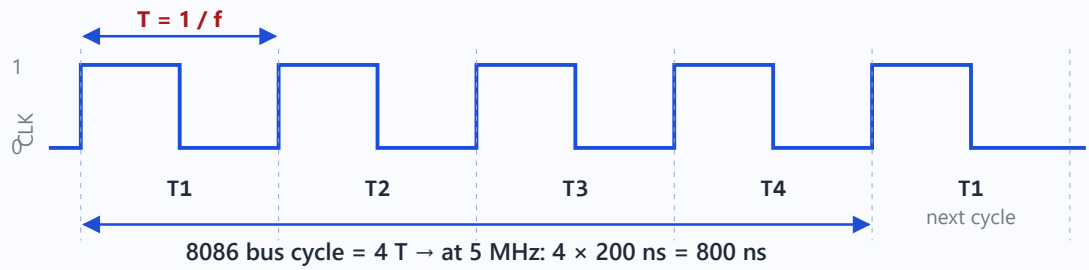


Figure 1.1.7: A square-wave clock signal. One period is *T*; one 8086 bus cycle consists of four T states, T1 to T4. At 5 MHz one bus cycle takes *4 × 200 ns = 800 ns*.

f	*T = 1/f*	*4T*
5 MHz (8086)	200 ns	800 ns
10 MHz (8086-1)	100 ns	400 ns
3 GHz	0.333 ns	1.33 ns

Table 1.1.4: Period and the duration of a four-cycle bus cycle for three clock frequencies.

Example 1.1.4

On a 5 MHz 8086, what are the clock period and the duration of a four-cycle bus cycle?

f = 5 MHz = 5 × 10^6 Hz. The period is *T = 1 / (5 × 10^6) s = 0.2 × 10^{-6} s = 200 ns*. Because a bus cycle takes four cycles, it lasts *4 × 200 ns = 800 ns*. This means a 5 MHz 8086 can access memory at most *1 / 800 ns = 1.25* million times per second. Since 2 bytes arrive with each access over the 16-bit data bus, this is an upper limit of 2.5 MB per second.

1.1.3.3 Clock rate, CPI, and execution time

The clock period gives the duration of one cycle, but it does not by itself say how long a program will take. Two more quantities are needed: the number of instructions the program executes, *N*, and how many cycles an instruction takes on average.

Definition 1.1.6

Cycles per instruction (CPI) is the average number of clock cycles spent per instruction in a program. The **CPU time** of a program of N instructions is:

$$CPU\ time = N \times CPI \times T = N \times CPI / f$$

The number of instructions executed per second is f / CPI. When this number is given in millions, its unit is **MIPS** (million instructions per second).

The three factors of the formula correspond to three different levels of design. N is determined by the program and the compiler. CPI depends on the processor's architecture: pipelining, caches, and parallel execution units lower CPI. f depends on the fabrication technology and the circuit design. Improving one often worsens another. For example, simplifying instructions lowers CPI but increases N.

The interactive calculator in the slides applies this formula to four historical processors. The CPI values are only rough examples. 8086 instructions take around 10 cycles on average. Modern processors, on the other hand, can finish more than one instruction in a cycle; their CPI drops below 1. Table 1.1.5 collects the calculator's results for a program of 1 million instructions.

Processor	f	CPI	T	$4T$	MIPS	Time
8086	5 MHz	10	200 ns	800 ns	0.5	2 s
80386	33 MHz	4	30.3 ns	121 ns	8.25	121 ms
Pentium	66 MHz	1.5	15.2 ns	60.6 ns	44	22.7 ms
Core i7	3.33 GHz	0.5	0.30 ns	1.20 ns	6660	0.15 ms

Table 1.1.5: Period, bus cycle, speed, and execution time for N = 1 million instructions. The values were obtained with the calculator's presets.

In the last column of the table there is a gap of about 13,000 times between the 8086 and the Core i7. Only about 670 times of this gap comes from clock frequency ($3330 / 5 = 666$). The remaining factor of 20 comes from the drop in CPI, that is, from advances in architecture ($10 / 0.5 = 20$). The 4T column is for comparison only; the buses of modern processors do not work with four-cycle bus cycles like the 8086's.

Example 1.1.5

How many seconds does a program of $N = 2 \times 10^9$ instructions take on a 3 GHz processor with CPI = 1.5?

$$time = N \times CPI / f = 2 \times 10^9 \times 1.5 / (3 \times 10^9) = 3 \times 10^9 / (3 \times 10^9) = 1\ s$$

> The processor executes $f / CPI = 3 \times 10^9 / 1.5 = 2 \times 10^9$ instructions per second, that is, 2000 MIPS; 2 billion instructions therefore take exactly 1 second.

1.1.3.4 Counting the clock cycles of a loop

CPI is an average. To find the running time of a real program, the cycle counts of the individual instructions are added up. Intel's 8086 datasheet gives a cycle count for each instruction. The following loop computes the sum 4 + 3 + 2 + 1; the comment on each instruction gives its cycle count.

```
org 100h
mov cx, 4        ; 4 cycles
mov ax, 0        ; 4 cycles
again: add ax, cx ; 3 cycles
loop again       ; 17 / 5 cycles
hlt              ; 2 cycles
```

Cycle counts vary with the type of instruction. A `mov` that loads an immediate value into a register takes 4 cycles. A register-to-register `add` takes 3 cycles; it is short because there is no memory access. The `loop` instruction has two durations: a taken `loop` takes 17 cycles and a not-taken `loop` takes 5. The difference comes from the jump emptying the prefetch queue. The bytes waiting in the queue are now invalid, because the flow has gone to another address; the BIU has to start fetching again from the target address. `hlt` takes 2 cycles.

Step	Instruction	Cycles	Total cycles	Time	CX	AX
1	`mov cx, 4`	4	4	0.8 µs	4	?
2	`mov ax, 0`	4	8	1.6 µs	4	0
3	`add ax, cx`	3	11	2.2 µs	4	4
4	`loop` (taken)	17	28	5.6 µs	3	4
5	`add ax, cx`	3	31	6.2 µs	3	7
6	`loop` (taken)	17	48	9.6 µs	2	7
7	`add ax, cx`	3	51	10.2 µs	2	9
8	`loop` (taken)	17	68	13.6 µs	1	9
9	`add ax, cx`	3	71	14.2 µs	1	10
10	`loop` (not taken)	5	76	15.2 µs	0	10
11	`hlt`	2	78	15.6 µs	0	10

Table 1.1.6: Clock-cycle count of the loop (5 MHz, 1 cycle = 200 ns). Each row gives the state after that instruction.

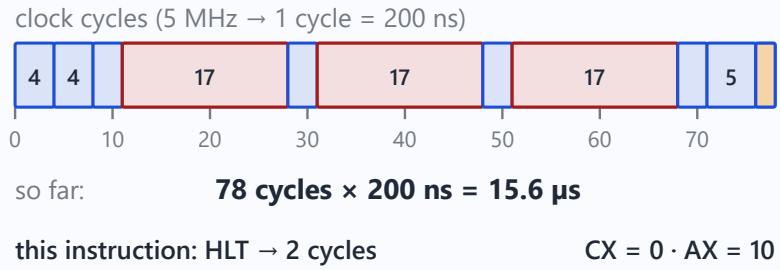


Figure 1.1.8: The timeline bar of the loop. Each rectangle is the cycle count of one instruction: 4, 4, then three times 3 + 17, and finally 3 + 5 and 2. The total of 78 cycles comes to 15.6 µs at 5 MHz.

The total can also be found with a formula. The body runs four times; `loop` jumps three times and falls through once:

$$4 + 4 + 4 \times 3 + 3 \times 17 + 5 + 2 = 8 + 12 + 51 + 5 + 2 = 78 \text{ cycles}$$

$$78 \times 200 \text{ ns} = 15{,}600 \text{ ns} = 15.6 \text{ µs}$$

The result is striking: of the 78 cycles, 51, more than half, are spent in the taken `loop` instructions. The addition, the real work, takes only 12 cycles. Loop overhead dominates the running time in loops with small bodies. This is why, where speed is needed, the loop body is written unrolled a few times to reduce the number of jumps. The average CPI of this program can also be computed: since 11 instructions take 78 cycles, *CPI = 78 / 11 ≈ 7.1*.

The cycle counts in the datasheet are best-case values. They assume that the instruction bytes are waiting ready in the queue and that the memory adds no wait states. The real time also includes fetch stalls when the queue is empty and Tw states added for slow memory; the measured time can therefore come out a little longer. In every case the time is *cycle count × T*.

Common mistake

When computing the time of a loop, all passes are wrongly counted of the `loop` instruction with the same duration. A taken and a not-taken `loop` take different times (17 and 5 cycles). In a loop that runs *n* times, `loop` jumps *n − 1* times and falls through on the last pass.

1.1.3.5 The growth of clock rates

The clock rates of Intel processors rose from megahertz to gigahertz in forty years. In the 16-bit era the 8086 was produced in 5, 8, and 10 MHz versions. The 8088 ran at 5 and 8 MHz; the 8088 in the IBM PC runs at 4.77 MHz. The 80186/80188 started at 6 MHz, and the 80286 started with 6, 8, 10, and 12.5 MHz versions. In the 32-bit era the 80386 ran in the 16–33 MHz range and the 80486 in the

16–100 MHz range. The DX2 and DX4 versions of the 80486 use a clock multiplier: the on-chip clock is two or three times the external bus clock. The Pentium started at 60 and 66 MHz. In the 64-bit era the Core 2 runs in the 1.2–3 GHz range. The first-generation Core i7 is in the 2.66–3.33 GHz range, the Core i5 in 2.4–3.6 GHz, and the Core i3 in 2.93–3.33 GHz.

The rise from 5 MHz to about 3 GHz is an increase of roughly 600 times in thirty years. Around 2005 this increase stopped. As the clock frequency rises, more transistors switch every second; dynamic power grows in proportion to the frequency and to the square of the supply voltage. The limit of removing the heat the chip produces had been reached. After that, performance gains came mostly from increasing the number of cores and finishing more instructions per cycle, that is, from lowering CPI.

1.1.3.6 Bit width

Definition 1.1.7

The **bit width** (word length) of a processor is the number of bits of the word it can process at once: the size of the registers and the ALU, and usually the width of the data bus as well. This is what "*n*-bit processor" means. With n bits, the unsigned range $0 \ldots 2^n - 1$ can be represented.

Word	Unsigned range
8 bits	0 … 255
16 bits	0 … 65,535
32 bits	0 … 4,294,967,295
64 bits	0 … ≈ 1.8×10^{19}

Table 1.1.7: Unsigned range by word width.

Bit width limits the size of numbers but does not stop a processor from working with larger numbers. An 8-bit processor can also add 32-bit numbers; it just does so in several steps. As the width grows, the same job finishes in fewer instructions.

The width of the address bus is a separate quantity; it determines the amount of memory that can be reached: n lines address 2^n bytes. Table 1.1.8 gives examples from Intel processors.

Address bus	Memory	Example
16 bits	64 KB	8080, 8085
20 bits	1 MB	8086, 8088
24 bits	16 MB	80286
32 bits	4 GB	80386, 80486

Address bus	Memory	Example
36 bits	64 GB	Pentium Pro (PAE)

Table 1.1.8: Address bus width and addressable memory.

In the processors in the table, data width and address width are not always equal. The 8-bit 8080 has a 16-bit address bus. The 16-bit 8086 has a 20-bit address bus. The 20-bit address is produced by combining two 16-bit values, the segment and the offset: *physical address = segment × 16 + offset*. For example, if the segment is `1234h` and the offset is `0010h`, the physical address is `12340h + 0010h = 12350h`.

1.1.3.7 A 32-bit addition on a 16-bit processor

To see the practical effect of bit width, let us add two 32-bit numbers on the 16-bit 8086: *a = 0001FFFFh = 131,071* and *b = 00030002h = 196,610*. These numbers do not fit in a 16-bit register. Each is split into two halves and held in a register pair. For *a* the high half goes into DX and the low half into AX; this pair is written DX:AX. *b* goes into the pair CX:BX.

```
org 100h
mov ax, 0FFFFh ; a: low half
mov dx, 0001h  ; a: high half
mov bx, 0002h  ; b: low half
mov cx, 0003h  ; b: high half
add ax, bx     ; low halves
adc dx, cx     ; high + CF
ret
```

The first four instructions load the numbers into registers. The leading 0 in `0FFFFh` is there so the assembler does not mistake the number for a label name; hexadecimal numbers that start with a letter are written this way. `add ax, bx` adds the low halves. `adc` (add with carry) adds the high halves and also adds the carry from the previous addition, that is, CF. When adding on paper digit by digit, `adc` plays for 16-bit "digits" the same role as adding the carry into the next digit.

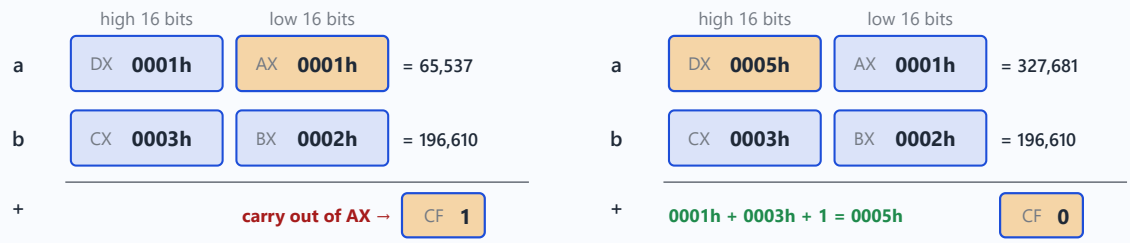


Figure 1.1.9: Two moments of the 32-bit addition. On the left, `add ax, bx`: bit 17 of the sum `FFFFh + 0002h` does not fit in AX; AX = `0001h`, CF = 1. On the right, `adc dx, cx`: `0001h + 0003h + 1 = 0005h`; the result is DX:AX = `00050001h`.

Instruction	DX	AX	CX	BX	CF	Note
`mov ax, 0FFFFh`	????	FFFFh	????	????	0	low half of *a*
`mov dx, 0001h`	0001h	FFFFh	????	????	0	DX:AX = 131,071
`mov bx, 0002h`	0001h	FFFFh	????	0002h	0	low half of *b*
`mov cx, 0003h`	0001h	FFFFh	0003h	0002h	0	CX:BX = 196,610
`add ax, bx`	0001h	0001h	0003h	0002h	1	`FFFFh + 0002h = 1 0001h`
`adc dx, cx`	0005h	0001h	0003h	0002h	0	`0001h + 0003h + 1`

Table 1.1.9: Trace of the 32-bit addition. Each row gives the state after that instruction.

The critical step is `add ax, bx`. The result `FFFFh + 0002h = 10001h` is 17 bits. The topmost 1 bit does not fit in the 16-bit AX: `0001h` remains in AX and the overflowing bit is written to CF. This instruction updates the other flags too: ZF = 0 because the result is not zero, and SF = 0 because the top bit is 0. In the signed interpretation `FFFFh` is −1, so *−1 + 2 = 1* is a correct result and OF stays 0. In the low four bits *Fh + 2h = 11h* overflows, so AF = 1. `adc dx, cx` then computes *0001h + 0003h + CF(1) = 0005h*; there is no new overflow, so CF = 0. The result is DX:AX = `00050001h`. Decimal check: *5 × 65,536 + 1 = 327,681 = 131,071 + 196,610*.

The 16-bit processor did the 32-bit addition in two instructions. A 32-bit processor, for example the 80386, does the same job with a single `add`: `add eax, ebx`. A 64-bit addition takes four instructions on the 8086 (an `add` and three `adc`), two on the 80386, and one on a 64-bit processor. This is the effect of bit width on performance.

Common mistake

If the high halves are added with `add` as well, the carry from the low half is lost, and the result comes out as DX:AX = `00040001h` = 262,145, exactly 65,536 less than the correct result. Also, no instruction that changes CF (a `cmp`, for example) may be placed between `add` and `adc`; `mov`, however, changes no flags and may come in between.

1.1.4 Instruction set architectures

Processors can be classified in many ways. Three questions stand out: How are the instructions designed? How much data does an instruction process? What is the processor optimized for? This section takes up the first question, the instruction set architecture: CISC, RISC, and EPIC. Pipelining, the most important technique these architectures rely on, is also followed step by step.

1.1.4.1 Overview of processor types

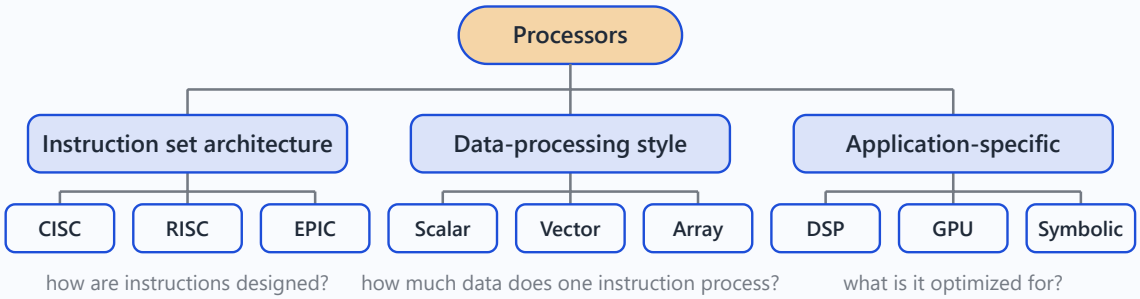


Figure 1.1.10: Classifying processors by three criteria. By instruction set architecture: CISC, RISC, and EPIC; by data-processing style: scalar, vector, and array; by application: DSP, GPU, and symbolic processors.

The **instruction set** of a processor is the full collection of instructions it can execute. The **instruction set architecture** (ISA) covers these instructions and also the registers, addressing modes, data types, and the way instructions are encoded in memory. The ISA is the contract between software and hardware: two processors that implement the same ISA run the same machine code, however different their internal circuits may be. The three approaches take their names from the following:

- **CISC**: Complex Instruction Set Computer.
- **RISC**: Reduced Instruction Set Computer.
- **EPIC**: Explicitly Parallel Instruction Computing.

1.1.4.2 CISC

The CISC approach aims to perform complex operations with a single instruction. Its roots go back to the 1960s and 1970s. In those years memory was expensive and slow, and programs were mostly written by hand in assembly. Doing a job with one instruction both saved memory and made the programmer's work easier.

Definition 1.1.8

In a **CISC** architecture a single instruction can include several low-level operations, such as loading from memory, an arithmetic operation, and storing back to memory. Instructions have several addressing modes and direct memory operands. Instruction lengths vary; the number of registers is relatively small; the control unit is usually microprogrammed.

The 8086 is a typical CISC processor. Its instructions are 1 to 6 bytes long; the number of bytes after the opcode varies with the operands. The control unit therefore learns how many bytes an instruction has only after decoding its first bytes, so decoding is complex. There are four general-purpose

registers (AX, BX, CX, DX). Many instructions can use memory directly as an operand. Two typical examples:

```
ADD [BX], AX   ; mem = mem + AX
REP MOVSB      ; copy CX bytes
```

`ADD [BX], AX` adds AX to the memory word BX points to: in a single instruction it reads memory, adds, and writes the result back to memory. `REP MOVSB` copies a block of memory CX bytes long from the place SI points to the place DI points to; one instruction takes the place of a loop. Other examples of the CISC family are the Intel 80386, 80486, and Pentium series, the Motorola 68000, 68020, and 68040, the DEC VAX, and the IBM System/360.

1.1.4.3 RISC

Measurements made in the late 1970s gave an interesting result. The code compilers generated used only a very small fraction of the complex instructions in CISC instruction sets. Most of a program consisted of simple load, store, add, and jump instructions. The complex instructions, meanwhile, enlarged the control unit and slowed the simple instructions too. The RISC approach was born from this observation.

Definition 1.1.9

A **RISC** architecture has a small number of simple, fixed-length instructions (for example, all 32 bits). Only **load/store** instructions access memory; arithmetic and logic operations are done only between registers. To reduce memory accesses there are many registers (32 or more). The control unit is hardwired and instructions execute in a pipeline. The goal, in the ideal case, is to complete one instruction every clock cycle (CPI ≈ 1).

Fixed-length instructions make decoding easier: every instruction starts at the same place, and the opcode and register numbers are always in the same bit fields. Because only load/store instructions access memory, the great majority of instructions finish in the same time. This regularity is a precondition for pipelining. The complexity does not disappear; it moves to the compiler: the work of one CISC instruction takes several instructions in RISC, and arranging those instructions in a suitable order is the compiler's job.

Examples of RISC are the IBM RS/6000 (POWER), the Motorola MC88100, the DEC Alpha 21064, 21164, and 21264, MIPS, and SPARC. The most widespread RISC family today is ARM: nearly all phones and tablets, and Apple's M-series computers, contain ARM cores. RISC-V is an open instruction set that requires no license fee and is spreading quickly both in academia and in embedded systems.

The RISC goal of "one instruction per cycle" does not mean an instruction finishes in a single cycle. An instruction still passes through several stages. Thanks to the pipeline, one instruction completes every cycle; what increases is not the time of a single instruction (its latency) but the number of instructions finished per unit time (the **throughput**).

1.1.4.4 Pipelining

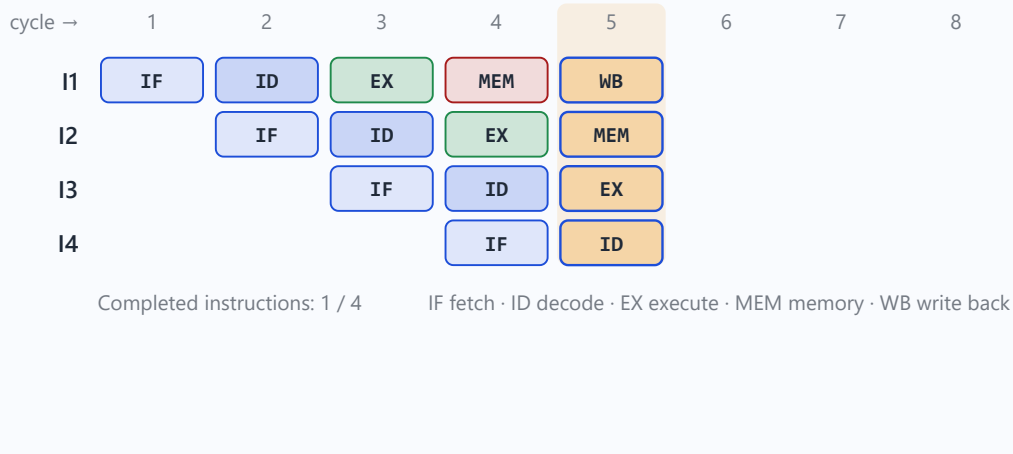
Definition 1.1.10

A **pipeline** is a technique that divides instruction execution into consecutive stages and executes different stages of consecutive instructions in the same cycle. Each stage is a separate hardware unit and hands its work to the next stage every cycle.

The classic RISC pipeline has five stages: IF (instruction fetch), ID (instruction decode and register read), EX (execute), MEM (memory access), and WB (write back the result to a register). An assembly line in a factory is a good analogy. Even if the complete assembly of a car takes five hours, the line turns out one car every hour, because five stations work on five different cars at the same time.

Figure 1.1.11 shows four independent instructions (I1–I4) passing through the pipeline. The cycles proceed as follows:

- **Cycle 1:** I1 is fetched (IF). The other stages are idle.
- **Cycle 2:** while I1 is decoded (ID), I2 is fetched. Two instructions are in different stages at the same time.
- **Cycle 3:** I1 executes, I2 is decoded, I3 is fetched: three instructions in three stages.
- **Cycle 4:** four instructions are in the pipeline at once.
- **Cycle 5:** I1 reaches the WB stage and completes. The pipeline is full; from now on one instruction finishes every cycle.
- **Cycles 6, 7, 8:** I2, I3, and I4 complete in turn.



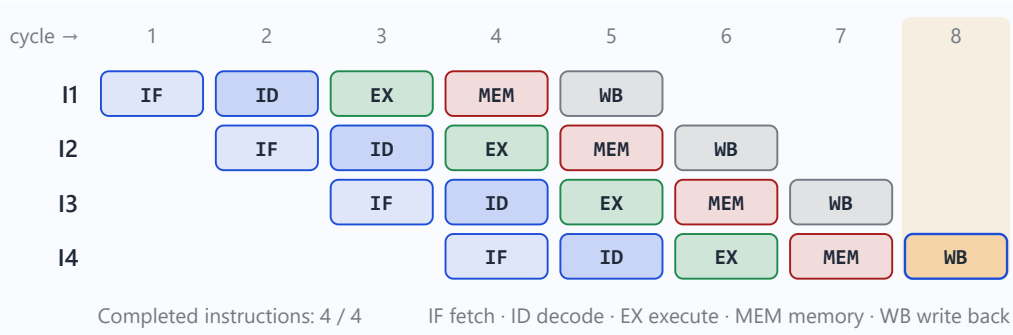


Figure 1.1.11: A five-stage pipeline. Top, cycle 5: the pipeline is full and I1 has completed. Bottom, cycle 8: all four instructions have finished.

Theorem 1.1.1 (Pipeline completion time)

In a k-stage pipeline, n instructions with no dependencies among them finish in

$$k + n - 1$$

cycles. Without the pipeline each instruction would take k cycles, and a total of $k \times n$ cycles would be needed.

The reasoning behind the formula is simple. The first instruction passes through the pipeline from start to finish in k cycles. After that, one more instruction finishes every cycle; the remaining $n - 1$ instructions need $n - 1$ more cycles. The total is $k + (n - 1)$. In the example $k = 5$, $n = 4$: $5 + 4 - 1 = 8$ cycles. Without the pipeline, $4 \times 5 = 20$ cycles would be needed.

Example 1.1.6

In a 5-stage pipeline, in how many cycles do 100 instructions finish? How many cycles are needed without the pipeline?

With the pipeline, $k + n - 1 = 5 + 100 - 1 = 104$ cycles. Without the pipeline, $5 \times 100 = 500$ cycles. The speedup is $500 / 104 \approx 4.8$ times. As the number of instructions grows, the speedup $k\,n / (k + n - 1)$ approaches k, the number of stages: for 1000 instructions, $5000 / 1004 \approx 4.98$.

This ideal picture does not always hold. An instruction may need a result that the instruction just before it has not yet written (a data dependency). Until the outcome of a conditional jump is known, it is not known which instruction to fetch (a control dependency). In these cases empty cycles enter the pipeline; this is called a pipeline **stall**. To reduce stalls, modern processors forward results directly between stages and predict the outcome of jumps in advance.

The 8086's pipeline is much simpler than this five-stage structure: there are only two stages, fetch (BIU) and execute (EU). The 17 cycles a taken `loop` takes are also a price of the pipeline: the jump invalidates the prefetched bytes and requires the queue to be refilled.

> **Common mistake**
>
> It is often thought that a pipeline speeds up a single instruction. The time of a single instruction (its latency) is not shortened by the pipeline; it even grows a little because of the registers between stages. What shrinks is the total time of many instructions; the pipeline increases throughput.

1.1.4.5 The same job, two approaches

It is instructive to see the difference between the two approaches on a single task: adding the value of a register to a number in memory, that is, `memory[addr] += x`. On a CISC processor (the 8086) this job is one instruction:

```
ADD [BX], AX     ; 1 instruction
                 ; read, add, write
```

On a RISC processor (ARM) it takes three instructions. `r1` holds the address and `r0` the value to add; in ARM assembly, comments start with the `@` sign:

```
LDR r2, [r1]      @ load
ADD r2, r2, r0    @ add
STR r2, [r1]      @ store
```

`LDR` (load register) loads the word in memory into `r2`. `ADD r2, r2, r0` is a three-operand addition: `r2 ← r2 + r0`. `STR` (store register) writes the result back to memory. The CISC code is short and takes little space in memory, but the instruction takes many cycles. The three RISC instructions are simple, and each flows smoothly through the pipeline. Because the summed value stays in a register, there is no need to go back to memory if other operations will be done on this number.

The two approaches have blended together today. Since the Pentium Pro, x86 processors translate complex instructions internally into simple RISC-like **micro-operations** (μops) and execute these micro-operations in a pipeline. From the outside the instruction set is CISC; inside, execution is like RISC.

1.1.4.6 A CISC instruction inside: code trace

To see how much work a single CISC instruction does inside, let us trace the following program. The program adds 7 twice to the variable `num` in memory.

```
org 100h
mov bx, offset num ; address
mov ax, 7          ; value
add [bx], ax       ; read, add, write
add [bx], ax       ; again
ret
num dw 0123h
```

The program starts at address `0100h` with `org 100h`. Adding up the instruction lengths gives the address of the variable `num`: `mov bx, offset num` is 3 bytes (0100h–0102h), `mov ax, 7` is 3 bytes (0103h–0105h), the two `add [bx], ax` are 2 bytes each (0106h–0109h), and `ret` is 1 byte (010Ah). `num` is therefore at `010Bh`. The `offset` operator makes the assembler substitute the variable's offset, that is, `010Bh`. `dw` (define word) reserves a word in memory and sets its initial value to `0123h`.

`add [bx], ax` is a single instruction, but it does three jobs inside:

1. **Read.** The BIU fetches the word (`0123h`) at address `010Bh`, which BX points to, with a memory-read bus cycle and puts it in a temporary register inside the processor. This temporary register is not visible to the programmer.
2. **Add.** The ALU computes `0123h + 0007h = 012Ah` and updates the flags.
3. **Write.** The result is written back to memory with a second bus cycle: `num = 012Ah`.

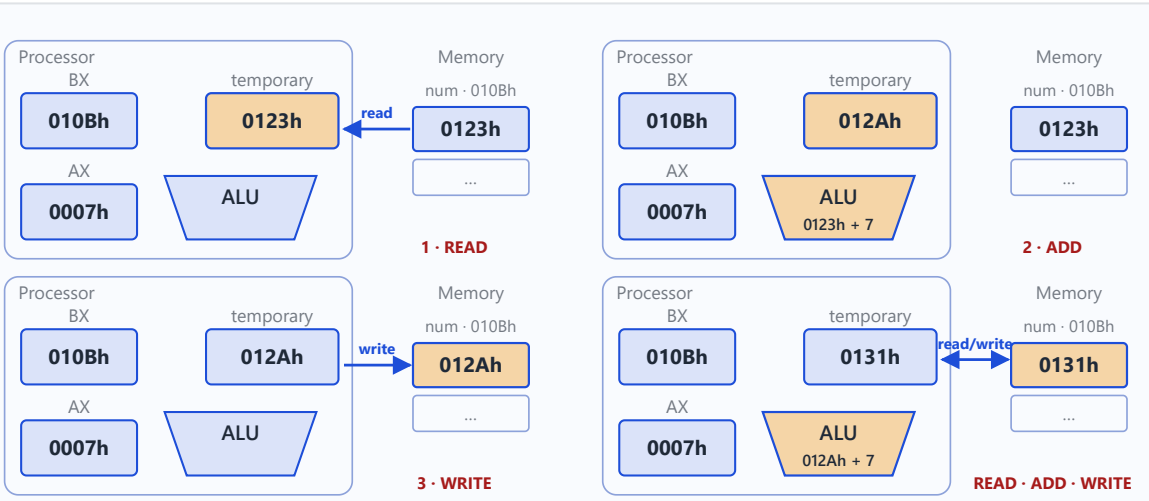


Figure 1.1.12: The three jobs the `add [bx], ax` instruction does inside. Top row: read (temporary = 0123h) and add (0123h + 7 = 012Ah); bottom row: write (num = 012Ah) and the second execution of the same instruction (num = 0131h).

Instruction	BX	AX	num (010Bh)	Bus cycles	Flags
initial	?	?	0123h	—	—
`mov bx, offset num`	010Bh	?	0123h	—	unchanged
`mov ax, 7`	010Bh	0007h	0123h	—	unchanged
`add [bx], ax`	010Bh	0007h	012Ah	read + write	ZF=0 CF=0 SF=0 OF=0 AF=0
`add [bx], ax`	010Bh	0007h	0131h	read + write	ZF=0 CF=0 SF=0 OF=0 AF=1

Table 1.1.10: Trace of the `add [bx], ax` program. Each row gives the state after that instruction.

In the second addition the low four bits *Ah + 7h = 11h* overflow, so AF = 1; in the first, *3h + 7h = Ah* fits in four bits, so AF = 0. In both results the top bit is 0 and there is no overflow out of 16 bits.

The cost of this instruction shows up in the cycle count. According to the 8086 datasheet, an `add` with a memory destination takes 16 cycles plus the effective-address calculation; for `[bx]` the effective-address calculation is 5 cycles, so the total is 21 cycles. Of these 21 cycles, 8 are the two bus cycles (read and write, four cycles each). A register-to-register `add`, by contrast, takes 3 cycles. A CISC instruction thus hides two memory accesses and one addition in a single instruction. On a RISC processor the same job is three instructions (`LDR`, `ADD`, `STR`) and the intermediate value becomes visible in a register.

1.1.4.7 RISC versus CISC

Criterion	RISC	CISC
Instruction set	small and simple	large and complex
Instruction length	fixed	variable
Cycles per instruction	low (≈ 1, with a pipeline)	high (variable)
Memory access	load/store only	most instructions can access memory
Number of registers	many	few
Program size	longer	shorter
Power consumption	low	relatively high
Flexibility	burden on the compiler	wide range of instructions, flexible use

Table 1.1.11: Comparison of RISC and CISC processors.

The summary of the table is this: RISC runs a small variety of instructions fast and delivers high performance at low power. CISC offers a large variety of instructions, but its individual instructions are slower. The comparison can also be read through the CPU-time formula. CISC reduces N but increases CPI; RISC reduces CPI but increases N. The product decides which one wins. In today's high-performance processors the internal structures of the two families have come very close; the difference remains mostly in code density, decoder complexity, and power consumption.

1.1.4.8 EPIC

Superscalar processors start more than one instruction in the same cycle. It is the hardware that finds out, at run time, which instructions are independent of each other and can execute together. This hardware is a large, power-hungry circuit. The EPIC approach takes this decision away from the hardware and gives it to the compiler.

Definition 1.1.11

In an **EPIC** architecture the compiler, not the hardware, decides which instructions can run in parallel and writes this explicitly into the code. Instructions are encoded in 128-bit **bundles**: each bundle holds three 41-bit instructions and a 5-bit template.

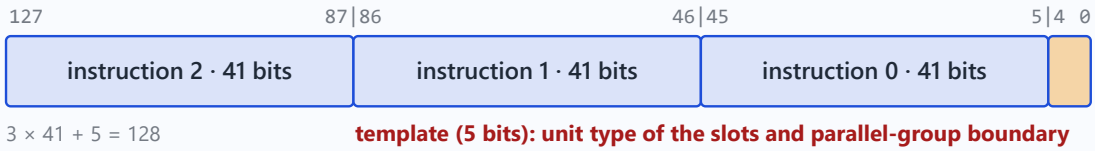


Figure 1.1.13: A 128-bit EPIC bundle. Bits 127–87 are instruction 2, bits 86–46 instruction 1, bits 45–5 instruction 0, and bits 4–0 the template. *3 × 41 + 5 = 128*.

The template field says which kind of execution unit (integer, memory, floating-point, branch) each of the three slots in the bundle goes to, and where a group of instructions that can execute in parallel ends. The processor reads this information and dispatches the instructions straight to the units; no large circuit is needed to search for dependencies. In this way many instructions can execute at once without needing a high clock frequency. The bit boundaries are consistent too: since each instruction takes 41 bits, instruction 0 spans bits 5 to 45, instruction 1 bits 46 to 86, and instruction 2 bits 87 to 127.

The only well-known commercial example of EPIC is the IA-64, the joint architecture of Intel and HP, and the Itanium processors that implement it (2001–2021). The approach did not achieve the expected success. Situations the compiler cannot know at compile time, for example whether a memory access will hit in the cache, limited the parallelism; producing good code put too much burden on the compiler. The Itanium also ran existing x86 software slowly. In the market, AMD's x86-64 architecture, which extended x86 to 64 bits, won.

1.1.5 Processors by data-processing style and application

The instruction set architecture describes how instructions are designed. A second way to tell processors apart is to look at how many data items an instruction processes: a scalar processor handles one, while vector and array processors handle many. A third way is what the processor is

optimized for: signal processing, symbolic computation, or graphics. This section introduces these types in turn and uses two small 8086 programs to show the cost of scalar processing.

1.1.5.1 Scalar and vector processing

Adding two arrays element by element, that is, *C[i] = A[i] + B[i]*, is a task that comes up constantly in scientific computing and in audio and image processing. A **scalar** processor does this job with a separate add instruction for each element. A **vector** processor adds all the elements of the array with a single instruction.

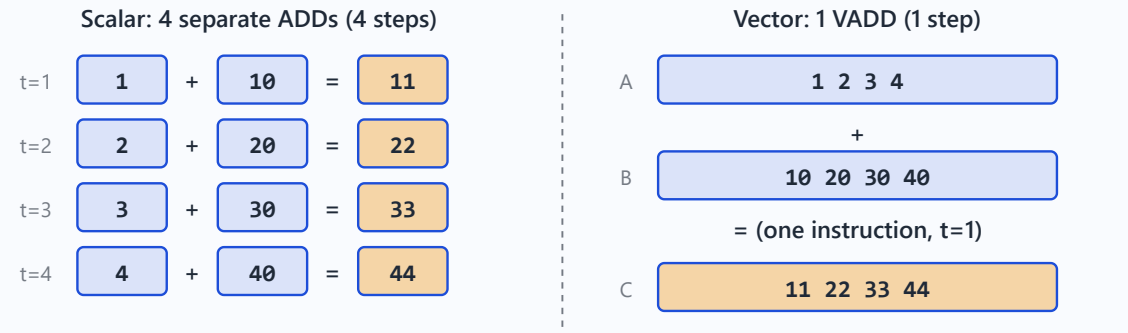


Figure 1.1.14: Adding arrays of four elements. A scalar processor computes it in four steps with four separate `ADD` instructions; a vector processor computes it in a single step with one `VADD` instruction: *C = (11, 22, 33, 44)*.

The gain of vector processing does not come only from the addition itself. A scalar processor fetches and decodes an instruction again for each element, decrements and tests the loop counter, and increments the index. A vector processor pays this overhead once. With four elements the gain looks small, but with vectors of a thousand elements the difference becomes very large.

1.1.5.2 A scalar loop: code trace

The 8086 is a scalar processor. The following program computes the sum $c = a + b$ for byte arrays of four elements.

```
org 100h
mov si, 0
mov cx, 4
again: mov al, a[si]  ; read one item
add al, b[si]         ; one add
mov c[si], al         ; write one item
inc si
loop again
ret
a db 3, 5, 7, 9
b db 10, 20, 30, 40
c db 4 dup(0)
```

SI is the common index of the arrays; CX is the loop counter. The notation `a[si]` denotes the byte at the address found by adding SI to the start address of array `a`. In the loop body three instructions do the real work: `mov al, a[si]` reads one element, `add al, b[si]` does one addition, and `mov c[si], al` writes the result. The remaining two, `inc si` and `loop again`, are loop overhead. `4 dup(0)` reserves a space of four bytes filled with zeros.

Pass	Operation	AL	array c	SI	CX	Instructions executed (total)
initial	—	?	0, 0, 0, 0	0	4	2
1	3 + 10	13	13, 0, 0, 0	1	3	7
2	5 + 20	25	13, 25, 0, 0	2	2	12
3	7 + 30	37	13, 25, 37, 0	3	1	17
4	9 + 40	49	13, 25, 37, 49	4	0	22

Table 1.1.12: Trace of the scalar loop. Each row gives the state at the end of one loop pass.

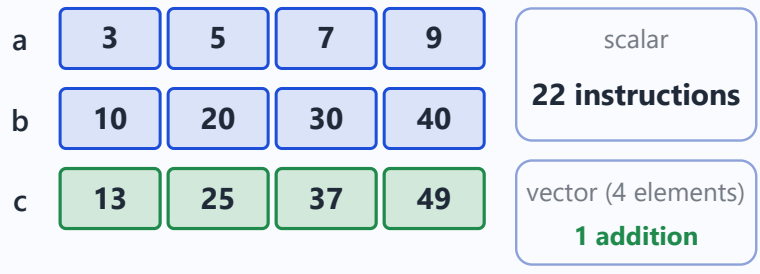


Figure 1.1.15: When the loop ends, *c = (13, 25, 37, 49)*. The scalar processor has executed 22 instructions; a vector processor does the same four additions with a single vector-add instruction.

The count goes like this: two instructions before the loop (`mov si, 0` and `mov cx, 4`), and five instructions per pass. Over four passes this comes to *2 + 4 × 5 = 22* instructions. Only four of these 22 instructions are additions. Eight are reads and writes, eight are loop overhead, and two are setup. For

every element three instructions are fetched and decoded, and the loop overhead is paid again for every element. This is the overhead a vector processor eliminates.

1.1.5.3 The vector processor

Definition 1.1.12

A **vector processor** is a processor that performs mathematical operations with a single instruction on arrays of data called vectors. It processes several data elements at the same time, or one after another in a deep pipeline. It has long vector registers and vector load/store units.

Figure 1.1.16: Vector addition. The eight elements in vector registers V1 and V2 are added with a single `VADD V3, V1, V2` instruction and written to register V3.

For this kind of work a vector processor is much faster than a scalar processor, which works with a single data element. Applications that work with large arrays, such as weather forecasting, physics simulations, and artificial-intelligence models, therefore benefit from vector processing. The famous historical example is the Cray-1 supercomputer (1976). Today vector processing has entered general-purpose processors as extensions: SSE and AVX on x86, NEON and SVE on ARM, and the V extension on RISC-V. For example, a 256-bit AVX register holds eight 32-bit numbers and adds all eight with one instruction.

1.1.5.4 The array processor and Flynn's taxonomy

Definition 1.1.13

An **array processor** is a processor in which a single control unit broadcasts the same instruction to many **processing elements** (PEs). Each PE works on data in its own local memory. This arrangement is also called **SIMD** (single instruction, multiple data).

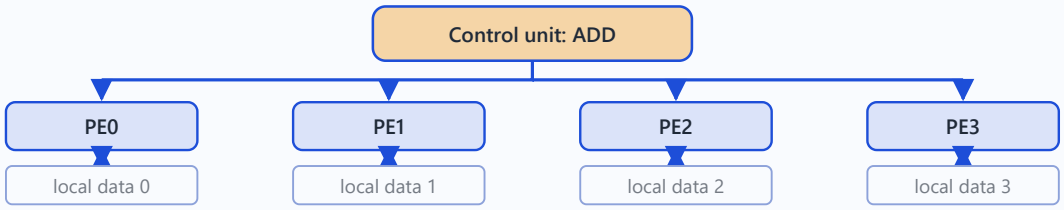


Figure 1.1.17: An array processor. The control unit broadcasts the same `ADD` instruction to four processing elements (PE0–PE3); each processing element works on its own local data.

The difference between a vector processor and an array processor lies mostly in the organization of the hardware. A vector processor can process elements one after another in a single deep pipeline. In an array processor there is a separate processing element for each element, and the elements really are processed at the same time. Array processors are optimized for parallel computation. They are used in supercomputers, servers, and GPUs; they are rare in general-purpose computing. A historical example is the ILLIAC IV.

The classification proposed by Michael Flynn in 1966 divides processors into four groups by the number of instruction and data streams:

Class	Meaning	Example
SISD	single instruction, single data	classic 8086
SIMD	single instruction, multiple data	array and vector processors, GPUs
MISD	multiple instruction, single data	rare; fault-tolerant systems
MIMD	multiple instruction, multiple data	multicore processors, clusters

Table 1.1.13: Flynn's taxonomy.

The 8086 processes a single data element per instruction and executes a single instruction stream; it is in the SISD class. In a multicore processor each core executes its own program on its own data; this is MIMD. A desktop processor today falls into several classes at once: each core does SIMD work with AVX instructions, and the cores together work in a MIMD arrangement.

1.1.5.5 Scalar and superscalar processors

Definition 1.1.14

A **scalar processor** processes scalar, that is, non-vector, data, one data element per instruction.

Simple scalar processors work only with integers. Powerful ones also handle floating-point numbers; in these, the integer unit (ALU) and the floating-point unit (FPU) are on the same chip. A scalar processor can have a CISC or a RISC architecture; the two classifications are independent of each

other. The most common type in general-purpose computing is the scalar processor.

A scalar processor can also execute several instructions at once. A **superscalar** processor is a scalar processor that starts more than one instruction per clock cycle by means of several pipelines or execution units. Each instruction still works on a single data element; the parallelism lies in the instruction stream, not in the data. The first superscalar processor in the x86 line is the Pentium: it has two pipelines named U and V, and two suitable instructions enter one pipeline each in the same cycle.

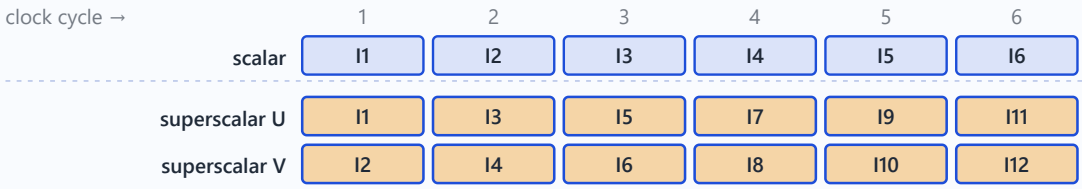


Figure 1.1.18: In a scalar processor one instruction starts every clock cycle (I1–I6 in 6 cycles). The two-pipeline superscalar Pentium issues one instruction to each of the U and V pipelines every cycle (I1–I12 in 6 cycles).

In the ideal case in the figure, the superscalar processor starts twice as many instructions in the same time; CPI drops to 0.5. In practice this ratio falls. Two instructions can run in the same cycle only if they are independent of each other: if the second uses the result of the first, it has to wait. On the Pentium the instructions the V pipeline could execute were also limited.

1.1.5.6 The digital signal processor (DSP)

Definition 1.1.15

A **digital signal processor** (DSP) is a processor optimized for operations on digital signals. A digital signal is a sequence of **samples** that represent sensory information such as sound and images. A DSP has an instruction set and hardware specialized for signal processing: multiply-accumulate (MAC) in a single cycle, circular-buffer addressing, and zero-overhead loops.

DSPs are used in audio processing and filtering, in image processing, and in telecommunications infrastructure (modems, base stations). Well-known examples are the Texas Instruments TMS320 and Analog Devices SHARC families.

The most basic operation in signal processing is the digital filter. A **FIR filter** (finite impulse response) computes each output sample as a weighted sum of the last N input samples:

$$y[n] = \Sigma_{k=0}^{N-1} h[k] \cdot x[n-k]$$

Here *h[k]* are the filter coefficients and *x* the input samples. Each output sample needs *N* multiplications and *N* additions. For a 48 kHz audio signal, a filter with 100 coefficients needs 4.8 million multiplications and 4.8 million additions per second. A DSP performs each term in a single cycle with one **MAC** (multiply-accumulate) instruction:

$$A \leftarrow A + h[k] \times x[n-k]$$

Here *A* is the accumulator. Circular-buffer addressing automatically wraps the address back to the start when the end of the buffer holding the last *N* samples is reached, so no bounds checking is done in software. A zero-overhead loop keeps the loop counter in hardware and spends no cycles on the jump at the end of the loop.

1.1.5.7 Multiply-accumulate: code trace

How the same computation is done on the 8086 shows clearly why the DSP is a separate type. The following program computes a filter sum with three coefficients: $y = h[0]{\cdot}x[0] + h[1]{\cdot}x[1] + h[2]{\cdot}x[2]$.

```
org 100h
mov si, 0
mov bx, 0        ; sum
mov cx, 3        ; term count
again: mov al, h[si]
mul x[si]        ; AX = h × x
add bx, ax       ; accumulate
inc si
loop again
ret
h db 2, 3, 1
x db 10, 20, 30
```

BX serves as the accumulator and starts at zero; CX is the number of terms. `mov al, h[si]` brings the coefficient into AL. `mul x[si]` is a byte multiplication: it multiplies AL by the byte in memory and writes the 16-bit result to AX. One operand of `mul` is always implicitly AL (for a byte multiplication); it is not written in the instruction. `add bx, ax` adds the product to the sum.

Instruction	SI	AL / AX	BX (sum)	CX
setup	0	?	0	3
`mov al, h[0]`, `mul x[0]`	0	AX = 2 × 10 = 20	0	3
`add bx, ax`, `inc si`, `loop`	1	AX = 20	20	2
`mov al, h[1]`, `mul x[1]`	1	AX = 3 × 20 = 60	20	2
`add bx, ax`, `inc si`, `loop`	2	AX = 60	80	1
`mov al, h[2]`, `mul x[2]`	2	AX = 1 × 30 = 30	80	1

Instruction	SI	AL / AX	BX (sum)	CX
`add bx, ax` , `inc si` , `loop`	3	AX = 30	110	0

Table 1.1.14: Trace of the multiply-accumulate loop. Each row gives the state after that instruction.

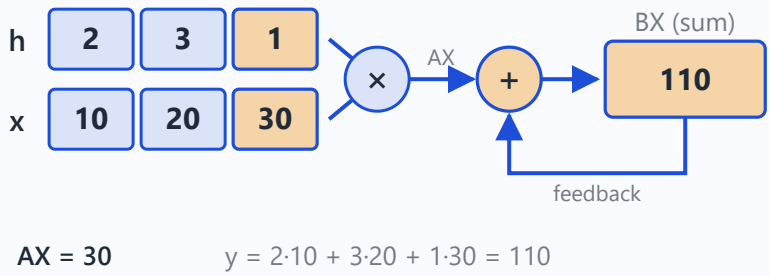


Figure 1.1.19: When the loop ends, the sum is BX = 110: *2·10 + 3·20 + 1·30 = 110*. A separate `mul` and `add` were needed for every term.

Since none of the three products exceeds 255, AH = 0 after each `mul` ; in that case the 8086 sets the CF and OF flags to 0. Had the result spilled into a high byte, both would have been 1.

The difference in time is large. On the 8086 a `mul` of a byte from memory alone takes 76–83 cycles plus the effective-address calculation; that is, more than 70 cycles. For one term, `mov` , `mul` , `add` , `inc` , and `loop` together take more than a hundred cycles. A DSP does the same term with a single MAC instruction in one cycle, and thanks to its zero-overhead loop it does not pay the loop overhead either.

1.1.5.8 The symbolic processor

Definition 1.1.16

A **symbolic processor** is a processor designed to compute on symbols, lists, and rules instead of numbers. It is also called a PROLOG or LISP processor.

Symbolic processors concentrate on semantic computation and on the rules of symbolic logic. They process mathematical expressions as symbols; they can simplify complex expressions and take derivatives. Application areas are expert systems, machine intelligence, and rule-based inference. The following Prolog fragment is an example of this kind of computation:

```
father(ali, veli).
grandpa(X, Z) :- father(X, Y),
                 father(Y, Z).
```

The first line is a fact: Ali is Veli's father. The rule that follows requires, for X to be Z's grandfather, that X be the father of some Y and that Y be the father of Z. The program does not compute numbers; it draws conclusions by looking for matches among facts and rules. The main parts of this work in hardware are list processing, matching, and backtracking.

The LISP machines of the 1980s (for example those of Symbolics) and Japan's Fifth Generation Computer project were of this kind. As general-purpose processors got faster, the advantage of special hardware disappeared. Today this work is done in software on general-purpose processors.

1.1.5.9 The graphics processing unit (GPU)

Definition 1.1.17

A **graphics processing unit** (GPU) is a processor made up of many simple cores, designed to accelerate two- and three-dimensional graphics.

A screen frame consists of millions of pixels, and the color of each pixel is computed with the same formula, only on different data. The GPU exploits this structure: thousands of simple cores run the same program on different pixels. This arrangement is a form of SIMD; GPU vendors call it SIMT (single instruction, multiple threads). The GPU produces real-time, high-quality graphics; it is effective at rendering tasks such as shading, texturing, and rasterization. It is used in graphics cards and accelerator cards.

Figure 1.1.20: A CPU consists of a few large cores with a large cache and control circuitry; a GPU consists of hundreds or thousands of small cores.

CPUs and GPUs are designed for different goals. A CPU has a few powerful cores; the aim is to reduce the latency of a single thread and to run complex, branch-heavy flows quickly. Most of the chip area goes to cache and control circuitry. A GPU has thousands of simple cores; the aim is to increase total throughput, not the time of a single job. Any job that works on parallel data benefits from this structure. **GPGPU** (general-purpose computing on GPU) is the use of the GPU, through programming environments such as CUDA or OpenCL, for non-graphics work such as scientific computing and deep learning.

1.1.5.10 Summary of the types

Type	Core idea	Strength	Example
CISC	rich, complex instructions	short code, compatibility	8086, Pentium, 68000
RISC	simple instructions + pipeline	speed, low power	ARM, MIPS, RISC-V
EPIC	the compiler determines parallelism	instruction-level parallelism	IA-64 Itanium
Scalar	one data item per instruction	general purpose	most CPU cores
Vector	a data array per instruction	scientific computing	Cray-1, AVX
Array (SIMD)	many PEs, same instruction	massive parallelism	ILLIAC IV, GPU
DSP	MAC, special addressing	real-time signals	TMS320, SHARC
Symbolic	symbol/rule processing	artificial intelligence (classic)	LISP machines
GPU	thousands of simple cores	graphics, deep learning	NVIDIA, AMD Radeon

Table 1.1.15: Summary of the processor types.

The classes in the table are not mutually exclusive. An ARM core is both RISC and scalar, and it also does vector processing with the NEON extension. A Core i7 is CISC on the outside and RISC-like on the inside, superscalar, vector-capable with AVX, and, being multicore, MIMD. The classes should be read as labels telling which design ideas a processor uses.

1.1.6 Evolution of the Intel x86 family

The 8086 was not born in a vacuum. Before it lies a short but rapid history of 4-bit and 8-bit processors, and after it a family that is still going today. This section follows that history starting from the transistor, lists the innovations each processor generation brought, and discusses the strengths and weaknesses of the x86 architecture.

1.1.6.1 From the transistor to the microprocessor

The history of the microprocessor can be summarized by three inventions.

- **1947, the transistor.** At Bell Laboratories, John Bardeen, Walter Brattain, and William Shockley invented the transistor; the invention was announced in 1948. The transistor did the switching and amplifying that the vacuum tube did, with a semiconductor that was much smaller, more durable, and used less power. The microelectronics revolution began this way.
- **1958, the integrated circuit.** At Texas Instruments, Jack Kilby demonstrated the first **integrated circuit** (IC). Many transistors and their connections could be fabricated together on a single piece of semiconductor. Complex circuits thus became small and cheap.
- **1971, the first microprocessor.** The Intel 4004 fit an entire CPU on a single chip. The microprocessor could now take on the job of a computer's central processing unit by itself.

The result is visible everywhere today. Computers shrank from the size of a room to a desktop, and from there to a pocket. At every step they became smaller, more powerful, faster, and cheaper.

1.1.6.2 Timeline

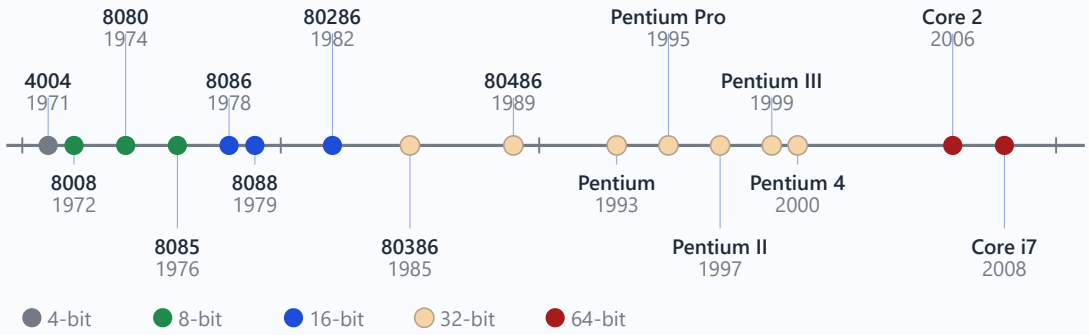


Figure 1.1.21: Timeline of Intel microprocessors: from the 4004 in 1971 to the Core i7 in 2008. The colors separate the 4-, 8-, 16-, 32-, and 64-bit generations.

The name **x86** comes from the common “86” ending of the names 80186, 80286, 80386, and 80486 that followed the 8086. All members of the family can run 8086 code. This property is called **backward compatibility**, and it is the basis of x86’s success of more than forty years: every new processor runs the software users already have, unchanged. There is also a consequence: when a present-day Core i7 is powered on, it starts in **real mode**, behaving like an 8086 and working with 20-bit addresses. The wider features are reached only when software switches the processor to another mode.

1.1.6.3 4-bit and 8-bit processors

Processor	Year	Clock	Transistors	Instruction rate / note
4004 (4-bit)	1971	740 kHz	2,300	≈ 60,000 instructions/s; Ted Hoff, Federico Faggin, Stanley Mazor
4040 (4-bit)	1974	740 kHz	≈ 3,000	an enhanced version of the 4004 (interrupts, more instructions)
8008 (8-bit)	1972	500 kHz	3,500	≈ 50,000 instructions/s; the first 8-bit processor
8080 (8-bit)	1974	2 MHz	6,000	≈ 10 times the speed of the 8008; 16-bit address (64 KB); Altair 8800
8085 (8-bit)	1976	3 MHz	6,500	single +5 V supply; 16-bit address bus; ≈ 769,000 instructions/s

Table 1.1.16: 4-bit and 8-bit Intel microprocessors.

The 4004 was designed for a Japanese calculator company. Its 4-bit word was enough to handle decimal digits as BCD: each digit fits in 4 bits. With the 8008 the move to 8 bits was made; a byte could now hold a character. The 8080, with both its speed and its 64 KB address space, was a real general-purpose processor and was used in the Altair 8800, one of the first personal computers. The

8085 reduced the 8080's design, which needed several supply voltages, to a single +5 V supply. The shortest instruction of the 8085 takes 1.3 µs at 3 MHz; this comes to about *1 / 1.3 µs ≈ 769,000* instructions per second.

1.1.6.4 16-bit processors

Processor	Year	Clock	Transistors	Feature
8086	1978	5 · 8 · 10 MHz	29,000	16-bit data bus, 20-bit address bus (1 MB); hardware multiply/divide; ≈ 0.33–0.75 MIPS
8088	1979	5 · 8 MHz	29,000	a cheap version of the 8086 with an 8-bit external data bus; IBM PC (1981, 4.77 MHz)
80186 / 80188	1982	from 6 MHz	≈ 55,000	on-chip interrupt controller, clock generator, local bus controller, counters, DMA; the 80188 is a cheap (8-bit external bus) version
80286	1982	6–12.5 MHz	134,000	16-bit data, 24-bit address bus (16 MB); protected mode; ≈ 0.9–2.7 MIPS

Table 1.1.17: 16-bit Intel microprocessors.

8086 instructions take more than 10 cycles on average. At 5 MHz this gives about 0.33 MIPS, which means an average of 15 cycles per instruction (*5 / 15 ≈ 0.33*). The 8088 has the same internal structure and instruction set as the 8086, and its address bus is also 20 bits. The difference is in the external data bus: the 8088 connects to the outside with an 8-bit bus and transfers a word in two bus cycles. Its prefetch queue is also 4 bytes, not 6. IBM chose the 8088 for the PC so it could use cheap 8-bit peripheral chips, and with this decision x86 became the standard of personal computers.

The 80186 took the units that were built from separate chips around the 8086 (interrupt controller, clock generator, timers, DMA) into the chip itself. In this respect it is a design that approaches a microcontroller, and it was used mostly in embedded systems. The 80286 brought two innovations: a 24-bit address bus and **protected mode**.

1.1.6.5 32-bit processors

Processor	Year	Clock	Transistors	Feature
80386	1985	16–33 MHz	275,000	32-bit data and address bus (4 GB), paging; versions: 386DX, 386SX (1988; 16-bit external data, 24-bit address), 386SL (laptop)
80486	1989	16–100 MHz	1.2 million	on-chip 8 KB cache, integrated FPU, 5-stage pipeline; versions: 486DX, 486SX (no FPU), DX2, DX4
Pentium	1993	60 · 66 MHz (first)	3.1 million	superscalar (2 pipelines); separate 8 KB instruction + 8 KB data caches; 64-bit external data bus

Table 1.1.18: 32-bit Intel microprocessors.

The Pentium's external data bus is 64 bits wide, but the Pentium is a 32-bit processor: its registers and ALU are 32 bits. The wide external bus is there to fill the cache from memory faster. This example shows once more that the definition of an "*n*-bit processor" rests on the width of the registers and the ALU.

1.1.6.6 64-bit processors

Processor	Year	Clock	Transistors	Feature
Core 2	2006	1.2–3 GHz	291 million (Duo)	64 KB L1 per core (32 KB instruction + 32 KB data), 2–4 MB shared L2; versions: Duo, Quad, Extreme
Core i7	2008	2.66–3.33 GHz	731 million	4 cores, Hyper-Threading, integrated memory controller, 8 MB L3
Core i5	2009	2.4–3.6 GHz	—	mid-range; Turbo Boost
Core i3	2010	2.93–3.33 GHz	—	entry level; 2 cores, integrated graphics

Table 1.1.19: 64-bit Intel microprocessors. The clock ranges are for first-generation desktop and mobile models.

64-bit x86, that is, x86-64, first appeared with AMD's AMD64 architecture (Opteron and Athlon 64, 2003). Intel adopted this architecture in 2004 with the Pentium 4. This is one of the rare moments in x86 history when the direction of the family was set by Intel's rival rather than by Intel; Intel's own 64-bit path, the IA-64 (Itanium), had not caught on in the market.

1.1.6.7 Transistor count and Moore's law

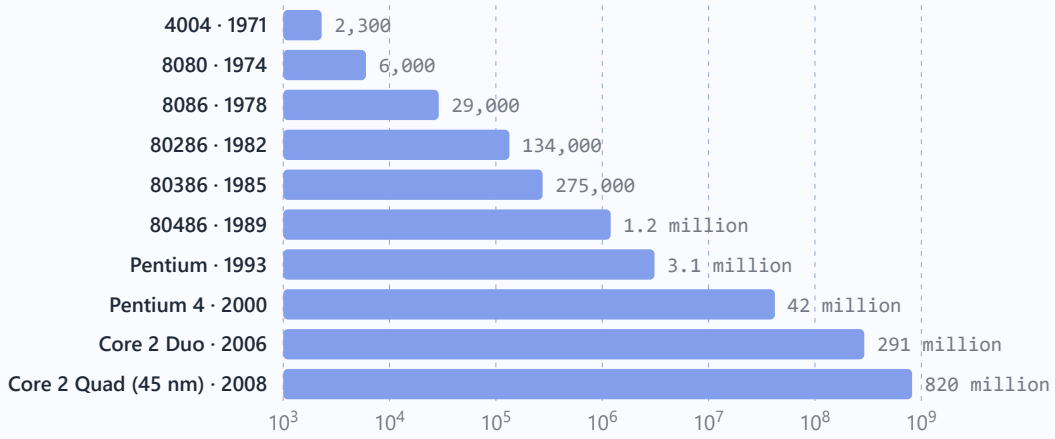


Figure 1.1.22: Transistor counts on a logarithmic scale: 2,300 in the 4004, 29,000 in the 8086, 3.1 million in the Pentium, and 820 million in the Core 2 Quad (45 nm). A straight line on a logarithmic scale means constant-ratio (exponential) growth.

Definition 1.1.18

Moore's law is an observation by Gordon Moore, one of the founders of Intel. In the form Moore put forward in 1965 and corrected in 1975, the number of transistors per chip doubles roughly every two years.

The observation is not a law of nature but a target the semiconductor industry held to for decades. It is easy to check. From the 4004 (1971, 2,300 transistors) to the Core 2 Quad (2008, 820 million), 37 years passed. The ratio is $820 \times 10^6 / 2300 \approx 356{,}500$ and lies between $2^{18} = 262{,}144$ and 2^{19}; that is, about 18 doublings. Doubling every two years over 37 years comes to $37 / 2 \approx 18.5$ doublings. Over this range the observation held with surprising accuracy. The growth in transistor count is the raw material of all the innovations described in this chapter, such as caches, pipelines, and multiple cores.

1.1.6.8 The 8080 and the 8086

The structure, organization, and technology of microprocessors changed radically in a short time. This change runs from the first general-purpose microprocessor, the 8080, to the 29,000-transistor 8086, and from there to the four-core Core 2 with 820 million transistors. The processor of this course is the 8086; comparing it with its predecessor, the 8080, shows which step was taken and why.

The 8080 (1974) is cited as the first general-purpose microprocessor. Its data bus is 8 bits and its address bus 16 bits; it addresses 64 KB of memory. It was used in the first personal computers, for example the Altair 8800.

The 8086 (1978) is a 16-bit processor: its data bus is 16 bits, and it has more 16-bit registers than the 8080. Its address bus is 20 bits and it addresses 1 MB of memory. It works only in real mode; there is no memory protection. Its most important innovation is that it **prefetches** instructions before executing them and holds them in a 6-byte instruction queue. While the BIU fills the queue, the EU executes the previous instruction; fetching and execution overlap. The 8086 has no cache; the speed gain comes from this queue. An on-chip cache came to x86 only with the 80486.

Example 1.1.7

Roughly what is the gain from the queue? On the 8086, a 2-byte instruction would, without the queue, first be fetched with one bus cycle (4 cycles) and then executed. For a 3-cycle instruction such as `add ax, cx` this would come to $4 + 3 = 7$ cycles. When the queue is full, the fetch has already been done while earlier instructions executed, and the instruction finishes in 3 cycles. This is why the cycle counts in the datasheet are given under the assumption "queue full." On jumps the queue is flushed, so the gain is lost; this is the reason a taken `loop` takes 17 cycles.

1.1.6.9 The 80286, 80386, and the 80486

The 80286 (1982) addresses 16 MB of memory with its 24-bit address bus. Besides real mode, it offers a 16-bit protected mode. Its data bus and registers are still 16 bits.

The 80386 (1985) is Intel's first 32-bit microprocessor. It supports multitasking in hardware. It brings 32-bit protected mode, **paging**, and virtual 8086 mode; it addresses 4 GB of memory. Virtual 8086 mode lets an operating system running in protected mode run old 8086 programs, each in its own isolated environment.

The 80486 (1989) brings an on-chip cache (8 KB) and an instruction pipeline. It adds page write protection (the WP bit of the CR0 register). It contains an integrated math coprocessor (FPU) for complex arithmetic; this unit had previously been a separate chip, the 80387.

Definition 1.1.19

Protected mode is the operating mode that isolates each program's memory from the others. If a program tries to access memory that was not allocated to it, the processor blocks the access and notifies the operating system. A program's bug therefore cannot crash the operating system or other programs. This is the foundation of modern operating systems.

The real mode of the 8086 has no such protection. A program can write anywhere in the 1 MB, for example to the operating system's own code or to the interrupt vector table. This is why it was common in the DOS era for a faulty program to lock up the whole computer.

1.1.6.10 The Pentium family

The Pentium (1993) is superscalar; two instructions can run at the same time in two pipelines (U and V). It uses **branch prediction** in x86 for the first time: it predicts the outcome of a conditional jump from its earlier behavior and fills the pipeline along the predicted path. The page size extension (PSE) improves paging by using 4 MB pages alongside 4 KB pages. Instruction and data caches are separate.

The Pentium Pro (1995) changes execution fundamentally with three techniques: advanced branch prediction, **out-of-order execution** through data-flow analysis, and **speculative execution**. In out-of-order execution, an instruction whose operands are ready executes without waiting for an earlier instruction that is waiting on a result; the results are still committed in program order. In speculative execution, instructions on the predicted branch path are executed before the outcome is confirmed; if the prediction turns out wrong, the results are discarded. Intel called the combined use of these three techniques "dynamic execution." The Pentium Pro puts the level 2 (L2) cache inside the same package. With PAE (physical address extension), 32-bit virtual addresses are translated to 36-bit physical addresses and up to 64 GB of memory can be reached.

The Pentium II (1997) combined MMX technology, integer SIMD instructions for multimedia, with the Pentium Pro core. It handles video, audio, and graphics data efficiently. MMX had first appeared in early 1997 with the "Pentium MMX"; the Pentium II took it over.

The Pentium III (1999) brought SSE (Streaming SIMD Extensions): 70 new instructions and floating-point SIMD. It supports three-dimensional graphics software. The highest clock speed is 1.4 GHz.

The Pentium 4 (2000) added floating-point improvements for multimedia (first SSE2, then SSE3). With a very deep pipeline called NetBurst it reached up to 3.8 GHz and brought Hyper-Threading. Hyper-Threading lets a single core execute two threads at the same time; the operating system sees one core as two logical processors. The very deep pipeline allowed high frequency, but the number of stages flushed on every mispredicted branch also grew, and power consumption rose. The clock-speed race reached its limit with this generation.

1.1.6.11 Core and Core 2

The Core (Core Duo, 2006) is a low-power design of mobile origin. Two cores sit on the same die and share the L2 cache; it is the first Intel processor in which two cores on the same die share a cache. Virtualization technology (VT-x) was added with this generation.

The Core 2 (2006) uses the 64-bit architecture (Intel 64): a 64-bit register set and addressing modes. The Core 2 Quad offers four cores in a single package; inside the package there are two dual-core dies.

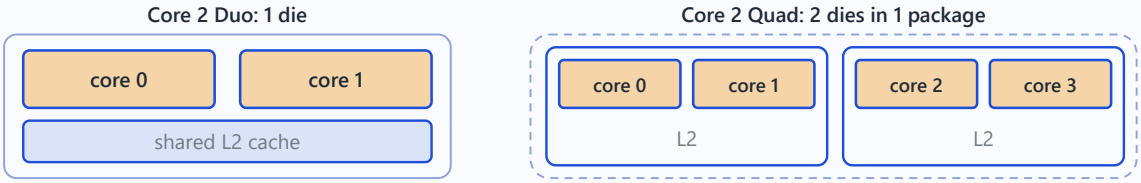


Figure 1.1.23: The Core 2 Duo has two cores and a shared L2 cache on a single die. The Core 2 Quad carries two dual-core dies in one package; each die has its own L2 cache.

Intel's first dual-core desktop processor was the Pentium D, released in 2005. In this processor the two cores had separate L2 caches and communicated with each other over the external bus; in the 2006 version the two cores were placed in the same package as two separate dies. Sharing a cache lets cores use common data without going to memory.

A **die** and a **package** are different things. A die is a single piece of chip cut from the silicon wafer. The package is the outer casing that carries the die (or several dies) and has the pins and the heat spreader. The "processor" in the user's hand is the package.

1.1.6.12 Strengths and weaknesses of the x86 architecture

There are four reasons for x86's long life.

- **Compatibility**: old software runs unchanged on new x86 processors. Code written for the 8086 in 1978 can still run today.
- **Performance**: every new processor generation has been faster and more efficient than the one before.
- **Versatility**: a wide range of uses, from personal computers to servers, and from embedded systems to mobile devices.
- **Broad industry support**: a large community of hardware and software vendors supports x86; this support has been the driving force of innovation and progress.

The same history also explains the architecture's weak points.

- **Complex instruction set**: thousands of instructions and variable-length encoding complicate the decoder hardware; optimizing code for performance becomes harder. Backward compatibility requires keeping even old instructions that are no longer used.
- **Power consumption**: the evolution of x86 brought a significant increase in power consumption. In devices where battery life is critical this is a major problem.
- **Heat dissipation**: the more powerful processors get, the hotter they run; cooling costs and noise rise.
- **Cost and licensing**: only companies with a license agreement (for example AMD) can make x86 processors; competition is limited.

For these reasons ARM (RISC) dominates phones and tablets, while x86 is strong on the desktop and in servers. In recent years ARM-based processors have entered laptops and servers too.

1.1.7 Modern processor features

The difference between the 8086 and today's processors is not only in clock speed. This section first shows with a numerical example why clock speed alone is not a measure of performance. It then lists the features that determine the performance of today's processors and explains the memory hierarchy.

1.1.7.1 Clock speed alone is not performance

The CPU-time formula is the basis of this section:

$$CPU\ time = N \times CPI \times T = N \times CPI / f$$

Here N is the number of instructions executed, CPI is the average number of clock cycles per instruction, and f is the clock frequency ($T = 1/f$). The formula makes it clear that clock frequency is only one of three factors.

Example 1.1.8

The same program ($N = 10^9$ instructions) runs on two processors. Processor A: $f = 2$ GHz, CPI = 2. Processor B: $f = 1.5$ GHz, CPI = 1. Which is faster?

$$A: 10^9 \times 2 / (2 \times 10^9) = 1 \text{ s}$$
$$B: 10^9 \times 1 / (1.5 \times 10^9) \approx 0.67 \text{ s}$$

B, with the lower clock, is $1 / 0.67 = 1.5$ times, that is, 50% faster. A executes $2 \times 10^9 / 2 = 10^9$ instructions per second, and B executes $1.5 \times 10^9 / 1 = 1.5 \times 10^9$.

Clock speed is "cycles per second," not "instructions per second." Instructions per second is f / CPI, and architecture (pipelining, caches, superscalar design) lowers CPI. The move from the Pentium 4 to the Core generation is a historical example of this. Core processors had lower clock speeds, but their CPI was lower too. They ran faster than the Pentium 4 and also used less power.

Common mistake

It is a mistake to compare two processors by clock speed alone. Clock speed is enough only to compare two processors of the same architecture with the same CPI. Across different architectures CPI, and even N for the same program, can differ; the right comparison is the time of the same program on the two processors.

1.1.7.2 Features that determine performance

Clock speed: the number of clock cycles per second (GHz). The instruction rate also depends on CPI.

Instruction set architecture (ISA): architectures such as CISC and RISC affect efficiency and complexity. The ISA is the contract between software and hardware.

Cache: a small, very fast memory that keeps frequently used data next to the processor. Its levels are called L1, L2, and L3.

Multiple cores: let several tasks truly run at the same time and raise total performance. To speed up a single program, however, the program must be divided into threads.

Virtualization: lets several operating systems, that is, virtual machines, run on the same hardware isolated from one another. Hardware support is called VT-x on Intel and AMD-V on AMD.

Power management: when load falls, the clock frequency and supply voltage are lowered; this is called DVFS (dynamic voltage and frequency scaling). Idle cores are put into sleep states (C-states). Intel's SpeedStep and Turbo Boost technologies work on this principle: Turbo Boost briefly raises the frequency of some cores when the thermal and power limits allow. Since dynamic power is roughly proportional to $C\,V^2\,f$, lowering the voltage reduces power quadratically.

Graphics processing: modern processors include an integrated graphics processing unit (iGPU); graphics-intensive tasks are handled faster and more efficiently without a separate graphics card.

Security features: hardware-level encryption (AES-NI instructions), secure boot, and isolated secure regions (TPM, SGX, ARM TrustZone).

Connectivity: especially in mobile and embedded chips, network units such as Wi-Fi, Bluetooth, and Ethernet are inside the chip.

Machine learning: special units have been added for artificial intelligence: neural processing units (NPUs), matrix accelerators (Intel AMX), and vector dot-product instructions.

System on chip (SoC): the CPU, GPU, NPU, memory controller, and I/O are combined on one chip. This trend is a return, on a larger scale, to the microcontroller's "everything on one chip" idea. The processors in phones and Apple's M series are SoCs.

1.1.7.3 Caches and the memory hierarchy

Over the years processor speed has grown much faster than memory speed. While a present-day processor finishes several instructions in one cycle, a read from main memory takes 50–100 ns, that is, hundreds of cycles. If every access went to main memory, the processor would spend most of its time waiting. The solution is to use memories of different speeds and sizes together in a hierarchy.

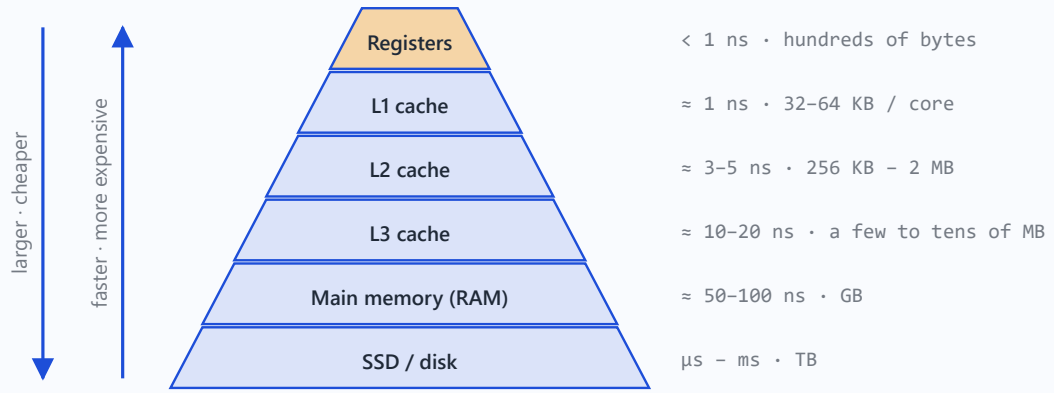


Figure 1.1.24: The memory hierarchy: registers, L1, L2, and L3 cache, main memory, and disk. Going up, speed increases and cost per byte rises; going down, capacity grows and cost falls. The times are approximate.

Definition 1.1.20

A **cache** is a small, fast memory that keeps a copy of the frequently used part of the data in main memory next to the processor. The **principle of locality** is the tendency of programs to access the same data again within a short time (temporal locality) and to access data at nearby addresses one after another (spatial locality).

The reason a cache works is the principle of locality. The instructions of a loop are read again on every pass; the elements of an array are read from consecutive addresses. The cache keeps this small working set close to the processor, and the great majority of accesses are found in the cache. The average access time therefore falls far below that of RAM.

Example 1.1.9

Access to the cache takes 1 ns and access to main memory takes 100 ns. If 95% of accesses are found in the cache, what is the average access time?

95% of accesses take 1 ns. In the remaining 5%, the cache is looked at first (1 ns), and when the data is not found, main memory is accessed (100 ns):

$$\text{average} = 1 + 0.05 \times 100 = 1 + 5 = 6 \text{ ns}$$

The 100 ns access time of main memory falls to 6 ns on average; an improvement of about 17 times. If the hit rate rises to 99%, the average becomes $1 + 0.01 \times 100 = 2$ ns.

The 8086 had no cache; a memory access took only four clock cycles anyway, and processor and memory speeds were close to each other. This balance broke down in the late 1980s. The 80486 brought the on-chip cache, the Pentium separate instruction and data caches, the Pentium Pro the L2 in the same package, and the Core i7 an L3 shared among the cores.

Common mistake

It is often thought that a cache speeds up the RAM or compresses the data. A cache does not speed up the RAM itself and does not compress data. It only keeps a copy of the frequently used data nearby; because most accesses are served without ever going to RAM, the average time falls. In a program with poor locality, for example one that accesses addresses scattered randomly in memory, the gain from the cache is small too.

1.1.8 Common mistakes

- **Confusing the data bus with the address bus.** The address bus determines the addressable memory (2^n bytes); the data bus determines the number of bits carried in one access. The 16-bit 8086 addresses 1 MB.
- **Assuming IP grows by 1 with every instruction.** IP holds a byte address and grows by the length of the instruction; 8086 instructions are 1–6 bytes.
- **Confusing the instruction cycle, the bus cycle, and the clock cycle.** The clock cycle is the smallest unit of time. A bus cycle on the 8086 is at least 4 clock cycles. An instruction cycle can contain one or several bus cycles and can also include clock cycles that are not bus cycles.
- **Counting a taken and a not-taken `loop` with the same duration.** On the 8086 a taken `loop` takes 17 cycles and a not-taken one takes 5; in a loop with n passes there are $n - 1$ jumps.
- **Adding the high half with `add` in a multiword addition.** The carry from the low half is added only by `adc`.
- **Thinking the pipeline speeds up a single instruction.** The pipeline raises throughput, not latency; n instructions finish in $k + n - 1$ cycles.
- **Equating clock speed with performance.** The time is $N \times CPI / f$; a processor with a low clock but a low CPI can be faster.
- **Basing the definition of "*n*-bit processor" on the external data bus.** The definition rests on register and ALU width: the 8088's external bus is 8 bits but the processor is 16-bit; the Pentium's external bus is 64 bits but the processor is 32-bit.
- **Thinking the cache speeds up the RAM.** The cache only keeps a copy of frequently used data nearby; its gain comes from the principle of locality.

1.1.9 Exercises and solutions

These exercises are the worked versions of the questions on the exercise slide of the presentation.

1.1.9.1 Clock period and bus cycle

Question. On a 5 MHz 8086, what are the clock period and the duration of a 4 T bus cycle?

Solution. $T = 1 / f = 1 / (5 \times 10^6)\ s = 200\ ns$. A bus cycle is four cycles: $4 \times 200\ ns = 800\ ns$.

1.1.9.2 A 24-bit address bus

Question. How many bytes of memory can a 24-bit address bus address? Which processor has one?

Solution. $2^{24} = 16{,}777{,}216$ bytes. Since 1 MB = 2^{20} bytes, this is $2^4 = 16$ MB. The 80286 has a 24-bit address bus.

1.1.9.3 100 instructions on a five-stage pipeline

Question. In a 5-stage pipeline, in how many cycles do 100 instructions finish? How many cycles are needed without the pipeline?

Solution. $k + n - 1 = 5 + 100 - 1 = 104$ cycles. Without the pipeline, $5 \times 100 = 500$ cycles are needed. The speedup is $500 / 104 \approx 4.8$ times.

1.1.9.4 Execution time

Question. How many seconds does a program of $N = 2 \times 10^9$ instructions take on a 3 GHz processor with CPI = 1.5?

Solution. $2 \times 10^9 \times 1.5 / (3 \times 10^9) = 1\ s$.

1.1.9.5 Microprocessor or microcontroller?

Question. For a washing-machine controller and for a laptop, is a microprocessor or a microcontroller chosen? Why?

Solution. For the washing machine, a microcontroller: it is cheap, uses little power, and does a single fixed job; memory and I/O are inside the chip. For the laptop, a microprocessor: high performance, gigabytes of memory, and an operating system are needed.

1.1.9.6 An EPIC bundle

Question. How are the 128 bits of an EPIC bundle divided?

Solution. Three 41-bit instructions take $3 \times 41 = 123$ bits; the remaining 5 bits are the template: $123 + 5 = 128$.

1.1.10 Summary

- The **microprocessor** is a programmable CPU chip that runs the instructions in memory with the fetch–decode–execute cycle. The **microcontroller** also carries memory and peripherals on the same chip; it is used in cheap, low-power systems dedicated to a single job.
- The CPU talks to the outside world over three buses: the **address bus** (unidirectional; n lines address 2^n bytes), the **data bus** (bidirectional), and the **control bus**. The 8086 has a 20-bit address bus (1 MB) and a 16-bit data bus.
- The **instruction cycle** consists of the fetch, decode, execute, and result stages. **IP** holds the address of the next instruction and grows by the length of the instruction; jumps write a new value into IP.

- The CPU consists of the **ALU**, the **control unit**, the **registers**, and the **bus interface**. On the 8086 the BIU prefetches instructions and holds them in a 6-byte queue, and the EU executes them.
- The period is $T = 1/f$; on the 8086 a bus cycle is at least 4 T. $CPU\ time = N \times CPI / f$. Clock speed alone is not a measure of performance.
- The time of a loop is found by adding up the cycle counts of the instructions; in loops with small bodies, the cost of the jumps determines the time.
- A 16-bit processor holds 32-bit numbers in register pairs and adds them with `add` + `adc`.
- **CISC** offers rich, variable-length instructions; **RISC** offers simple fixed-length instructions + load/store + a **pipeline**; **EPIC** offers compiler-driven parallelism. In a k-stage pipeline, n instructions finish in $k + n - 1$ cycles.
- **Vector** and **array (SIMD)** processors process many data items with one instruction; a **DSP** does multiply-accumulate in one cycle; a **GPU** exploits data parallelism with thousands of simple cores; **symbolic** processors process rules and symbols.
- Intel 4004 (1971) → 8080 (8-bit) → 8086 (1978, 16-bit, 1 MB) → 80286 (protected mode) → 80386 (32-bit) → 80486 (cache, FPU) → Pentium (superscalar) → Core 2 (64-bit, multicore). **Backward compatibility** is x86's greatest strength.
- Modern processors gain performance from **caches**, multiple cores, virtualization, power management, and special accelerators. The gain from a cache comes from the **principle of locality**.

1.1.11 Key terms

microprocessor microcontroller (MCU) central processing unit (CPU) chip integrated circuit (IC)
bus address bus data bus control bus bus cycle T state wait state instruction cycle
fetch / decode / execute opcode operand immediate value instruction pointer (IP)
program counter (PC) arithmetic logic unit (ALU) control unit register flag
bus interface unit (BIU) execution unit (EU) prefetch queue microcode clock signal clock cycle
clock speed cycles per instruction (CPI) word length word segment offset carry flag (CF)
instruction set CISC (complex instruction set computer) RISC (reduced instruction set computer)
pipeline pipeline stall scalar processor vector processor SIMD (single instruction, multiple data)
digital signal processor (DSP) graphics processing unit (GPU) floating-point unit (FPU) real mode
protected mode branch prediction cache memory hierarchy

1.1.12 Test Yourself

1. The loop examined in the clock-cycle count (`mov cx, 4` … `hlt`, 78 cycles) takes how many

microseconds on a 10 MHz 8086?

Answer

At 10 MHz, *T = 100 ns*. The cycle count does not depend on the frequency: *78 × 100 ns = 7.8 µs*. The time is half the 15.6 µs at 5 MHz.

2. If `mov cx, 5` is written instead of `mov cx, 4` in the same loop, how many cycles does the loop take and what is AX?

Answer

The body runs five times; `loop` jumps four times and falls through once: *4 + 4 + 5 × 3 + 4 × 17 + 5 + 2 = 98* cycles. AX = 5 + 4 + 3 + 2 + 1 = 15. At 5 MHz the time is *98 × 200 ns = 19.6 µs*.

3. If the program in the IP trace started at address `0200h` instead of `0100h`, would the machine code of `loop again` change?

Answer

It would not. `loop` stores not the target address but the displacement relative to the next instruction. The label would be at `0206h` and the next instruction at `020Ah`; the displacement is still *−4* (`FCh`) and the code is still `E2 FC`.

4. DX:AX = `0000FFFFh` and CX:BX = `00000001h`, and `add ax, bx` and `adc dx, cx` are executed. What are AX, DX, CF, and ZF after each instruction?

Answer

`add ax, bx` : `FFFFh + 0001h = 10000h` ; AX = `0000h` , CF = 1, ZF = 1. `adc dx, cx` : *0000h + 0000h + 1 = 0001h*; DX = `0001h` , CF = 0, ZF = 0. The result is DX:AX = `00010000h` = 65,536.

5. With AL = `05h` , `add al, 0FBh` is executed. What are AL, ZF, and CF?

Answer

`05h + FBh = 100h` . The 9th bit does not fit in AL: AL = `00h` , CF = 1. Because the result is zero, ZF = 1. In the signed interpretation *5 + (−5) = 0* is a correct result, and OF stays 0.

6. In a 6-stage pipeline, in how many cycles do 50 independent instructions finish? How many cycles

are needed without the pipeline?

Answer

$6 + 50 - 1 = 55$ cycles. Without the pipeline, $6 \times 50 = 300$ cycles. The speedup is $300 / 55 \approx 5.5$ times.

7. If the scalar loop ($c = a + b$ with four elements) were applied to arrays of 100 elements, how many instructions would execute? How many of them would be additions?

Answer

$2 + 100 \times 5 = 502$ instructions; 100 of them are additions. The remaining 402 instructions are reads, writes, loop overhead, and setup.

8. A processor runs at 2.4 GHz with CPI = 0.8. How many million instructions does it execute per second? How many seconds does a program of 6×10^9 instructions take?

Answer

$f / CPI = 2.4 \times 10^9 / 0.8 = 3 \times 10^9$ instructions/s, that is, 3000 MIPS. The time is $6 \times 10^9 / (3 \times 10^9) = 2$ s.

9. Access to the L1 cache takes 2 ns and access to main memory takes 80 ns, and 90% of accesses are found in the cache. What is the average access time?

Answer

$2 + 0.10 \times 80 = 2 + 8 = 10$ ns. This is an 8-fold improvement over main memory.

10. How many address lines are needed to address 4 GB of memory? How many lines would have to be added to the 8086's address bus?

Answer

4 GB = $2^2 \times 2^{30} = 2^{32}$ bytes, that is, 32 lines. 12 lines would have to be added to the 8086's 20; the 80386's address bus is 32 bits.

1.1.13 Further reading

- B. B. Brey, *The Intel Microprocessors* (8th edition), Chapters 1–2: covers the history of the Intel family from the 8086 to the Pentium 4, the memory and I/O organization, and the register model, with the 8086 at the center; it is the main source for this course.

- W. Stallings, *Computer Organization and Architecture* (10th edition), Chapters 2–3 and 14–16: treats in detail the evolution of the computer, the instruction cycle, bus structures, pipelining, and RISC and superscalar processors.
- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Chapters 1 and 4: presents the CPU-time formula and performance comparison, the five-stage RISC pipeline, and dependencies with clear examples.
- Intel Corporation, *8086 16-Bit HMOS Microprocessor* datasheet: contains the 8086's pins, the bus-cycle timing (T1–T4), and the cycle counts of the instructions; the cycle counts in this chapter are taken from this source.

1.1.14 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 3rd edition, Pearson.
2. B. B. Brey, *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium …*, 8th edition, Pearson.
3. W. Stallings, *Computer Organization and Architecture*, 10th edition, Pearson.
4. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.
5. Intel Corporation, *8086 16-Bit HMOS Microprocessor* datasheet; *Intel 64 and IA-32 Architectures Software Developer's Manual*.

PART I · CHAPTER 1.2

Number Systems

Everything inside a microprocessor is made of bits: registers, memory cells, the lines of the address and data buses, flags. The pattern 1111 1011 sitting in a register can, depending on context, be 251, be −5, be part of a character, or be the exponent field of a floating-point number. The pattern itself carries no meaning; the meaning comes from the representation the programmer chooses. This chapter collects the number representations needed before moving on to 8086 programming. It starts with positional number systems and converts between bases both by hand and with 8086 instructions. It then examines the two's complement representation of negative numbers, binary and hexadecimal arithmetic, and the difference between carry and signed overflow. The last sections cover the hardware counterpart of the bit (voltage levels, flip-flops), the IEEE 754 floating-point format, BCD, byte order, and the Gray code.

LEARNING OUTCOMES

Read positional systems
Compute the value of a decimal, binary, or hexadecimal number from its digit weights; state the ranges of the units bit, nibble, byte, and word; use the assembly suffixes `b`, `h`, and `o` correctly.

Convert between bases
Convert a decimal integer with the divide-and-remainder method and the fractional part with repeated multiplication by 2; move between binary and hexadecimal by grouping bits in fours; trace programs that do the same job with the `DIV` and `SHR` instructions.

Use signed numbers
Write negative numbers in two's complement; compare it with one's complement and sign-magnitude; explain the 8-bit and 16-bit signed ranges, sign extension, and fixed-point fractions.

Do arithmetic and interpret flags
Add and subtract in binary and hexadecimal; turn subtraction into addition of a complement; find, with justification, the values of the CF and OF flags after an operation.

Decode encodings
Decode and encode a 32-bit IEEE 754 pattern; do BCD addition with the correction step; write the little-endian memory layout of a multibyte value; produce a Gray code and convert it back.

Connect bits to hardware
Explain how voltage levels correspond to bits, how the types of flip-flops behave, and how a register is built from D flip-flops.

1.2.1 Representing numbers

A number is an abstract concept. Five apples, five fingers, and five days share the same quantity; but to write that quantity on paper, on stone, or in a circuit, a representation is needed. The choice of representation determines how easy it will be to do things with numbers. This section uses two old representations to explain why positional systems took the form they have today.

1.2.1.1 From tally sticks to symbols

People first counted with concrete objects: pebbles, knots, sticks. Someone counting with sticks shows five as `| | | | |`. This representation has one advantage: the value equals the number of symbols, and it can be read without knowing any rules. Its drawback is just as clear. A hundred takes a hundred strokes. To compare two numbers you have to count the strokes one by one.

In the Roman Empire a partial solution to this problem was found. Small numbers were still written with strokes (`III` is three), but separate symbols were used for certain groups: V stood for five sticks and X for ten. The representation we use today rests on a completely different idea: the value of a digit depends not only on the digit itself but also on its **position** in the number. The 7 in 754 is not seven but seven hundred. Figure 1.2.1 puts the three representations side by side.

Figure 1.2.1: Three representations of the same idea. On the left five tally sticks, in the middle Roman symbols (III = 3, V = 5, X = 10), on the right positional decimal notation: 754 = 7·100 + 5·10 + 4.

1.2.1.2 Roman numerals

The Roman system has seven symbols. Table 1.2.1 lists them with their values. A number is read by adding the values of the symbols and, where needed, subtracting.

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

Table 1.2.1: Roman symbols and their values.

Definition 1.2.1

Roman numerals are read with an addition and subtraction rule. A small symbol after a larger one is added: VI = 5 + 1 = 6. A small symbol before a larger one is subtracted: IV = 5 − 1 = 4. Subtraction is used in only six pairs: IV (4), IX (9), XL (40), XC (90), CD (400), and CM (900).

The safe way to read is to split the number from left to right into groups of value. Each group is either symbols that are simply added or one of the six subtraction pairs. The values of the groups are found and added.

Example 1.2.1

Let us read MCMLXXXVI. From left to right the groups are M, CM, LXXX, and VI. M = 1000. CM is a subtraction pair: because C comes before M it gives 1000 − 100 = 900. LXXX = 50 + 10 + 10 + 10 = 80. VI = 5 + 1 = 6. The total is 1000 + 900 + 80 + 6 = **1986**.

For MMVII the groups are M, M, and VII: 1000 + 1000 + 7 = **2007**. MMVI, written with one I fewer, is 2006.

Figure 1.2.2: Splitting MCMLXXXVI into groups: M = 1000, CM = 900, LXXX = 80, VI = 6; total 1986.

The Roman system has two shortcomings, and both make arithmetic hard. The first is that there is no symbol for **zero**. The second is that there is no notion of **place value**: X is ten everywhere. So it is impossible to write two Roman numerals one under the other and add them column by column. To add MCMLXXXVI and MMVII you first have to convert both to their values. Multiplication and division are even more laborious. The positional system removes both shortcomings at once.

Common mistake

Applying the subtraction rule to every small-large pair. IC is not 99; 99 is written XCIX (90 + 9). When reading MCMXLIV, it is also common to read XL like LX (60): in XL the X comes before the L, so it is subtracted and the group is 40. The correct reading is 1000 + 900 + 40 + 4 = 1944.

1.2.2 Positional number systems

In a positional system every digit position has a **weight**, and the digit is multiplied by this weight. The weights are powers of the base. In the decimal system the weights go 1, 10, 100, 1000; in binary, 1, 2, 4, 8. This section treats the three bases (10, 2, 16) with the same formula, defines the data units of the computer world, and shows how numbers are written in assembly.

1.2.2.1 The decimal system and the general formula

The system we use in daily life is the **decimal** system. Its base is 10 and it has ten digits: 0, 1, 2, …, 9. In the number 754 the 4 is in the ones place, the 5 in the tens place, and the 7 in the hundreds place. The value is the sum of each digit multiplied by its own place weight: 7 × 100 + 5 × 10 + 4 × 1 = 754. Figure 1.2.3 shows this sum place by place.

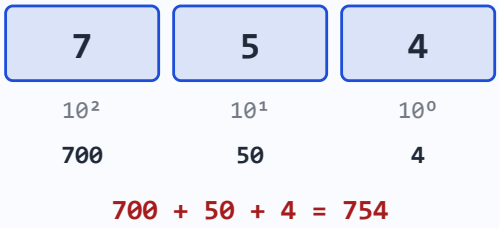


Figure 1.2.3: The digits and weights of 754. $7 \times 10^2 = 700$, $5 \times 10^1 = 50$, $4 \times 10^0 = 4$; total 754.

Definition 1.2.2

A **positional number system** is a system in which the value of a digit is determined by the position it occupies. The value of an n-digit number $d_{n-1} \ldots d_1\, d_0$ written in **base** (radix) b is

$$N = d_{n-1} \times b^{n-1} + \ldots + d_1 \times b^1 + d_0 \times b^0 = \Sigma_{i=0..n-1}\, d_i \times b^i$$

Here each digit d_i lies between 0 and $b - 1$; i is the position of the digit and is counted from the right starting at 0.

The formula is the same for every base; only b changes. So there is no need to learn a new rule for binary and hexadecimal numbers. The set of digits changes with the base: base 10 has ten digits, base 2 has two, and base 16 has sixteen. A digit can never be equal to or greater than the base. There is no digit 2 in a binary number, and no single digit “10” in a decimal one.

1.2.2.2 The importance of position, and zero

754 and 547 consist of the same three digits, but their values are very different. The 7 has the value 700 when put on the far left and the value 7 when put on the far right. Figure 1.2.4 shows the two numbers side by side. Where X is ten everywhere in the Roman system, in the positional system the same digit takes the value 700, 70, or 7 depending on its position.

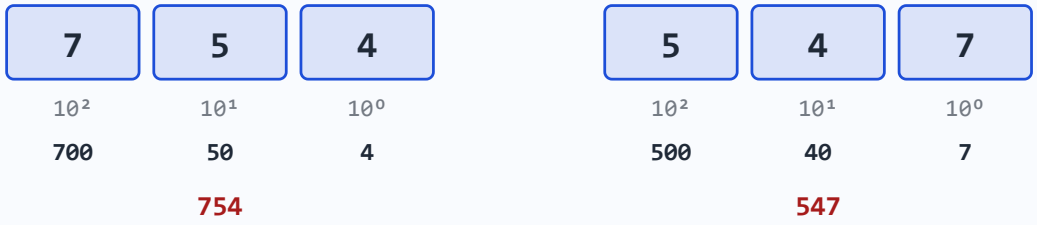


Figure 1.2.4: The same digits in different positions: 754 = 700 + 50 + 4, 547 = 500 + 40 + 7.

The rightmost position is the “ones” place in every base. The reason is that the zeroth power of every nonzero number is 1: $x^0 = 1$. In mathematics 0^0 is considered indeterminate; in number systems and polynomials it is taken as 1 by convention. Since the base is never 0, this detail causes no problem in practice.

The digit **zero** is essential for a positional system to work. In the number 705 the tens place is empty. If this gap is not filled with a zero, 705 and 75 cannot be told apart. Here zero is a placeholder meaning “nothing here.” The Roman system did not need a zero, because the value of its symbols was independent of position.

1.2.2.3 The binary system

The reason computers use binary instead of decimal is electronics. Keeping a circuit element at one of ten different voltage levels and telling these levels apart reliably in the presence of noise is hard. An element with two states is easy to build. A transistor is either conducting or cut off. A line is either at high voltage or low. A capacitor is either charged or empty. These two states are named 1 and 0. Figure 1.2.5 shows how the voltage on a line carries the values 1 and 0 over time.

Figure 1.2.5: A binary signal. High voltage (5 V) represents 1 and low voltage (0 V) represents 0; the sequence drawn is 1011 0010.

Definition 1.2.3

The **binary** system is the positional system with base 2. Its digits are only 0 and 1. The place weights are powers of 2: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, … Each digit of a binary number is called a **bit** (binary digit).

The binary system uses many more digits than decimal. The number 1000 is four digits in decimal and takes ten in binary. In return, because each digit has only two values, the rules of arithmetic are very simple. The addition table has four rows and the multiplication table also has four. This simplicity makes it possible to build an adder circuit from a few logic gates.

1.2.2.4 Bit, nibble, byte, and word

Bits are processed not one at a time but in groups of fixed width. Each group width has a name. Figure 1.2.6 splits a 32-bit value into these groups; Table 1.2.2 gives the range of each unit and its counterpart on the 8086.

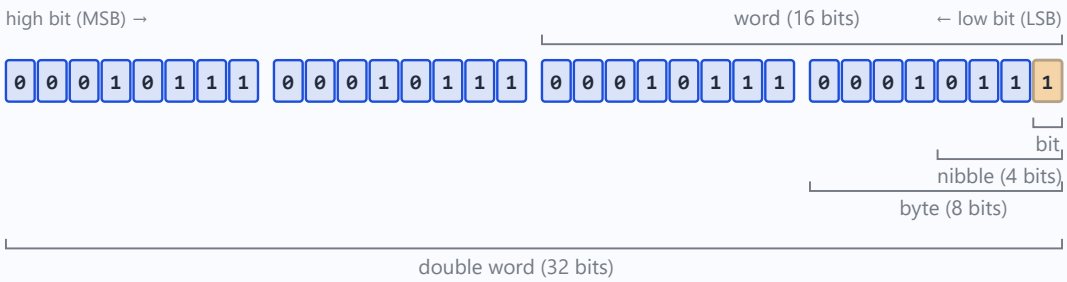


Figure 1.2.6: The groups of a 32-bit value: the single rightmost digit is a bit, 4 bits a nibble, 8 bits a byte, 16 bits a word, 32 bits a double word. The leftmost bit is the most significant bit (MSB) and the rightmost the least significant bit (LSB).

Unit	Bits	Unsigned range	On the 8086
bit	1	0 … 1	a flag (CF, ZF …)
nibble	4	0 … 15 (0h … Fh)	one hexadecimal digit, one BCD digit
byte	8	0 … 255	AL, AH, BL …; `DB`
word	16	0 … 65,535	AX, BX, CX, DX …; `DW`
double word	32	0 … 4,294,967,295	the DX:AX pair; `DD`

Table 1.2.2: Data units, their unsigned ranges, and their counterparts on the 8086.

The third column of the table follows from a single rule: with n bits, 2^n different patterns can be written. Counted from 0, the largest value is $2^n - 1$. For 8 bits there are $2^8 = 256$ patterns and the largest value is 255. For 16 bits there are 65,536 patterns and the largest value is 65,535. These limits

come up constantly in 8086 programming. You cannot store 256 in a byte; you cannot store 70,000 in a word.

Definition 1.2.4

A **nibble** is a group of 4 bits. A **byte** is a group of 8 bits and the smallest addressable unit of memory. A **word** is a group of 16 bits on the 8086, and a **double word** is a group of 32 bits. The leftmost bit of a group is called the **most significant bit** (MSB), and the rightmost bit the **least significant bit** (LSB).

The word width depends on the processor. The 8086 is a 16-bit processor: its general-purpose registers are 16 bits and its data bus carries 16 bits in one bus cycle. The AX register consists of two bytes. The high byte is AH and the low byte is AL. If AX = 1234h, then AH = 12h and AL = 34h. A 32-bit value does not fit in a single register on the 8086; the multiply and divide instructions hold such values in the DX:AX pair. In this pair DX carries the high word and AX the low word.

1.2.2.5 Reading binary numbers

In assembly source code, binary numbers end with the letter `b`: `101b` is five. To find the value of a binary number the positional formula is applied. Since the digits are only 0 and 1, no multiplication is needed. Positions that hold 0 add nothing to the sum, and positions that hold 1 add their weights. The shortcut is this: add up only the weights of the positions that hold 1.

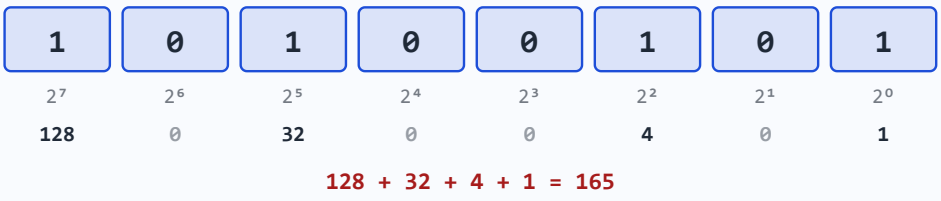


Figure 1.2.7: The value of 10100101b. The weights of the positions that hold 1 are 128, 32, 4, and 1; the total is 165.

Knowing the 8-bit weights by heart speeds up reading: from left to right 128, 64, 32, 16, 8, 4, 2, 1. Figure 1.2.7 works out the example 10100101b: 128 + 32 + 4 + 1 = 165. In a number with fewer than 8 digits the weights still start from the right. The missing digits on the left are taken as 0.

Example 1.2.2

Let us convert 1111011b to decimal. The number has seven digits; the weights go from 2^0 to 2^6 starting at the right. Let us add the digits one by one from right to left and follow the running total:

1. Digit 0: $1 \times 2^0 = 1$, running total 1.
2. Digit 1: $1 \times 2^1 = 2$, running total 3.
3. Digit 2: $0 \times 2^2 = 0$, running total 3.
4. Digit 3: $1 \times 2^3 = 8$, running total 11.
5. Digit 4: $1 \times 2^4 = 16$, running total 27.
6. Digit 5: $1 \times 2^5 = 32$, running total 59.
7. Digit 6: $1 \times 2^6 = 64$, running total **123**.

The result is $(1111011)_2 = 1 + 2 + 0 + 8 + 16 + 32 + 64 = 123$. By the shortcut: if all seven bits were 1 the value would be 127; since the 2^2 position is 0, 127 − 4 = 123.

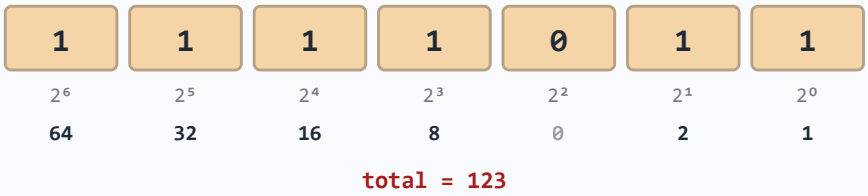


Figure 1.2.8: The place weights of 1111011b and the final running total: 64 + 32 + 16 + 8 + 0 + 2 + 1 = 123.

The trick in the last sentence of the example is a general rule. The number whose n bits are all 1 is $2^n - 1$. The value of a number with a few 0 bits can be found by subtracting the weights of the 0 bits from this upper bound. For numbers that are mostly 1s this route is faster.

1.2.2.6 The hexadecimal system

Binary numbers are long and tiring to read. Writing a 16-bit address as a string of 0s and 1s takes space and makes mistakes easy. The decimal system gives a short notation, but converting between decimal and binary needs division and multiplication. The **hexadecimal** system combines two advantages: it is short, and converting it to binary needs no computation.

Definition 1.2.5

The **hexadecimal** system is the positional system with base 16. It has sixteen digits: 0–9, and the letters A, B, C, D, E, F for the values 10–15. The place weights are powers of 16: 1, 16, 256, 4096, 65,536, …

Because $16 = 2^4$, each hexadecimal digit corresponds to exactly four bits, that is, one nibble. Table 1.2.3 gives the decimal and hexadecimal counterparts of the sixteen nibbles. This is the only table that must be known by heart when working with number systems. A byte is written with two hexadecimal digits and a word with four: 1010 0101b = A5h.

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	`0000`	0	8	`1000`	8
1	`0001`	1	9	`1001`	9
2	`0010`	2	10	`1010`	A
3	`0011`	3	11	`1011`	B
4	`0100`	4	12	`1100`	C
5	`0101`	5	13	`1101`	D
6	`0110`	6	14	`1110`	E
7	`0111`	7	15	`1111`	F

Table 1.2.3: The sixteen nibbles and hexadecimal digits.

In assembly, hexadecimal numbers end with the letter `h`: `5Fh` = 5 × 16 + 15 = 95. A **0** is put in front of hexadecimal numbers that start with a letter: `0E120h`. This zero does not change the value of the number, but it tells the assembler that it is a number. Without the zero the assembler takes `E120h` for the name of a variable or a label. Table 1.2.4 summarizes the consequences of this rule.

Notation	Meaning
`MOV AL, 0FFh`	AL = 255; correct notation
`MOV AL, FFh`	error: `FFh` is read as an undefined name
`MOV AX, 10`	decimal 10 (= 0Ah); without a suffix the number is decimal

Table 1.2.4: Writing hexadecimal constants.

The decimal value of a hexadecimal number is again found with the positional formula. For `1234h` the weights are 4096, 256, 16, and 1: 1 × 4096 + 2 × 256 + 3 × 16 + 4 × 1 = 4096 + 512 + 48 + 4 = 4660. Figure 1.2.9 shows this calculation. Digits that are letters are first converted to their decimal values and then multiplied.

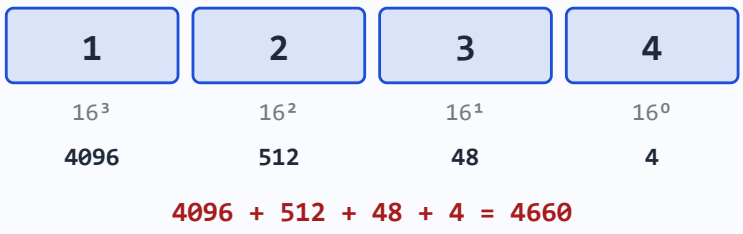


Figure 1.2.9: 1234h = $1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0$ = 4096 + 512 + 48 + 4 = 4660.

Example 1.2.3

Let us convert A12Fh to decimal. First the letters are decoded: A = 10, F = 15. The weights are $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, and $16^3 = 4096$. Starting from the right:

1. F × 16^0 = 15 × 1 = 15; running total 15.

2. $2 \times 16^1 = 32$; running total 47.
3. $1 \times 16^2 = 256$; running total 303.
4. $A \times 16^3 = 10 \times 4096 = 40{,}960$; running total **41,263**.

The result is $(A12F)_{16} = 41{,}263$. Since the value is less than 65,535, it fits in a 16-bit register: `MOV AX, 0A12Fh` is a valid instruction.

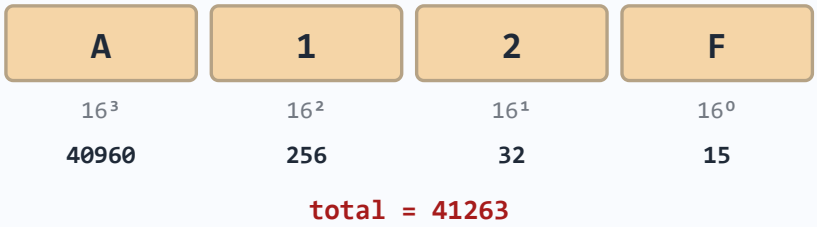


Figure 1.2.10: The digit contributions of A12Fh: 40,960 + 256 + 32 + 15 = 41,263.

1.2.2.7 Grouping in fours between binary and hexadecimal

Because each hexadecimal digit corresponds to four bits, converting between binary and hexadecimal is as easy as looking a word up in a dictionary. The binary number is split into groups of four starting from the right. If the leftmost group is shorter than four bits, 0s are added on its left. Each group is converted to a single hexadecimal digit using Table 1.2.3.

Example 1.2.4

Let us convert 1010 1011 0101 1100b to hexadecimal. The number is already split into groups of four. 1010 = 8 + 2 = 10, that is, A. 1011 = 8 + 2 + 1 = 11, that is, B. 0101 = 4 + 1 = 5. 1100 = 8 + 4 = 12, that is, C. The result is AB5Ch. Because it starts with A, it is written `0AB5Ch` in assembly.

The reverse direction uses the same table: each hexadecimal digit is expanded to four bits. `3Eh` → 0011 1110b. An example whose bit count is not a multiple of four: 1101011b is seven bits. Split from the right, the groups are 110 and 1011. A 0 is added to the left group: 0110 1011 → 6Bh.

Figure 1.2.11: The groups of four and hexadecimal digits of a 16-bit binary number: 1010 → A, 1011 → B,

0101 → 5, 1100 → C.

This is why programmers write binary data in hexadecimal almost always. A byte is exactly two hexadecimal digits, a word exactly four. 8086 debuggers and emulators show register and memory contents in hexadecimal. When a register's value is read in hexadecimal, which bits are 1 is visible at a glance. For example, if AL = 80h only bit 7 is one; if AL = 0Fh the four bits of the low nibble are one.

The same method works for the **octal** system. Because $8 = 2^3$, each octal digit corresponds to three bits and the binary number is split into groups of three. 0100 0001b is written with groups of three as 001 000 001; this is 101o. The octal system was common on old computers; it is rarely used in 8086 programming.

Common mistake

Splitting the groups of four starting from the left. If 1101011b is split from the left, the groups 1101 and 011 come out; if a 0 is added to the right group on its right, 1101 0110 = D6h is found. This is wrong: a zero added on the right doubles the number. Groups always start from the right, that is, from the ones place; missing bits are added only on the left.

1.2.2.8 Number notations in assembly

The emu8086 assembler accepts constants in four bases and as characters. The base is determined by the ending at the end of the number, that is, the **suffix**. Table 1.2.5 lists five different notations that load 65 into AL, and one negative constant.

Notation	Base	Value going to AL
`MOV AL, 65`	decimal (no suffix or `d`)	65 = 41h
`MOV AL, 41h`	hexadecimal	65 = 41h
`MOV AL, 01000001b`	binary	65 = 41h
`MOV AL, 101o`	octal (`o` or `q`)	65 = 41h
`MOV AL, 'A'`	ASCII character	65 = 41h
`MOV AL, -1`	decimal, signed	FFh

Table 1.2.5: Different notations of the same value.

The first five rows all produce the same machine code: `B0 41`. Here B0h is the opcode of the instruction "load a constant byte into AL"; 41h is the constant itself. The assembler converts the number in the source code to binary whatever base it finds it in, and embeds it in the instruction. The processor never sees the source code; it sees only the bits `1011 0000 0100 0001`. The base is only a writing convenience for people.

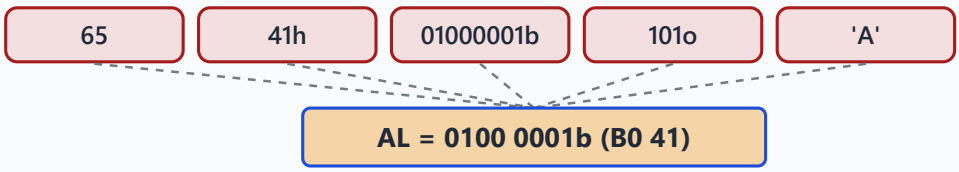


Figure 1.2.12: Five different notations turn into the same byte: 0100 0001b = 41h. The machine code of the instruction is B0 41 in every case.

The last row is interesting too: `MOV AL, -1` loads FFh into AL. The assembler converts the negative constant to two's complement form: for −1, 256 − 1 = 255 = FFh. The bit pattern FFh that goes into AL is the same pattern `MOV AL, 255` loads. To the processor there is no difference between the two.

Common mistake

Forgetting the suffix. `MOV AL, 41` loads decimal 41, that is 29h, into AL, not 41h. If the `h` suffix is left off when a hexadecimal value is copied from a memory dump or a table, the program runs with the wrong value and gives no error. The assembler cannot notice this, because 41 is a valid decimal number.

1.2.3 Base conversions

Converting a number from another base to decimal is done with the positional formula: the digits are multiplied by their weights and added. The reverse direction, from decimal to another base, needs a separate method. This section presents the divide-and-remainder method for the integer part and the multiplication method for the fractional part. It shows why both methods work and traces two small 8086 programs that do the same job, step by step.

1.2.3.1 The divide-and-remainder method

Definition 1.2.6

The **divide-and-remainder method** converts a decimal integer to base b. The number is divided by b; the quotient is kept and the remainder is written down as a digit. Division continues on the quotient until it becomes 0. Reading the remainders from last to first (bottom to top) gives the representation of the number in base b: the first remainder is the lowest digit.

Why the method works can be seen from the positional formula. Write $N = d_k b^k + \ldots + d_1 b + d_0$. Every term except d_0 is a multiple of b. So when N is divided by b, the remainder is exactly d_0 and the quotient is $d_k b^{k-1} + \ldots + d_1$. The quotient is the number with its rightmost digit deleted. Dividing it by b again gives d_1. Each division produces one digit and shortens the number by one digit. Because the remainder is always between 0 and $b - 1$, every remainder is a valid digit.

In the decimal system the same method is known without being named. Dividing 754 by 10 gives quotient 75 and remainder 4; the remainder is the last digit. Dividing 75 by 10 gives remainder 5 and quotient 7. The method is just this familiar operation done in another base.

Example 1.2.5

Let us convert 39 to hexadecimal. 39 ÷ 16 = 2, remainder 7; 7 is the ones digit. 2 ÷ 16 = 0, remainder 2. The quotient is 0, so we stop. Reading the remainders from bottom to top: **27h**. Check: 2 × 16 + 7 = 39.

Figure 1.2.13: Repeated division of 39 by 16. The remainders are 7 and 2; read from bottom to top they give 27h.

Example 1.2.6

A larger number: let us convert 43,868 to hexadecimal.

1. 43,868 ÷ 16 = 2741, remainder 12 → C.
2. 2741 ÷ 16 = 171, remainder 5.
3. 171 ÷ 16 = 10, remainder 11 → B.
4. 10 ÷ 16 = 0, remainder 10 → A. Quotient 0: stop.

Reading the remainders from bottom to top gives AB5Ch. Because it starts with A, it is written **0AB5Ch** in assembly. Check: 10 × 4096 + 11 × 256 + 5 × 16 + 12 = 40,960 + 2816 + 80 + 12 = 43,868. This is the same number as 1010 1011 0101 1100b in the grouping example.

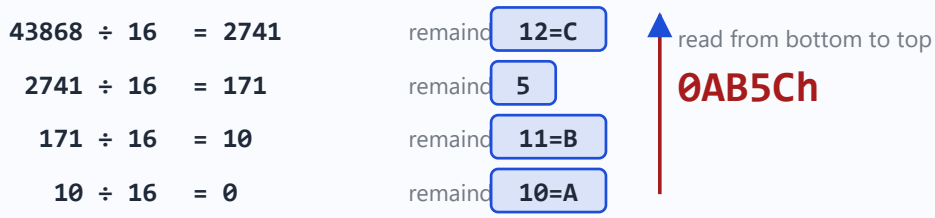


Figure 1.2.14: The division ladder of 43,868. The remainders come out as C, 5, B, A in turn; read from bottom to top they give 0AB5Ch.

To compute the remainder at each division step, subtract the product of quotient and divisor from the number: 43,868 − 2741 × 16 = 43,868 − 43,856 = 12. When working with a calculator, the decimal result can be used too. 43,868 / 16 gives 2741.75. The integer part is the quotient; the fractional part, multiplied by 16, gives the remainder: 0.75 × 16 = 12.

1.2.3.2 Code trace: hexadecimal digits with DIV

The divide-and-remainder method is a loop, and it turns directly into 8086 code. The 8086 `DIV` instruction gives the quotient and the remainder in one instruction. The following program repeatedly divides 2989 by 16 and writes the remainders into an array named `rem`.

```
ORG 100h
MOV AX, 2989   ; number
MOV BX, 16     ; base
MOV SI, 0      ; counter
again: MOV DX, 0 ; DX:AX
DIV BX         ; quot, rem
MOV rem[SI], DL
INC SI
CMP AX, 0      ; quotient 0?
JNE again
RET
rem DB 4 DUP(0)
```

Here is what the program does, line by line. `ORG 100h` is not an instruction but a **directive** given to the assembler. It declares that the program, as a .COM file, will start at offset 100h of the segment. `MOV AX, 2989` puts the number to be divided into AX; in hexadecimal 2989 is 0BADh. `MOV BX, 16` puts the divisor, that is, the target base, into BX. Had 2 or 8 been written into BX, the same code would have produced binary or octal digits. `MOV SI, 0` clears the index that holds the position of the next remainder in the array.

The line labeled `again` is the top of the loop and clears DX. The reason is how `DIV` works. `DIV BX`, whose operand is a 16-bit register, divides the 32-bit DX:AX pair by BX. It writes the quotient to AX and the remainder to DX. If DX holds a leftover value from the previous pass, the dividend becomes

65,536 × DX + AX and the result comes out completely wrong. If the quotient does not fit in 16 bits, the processor also raises a divide-error interrupt. This is why DX = 0 is set before every division.

`MOV rem[SI], DL` writes the low byte of the remainder into element SI of the array. Because the remainder is less than 16, it fits in DL and DH is always 0. `INC SI` increments the index. `CMP AX, 0` subtracts 0 from AX, stores the result nowhere, but sets the flags: if the quotient is 0, ZF = 1, otherwise ZF = 0. `JNE again` ("jump if not equal") returns to the top of the loop while ZF = 0. When the quotient becomes 0 the jump is not taken and `RET` ends the program. The last line defines the four-byte array `rem`, initially zero.

Table 1.2.6 gives the state of the program after each important step. Register values are written in hexadecimal.

Step	AX	DX	SI	`rem` (+0 … +3)	Note
first three MOVs	0BADh	0000h	0	00 00 00 00	BX = 0010h
pass 1: DIV BX	00BAh	000Dh	0	00 00 00 00	2989 ÷ 16 = 186, remainder 13 (D)
MOV, INC, CMP	00BAh	000Dh	1	0D 00 00 00	AX ≠ 0 → ZF = 0, JNE jumps
pass 2: DIV BX	000Bh	000Ah	1	0D 00 00 00	186 ÷ 16 = 11, remainder 10 (A)
MOV, INC, CMP	000Bh	000Ah	2	0D 0A 00 00	AX ≠ 0 → JNE jumps
pass 3: DIV BX	0000h	000Bh	2	0D 0A 00 00	11 ÷ 16 = 0, remainder 11 (B)
MOV, INC, CMP	0000h	000Bh	3	0D 0A 0B 00	AX = 0 → ZF = 1, JNE falls through
RET	0000h	000Bh	3	0D 0A 0B 00	three digits found

Table 1.2.6: Trace of the DIV program (2989 = 0BADh, base 16).

The loop runs three times, because 2989 has three digits in hexadecimal. The final value of SI gives the number of digits. In memory the remainders sit in the order 0D 0A 0B, starting from the lowest digit. Read from last to first they give B, A, D, that is, **0BADh**. Check: 11 × 256 + 10 × 16 + 13 = 2816 + 160 + 13 = 2989. Figure 1.2.15 shows the final state.

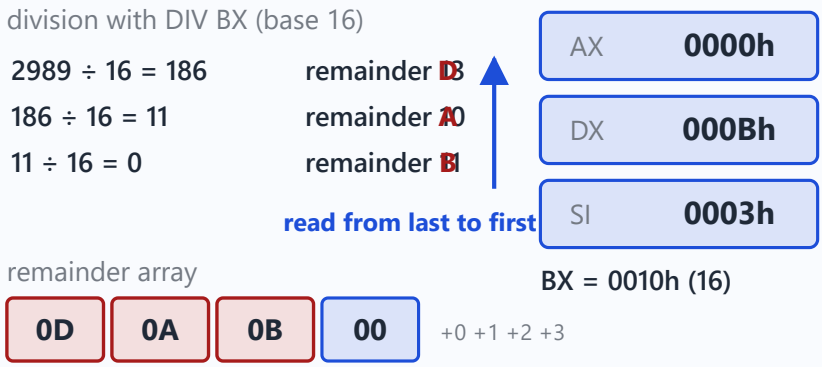


Figure 1.2.15: The registers and the `rem` array after the loop ends. The remainders of the three divisions are D, A, and B; read from last to first they give 0BADh.

The array in the program is four bytes. A 16-bit number has at most four digits in hexadecimal (FFFFh), so four bytes are always enough. With base 2, 16 digits would be needed and the array would have to be defined as `16 DUP(0)`. To print the remainders on the screen, each must be converted to an ASCII character: add 30h for 0–9 and 37h for A–F (0Dh + 37h = 44h = 'D').

A note on the speed of the program is in order. On the 8086 a 16-bit `DIV` is one of the slowest instructions; with a register operand it takes about 144–162 clock cycles. In the same time dozens of `MOV` or `ADD` instructions could run. When the divisor is a power of 2 there is no need for division at all: the shift instructions do the same job much faster. Division by 2 is the most common example.

1.2.3.3 Dividing by the binary base

The divide-and-remainder method works in every base. For base 2 the divisor is 2 and the remainder is always 0 or 1. The remainder of odd numbers is 1 and that of even numbers is 0.

Example 1.2.7

Let us convert 45 to binary.

1. 45 ÷ 2 = 22, remainder 1.
2. 22 ÷ 2 = 11, remainder 0.
3. 11 ÷ 2 = 5, remainder 1.
4. 5 ÷ 2 = 2, remainder 1.
5. 2 ÷ 2 = 1, remainder 0.
6. 1 ÷ 2 = 0, remainder 1. Quotient 0: stop.

Reading the remainders from bottom to top gives **101101b**. Check: 32 + 8 + 4 + 1 = 45. In an 8-bit register this number sits as 0010 1101b = 2Dh.

Figure 1.2.16: Repeated division of 45 by 2. The six remainders, read from bottom to top, give 101101b.

For large numbers division by 2 takes long: a number like 43,868 needs sixteen steps. A faster way is to convert the number to hexadecimal first and then expand each digit into four bits. Since 45 ÷ 16 = 2, remainder 13 (D), 45 = 2Dh. Expanding 2 = 0010 and D = 1101 gives 0010 1101b. This route ends with two divisions and a table lookup.

1.2.3.4 Code trace: the remainders come from CF

Shifting a binary number right by one bit divides it by 2. Just as deleting the last digit of a decimal number divides it by 10, dropping the last bit of a binary number divides it by 2. The bit that falls out is the remainder of the division. The 8086 `SHR` (shift right, logical) instruction writes this dropped bit into the carry flag (CF). The following program uses this observation: without `DIV`, it drops the binary digits of 45 one by one into CF and collects them in BL.

```
ORG 100h
MOV AL, 45     ; dividend
MOV CX, 8      ; 8 bits
bit: SHR AL, 1 ; rem → CF
RCR BL, 1      ; CF → BL
LOOP bit
RET
```

`MOV AL, 45` puts the dividend into AL: AL = 0010 1101b = 2Dh. `MOV CX, 8` sets up the loop counter; there will be one pass for each of the eight bits of AL. `SHR AL, 1` shifts all the bits of AL one place to the right. A 0 enters the leftmost bit and the rightmost bit falls into CF. As a result AL holds the quotient and CF holds the remainder.

`RCR BL, 1` (rotate through carry right) rotates BL together with CF like a 9-bit ring. The bit in CF enters bit 7 of BL, and bit 0 of BL goes out into CF. In this way the remainder produced by SHR is placed at the far left of BL. In every pass the old remainders in BL move one place right and the new one enters at the far left. `LOOP bit` decrements CX; if CX is not zero it returns to the label `bit`. This instruction does the job of “decrement CX, jump if not zero” in a single instruction and changes no flags.

Pass	Quotient	SHR remainder	AL	BL (binary)	BL	CX
start	—	—	2Dh	`0000 0000`	00h	8
1	45 ÷ 2 = 22	1	16h	`1000 0000`	80h	7
2	22 ÷ 2 = 11	0	0Bh	`0100 0000`	40h	6
3	11 ÷ 2 = 5	1	05h	`1010 0000`	A0h	5
4	5 ÷ 2 = 2	1	02h	`1101 0000`	D0h	4
5	2 ÷ 2 = 1	0	01h	`0110 1000`	68h	3
6	1 ÷ 2 = 0	1	00h	`1011 0100`	B4h	2
7	0 ÷ 2 = 0	0	00h	`0101 1010`	5Ah	1

Pass	Quotient	SHR remainder	AL	BL (binary)	BL	CX
8	0 ÷ 2 = 0	0	00h	`0010 1101`	2Dh	0

Table 1.2.7: Trace of the SHR/RCR program. Each row is the state after the `LOOP` instruction of that pass; “SHR remainder” is the bit that fell into CF in that pass.

Table 1.2.7 gives the eight passes. The program does not clear BL; the table assumes BL is 0 at the start. The result does not depend on this assumption: in eight passes all the old bits of BL go out into CF and are overwritten by the remainder of the next SHR. In every pass RCR shifts bit 0 of BL out into CF. This bit is always 0 until the eighth pass, so at the moment of `LOOP` CF appears as 0 in every pass. CF = 1 is seen only in the short moment between SHR and RCR: in passes 1, 3, 4, and 6. The first remainder reaches bit 0 of BL by shifting right eight times over the eight passes. The last remainder is placed in bit 7 of BL. When the program ends, BL = 0010 1101b = 2Dh = 45.

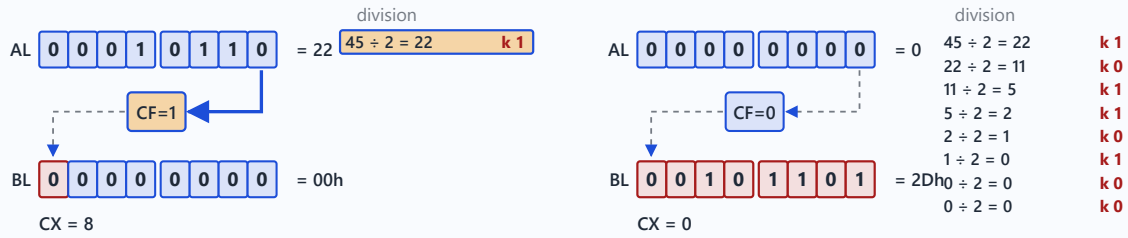


Figure 1.2.17: On the left the first `SHR AL, 1` : 45 ÷ 2 = 22 (AL = 16h), and the dropped bit gives CF = 1. On the right when the loop ends: AL = 0, BL = 0010 1101b = 2Dh.

The result may look surprising at first: the program took 45 and produced 45 again. That is exactly what makes it instructive. AL already holds a binary number; to “convert it to binary” there is no need to rewrite its bits. The program shows that the remainders of division by 2 are the very bits of the number. Arranged from last to first, the remainders reveal the binary representation of the number. The same idea is used to process the bits of a number one at a time: the bit dropped by SHR is read from CF and a decision is made with `JC` or `JNC`.

Common mistake

Using ROR instead of RCR. `ROR BL, 1` rotates BL independently of CF: bit 0 moves to bit 7 and is only copied to CF. The remainder SHR dropped into CF never enters BL; the program merely rotates BL’s initial value. To carry the remainder, the rotation must go through the carry (RCR), or a separate instruction that reads CF after SHR must be used.

1.2.3.5 The fractional part: multiplying by 2

The positional formula extends to the right of the radix point too. The weights of the digits after the point are negative powers of the base. In decimal, $0.375 = 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$. In binary, the weights after the point go $2^{-1} = 0.5$, $2^{-2} = 0.25$, $2^{-3} = 0.125$, $2^{-4} = 0.0625$. 0.101b = 0.5 + 0.125 = 0.625.

Definition 1.2.7

The **multiplication method** converts a decimal fraction between 0 and 1 to binary. The fraction is multiplied by 2; the integer part of the product (0 or 1) becomes the next fraction bit. The process continues with the fractional part of the product. It stops when the fraction becomes 0 or the desired number of bits is reached. The bits are read **from top to bottom**, that is, in the order they were produced: the first bit is the 2^{-1} place.

The justification of the method mirrors the division method. Write $x = 0.f_1f_2f_3\ldots$ (binary). $2x$ shifts the point one place to the right: $f_1.f_2f_3\ldots$. The integer part of the product is f_1, and the fractional part is the number made of the remaining bits. Each multiplication moves one bit to the left of the point. In division the first remainder was the lowest digit; in multiplication the first integer part is the highest fractional digit. This is why the reading direction is reversed.

Example 1.2.8

Let us convert 0.625 to binary.

1. 0.625 × 2 = 1.25: integer part **1** (the 2^{-1} bit); continue with fraction 0.25.
2. 0.25 × 2 = 0.5: integer part **0** (the 2^{-2} bit); fraction 0.5.
3. 0.5 × 2 = 1.0: integer part **1** (the 2^{-3} bit); fraction 0, stop.

Reading the bits from top to bottom gives **0.101b**. Check: 1/2 + 1/8 = 0.5 + 0.125 = 0.625.

For a number that has both an integer and a fractional part, the two parts are converted separately and combined. For 25.625, since 25 = 11001b (by the division method) and 0.625 = 0.101b, 25.625 = **11001.101b**.

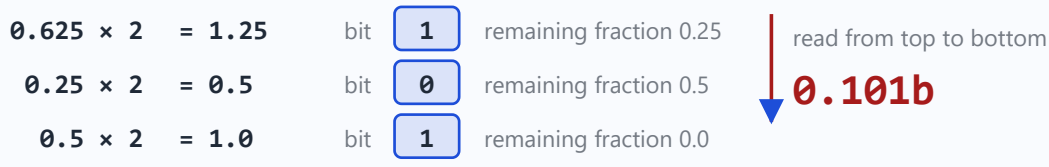


Figure 1.2.18: Repeated multiplication of 0.625 by 2. The integer parts are 1, 0, 1; read from top to bottom they give 0.101b.

Fractional binary numbers are converted to hexadecimal by grouping in fours as well. The only difference is that the groups are formed outward from the point in both directions. The integer part is grouped from the point to the left and the fractional part from the point to the right. In the fractional part, missing bits are added as 0s on the right; a zero added to the right of the point does not change the value. For 1101.011b the groups are 1101 and 0110, and the result is D.6h.

1.2.3.6 Repeating fractions: 0.1

Not every decimal fraction can be written finitely in binary. 1/3 goes on forever in decimal as 0.333…; in the same way 0.1 goes on forever in binary. The multiplication method shows this clearly.

Example 1.2.9

Let us convert 0.1 to binary.

1. 0.1 × 2 = 0.2 → bit 0; fraction 0.2.
2. 0.2 × 2 = 0.4 → bit 0; fraction 0.4.
3. 0.4 × 2 = 0.8 → bit 0; fraction 0.8.
4. 0.8 × 2 = 1.6 → bit 1; fraction 0.6.
5. 0.6 × 2 = 1.2 → bit 1; fraction **0.2**.
6. 0.2 × 2 = 0.4 → bit 0; the second step repeats.

After the fifth step the fraction returned to 0.2. 0.2 had been seen at the second step; from here on steps 2–5 repeat forever, and the bit pattern 0011 recurs. The result is 0.1 = 0.0 0011 0011 0011…b. If it is stored with 8 fraction bits, 0.0001 1001b = 0.097 656 25 is obtained.

Figure 1.2.19: Converting 0.1 to binary. When the fraction returns to 0.2, the pattern 0011 repeats forever.

A single rule decides which fractions can be written finitely. If a fraction in lowest terms is p/q, then for it to be finite in binary, q must be a power of 2. 0.625 = 5/8, and since $8 = 2^3$ it is written exactly with three bits. 0.1 = 1/10, and 10 has the factor 5; so it is not finite. The same rule holds in the

decimal system: fractions whose denominator contains only the factors 2 and 5 are finite, and 1/3 is infinite.

The practical consequence of this rule is important. Because a computer uses a limited number of bits, it can store values like 0.1 only approximately; the extra bits are cut off or rounded. In the IEEE 754 floating-point format too, 0.1 is stored not exactly but approximately. This is why in many programming languages 0.1 + 0.2 does not come out exactly equal to 0.3. In applications such as money, where decimal fractions must be kept exactly, BCD or an integer number of cents is used.

Common mistake

Reading the fraction bits from bottom to top, as in the division method. For 0.625 the bits come out in the order 1, 0, 1, and because it is symmetric, the wrong reading goes unnoticed. For 0.375, however, the bits are 0, 1, 1: the correct result is 0.011b = 0.375; the reverse reading gives 0.110b = 0.75. In division the remainders are read in reverse, in multiplication the integer parts are read straight.

1.2.3.7 Using the conversions together

The base converter in the slides shows an entered number in all four bases at once and lists the division steps for the chosen base. The route to follow when doing the same job by hand depends on the source and the target. Table 1.2.8 summarizes the shortest routes.

Source → target	Method
binary/hexadecimal/octal → decimal	positional formula: sum of digit × weight
decimal → hexadecimal	divide by 16 and take remainders; for fractions, multiply by 16
decimal → binary	first to hexadecimal, then expand each digit to 4 bits; or divide by 2
binary ↔ hexadecimal	groups of four from the right; for fractions, outward from the point
binary ↔ octal	groups of three from the right
hexadecimal ↔ octal	via binary: first expand to 4 bits, then group in threes

Table 1.2.8: Short routes for conversion between bases.

Example 1.2.10

Let us write 2024 in hexadecimal, binary, and octal. 2024 ÷ 16 = 126, remainder 8; 126 ÷ 16 = 7, remainder 14 (E); 7 ÷ 16 = 0, remainder 7. The result is 7E8h. Each digit is expanded to four bits: 0111 1110 1000b. For octal, the same bits are grouped in threes from the right: 011 111 101 000 → 3750o. Check: 7 × 256 + 14 × 16 + 8 = 1792 + 224 + 8 = 2024 and 3 × 512 + 7 × 64 + 5 × 8 + 0 = 2024.

1.2.4 Signed numbers

Up to here all numbers were zero or positive. Real programs, however, also work with negative values such as temperature differences, debts, and changes of direction. On paper a negative number is written with a minus sign in front. A register has no separate place for a minus sign; it has only bits. This section examines how negative numbers can be represented with bits and treats in detail the two's complement representation the 8086 uses.

1.2.4.1 Pattern and interpretation

Is `0FFh` positive or negative? This question cannot be answered by looking at the bit pattern. The pattern 1111 1111 is 255 if read as unsigned and −1 if read in two's complement. The processor does not know which of the two interpretations is meant, and it does not need to. The interpretation is determined by the instruction that uses the pattern, or by the programmer.

Definition 1.2.8

In the **unsigned** interpretation, n-bit patterns represent the values between 0 and $2^n - 1$. In the **signed** interpretation half of the patterns are set aside for negative numbers. In the **two's complement** representation used by the 8086, the negative number $-x$ is written with the pattern of the unsigned value $2^n - x$.

With 8 bits there are 256 different patterns. In two's complement the first 128 of them (00h … 7Fh, that is, 0 … 127) represent zero and the positive numbers. The next 128 (80h … FFh, that is, unsigned 128 … 255) represent the negative numbers. To write −5, 5 is subtracted from the number of patterns: 256 − 5 = 251 = FBh. For −1, 256 − 1 = 255 = FFh; for −128, 256 − 128 = 128 = 80h. Figure 1.2.20 shows the two interpretations of the 256 patterns on the same axis.

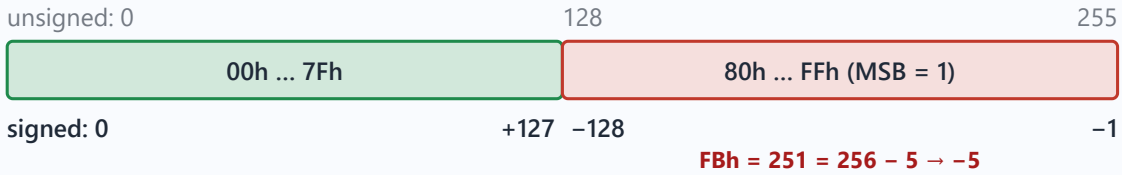


Figure 1.2.20: Two interpretations of the 256 8-bit patterns. 00h … 7Fh is 0 … 127 in both interpretations. 80h … FFh is 128 … 255 in the unsigned interpretation and −128 … −1 in the signed one; all of these patterns have MSB = 1.

The two interpretations share a property: in all negative numbers the most significant bit is 1, and in all positive ones it is 0. That is why the MSB is called the **sign bit**. After every arithmetic and logic operation the 8086 copies the MSB of the result into the **sign flag** (SF). If SF = 1, the result is negative in the signed interpretation.

1.2.4.2 The number circle

A fixed-width register behaves not like a line but like a circle. Consider a 4-bit register. Starting at 0000 and adding 1 repeatedly, we reach 0001, 0010, … 1111. Adding 1 to 1111 gives 1 0000; the fifth bit does not fit in the register and the register returns to 0000. Figure 1.2.21 arranges the sixteen patterns on a circle. Moving one step clockwise is adding 1, and one step counterclockwise is subtracting 1.

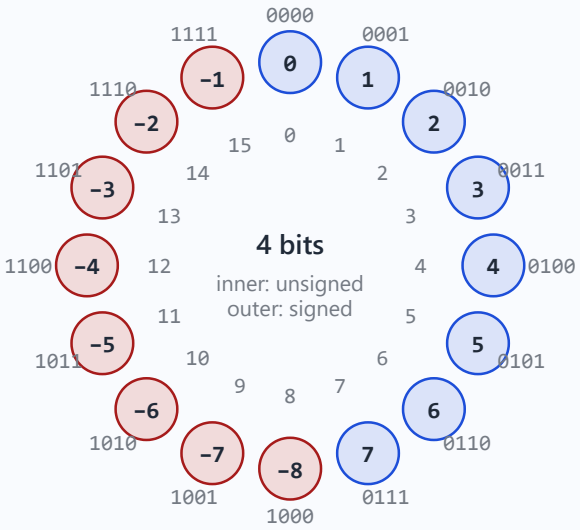


Figure 1.2.21: The 4-bit number circle. Next to each pattern, the unsigned value is written on the inside and the two's complement value on the outside. 0000–0111 is 0–7 in both interpretations; 1000–1111 is unsigned 8–15 and signed −8 … −1.

The circle clearly shows two "break" points. The first is between 1111 and 0000. In the unsigned interpretation this is where the count jumps from 15 to 0. If an addition crosses this point, the result did not fit in the unsigned interpretation, and the 8086 reports this with the **carry flag** (CF). The second is between 0111 and 1000. In the signed interpretation this is where the count jumps from +7 to −8. If an operation crosses this point, the result did not fit in the signed interpretation, and the 8086 reports this with the **overflow flag** (OF).

The most important lesson of the circle is this: the addition and subtraction circuit is the **same** for both interpretations. Moving one step on the circle does not depend on which interpretation the pattern is read with. The operation 1011 + 0001 = 1100 is 11 + 1 = 12 in the unsigned interpretation and −5 + 1 = −4 in the signed one; both results are correct. The processor uses a single adder and sets separate flags for the two interpretations. It is up to the program to decide which flag to look at.

1.2.4.3 Three signed representations

Two's complement is not the only way to represent negative numbers. Three representations have been used historically. Table 1.2.9 compares the sixteen 4-bit patterns under four interpretations.

Binary	Unsigned	Sign-magnitude	One's complement	Two's complement
`0000`	0	0	0	0
`0001`	1	1	1	1
`0010`	2	2	2	2
`0011`	3	3	3	3
`0100`	4	4	4	4
`0101`	5	5	5	5
`0110`	6	6	6	6
`0111`	7	7	7	7
`1000`	8	−0	−7	**−8**
`1001`	9	−1	−6	−7
`1010`	10	−2	−5	−6
`1011`	11	−3	−4	−5
`1100`	12	−4	−3	−4
`1101`	13	−5	−2	−3
`1110`	14	−6	−1	−2
`1111`	15	−7	−0	−1

Table 1.2.9: Four interpretations of 4-bit patterns.

Definition 1.2.9

In the **sign-magnitude** representation the top bit gives the sign and the remaining bits give the absolute value of the number. In 4 bits, 0101 = +5 and 1101 = −5. It is the direct counterpart of the minus sign on paper.

Sign-magnitude is the easiest to read, but it has two serious problems. First, zero has two representations: 0000 = +0 and 1000 = −0. To test whether a number is zero, both patterns have to be checked. Second, the addition circuit gets complicated. In +5 + (−3), adding 0101 and 1011 directly gives 1 0000, which is not the expected +2. The adder would first have to compare the signs and then add the magnitudes or subtract the smaller from the larger. This representation is not used for integers today. The sign bit of floating-point numbers is based on the same idea.

1.2.4.4 One's complement

Definition 1.2.10

In the **one's complement** representation the negative of a number is obtained by **inverting all the bits** of the number (0 → 1, 1 → 0). In n bits the pattern of $-x$ is $(2^n - 1) - x$. On the 8086 the `NOT` instruction produces the one's complement of an operand.

An 8-bit example: +45 = 0010 1101b. Inverting all the bits gives 1101 0010b = D2h. In one's complement this pattern is −45. The same pattern in two's complement represents −46; the two representations differ by exactly one. Figure 1.2.22 shows the operation bit by bit.

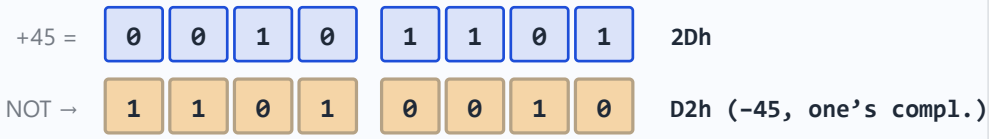


Figure 1.2.22: Inverting all the bits of +45 = 2Dh gives D2h. In one's complement, D2h = −45.

One's complement largely solves the addition problem of sign-magnitude, but it leaves two flaws. Zero still has two representations: 0000 0000 = +0 and 1111 1111 = −0. Also, if a carry comes out of the top bit in an addition, that carry has to be added back to the lowest bit of the result; this is called the **end-around carry**. In 4 bits, −2 + 5: 1101 + 0101 = 1 0010. Adding the carried 1 back gives 0010 + 1 = 0011 = 3. Some old computers (the CDC 6600, UNIVAC 1100) used this representation. Today one's complement appears not as a number representation by itself but as an intermediate step of two's complement and as the bitwise NOT operation.

1.2.4.5 Two's complement

Two's complement removes both flaws of one's complement in a single move: 1 is added to the inverted pattern. This addition eliminates the −0 pattern and removes the need for the end-around carry.

Definition 1.2.11

The **two's complement** of a number is found by first inverting all its bits and then adding 1 to the result: $-x = NOT\ x + 1$. In n bits this pattern is $2^n - x$. On the 8086 the `NEG` instruction produces the two's complement of its operand in one step; it does the operation as the subtraction $0 - x$.

It is easy to show that the two formulas give the same pattern. The pattern in which all n bits are 1 is $2^n - 1$. Inverting the bits of a number is the same as subtracting it from this pattern, because 1 − 0 = 1 and 1 − 1 = 0 and no borrow is needed in any position. Hence $NOT\ x = (2^n - 1) - x$ and $NOT\ x + 1 = 2^n - x$.

Example 1.2.11

Let us find the 8-bit two's complement representation of −45. +45 = 0010 1101b. Inverting all the bits gives 1101 0010b. Adding 1 gives 1101 0011b = **D3h**. Check: D3h = 211 = 256 − 45. A second check is done by addition: D3h + 2Dh = 100h; in 8 bits the result is 00h, that is, −45 + 45 = 0.

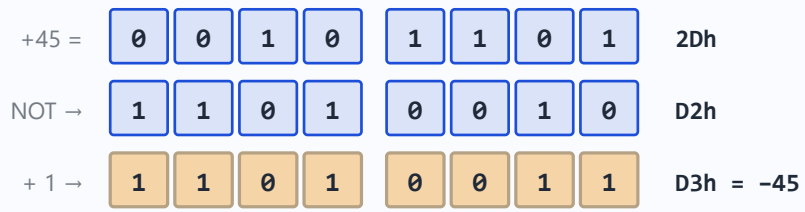


Figure 1.2.23: The two's complement of −45: 0010 1101 → NOT → 1101 0010 → +1 → 1101 0011 = D3h.

There is a faster shortcut for hand calculation. Starting from the right, the bits up to and including the first 1 are copied unchanged, and all the remaining bits on the left are inverted. For 0010 1101 the rightmost bit is already 1; it stays as it is, and the seven bits on the left are inverted: 1101 001**1**. For 0001 1000 (24) the first 1 from the right is at bit 3: the 1000 on the right stays unchanged and the 0001 on the left is inverted: 1110 1000 = E8h = 256 − 24. The reason for the shortcut is that adding 1 clears the run of 1s on the right and turns the first 0 into a 1.

Two's complement has two important properties. First, the two's complement of a number's two's complement is the number itself: $2^n - (2^n - x) = x$. The two's complement of D3h is 2Dh again. So the magnitude of a negative number is found with the same operation. Second, zero has a single representation. The two's complement of 0000 0000 is 1111 1111 + 1 = 1 0000 0000; when the ninth bit is dropped, 0000 0000 remains. The −0 pattern of one's complement (1111 1111) is −1 in two's complement.

In two's complement the signed value of a pattern can also be computed directly. The weight of the MSB is taken as negative and the other bits are added as usual. In 8 bits the weights are −128, 64, 32, 16, 8, 4, 2, 1. For D3h = 1101 0011b this gives −128 + 64 + 16 + 2 + 1 = −45. This interpretation is useful for fixed-point fractions too.

1.2.4.6 Code trace: NOT + 1 versus NEG

The following program computes the two's complement of the same number in two ways and compares the results. Then it adds −45 + 45 and repeats the same operation in 16 bits. The program is a good example for seeing the effect of each instruction on the flags.

```
ORG 100h
MOV AL, 45       ; +45
MOV BL, AL
NOT BL           ; ones' comp.
INC BL           ; + 1
NEG AL           ; one instr.
CMP AL, BL       ; equal?
ADD AL, 45       ; -45 + 45
MOV AX, 45
NEG AX           ; -45 in 16 bits
RET
```

`MOV AL, 45` loads 2Dh into AL; `MOV BL, AL` copies the same value into BL. MOV instructions change no flags. `NOT BL` inverts all the bits of BL: BL = D2h. NOT affects no flag either. `INC BL` adds 1 to BL: BL = D3h. INC sets the ZF, SF, OF, AF, and PF flags but does not touch CF. The result is not zero (ZF = 0) and its MSB is 1 (SF = 1).

`NEG AL` computes AL as 0 − AL and gives the same result in one instruction: AL = D3h. NEG sets CF = 1 if the operand is not zero, because subtracting a nonzero number from 0 requires a borrow in the unsigned interpretation. `CMP AL, BL` computes the difference AL − BL but does not store the result. Since D3h − D3h = 0, ZF = 1, CF = 0, SF = 0. ZF = 1 shows that the two routes produced the same bit pattern.

`ADD AL, 45` adds −45 and +45. D3h + 2Dh = 100h. The ninth bit does not fit in AL and falls into CF: CF = 1. AL = 00h; ZF = 1. In the signed interpretation the result is correct (−45 + 45 = 0), so OF = 0. This line shows that in two's complement CF = 1 does not mean an error. `MOV AX, 45` loads 002Dh into AX. `NEG AX` computes the 16-bit two's complement: 65,536 − 45 = 65,491 = **FFD3h**. The low byte is again D3h; the high byte has become FFh.

Instruction	AX	BL	CF	ZF	SF	OF	Note
`MOV AL, 45`	002Dh	00h	0	0	0	0	AL = +45
`MOV BL, AL`	002Dh	2Dh	0	0	0	0	copy
`NOT BL`	002Dh	D2h	0	0	0	0	one's complement; flags unchanged
`INC BL`	002Dh	D3h	0	0	1	0	D3h = −45; CF untouched
`NEG AL`	00D3h	D3h	1	0	1	0	0 − 2Dh; operand ≠ 0 → CF = 1
`CMP AL, BL`	00D3h	D3h	0	1	0	0	difference 0 → equal
`ADD AL, 45`	0000h	D3h	1	1	0	0	D3h + 2Dh = 100h
`MOV AX, 45`	002Dh	D3h	1	1	0	0	MOV leaves the flags alone
`NEG AX`	FFD3h	D3h	1	0	1	0	65,536 − 45 = FFD3h

Table 1.2.10: Trace of the NOT/NEG program. The flags are the values after each instruction; in the first row the initial value is taken as 0.

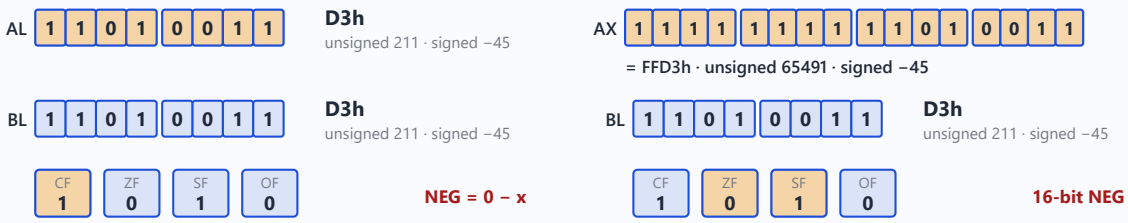


Figure 1.2.24: On the left after `NEG AL` : AL = D3h, BL = D3h, CF = 1, SF = 1. On the right after `NEG AX` : AX = FFD3h (unsigned 65,491, signed −45).

The last two rows show how the same number is written at different widths. −45 is D3h in 8 bits and FFD3h in 16 bits. The low byte of the 16-bit representation is the same as the 8-bit representation; the high byte consists only of copies of the sign bit (FFh). This observation is the basis of sign extension.

Common mistake

Confusing NOT with NEG. `NOT AL` gives the one's complement, and the result is 1 less than the true negative: with AL = 2Dh, NOT AL = D2h = −46. To change a number's sign, NEG must be used, or INC must be added after NOT. Another misconception is to take CF = 1 after NEG for an error; NEG always gives CF = 1 unless the operand is zero.

1.2.4.7 −5 + 5 = 0

The most basic rule a negative-number representation must satisfy is the equality $-x + x = 0$ from mathematics. In two's complement −5 = FBh = 1111 1011b. Adding 5 = 0000 0101b gives the sum 1 0000 0000b = 256. The ninth bit does not fit in the byte and is dropped; all of the remaining 8 bits are zero. Figure 1.2.25 shows the addition column by column.

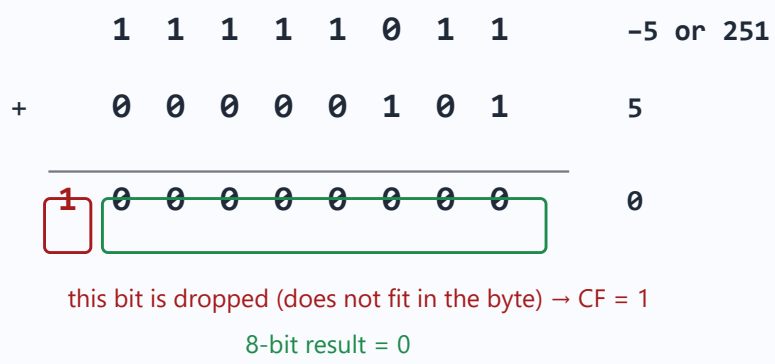


Figure 1.2.25: 1111 1011 (−5 or 251) + 0000 0101 (5) = 1 0000 0000. The ninth bit does not fit in the byte and falls into CF; the 8-bit result is 0.

This addition is an example of how the same adder circuit gives the correct result for both interpretations. In the unsigned interpretation the result of 251 + 5 = 256 did not fit in 8 bits; CF = 1 reports this. In the signed interpretation −5 + 5 = 0 is correct and OF = 0. This is why two's complement was chosen: no separate circuit is needed for negative numbers, and the adder works without knowing the meaning of the patterns. The mathematical reason is that an n-bit addition is really done modulo 2^n. In this modulus $2^n - x$ and $-x$ are in the same class.

1.2.4.8 Signed ranges and sign extension

Table 1.2.11 gives the unsigned and signed ranges of 8-, 16-, and 32-bit numbers. At every width half of the patterns are negative.

Width	Number of patterns	Unsigned	Signed	Negative patterns
8 bits	256	0 … 255	−128 … +127	80h … FFh
16 bits	65,536	0 … 65,535	−32,768 … +32,767	8000h … FFFFh
32 bits	4,294,967,296	0 … 4,294,967,295	−2,147,483,648 … +2,147,483,647	80000000h … FFFFFFFFh

Table 1.2.11: Unsigned and signed (two's complement) ranges.

The range is not symmetric: the negative side has one more value than the positive side. The reason is that zero sits on the positive side. In 8 bits there are 128 negative and 128 zero-or-positive patterns; the positives are 1 … 127 and the negatives are −1 … −128. This asymmetry has an interesting consequence: the positive counterpart of −128 does not fit in 8 bits. If the two's complement of 80h is taken with `NEG`, 0111 1111 + 1 = 1000 0000 = 80h comes out; the result does not change and the 8086 reports the overflow with OF = 1.

Definition 1.2.12

Sign extension carries a signed value into a wider register without changing its value: the old sign bit is copied into all of the newly added upper bits. **Zero extension**, by contrast, sets the upper bits to 0 and is correct for unsigned values. On the 8086, `CBW` sign-extends AL into AX and `CWD` sign-extends AX into DX:AX.

Why extension is done with the sign bit can be seen from the two's complement formula. −5 is 256 − 5 = FBh in 8 bits and 65,536 − 5 = FFFBh in 16 bits. The only difference between the two patterns is that the upper byte is FFh. For a positive number the upper byte is 00h: +5 is 05h and 0005h. In both cases the upper bits are copies of the sign bit. If FBh were zero-extended, 00FBh = +251 would result, and the value −5 would be lost.

Example 1.2.12

`CBW` executes with AL = 9Ch (−100). The MSB of AL is 1, so AH = FFh and AX = FF9Ch results. Check: 65,536 − 100 = 65,436 = FF9Ch. Had AL been 64h (+100), the MSB would be 0, so AH = 00h and AX = 0064h. `CWD` does the same job one width up: if AX = FF9Ch, DX = FFFFh.

1.2.4.9 Fixed-point signed fractions

Floating-point hardware is not always available for fractional numbers. The 8086 has no floating-point instruction; many small microcontrollers do not either. In such systems fractions are kept in integer registers as **fixed-point** numbers. The position of the point is decided in advance and is stored nowhere.

Definition 1.2.13

A **fixed-point number** is a binary number whose radix point is at a predetermined position. A number in **Q*m.f*** format consists of m integer bits, including the sign bit, and f fraction bits. Reading the pattern as an integer and dividing by 2^f gives the value of the number.

Consider the 8-bit Q5.3 format: the last three bits are the fraction. The place weights from left to right are −16, 8, 4, 2, 1, ½, ¼, ⅛. The weight of the leftmost bit is negative, as in two's complement. The representable values are between −16 and +15.875, and the difference between two consecutive values is 1/8 = 0.125.

Example 1.2.13

Let us write −6.75 in Q5.3 format. First +6.75 is written: 6.75 = 4 + 2 + ½ + ¼, that is, 00110.110b = 36h. For the negative, the two's complement is taken, ignoring the point. Inverting 0011 0110 gives 1100 1001; adding 1 gives 1100 1010 = **CAh**. With the point put back, 11001.010b is read.

The check is done with the weights: −16 + 8 + 1 + ¼ = −6.75. There is a shorter way too: multiply the number by $2^3 = 8$ and treat it as an integer. −6.75 × 8 = −54 and 256 − 54 = 202 = CAh.

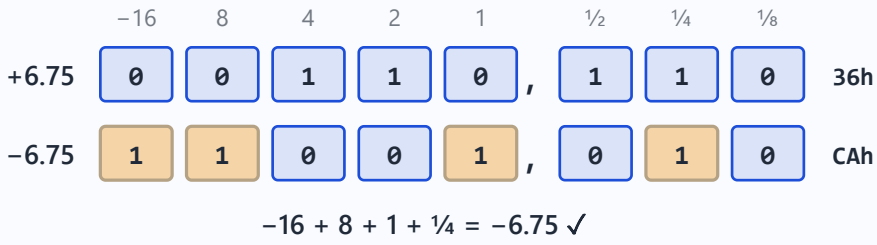


Figure 1.2.26: In Q5.3 format, +6.75 = 36h and −6.75 = CAh. The weight of the sign bit is −16: −16 + 8 + 1 + ¼ = −6.75.

The main advantage of fixed-point numbers is that addition is ordinary integer addition. To the processor CAh is an ordinary byte; the position of the point is only the programmer's convention. In the operation −6.75 + 2.5, 2.5 × 8 = 20 = 14h. CAh + 14h = DEh. DEh = 11011.110b, and by the weights −16 + 8 + 2 + 1 + ½ + ¼ = −4.25; this is the expected result. No extra hardware is needed for addition and subtraction. In multiplication, however, the number of fraction bits of the result doubles, and the result is rescaled by shifting right *f* bits.

The bit explorer in the slides shows the unsigned, signed, BCD, and Gray-code interpretations of an 8-bit pattern together. Watching how the four interpretations change when one of the bits is changed reinforces the distinction between pattern and interpretation.

1.2.5 Binary and hexadecimal arithmetic

The arithmetic logic unit (ALU) of a microprocessor does addition, subtraction, and bitwise operations on binary numbers. The programmer, however, often reads the results in hexadecimal. This section explains how to add and subtract by hand in the two bases, why subtraction reduces to addition, and how the CF and OF flags are set after an operation.

1.2.5.1 Binary addition and the full adder

Binary addition is done the same way as decimal addition: the numbers are written one under the other, you start at the lowest digit, and the columns are added in turn. The difference is that when a column sum reaches 2 it turns into a carry. For two digits in one column there are four cases: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10b. In the last case 0 is written in the column and a **carry** of 1 is passed to the next column. When the carry from the previous column is added too, the sum can be 3 (11b); in that case 1 is written in the column and a carry of 1 again comes out.

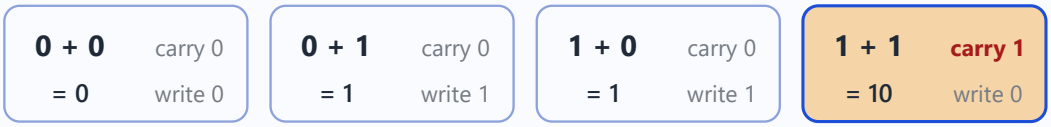


Figure 1.2.27: The four cases in adding two bits. A carry arises only in 1 + 1: 0 is written in the column and a carry of 1 comes out.

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.2.12: Truth table of the full adder. a and b are the bits being added, c_{in} is the carry in, s is the sum bit, and c_{out} is the carry out.

Table 1.2.12 gives all the cases of a single column. This table is the truth table of the **full adder** circuit known from digital design. The sum bit is the XOR of the three inputs: $s = a \oplus b \oplus c_{in}$. The carry out is 1 when at least two of the inputs are 1: $c_{out} = ab + c_{in}(a \oplus b)$. If eight full adders are chained, with the carry out of each connected to the carry in of the next, an 8-bit adder results. This structure is called a **ripple-carry adder**. The carry out of the last full adder is the bit written into the CF flag on the 8086.

The ripple-carry adder has a weakness: the result of the top bit has to wait for the carry to pass through all the gates from bottom to top. In a 16-bit addition the delay is about 16 times that of a single full adder. Fast processors therefore use circuits that compute the carry in advance (carry-lookahead). The logic is the same; only the carries are computed in parallel.

Example 1.2.14

Let us add 110001101b (397) and 100001011b (267). The columns are numbered from right to left.

1. Digit 0: 1 + 1 = 2 → write 0, carry 1.
2. Digit 1: 0 + 1 + 1 (carry) = 2 → write 0, carry 1.
3. Digit 2: 1 + 0 + 1 (carry) = 2 → write 0, carry 1.

4. Digit 3: 1 + 1 + 1 (carry) = 3 → write 1, carry 1.
5. Digit 4: 0 + 0 + 1 (carry) = 1 → write 1, no carry.
6. Digit 5: 0 + 0 = 0 → write 0.
7. Digit 6: 0 + 0 = 0 → write 0.
8. Digit 7: 1 + 0 = 1 → write 1.
9. Digit 8: 1 + 1 = 2 → write 0, carry 1.
10. Digit 9: only the carry 1 → write 1.

The result is **1010011000b**. Check: 512 + 128 + 16 + 8 = 664 = 397 + 267. The numbers added were 9 bits and the result is 10 bits. In a 9-bit adder the topmost 1 does not fit and goes out as a carry.

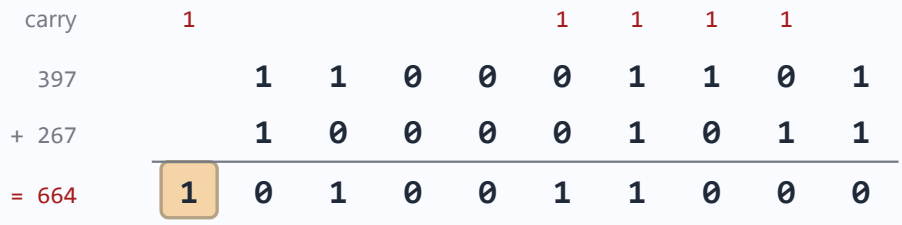


Figure 1.2.28: 110001101b + 100001011b = 1010011000b (397 + 267 = 664). The top row shows the carry coming into each column.

1.2.5.2 Binary subtraction: adding the complement

Subtraction can also be done column by column by borrowing. Processors, however, usually do not use a separate subtractor circuit. Subtraction is turned into addition: $A - B = A + (-B)$. −B is the two's complement of B, and the two's complement is found by inverting the bits and adding 1. Inverting the bits is done with one XOR gate per bit, and adding 1 is done by setting the carry in of the lowest full adder to 1. In this way the same adder both adds and subtracts.

Example 1.2.15

Let us compute 1011 − 0110 (11 − 6) in 4 bits. A = 1011, B = 0110. The bits of B are inverted: 1001. Adding 1 gives 1010; this is −6 in 4 bits. It is added to A: 1011 + 1010 = 10101. The result is 5 bits. The leftmost 1, which does not fit in 4 bits, is dropped and **0101** = 5 remains. Check: 11 − 6 = 5.

A =		1	0	1	1	11
B =		0	1	1	0	6
NOT B =		1	0	0	1	one's complement
+1 →		1	0	1	0	−B (two's compl.)
A + (−B) =	~~1~~	0	1	0	1	5 bits: the carry on the left is dropped
result =		0	1	0	1	5 ✓

Figure 1.2.29: Doing 1011 − 0110 with two's complement: NOT B = 1001, −B = 1010, A + (−B) = 1 0101; when the carry on the left is dropped, 0101 = 5.

In this example the carry that was dropped is 1 and the result is correct. What would happen if A < B? For 0110 − 1011 (6 − 11), the two's complement of 1011 is 0101. 0110 + 0101 = 1011, and no carry comes out. 1011 is −5 in the signed interpretation; the correct result is also −5. The absence of a carry shows that in the unsigned interpretation the subtraction needed a borrow, that is, A is smaller than B.

The 8086 stores this relationship inverted in the CF flag. After `SUB` and `CMP`, CF shows whether a **borrow** was taken in the unsigned interpretation. If a carry came out in addition by complement, there is no borrow and CF = 0. If no carry came out, there is a borrow and CF = 1. Thanks to this rule, CF = 1 after `CMP AX, BX` means AX < BX in the unsigned interpretation. `JB` ("jump if below") looks at exactly this flag.

1.2.5.3 Hexadecimal addition

Hexadecimal numbers are also added column by column. The values of the digits are thought of in decimal (A = 10, …, F = 15). If a column sum is less than 16 it is written directly. If it is 16 or more, 16 is subtracted from the sum, the remainder is written in the column, and a carry of 1 is passed to the left. The rule is the same as in decimal addition; only the threshold is 16 instead of 10.

Example 1.2.16

Let us find the sum 9A7Ch + 5B96h.

1. C + 6 = 12 + 6 = 18 ≥ 16 → 18 − 16 = 2, write 2, carry 1.
2. 7 + 9 + 1 = 17 ≥ 16 → 17 − 16 = 1, write 1, carry 1.
3. A + B + 1 = 10 + 11 + 1 = 22 ≥ 16 → 22 − 16 = 6, write 6, carry 1.
4. 9 + 5 + 1 = 15 < 16 → write F, no carry.

The result is **F612h**. Check in decimal: 9A7Ch = 39,548, 5B96h = 23,446, and 39,548 + 23,446 = 62,994 = F612h. On the 8086, with AX = 9A7Ch and BX = 5B96h, `ADD AX, BX` gives the same result. Since no carry comes out of the leftmost column, CF = 0.

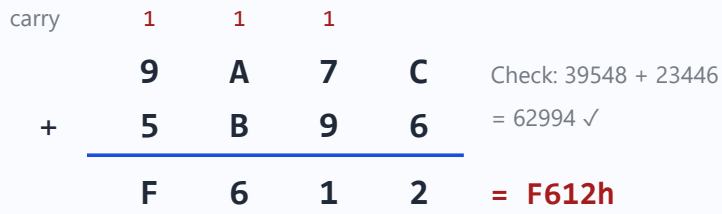


Figure 1.2.30: 9A7Ch + 5B96h = F612h. A carry comes out of three columns and not out of the leftmost one.

There is one more detail in this addition. In the signed interpretation 9A7Ch is negative (MSB = 1) and 5B96h is positive. The sum of two numbers of different signs never produces signed overflow; so OF = 0. F612h is −2542 in the signed interpretation: −25,988 + 23,446 = −2542.

1.2.5.4 Hexadecimal subtraction

In hexadecimal subtraction, if the upper digit is smaller than the lower one, 1 is **borrowed** from the column on the left. The borrowed 1 is worth 16 in the column on the right. The digit of the lending column decreases by 1.

Example 1.2.17

Let us find the difference 9A7Ch − 5B96h.

1. C − 6 = 12 − 6 = 6; no borrow.
2. 7 − 9 is not possible: borrow from the left. 7 + 16 − 9 = 14 = E; the A on the left drops to 9.
3. 9 − B is not possible: borrow again. 9 + 16 − 11 = 14 = E; the 9 on the left drops to 8.
4. 8 − 5 = 3; no borrow.

The result is **3EE6h**. Check: 39,548 − 23,446 = 16,102 = 3EE6h. Since no borrow is needed in the leftmost column, CF = 0 after `SUB AX, BX`.

Figure 1.2.31: 9A7Ch − 5B96h = 3EE6h. The "−1" marks show the columns that lend.

1.2.5.5 15's and 16's complement

The hexadecimal counterparts of one's and two's complement in binary are the 15's and 16's complement. They turn hexadecimal subtraction into addition, just as in binary.

Definition 1.2.14

The **15's complement** of a k-digit hexadecimal number is found by subtracting each digit from F and equals $16^k - 1 - x$. The **16's complement** is found by adding 1 to the 15's complement and equals $16^k - x$. Since a 4-digit hexadecimal number is 16 bits, the 16's complement is the same pattern as the 16-bit two's complement.

Example 1.2.18

The 15's complement of 5B96h: F − 5 = A, F − B = 4, F − 9 = 6, F − 6 = 9 → A469h. Its 16's complement is A469h + 1 = A46Ah; this equals 10000h − 5B96h. Now 9A7Ch − 5B96h is computed as 9A7Ch + A46Ah: the sum is 13EE6h. The 1 beyond four digits is dropped and **3EE6h** remains. The same result as direct subtraction has been found.

Let us compute 5B96h − 9A7Ch in the opposite direction. The 16's complement of 9A7Ch is 10000h − 9A7Ch = 6584h. 5B96h + 6584h = C11Ah, and no carry beyond four digits **comes out**. The absence of a carry shows that the result is negative. To find the magnitude, the 16's complement of C11Ah is taken: 10000h − C11Ah = 3EE6h. The result is −3EE6h, that is, −16,102.

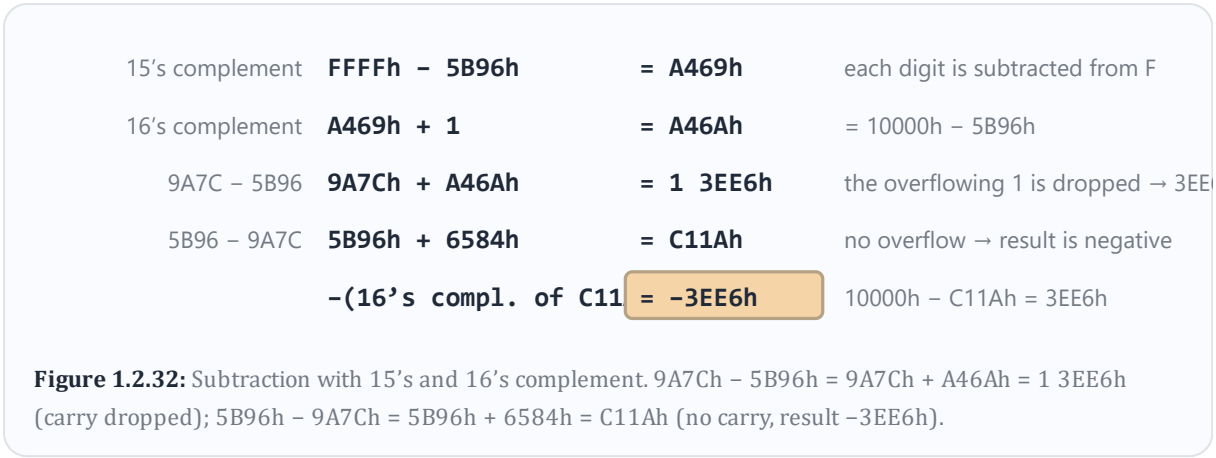


Figure 1.2.32: Subtraction with 15's and 16's complement. 9A7Ch − 5B96h = 9A7Ch + A46Ah = 1 3EE6h (carry dropped); 5B96h − 9A7Ch = 5B96h + 6584h = C11Ah (no carry, result −3EE6h).

On the 8086, with AX = 5B96h and BX = 9A7Ch, `SUB AX, BX` gives AX = C11Ah directly. Because a borrow occurred, CF = 1. C11Ah is −16,102 in the signed interpretation; if the programmer reads the result as signed, no extra step is needed. If it is read as unsigned, CF = 1 says the result is meaningless, because a larger number was subtracted from a smaller one.

1.2.5.6 Carry and signed overflow

In a fixed-width register the result of any operation may not fit. Not fitting happens in two different ways, and the 8086 reports them with two separate flags. Confusing these two flags is one of the most common mistakes in assembly programming.

Definition 1.2.15

The **carry flag** (CF) shows that in the unsigned interpretation the result did not fit in the destination: in addition a carry came out of the top bit, in subtraction a borrow was taken. The **overflow flag** (OF) shows that in the signed interpretation the result did not fit in the destination's range. The hardware sets OF as follows: if the carry *into* the sign bit and the carry *out of* the sign bit differ, OF = 1.

The sign of signed overflow is that the sign of the result comes out opposite to what is expected. If the sum of two positive numbers comes out negative, or the sum of two negative numbers comes out positive, overflow has occurred. The sum of two numbers of different signs, however, never overflows: the result always lies between the two numbers and so within the range. In subtraction the situation is reversed: since $A - B = A + (-B)$, the difference of numbers with different signs can overflow, and the difference of numbers with the same sign cannot. Table 1.2.13 gives the six cases with 8-bit examples.

Operation	Signs	Example	Signed interpretation	OF	CF
A + B	+ and +	`50h + 40h = 90h`	80 + 64 → −112 (wrong)	1	0
A + B	− and −	`9Ch + CEh = 6Ah`	−100 + (−50) → +106 (wrong)	1	1

Operation	Signs	Example	Signed interpretation	OF	CF
A + B	different	`64h + CEh = 32h`	100 + (−50) = 50	0	1
A − B	+ minus −	`64h - CEh = 96h`	100 − (−50) → −106 (wrong)	1	1
A − B	− minus +	`9Ch - 32h = 6Ah`	−100 − 50 → +106 (wrong)	1	0
A − B	same	`9Ch - CEh = CEh`	−100 − (−50) = −50	0	1

Table 1.2.13: The signed overflow rule in 8 bits. The CF column shows the unsigned interpretation of the same operation.

The CF column of the table shows that the two flags are independent of each other. In the first row CF = 0 but OF = 1: 80 + 64 = 144 fits in 255 in the unsigned interpretation and exceeds 127 in the signed one. In the third row CF = 1 but OF = 0: in the unsigned interpretation 100 + 206 = 306 does not fit in a byte, and in the signed interpretation 100 − 50 = 50 is correct. Which flag is meaningful depends on how the programmer interprets the numbers. A program working with unsigned numbers looks at CF, and one working with signed numbers looks at OF.

Example 1.2.19

Let us apply the hardware rule to 50h + 40h. When 0101 0000 + 0100 0000 is added, 1 + 1 = 10b occurs at bit 6: a carry of 1 goes from bit 6 into bit 7 (the sign bit). At bit 7, 0 + 0 + 1 = 1; no carry comes out of the sign bit. The carry in is 1 and the carry out is 0; since they differ, OF = 1. Because the carry out is 0, CF = 0. The result is 1001 0000 = 90h.

For 9Ch + CEh, 1001 1100 + 1100 1110 is added. At bit 6, 0 + 1 + (0 from bit 5) = 1; no carry goes into the sign bit. At bit 7, 1 + 1 = 10b; a carry comes out. Because the carry in is 0 and the carry out is 1, OF = 1, and because the carry out is 1, CF = 1. The result is 0110 1010 = 6Ah.

The logic of the rule can be explained like this. The carry out of the sign bit shows unsigned overflow. The carry into the sign bit shows that the sum of the lower seven bits exceeded 127. With two positive numbers there is no carry out; if there is a carry in, the result has corrupted the sign bit. With two negative numbers there is always a carry out; if there is no carry in, the result has dropped the sign bit to 0. If the carry in and carry out are equal, the sign bit has stayed correct.

When numbers of different widths are to be added, the shorter one must first be sign-extended. Suppose FBh (−5) is to be added to 0100h (256). Sign-extending FBh gives FFFBh: FFFBh + 0100h = 1 00FBh, and the 16-bit result is 00FBh = 251. The correct result is also −5 + 256 = 251. Had FBh been zero-extended, 00FBh + 0100h = 01FBh = 507 would have come out, which is wrong. On the 8086 `CBW` does this job in one instruction.

Common mistake

Reading CF = 1 in every case as "the result is wrong." When working with signed numbers CF is meaningless: in −5 + 5, CF = 1 but the result (0) is correct. Conversely, when working with unsigned numbers OF is meaningless: in 80 + 64, OF = 1 but the unsigned result (144) is correct. A flag is read only according to the interpretation the programmer has chosen.

1.2.5.7 Constant expressions in the assembler

The emu8086 assembler accepts a set of operators in constant expressions. These operators make programs easier to write: a mask or an address difference can be written as an expression instead of being computed by hand. Table 1.2.14 lists the supported operators with examples.

Operator	Meaning	Example → value	Run-time counterpart
`~`	inverts all the bits	`~0Fh` → F0h (8 bits)	`NOT`
`*`	multiplication	`3*4` → 12	`MUL`
`/`	integer division	`17/5` → 3	`DIV` (quotient)
`%`	remainder	`17%5` → 2	`DIV` (remainder)
`+`	addition	`40h+2` → 42h	`ADD`
`-`	subtraction, unary minus	`-5` → FBh	`SUB` , `NEG`
`<<`	shift left	`1<<4` → 16	`SHL`
`>>`	shift right	`80h>>3` → 10h	`SHR`
`&`	bitwise AND	`0F0h&3Ch` → 30h	`AND`
`^`	bitwise XOR	`0Fh^05h` → 0Ah	`XOR`
`|`	bitwise OR	`0F0h|0Fh` → FFh	`OR`

Table 1.2.14: Operators emu8086 supports in constant expressions.

These expressions are evaluated at **assembly time**. When `MOV AX, 3*4+1` is written, the assembler computes 13 and puts the instruction `MOV AX, 13` into the machine code. The processor does no multiplication while running. That is why expressions work only with constants: `MOV AX, BX*2` is invalid, because the value of BX is not known at assembly time. At run time the same operations are done with the instructions in the last column of the table. To multiply BX by 2, for example, `SHL BX, 1` is used.

1.2.6 The bit in hardware

So far bits have been treated as abstract 0s and 1s. In a microprocessor, however, every bit is a physical quantity: the voltage on a line or the stable state of a circuit. This section explains how voltage levels correspond to bits and how bits are stored in flip-flops. The topics are familiar from

digital design; here their relation to the 8086's registers and clock signal is emphasized.

1.2.6.1 Binary signals and voltage levels

The instructions and data of a program sit in memory as groups of digits. In circuits these digits exist as voltage and current. Digital computers use two kinds of signal: the **high** level represents 1 and the **low** level represents 0. In a circuit powered by 5 V like the 8086, the high level is close to 5 V and the low level close to 0 V.

The voltage does not have to be exactly 5 V or exactly 0 V. Each logic family defines threshold ranges for an input. In the TTL (transistor-transistor logic) family an input of 2.0 V or above is taken as 1, and an input of 0.8 V or below as 0. The region in between is undefined, and in a stably working circuit a signal is there only during a transition. On the output side the limits are tighter: a TTL output gives at least 2.4 V for 1 and at most 0.4 V for 0. The 0.4 V margin between output and input thresholds is the **noise margin**. Even if this much noise is added to the signal, the receiver reads the bit value correctly. This is why digital circuits are robust against noise. In an analog circuit every bit of noise added to the signal corrupts the value; in a digital circuit noise that does not cross the threshold leaves no trace.

The 8086's address and data buses are made of lines like these. The 20 address lines carry 20 bits at the same time, and the 16 data lines carry 16 bits at the same time. When a `MOV` instruction reads a word from memory, the memory chip pulls each of the 16 data lines to a high or low level. The processor samples these levels at a specific moment of the clock and takes them into its register.

1.2.6.2 The flip-flop

To hold a bit until the next operation, a circuit with memory is needed. A combinational circuit built from gates changes its output when its inputs change; it does not remember the past. Circuits with feedback do remember.

Definition 1.2.16

A **flip-flop** is the basic memory circuit that stores one bit and changes its state in step with the clock signal. It has two outputs: the stored bit Q and its complement Q'. The value at its input is loaded into the circuit at a specific moment of the clock signal and is held until the next such moment.

Figure 1.2.33 shows an SR flip-flop with a clock input. The heart of the circuit is two cross-coupled NOR gates. The output of each gate is fed back to the input of the other. This feedback creates two stable states: Q = 1, Q′ = 0 or Q = 0, Q′ = 1. While the inputs are 0 the circuit stays in the state it is in; it "remembers" the bit. The S and R inputs pass through AND gates with the clock signal. Thus the inputs take effect only while the clock is 1.

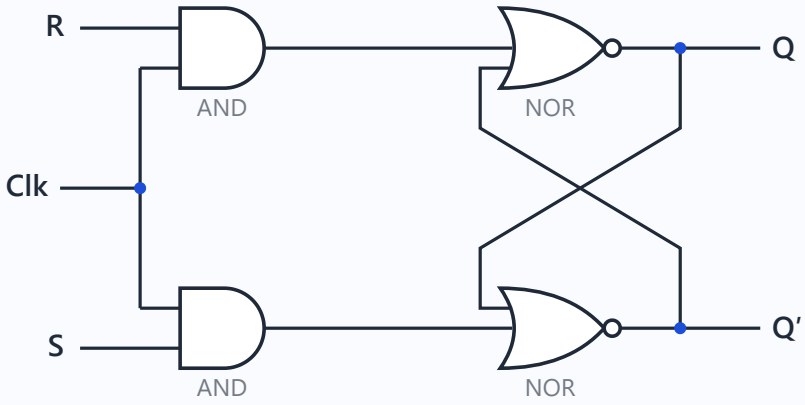


Figure 1.2.33: An SR flip-flop with a clock input. The R and S inputs pass through AND gates with Clk; two cross-coupled NOR gates produce the Q and Q′ outputs.

1.2.6.3 Types of flip-flops

In practice four types of flip-flops are used. All store one bit; the meaning of their inputs differs. Table 1.2.15 summarizes the behavior of the four types.

Type	Inputs	Behavior	Use
SR (set-reset)	S, R	S = 1 → Q = 1; R = 1 → Q = 0; S = R = 0 → no change	S = R = 1 is forbidden
D (data)	D	Q ← D	register and memory cell
JK	J, K	J = 1 → Q = 1; K = 1 → Q = 0; J = K = 0 → no change; J = K = 1 → Q toggles	has no forbidden state
T (toggle)	T	T = 1 → Q toggles; T = 0 → no change	counter, frequency divider

Table 1.2.15: Flip-flop types and their behavior at the clock pulse.

In an SR flip-flop the input S = R = 1 is forbidden. In that case the outputs of both NOR gates become 0, and Q and Q′ stop being each other's complement. If the inputs return to 0 at the same time, which state the circuit falls into depends on the gate delays and is unpredictable. The JK flip-flop turns this case into something useful: when J = K = 1, the output toggles at every clock pulse. This toggling is what makes the JK flexible. When the J and K inputs are tied together into a single T input, a T flip-flop results. The output of a T flip-flop held at T = 1 produces a full period every two clock pulses; that is, it divides the clock frequency by two. Chained T flip-flops form a binary counter.

Example 1.2.20

In a JK flip-flop Q = 0 initially, and three rising clock edges arrive while J = K = 1 is held. At every edge the output toggles: 0 → 1 → 0 → 1. After three edges Q = 1. Had the number of edges been even, Q would be 0 again.

1.2.6.4 The D flip-flop and the register

The type most used in microprocessors is the D flip-flop. It has a single data input, and at the rising edge of the clock the value on D is copied to Q. Figure 1.2.34 shows this behavior on a timing diagram.

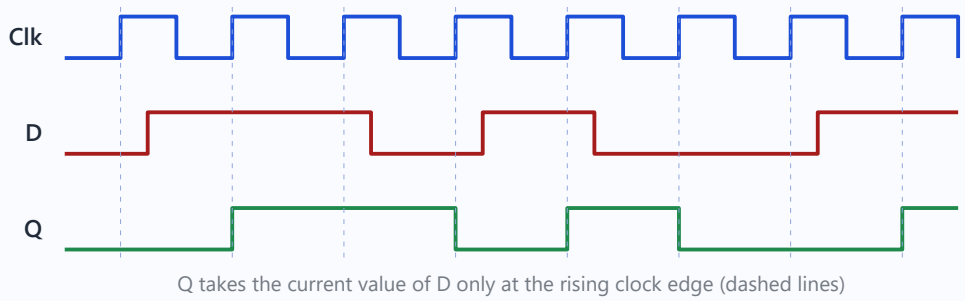


Figure 1.2.34: D flip-flop timing. Q takes the current value of D only at the rising clock edge (dashed lines); changes on D between edges do not affect Q.

In the diagram the D input changes several times between clock edges, but Q is unaffected by these changes. D is sampled only at the moment of the rising edge, and the value is held in Q until the next edge. This is a 1-bit memory. If 16 D flip-flops are placed side by side and driven by a common clock and a load signal, a 16-bit register results. The 8086's registers such as AX and BX are built this way in principle. A `MOV AX, BX` instruction is the transfer of BX's outputs over the internal data bus to AX's D inputs and their loading into AX at a clock edge. Each flag in the flags register is also a flip-flop.

The clock signal determines when all these loads happen. All internal operations of the 8086 are divided into clock cycles. With a 5 MHz clock one **clock cycle** takes 1 / 5,000,000 s = 200 ns. How many cycles an instruction takes depends on how many internal steps it consists of. A register-to-register `MOV` takes 2 cycles and a 16-bit `DIV` takes 144–162 cycles. A memory read requires a bus cycle of at least 4 cycles (T1–T4).

1.2.7 Floating-point numbers: IEEE 754

Because the position of the point is fixed, fixed-point representation can hold either very large or very small numbers, but not both together. In Q5.3 format the smallest step is 0.125 and the largest value is below 16. Scientific computation, however, needs both large numbers like 6×10^{23} and small ones

like 1.6×10^{-19}. When the idea of scientific notation is carried over to binary, the floating-point format results: a number is written as the product of a fraction and a power of 2, and the point "floats" according to the exponent.

1.2.7.1 The single-precision format

Definition 1.2.17

An **IEEE 754 single-precision floating-point number** is 32 bits and consists of three fields: a 1-bit **sign** s (bit 31), an 8-bit **exponent** e (bits 30–23), and a 23-bit **fraction** (mantissa) f (bits 22–0). The exponent is stored with a bias of 127. The value of a normalized number is

$$\text{value} = (-1)^s \times 1.f \times 2^{e-127}$$

Here the 1 is the **hidden bit**, which is not stored.

Figure 1.2.35: The fields of the IEEE 754 single-precision format: bit 31 is the sign, bits 30–23 the exponent, bits 22–0 the fraction. The pattern shown is 3E200000h.

Three details of the format need explanation. First, the sign is a separate bit: floating-point numbers are stored with sign-magnitude logic, not two's complement. To negate a number only bit 31 is inverted. Second, the exponent is stored **biased**: 127 is added to the true exponent. Exponents from −126 to +127 are thus written as unsigned numbers from 1 to 254. The benefit of the bias is that the patterns of two positive floating-point numbers can be compared like unsigned integers. Third, a normalized binary number always starts with 1: $1.\text{xxx} \times 2^k$. Because the leading 1 is always the same, it is not stored, and one more bit fits in the fraction field. This is why the 23-bit field gives 24 bits of precision.

The 8086 has no floating-point instruction. Floating-point operations are done either with the 8087 math coprocessor or with software routines written with integer instructions. The 8087 is a separate chip attached next to the 8086 that works on numbers in this format with its own register stack. The 64-bit double-precision format applies the same idea with 1 + 11 + 52 bits; its exponent bias is 1023.

1.2.7.2 Decoding

Example 1.2.21

Let us find the value of the pattern `0 01111100 01000000000000000000000`, that is, 3E200000h.

1. The sign bit is 0: the number is positive, the factor is (+1).
2. The exponent field is 0111 1100b = 124. The true exponent is 124 − 127 = −3: the factor is 2^{-3}.
3. The fraction field is 0100…0: $0 \times 2^{-1} + 1 \times 2^{-2} = 0.25$. With the hidden bit it is 1.25.
4. The value is $(+1) \times 1.25 \times 2^{-3} = 1.25 / 8 =$ **0.156 25**.

The hexadecimal form of the pattern is found by putting the fields together and grouping in fours: 0011 1110 0010 0000 … = 3E200000h.

1.2.7.3 Encoding

Example 1.2.22

Let us convert −6.5 to the 32-bit IEEE 754 format.

1. The number is negative: the sign bit is **1**.
2. The magnitude is converted to binary: 6 = 110b, 0.5 = 0.1b; 6.5 = 110.1b.
3. It is normalized: the point moves two places left, $110.1b = 1.101b \times 2^2$.
4. Exponent: 2 + 127 = 129 = **1000 0001b**.
5. Fraction: the hidden 1 is dropped, and the remaining 101 is padded on the right with 0s to 23 bits: **101**0 0000 0000 0000 0000 000.
6. The fields are combined: 1 10000001 10100000000000000000000 = 1100 0000 1101 0000 0000 … = **C0D00000h**.

Figure 1.2.36: The IEEE 754 pattern of −6.5: sign 1, exponent 1000 0001 (129), fraction 101 and zeros; C0D00000h.

In 8086 memory this pattern sits, starting from the lowest address, in the order 00 00 D0 C0; the lowest byte is written to the lowest address (little-endian order).

1.2.7.4 Special patterns

Two extreme values of the exponent field (0 and 255) are reserved for special meanings. Table 1.2.16 lists them.

Pattern	Meaning
e = 0, f = 0	±0 (+0 or −0 according to the sign bit)
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN, “not a number” (e.g. 0/0)
e = 0, f ≠ 0	denormalized, very small number: 0.f × 2^{-126}

Table 1.2.16: Special patterns of IEEE 754.

Zero cannot be written with the hidden-bit rule; 1.f is always at least 1. That is why a separate pattern is defined for zero. Having a separate sign bit gives rise to +0 and −0; the standard requires them to compare equal. The infinity and NaN values let a computation continue instead of stopping with an error. Denormalized numbers fill the gap in the region near zero. The smallest normalized number is 2^{-126}; values below it are stored without the hidden bit as 0.f × 2^{-126}.

The precision of floating-point numbers is limited. In single precision the 24-bit significand corresponds to about 7 decimal digits. A value like 0.1 that is not finite in binary is rounded to these 24 bits: in single precision the pattern 3DCCCCCDh stands for 0.100 000 001 49…. This small error accumulates when additions are done one after another.

1.2.8 Binary-coded decimal (BCD)

The binary system is efficient for computation, but people read decimal. To print a number on the screen it has to be divided by 10 each time to separate its digits; this needs a slow sequence of `DIV`s. In applications where numbers are displayed often and computed little, another encoding is used: each decimal digit is written separately in binary.

Definition 1.2.18

In **binary-coded decimal** (BCD) encoding each decimal digit is represented with four bits: 0 = 0000, …, 9 = 1001. The patterns 1010 … 1111 are invalid. **Packed** BCD puts two digits in a byte (25 → 25h); **unpacked** BCD puts one digit in each byte (25 → 02h 05h).

The difference between BCD and binary is seen in an example. 25 in BCD is 0010 0101: the left nibble shows 2 and the right one 5. The binary equivalent of the same number is 0001 1001 = 19h. In packed BCD the hexadecimal notation of a byte is the same as the digits of the decimal value: BCD 25 = 25h. This property makes BCD numbers very easy to read in a memory dump. The price is inefficient use of space: a byte holds 0 … 255 in binary but only 0 … 99 in packed BCD.

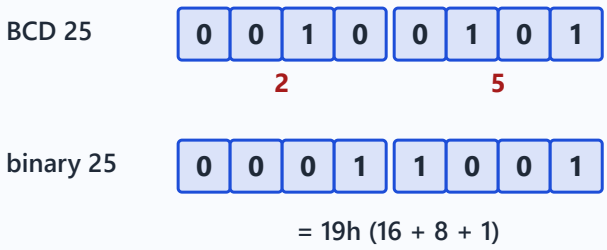


Figure 1.2.37: BCD and binary representation of 25. In BCD each nibble is a decimal digit (0010 0101); in binary the number is written as a whole (0001 1001 = 19h).

BCD is used where decimal digits are represented directly. The real-time clock (RTC) chip in computers keeps the hour, minute, and date in BCD; 14:35 is stored as the bytes 14h and 35h. Seven-segment displays are driven by decoders that take a BCD digit and light the corresponding segments. Calculators and financial software use BCD arithmetic so that decimal fractions such as 0.1 are stored exactly.

1.2.8.1 BCD addition and correction

If two packed BCD numbers are added with a binary adder, the result is not always valid BCD. When the sum of a nibble exceeds 9, the binary adder leaves it between 10 and 15; in the decimal system, however, a digit that reaches 10 should turn into a carry. This difference is fixed with a **correction**: 6 is added to the invalid digit. The number 6 is the difference between the 16 patterns of a nibble and the 10 values of a decimal digit. Adding 6 skips the six unused patterns (1010 … 1111) and passes the carry to the next nibble up.

Example 1.2.23

Let us add BCD 38 and BCD 45. 38 = 0011 1000, 45 = 0100 0101. The binary sum is 0111 1101 = 7Dh. The low nibble 1101 = 13 is invalid in BCD. 6 is added to this nibble: 7Dh + 06h = 83h = 1000 0011. The result is BCD **83**, and 38 + 45 = 83 is correct.

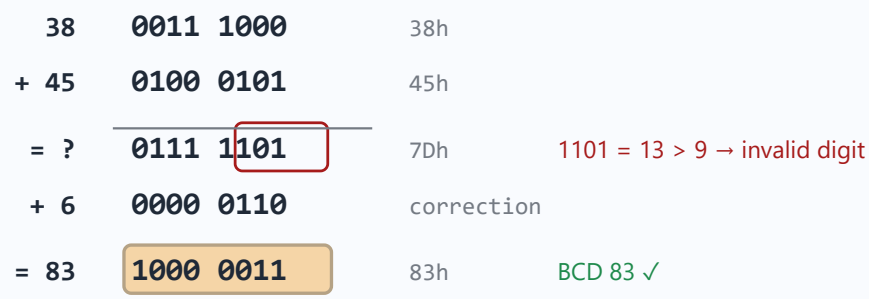


Figure 1.2.38: BCD 38 + 45. The low digit (13) of the binary sum 7Dh is invalid; adding 6 gives 83h, that is, BCD 83.

There is a second case that needs correction: the low nibble looks valid but has actually overflowed. In 29 + 18 the binary sum is 29h + 18h = 41h. The low nibble is 1 and looks valid; yet 9 + 8 = 17. A carry has come out of the low nibble into the next one up, taking the value 16 with it. The 8086 keeps this carry in the **auxiliary carry flag** (AF). If AF = 1, 6 is again added to the low nibble: 41h + 6 = 47h, that is, BCD 47 = 29 + 18. If the high digit exceeds 9 or produces a carry, 6 is added to it too, that is, 60h is added to the byte; in that case CF = 1 and the result has exceeded 99.

The 8086 applies all these rules with a single instruction: `DAA` (decimal adjust after addition). If `DAA` is run right after two packed BCD numbers are added with `ADD AL, BL`, AL is corrected to valid BCD. DAA looks at the low nibble of AL and AF, then the high nibble and CF, and adds 06h, 60h, or 66h. Its counterpart for subtraction is `DAS`. DAA works only on AL; multi-digit BCD numbers are processed byte by byte, added with carry (`ADC`) and applying DAA after each byte.

Common mistake

Looking only at “is the nibble greater than 9?” and forgetting AF. In 29h + 18h = 41h the low nibble is 1 and no correction seems needed; but the result should be 47, not 41. Another mistake is to read a BCD number like a hexadecimal number: 47h is 71 in hexadecimal but 47 in BCD. 2Fh is the binary equivalent of decimal 47 and is not BCD.

1.2.9 Byte order: little-endian and big-endian

Memory is addressed byte by byte. In what order the bytes of a 16- or 32-bit value are laid out when it is written to memory is a design decision. There are two opposite conventions.

Definition 1.2.19

In **little-endian** order the least significant byte of a multibyte value is written to the lowest address. In **big-endian** order the most significant byte is written to the lowest address. The 8086 and the whole x86 family are little-endian.

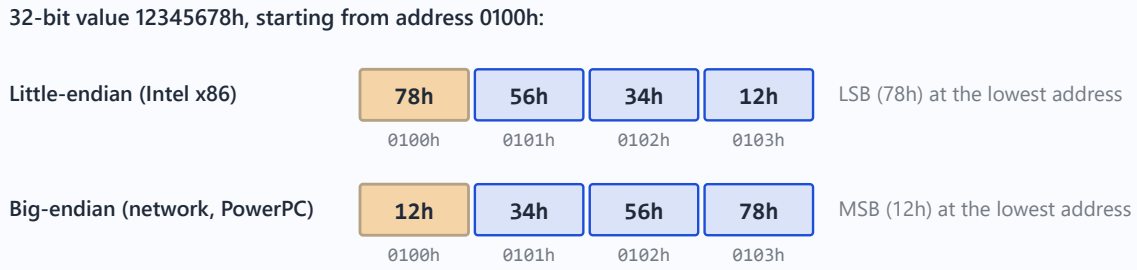


Figure 1.2.39: The layout of the value 12345678h starting at address 0100h in the two orders. Little-endian:

78 56 34 12; big-endian: 12 34 56 78.

Figure 1.2.39 shows the 32-bit 12345678h in the two orders. In little-endian, 78h is at address 0100h and 12h at address 0103h. When a memory dump is read from left to right, the value looks "reversed." In big-endian the bytes sit in the order they were written and the dump reads directly. Systems that use big-endian include the Motorola 68000, most PowerPC configurations, and network protocols. Internet protocols transmit multibyte fields in big-endian; this order is called **network byte order**. An x86 machine must swap the bytes of a 16-bit port number received from the network before using it.

Little-endian has one convenience: the address of a value points to its low byte regardless of the value's width. To read the low byte of a word at address 0100h, one still looks at address 0100h. On the 8086, the instruction `MOV AX, [0100h]` loads AL from 0100h and AH from 0101h. From the same address, `MOV AL, [0100h]` takes only the low byte and finds the low byte of the same value.

Example 1.2.24

Suppose the 32-bit A1B2C3D4h has been written to address DS:0300h with `DD 0A1B2C3D4h`. In little-endian order the bytes are 0300h: D4h, 0301h: C3h, 0302h: B2h, 0303h: A1h. Now consider the following three instructions. `MOV AX, [0302h]` takes AL from 0302h (B2h) and AH from 0303h (A1h): AX = A1B2h. `MOV BL, [0301h]` reads a single byte: BL = C3h. `MOV CX, [0301h]` takes CL from 0301h (C3h) and CH from 0302h (B2h): CX = B2C3h. The last read is a word read starting at an odd address; the 8086 does it with two separate bus cycles and spends four extra clock cycles.

Common mistake

Thinking little-endian stores the bits in reverse order. The ordering is at the byte level; the bits inside each byte do not change. 1234h is written to memory as 34 12, not as 43 21 or 2C 48. Another mistake is to expect 12h when reading a word defined with `DW 1234h` using `MOV AL, [address]`; the low address holds 34h.

1.2.10 The Gray code

In ordinary binary counting, more than one bit can change between two consecutive numbers. Going from 3 to 4, 011 → 100, three bits change at once. At the output of a rotary encoder or a counter these bits do not change at exactly the same moment physically. A value read during the transition may be an intermediate value that is neither 3 nor 4, such as 111 (7) or 000 (0). The Gray code solves this problem at the root.

Definition 1.2.20

The **Gray code** is an encoding in which only one bit changes between any two consecutive values. The Gray code of a binary number *B* is found with the formula *G = B XOR (B >> 1)*.

Decimal	Binary	Gray	Changed bit
0	`000`	`000`	—
1	`001`	`001`	1
2	`010`	`011`	2
3	`011`	`010`	1
4	`100`	`110`	3
5	`101`	`111`	1
6	`110`	`101`	2
7	`111`	`100`	1

Table 1.2.17: 3-bit binary and Gray codes. “Changed bit” is the position, counted from the right, of the bit that changes in the Gray code relative to the previous row.

Table 1.2.17 gives the 3-bit sequence: 000, 001, 011, 010, 110, 111, 101, 100. The transition from 3 to 4 is 010 → 110 in Gray code; only the top bit changes, and no wrong intermediate value can be read. The sequence also closes on itself: going from 7 (100) to 0 (000) also changes a single bit. This is why the Gray code is widely used in rotary encoders that measure the angle of a shaft. The same code also appears in some types of analog-to-digital converters and in the row-column ordering of Karnaugh maps. Single-bit changes make the transitions robust against electrical noise and timing differences.

1.2.10.1 From binary to Gray code and back

The meaning of the formula is this: the top bit of the Gray code is the same as in binary; each following bit is the XOR of that bit in binary and the bit to its left. *B >> 1* brings the bit to the left of each bit into its own position; the XOR compares the two bits. If a bit in the Gray code is 1, then that bit and the bit to its left differ in binary.

Example 1.2.25

Let us find the Gray code for B = 0110 (6). B is shifted right by one bit: B >> 1 = 0011. XOR is taken bit by bit: 0110 ⊕ 0011 = **0101**. The Gray code of 6 is 0101. Check by the rule: top bit 0; then 0 ⊕ 1 = 1, 1 ⊕ 1 = 0, 1 ⊕ 0 = 1 → 0101.

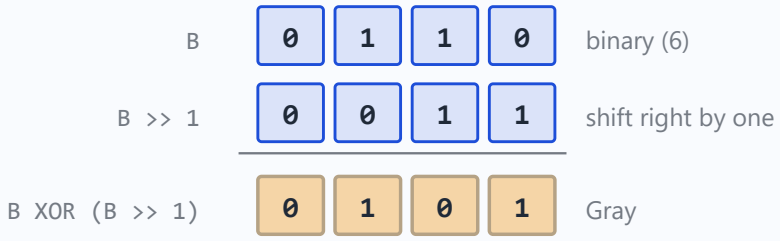


Figure 1.2.40: The Gray code of 6 = 0110: B >> 1 = 0011 and 0110 ⊕ 0011 = 0101.

Converting from Gray code back to binary goes the other way. The top bit of the binary equals the top bit of the Gray code. Each following bit is the XOR of the binary bit found just before it and the bit in the same position in the Gray code: $B_i = B_{i+1} \oplus G_i$. For 0101: $B_3 = 0$; $B_2 = 0 \oplus 1 = 1$; $B_1 = 1 \oplus 0 = 1$; $B_0 = 1 \oplus 1 = 0$. The result is 0110 = 6. The conversion back is done not in a single step like the forward direction but with a chain that proceeds bit by bit.

1.2.10.2 Code trace: Gray code with SHR and XOR

The formula is written on the 8086 with three instructions: copy the value, shift the copy right by one, and XOR the two values. The following program computes the Gray code of two consecutive numbers, 11 and 12.

```
ORG 100h
MOV AL, 1011b  ; 11
MOV BL, AL
SHR BL, 1      ; B >> 1
XOR BL, AL     ; G
MOV AL, 1100b  ; 12
MOV BH, AL
SHR BH, 1
XOR BH, AL     ; G
RET
```

`MOV AL, 1011b` loads 11 into AL. `MOV BL, AL` copies the value into BL; the shift will be done on the copy, and the value B is kept in AL. `SHR BL, 1` shifts BL right by one: 0000 1011 → 0000 0101. The rightmost 1 falls into CF and a 0 enters at the far left. `XOR BL, AL` compares the two values bit by bit: 0101 ⊕ 1011 = 1110. BL = 0Eh, the Gray code of 11. The XOR instruction always sets CF and OF to 0; it sets ZF and SF according to the result. The second half of the program does the same job for 12 on BH: 1100 → SHR → 0110 → XOR → 1010 = 0Ah.

Instruction	AL	BL	BH	Note
`MOV AL, 1011b`	0Bh	00h	00h	B = 11
`MOV BL, AL`	0Bh	0Bh	00h	copy

Instruction	AL	BL	BH	Note
`SHR BL, 1`	0Bh	05h	00h	0101b; dropped bit CF = 1
`XOR BL, AL`	0Bh	0Eh	00h	1011 ⊕ 0101 = 1110: Gray(11)
`MOV AL, 1100b`	0Ch	0Eh	00h	B = 12
`MOV BH, AL`	0Ch	0Eh	0Ch	copy
`SHR BH, 1`	0Ch	0Eh	06h	0110b; dropped bit CF = 0
`XOR BH, AL`	0Ch	0Eh	0Ah	1100 ⊕ 0110 = 1010: Gray(12)

Table 1.2.18: Trace of the Gray code program.

Figure 1.2.41: At the end of the program: Gray(11) = 1110 (BL), Gray(12) = 1010 (BH). The two codes differ only in bit 2.

The result confirms the basic property of the Gray code. In going from 11 to 12 three bits change in binary: 1011 → 1100. In Gray code only one bit changes: 1110 → 1010. In the program BL and BH are the two halves of the same BX register; when the program ends, BX = 0A0Eh. Because BH and BL can be addressed separately, the two 8-bit results sit side by side in a single 16-bit register.

1.2.11 Summary

- In a **positional system** the value of a number is the sum of each digit multiplied by its place weight, a power of the base: $N = \Sigma\, d_i \times b^i$. Roman numerals make arithmetic hard because they have no zero and no place value.
- Computers use the **binary** system because two-state elements are easy to build. n bits give 2^n patterns: a byte 0 … 255, a word 0 … 65,535.
- A **hexadecimal** digit is exactly one nibble; one moves between binary and hexadecimal by groups of four from the right. In assembly the suffixes `b`, `h`, and `o` indicate the base; a 0 is put in front of a hexadecimal number that starts with a letter. The base is only a way of writing: `65`, `41h`, and `'A'` produce the same code `B0 41`.

- The integer part is converted from decimal to another base with the **divide-and-remainder** method, and the remainders are read from bottom to top. The fractional part is converted with the **multiply-by-2** method, and the bits are read from top to bottom. Fractions whose denominator is not a power of 2 (0.1) are not finite in binary.
- `DIV BX` divides DX:AX, writes the quotient to AX and the remainder to DX; DX must be cleared beforehand. `SHR` divides by 2 and drops the remainder into CF; `RCR` carries CF into a register.
- A bit pattern carries no meaning; the program supplies the interpretation. In **two's complement**, $-x = NOT\ x + 1 = 2^n - x$; `NEG` does this in one instruction. The MSB is the sign bit; there is a single zero; in 8 bits the range is −128 … +127. **One's complement** (`NOT`) and sign-magnitude have two zeros.
- **Sign extension** copies the sign bit into the upper bits (`CBW` , `CWD`): −5 = FBh = FFFBh. In **fixed-point** numbers the position of the point is the programmer's convention; addition is ordinary integer addition.
- Binary addition is done with full adders; subtraction is turned into addition of the two's complement. In hexadecimal a carry arises when a column sum reaches 16; the 15's and 16's complement are the counterparts of one's and two's complement.
- **CF** shows that the result did not fit in the unsigned interpretation (carry or borrow), and **OF** shows that it did not fit in the signed interpretation. OF can be 1 only when numbers of the same sign are added or numbers of different signs are subtracted; the hardware compares the carry into and out of the sign bit.
- Bits in a circuit are voltage levels; threshold ranges provide a noise margin. A **flip-flop** stores one bit; in the SR type S = R = 1 is forbidden, and in the JK type J = K = 1 toggles. Registers are built from D flip-flops loaded at the rising clock edge.
- An **IEEE 754** single-precision number consists of 1 sign bit, 8 exponent bits (bias 127), and 23 fraction bits; the value is $(-1)^s \times 1.f \times 2^{e-127}$. The 8086 has no floating-point instruction; that job is left to the 8087 or to software.
- **BCD** holds each decimal digit in four bits; after an addition, 6 is added to an invalid digit or when AF = 1 (`DAA`). The 8086 is **little-endian**: the low byte is at the low address. In the **Gray code** consecutive values differ in a single bit: $G = B \oplus (B >> 1)$.

1.2.12 Key terms

positional number system base (radix) digit decimal binary hexadecimal octal suffix bit nibble byte word double word MSB / LSB signed / unsigned sign bit sign-magnitude one's complement two's complement sign extension fixed-point number carry borrow overflow carry flag (CF) overflow flag (OF) sign flag (SF) zero flag (ZF) auxiliary carry flag (AF) shift rotate flip-flop clock cycle floating-point number sign / exponent / fraction binary-coded decimal (BCD) byte order Gray code

1.2.13 Test Yourself

1. Write the number 300 in hexadecimal and binary. Does this value fit in AL?

   Answer

   300 ÷ 16 = 18, remainder 12 (C); 18 ÷ 16 = 1, remainder 2; 1 ÷ 16 = 0, remainder 1. The result is 12Ch; expanding the digits to 4 bits gives 1 0010 1100b. Because the value is greater than 255, it does not fit in the 8-bit AL; nine bits are needed. It fits in AX.

2. Convert the number 0.3125 to binary. Is the result finite? Why?

   Answer

   0.3125 × 2 = 0.625 → 0; 0.625 × 2 = 1.25 → 1; 0.25 × 2 = 0.5 → 0; 0.5 × 2 = 1.0 → 1. Reading the bits from top to bottom gives 0.0101b: 1/4 + 1/16 = 0.3125. The result is finite, because 0.3125 = 5/16 and the denominator is a power of 2.

3. What are the unsigned, two's complement, and one's complement interpretations of the pattern 1101 0110b?

   Answer

   Unsigned: 128 + 64 + 16 + 4 + 2 = 214. Two's complement: 214 − 256 = −42 (or with the weights −128 + 64 + 16 + 4 + 2 = −42). One's complement: inverting the bits gives 0010 1001 = 41; the interpretation is −41.

4. `CBW` executes with AL = 80h. What does AX become? What would have been found had the same value been zero-extended?

   Answer

   The MSB of AL is 1; CBW makes AH = FFh and AX = FF80h. This is −128 in 16 bits (65,536 − 128 = 65,408 = FF80h). Zero extension would have given 0080h = +128; the signed value would not have been preserved.

5. `ADD AL, 1` executes with AL = 7Fh. What are AL, CF, OF, SF, and ZF? Interpret the result as

unsigned and as signed.

Answer

0111 1111 + 1 = 1000 0000, that is, AL = 80h. No carry comes out of the top bit: CF = 0. A carry goes into the sign bit but none comes out: OF = 1. SF = 1, ZF = 0. In the unsigned interpretation 127 + 1 = 128 is correct; in the signed interpretation the result of +127 + 1 appears as −128, which is overflow. It is the 8-bit counterpart of the step from 0111 to 1000 on the number circle.

6. What value does F4h represent in Q5.3 format?

Answer

F4h = 11110.100b. By the weights, −16 + 8 + 4 + 2 + ½ = −1.5. Shortcut: F4h = 244, as signed 244 − 256 = −12; −12 / 8 = −1.5.

7. Write the number 0.75 in 32-bit IEEE 754 format.

Answer

0.75 = 0.11b = $1.1\text{b} \times 2^{-1}$. Sign 0; exponent −1 + 127 = 126 = 0111 1110b; fraction 1000…0. The pattern is 0 01111110 10000000000000000000000 = 0011 1111 0100 0000 … = 3F400000h.

8. AL = 66h and BL = 57h are packed BCD numbers. After `ADD AL, BL` and then `DAA`, what are AL and CF?

Answer

The binary sum is 66h + 57h = BDh. The low nibble D (13) is invalid: adding 6 gives C3h. The high nibble C (12) is also invalid: adding 60h gives 123h. AL = 23h, CF = 1. The result is BCD 123: 66 + 57 = 123; the hundreds digit is in CF.

9. `DW 1234h, 0ABCDh` has been defined starting at DS:0200h. What is AX after `MOV AX, [0201h]`?

Answer

In little-endian order the bytes are 0200h: 34h, 0201h: 12h, 0202h: CDh, 0203h: ABh. AL is loaded from 0201h (12h) and AH from 0202h (CDh): AX = CD12h. Because the read starts at an odd address, it takes two bus cycles.

10. What is the binary equivalent of the 4-bit number whose Gray code is 1001? Check the result with

the formula.

Answer

$B_3 = G_3 = 1$; $B_2 = 1 \oplus 0 = 1$; $B_1 = 1 \oplus 0 = 1$; $B_0 = 1 \oplus 1 = 0$. The binary equivalent is 1110 = 14. Check: $1110 \oplus 0111 = 1001$.

1.2.14 Further reading

- M. M. Mano and M. D. Ciletti, *Digital Design* (5th edition), Chapters 1, 4, and 5: number systems, complements, BCD and the Gray code, and adder circuits and flip-flops, treated in detail from the digital-design point of view.
- B. B. Brey, *The Intel Microprocessors* (8th edition), Chapter 1: number systems, signed numbers, BCD, and byte order are tied directly to the data formats of the 8086 family.
- D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, 23(1), 1991: the classic paper explaining IEEE 754's rounding, special values, and error accumulation.
- D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Section 4.1: gives the history of positional number systems and the methods of base conversion, with proofs.

1.2.15 References

1. M. Morris Mano, Michael D. Ciletti, *Digital Design*, 5th edition, Pearson, 2013.
2. Barry B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, 2009, Chapter 1.
3. IEEE Std 754-2019, *IEEE Standard for Floating-Point Arithmetic*.
4. emu8086 documentation, *Numbering systems tutorial*.
5. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
6. Şadi Çağatay Öztürk, Oğuz Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.
7. David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, 23(1):5–48, 1991.
8. Donald E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd edition, Addison-Wesley, 1997.

PART I · CHAPTER 2

8086 Architecture and Registers

The Intel 8086, introduced in 1978, is the 16-bit microprocessor that launched the x86 family. This chapter describes the system buses, the bus interface unit (BIU) and execution unit (EU), the prefetch queue, and how a 20-bit physical address is formed in segmented memory. It then studies the fourteen registers of the 8086 by group: the general-purpose registers AX, BX, CX and DX, the pointer and index registers SP, BP, SI and DI, the segment registers and IP. The flags register, with its status and control flags, and the instructions that change those flags complete the chapter.

NOTES IN THIS CHAPTER

2.1 8086 Architecture

2.2 8086 Registers

PART I · CHAPTER 2.1

8086 Architecture

Every line written in assembly language drives a concrete piece of hardware inside the processor: it loads a value into a register, makes the ALU perform an addition, or puts a 20-bit address on the address bus. That is why you need to know the inside of the 8086 before programming it. This chapter starts with the general features of the Intel 8086 and explains how the processor is connected to memory and to I/O devices over buses. It then examines two internal units, the bus interface unit (BIU) and the execution unit (EU), together with the prefetch queue and the pipeline. It shows the ALU's operations and the byte lengths of instructions in memory through code traces. The last sections cover the register set and the 8086's most distinctive feature, segmented memory: how a 20-bit physical address is produced from two 16-bit values, why the same byte can be named in many ways, and which access uses which segment.

LEARNING OUTCOMES

Explain the general features
State the data bus and address bus widths of the 8086, the amount of memory it can address, its place in the x86 family, and which chips make up an 8086 system.

Follow the buses
Separate the roles of the address, data, and control buses; show in code which values are on the buses during a memory or I/O read and write.

Describe the internal structure
Explain the roles of the BIU and EU, how the prefetch queue works, the gain from the pipeline, and the effect of a branch on the queue; trace the fetch and execution of an instruction on a block diagram.

Read machine code
Separate the opcode, ModR/M byte, displacement, and immediate fields of an instruction; find the value of IP at each step from the instruction lengths.

Group the registers
Divide the 14 registers into groups; explain the usage rules of the segment registers and the role of IP.

Compute physical addresses
Find the 20-bit physical address with `segment × 10h + offset`; normalize an address; show the wraparound at the 1 MB limit and the default segment rules with code traces.

2.1.1 Overview of the 8086

The Intel 8086 is a 16-bit microprocessor that came to market in 1978. It is the first member of the x86 processor family that runs in desktop and laptop computers today. Over more than forty years the processors of this family grew to 32 and 64 bits and their clock rates rose thousands of times. Even so, every new member kept running instructions written for the 8086. Learning the 8086's registers, address computation, and instruction format therefore means learning the lowest layer of today's processors.

This section first gives the 8086's basic numbers: how many bits of data it works with, how much memory it can address, and what technology it is made in. It then describes the 8086's place in the family and why the 20-bit address bus was chosen. Finally it shows that a microprocessor is not a computer by itself and what chips it needs around it.

2.1.1.1 Basic features

Calling a processor "16-bit" describes the width of the data it can work on at once. The 8086's general-purpose registers are 16 bits. The arithmetic logic unit (ALU) adds two 16-bit numbers in a single operation. The external data bus is also 16 lines wide, so the processor exchanges 2 bytes with memory at a time. The equality of these three widths makes the 8086 a "fully 16-bit" processor.

The address bus, on the other hand, has 20 lines. Since each line carries one bit, the processor can produce 2^{20} different addresses. 2^{20} = 1,048,576 bytes, that is, 1 MB. Addresses are written in hexadecimal with five digits: the lowest address is `00000h` and the highest is `FFFFFh`.

Feature	Value	Meaning
Data bus	16 bit	at most 2 bytes are carried in one bus cycle
Address bus	20 bits	2^{20} bytes = 1 MB of memory is addressed
Registers, ALU	16 bit	the word size is 16 bits
Package	40-pin DIP	dual in-line plastic or ceramic package
Clock	5–10 MHz	8086: 5 MHz, 8086-2: 8 MHz, 8086-1: 10 MHz
Transistors	about 29,000	HMOS (high-density n-channel MOS) technology
Supply	+5 V	a single supply voltage
Architecture	CISC	instructions 1–6 bytes long, instructions that operate on memory

Table 2.1.1: Basic features of the 8086.

Table 2.1.1 collects these numbers. The clock speed is the frequency of the square wave that synchronizes all the operations inside the processor. One period of a 5 MHz clock lasts 1 / 5,000,000 s = 200 ns. This time is called a **clock cycle**. On the 8086 the simplest instructions finish in a few clock cycles; complex instructions such as multiply and divide can take hundreds of cycles.

In a CISC (complex instruction set computer) design the instructions have different lengths, and an instruction can work directly with an operand in memory. For example, the `ADD AX, [BX]` instruction reads a word from memory and adds it to AX. This design aimed to keep programs short when memory was expensive: frequently used jobs were fitted into a single instruction.

The 8086 also has a sibling: the 8088. The 8088's registers and ALU are 16 bits, but its external data bus is 8 lines. To read a word from memory it makes two separate bus cycles. IBM chose the 8088 for its first personal computer in 1981; the 8-bit bus combined easily with the cheap 8-bit peripheral chips of those years. From the programmer's point of view the two processors run the same instructions.

Common mistake

The idea that "a 16-bit processor addresses 64 KB of memory" is wrong. The width of the data bus determines how much data is carried at a time, and the width of the address bus determines how much memory can be selected. The two are independent of each other. The 8086's data is 16 bits and its address 20 bits; so it addresses 1 MB, not 64 KB.

2.1.1.2 Its place in the x86 family

The processors that came after the 8086 were designed so that they could run programs written for their predecessors unchanged. This principle is called **backward compatibility**. Table 2.1.2 compares the first five members of the family.

Processor	Year	Internal / external data bus	Address bus	Memory	Notable feature
8086	1978	16 / 16 bits	20 bits	1 MB	6-byte prefetch queue
8088	1979	16 / 8 bits	20 bits	1 MB	4-byte queue; IBM PC (1981)
80186	1982	16 / 16 bits	20 bits	1 MB	integrated peripherals
80286	1982	16 / 16 bits	24 bits	16 MB	protected mode
80386	1985	32 / 32 bits	32 bits	4 GB	32-bit x86 (IA-32)

Table 2.1.2: The first members of the x86 family.

The memory column of the table comes directly from the address bus: 24 bits gives 2^{24} = 16,777,216 bytes (16 MB), and 32 bits gives 2^{32} = 4,294,967,296 bytes (4 GB). The 80186 adds peripherals such as a clock generator, timers, and an interrupt controller to the 8086 core, packing more work into a single chip for embedded systems. The 80286 brings protected mode: in this mode programs cannot access each other's memory, and all 16 MB can be used.

Definition 2.1.1

Real mode is the operating mode in which x86 processors behave like an 8086: addresses are computed as 20-bit physical addresses from a 16-bit segment and offset pair, there is no memory protection, and the directly accessible space is 1 MB. Since the 80286, every x86 processor starts in real mode when powered on.

All programs written with emu8086 in this course run in 8086 real mode. On a computer today, too, the processor is in real mode in the first moments of boot; the operating system later switches it to protected or 64-bit mode.

2.1.1.3 Why a 20-bit address?

Memory is an ordered sequence of cells, each holding one byte. Each cell has a number, that is, an address. To access a cell, the processor puts its address on the address bus. An n-line address bus can carry 2^n different bit patterns, so it can select 2^n different bytes.

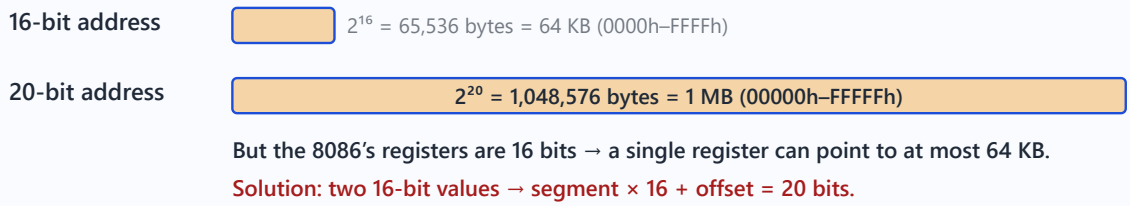


Figure 2.1.1: A 16-bit address selects 64 KB and a 20-bit address selects 1 MB. Because the 8086's registers are 16 bits, the 20-bit address is produced from two 16-bit values.

Example 2.1.1

Let us compare the capacity of 16-bit and 20-bit address buses.

1. 16 bits: 2^{16} = 65,536 bytes. Since 1 KB = 2^{10} = 1024 bytes, 65,536 / 1024 = 64 KB. Addresses `0000h` – `FFFFh`.
2. 20 bits: 2^{20} = 1,048,576 bytes. Since 1 MB = 2^{20} bytes, exactly 1 MB. Addresses `00000h` – `FFFFFh`.
3. Ratio: $2^{20} / 2^{16} = 2^4 = 16$. Four extra lines multiply the addressable memory by 16.

Each hexadecimal digit is four bits; 20 bits is exactly five digits. That is why physical addresses are always written with five digits.

The designers of the 8086 faced a contradiction here. In 1978, 64 KB of memory had become too small; a larger address space was needed. But the processor's registers were 16 bits, and a single register could point to only a 64 KB range. Widening the registers to 20 bits would have enlarged the chip and broken compatibility with software from 8-bit processors. The solution was to produce the address from two 16-bit values: one gives the start of a region of memory, that is, the **segment**, and the other gives the distance within that region, that is, the **offset**. The processor multiplies the segment value by 16 and adds the offset. This computation yields a 20-bit result. Figure 2.1.1 summarizes the idea.

2.1.1.4 Microprocessor and microcontroller

The 8086 is a **microprocessor**: it puts all the functions of the central processing unit (CPU) in a single integrated circuit. But it has no memory inside to store programs or data. Inside the chip there are only registers; intermediate and final results are kept in these registers. Program code, variables, and the stack are in external RAM and ROM chips. The processor reaches this memory and the I/O (input/output) devices over the **system bus**.

Definition 2.1.2

A **microprocessor** is an integrated circuit that puts the CPU's fetch, decode, and execute functions on a single chip but leaves memory and I/O units outside. A **microcontroller** (MCU), on the other hand, carries program memory (ROM or Flash), RAM, timers, and I/O ports on the same chip together with the CPU.

The two approaches have different uses. A microcontroller (8051, AVR, PIC) can work like a complete system by itself: once power and a crystal are connected, it reads its program from its own memory, lights lamps from its ports, and reads sensors. In embedded applications such as washing machines, remote controls, and sensor nodes, low cost and low power consumption stand out. Its processing power, however, is generally limited. A microprocessor is flexible: the amount of memory and the peripherals are chosen to suit the application, but the system consists of many chips.

Figure 2.1.2: The chips of an 8086-based system: the 8284A clock generator, the 8086, address latches (8282), data bus transceivers (8286), memory, and I/O devices.

Figure 2.1.2 shows the parts of the smallest 8086 system. The job of each chip is as follows:

- **The 8284A clock generator** gives the 8086 its clock signal (CLK). It divides the frequency of the external crystal by three: a 15 MHz crystal produces a 5 MHz clock. It also synchronizes the RESET and READY signals with the clock.
- **Address latches** (8282 or 74LS373) capture the address and hold it for the length of the bus cycle. The 8086's 40 pins are not enough to give every signal its own pin. For this reason the low 16 bits of the address and the data come out on the same pins (AD0–AD15) at different times. At the start of the cycle the pins carry the address; the processor produces an ALE (address latch enable) pulse and the latches capture the address. Then the same pins switch to data, but the memory keeps seeing the address in the latch. Three 8-bit latches are needed for 20 bits.
- **Data bus transceivers** (8286) amplify the data lines and set their direction: from memory to processor on a read, from processor to memory on a write. Two are used for 16 bits.
- **Memory** (RAM and ROM) holds the program and data; the **I/O devices** connect to the outside world, such as the keyboard, display, and ports.

These extra chips are the price of a microprocessor. In return the system's memory can be expanded up to 1 MB and any kind of peripheral can be added. Personal computers chose this path.

2.1.2 System buses

The processor, memory, and I/O devices are connected by parallel bundles of wires. Each bundle is called a **bus**. The width of a bus is the number of lines in it; each line carries one bit at a time. There are three buses in the system, and each has its own job.

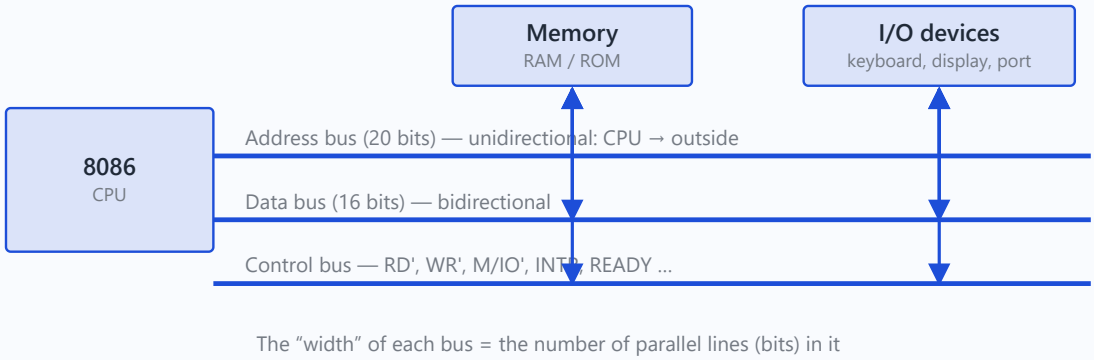


Figure 2.1.3: The processor, memory, and I/O devices are connected by address, data, and control buses. The address bus is unidirectional and the data bus is bidirectional.

2.1.2.1 Address, data, and control buses

Definition 2.1.3

The **system bus** is the whole of the three buses that connect the CPU, memory, and I/O devices. The **address bus** selects the memory location or I/O port to be accessed and works only from the processor outward. The **data bus** carries instruction and data bytes in both directions. The **control bus** consists of signals that tell the type of the operation (read, write, memory or I/O) and its timing.

Bus	What it carries	Direction	On the 8086
Address	the memory location to be read or written, the I/O port number	CPU → outside	20 bits → 1 MB of memory; for I/O 16 bits → 64 K ports
Data	instruction and data bytes between the processor and memory or I/O	bidirectional	16 bits, one word (2 bytes) at a time
Control	read, write, memory/I/O select, interrupt requests, status information	depends on the signal	RD', WR', M/IO', ALE, INTR, READY …

Table 2.1.3: The three buses of the 8086.

Only the processor produces the address; a memory or I/O device puts nothing on the address bus. That is why the address bus is unidirectional. Data, on the other hand, flows from outside to the processor on a read and from the processor outward on a write; the data bus is bidirectional. Some of the signals on the control bus are produced by the processor (RD', WR', M/IO', ALE), and some by external devices (INTR, the interrupt request, and READY, the ready signal).

The prime mark (') at the end of a signal name shows that the signal is **active low**: the signal does its job when it is at logic level 0. RD' = 0 means “read now,” and WR' = 0 means “write now.” The M/IO' signal selects the target of the access: memory when 1, I/O when 0. Memory chips respond only when M/IO' = 1, and I/O devices respond only when M/IO' = 0. In this way the same address and data lines serve two separate address spaces.

The I/O address space is smaller than memory. In an I/O access only the low 16 bits of the address (A15–A0) are used, and the A19–A16 lines are 0. There are 2^{16} = 65,536 port numbers. The `IN` and `OUT` instructions take the port number either as an 8-bit constant inside the instruction (0–FFh) or in the DX register (0–FFFFh).

2.1.2.2 The bus cycle

Each time the processor reads once or writes once to memory or I/O, it takes one **bus cycle**. On the 8086 a bus cycle consists of at least four clock cycles. These cycles are called T states and are named T1, T2, T3, and T4.

Definition 2.1.4

A **bus cycle** is the time period over which a single read or write operation with memory or I/O runs from start to finish. On the 8086 it consists of four T states (T1–T4); a slow device can have wait states (Tw) inserted between T3 and T4 by holding the READY signal at 0.

A memory read cycle proceeds in this order:

1. **T1:** The processor puts the 20-bit address on the address lines, announces the target with M/IO', and issues the ALE pulse. The latches capture the address.
2. **T2:** The AD0–AD15 pins are released from the address and become free for data. The processor makes the RD' signal 0: “read.”
3. **T3:** The memory puts the byte or word at the address on the data bus. The processor samples READY; if the device is ready it takes the data.
4. **T4:** RD' becomes 1 again and the memory releases the data bus. The cycle ends.

In a write cycle the order is the same; the difference is that the processor puts the data on the data bus starting at T2 and makes the WR' signal 0 instead of RD'. While WR' is active, the memory writes the value on the data bus into its cell.

Example 2.1.2

A 5 MHz 8086 reads 100 words from memory with no wait states. How long does it take at least?

1. One clock cycle: 1 / 5 MHz = 200 ns.
2. One bus cycle: 4 × 200 ns = 800 ns.
3. If the words are at even addresses, each word arrives in a single cycle: 100 × 800 ns = 80,000 ns = 80 µs.

This shows that the bus’s maximum transfer rate is 2 bytes / 800 ns = 2.5 MB/s. If the words were at odd addresses, each would take two cycles and the time would be 160 µs. The 8086’s memory is divided into two byte banks, even-addressed and odd-addressed. The bytes of a word starting at an odd address fall in different rows and cannot be read together in a single cycle.

Comparing these times with registers makes one thing clear: memory is slow. A register-to-register `MOV AX, BX` takes 2 clock cycles on the 8086. `MOV AX, [BX]`, which reads the same value from memory, takes 13 cycles including the address calculation and the bus cycle. That is why frequently used values are kept in registers and why the 8086’s internal structure is arranged to hide memory waiting.

2.1.2.3 Code trace: memory and I/O on the buses

The following program makes four accesses: it reads from memory, writes to an I/O port, reads from an I/O port, and writes to memory. Following which values are on the buses in each access makes the roles of the address, data, and control buses concrete. At the start of the program it is assumed that memory at address 10200h holds 5Ah and the I/O port numbered 51h holds 3Ch.

```
org 100h
mov ax, 1000h
mov ds, ax
mov bx, 0200h
mov al, [bx]     ; memory read
out 50h, al      ; I/O write
in al, 51h       ; I/O read
mov [bx+1], al   ; memory write
ret
```

`org 100h` is not an instruction but a directive given to the assembler. It declares that the program, in .COM form, starts at offset 100h of the segment. The first two instructions load 1000h into the DS register. The value is moved first to AX and then from AX to DS, because on the 8086 a constant cannot be loaded directly into a segment register. With DS = 1000h the data segment starts at physical address 10000h. `mov bx, 0200h` puts the offset into BX. These three instructions change only registers; no bus is used for data. The buses work only to fetch the instruction bytes.

`mov al, [bx]` is the first data access. The square brackets mean “the memory location BX points to.” The processor computes the physical address as DS × 10h + BX = 10000h + 0200h = 10200h and starts a memory read cycle: 10200h on the address bus, M/IO' = 1, RD' = 0. The memory puts 5Ah on the data bus and the processor writes it into AL.

`out 50h, al` sends the value in AL to port number 50h. This is an I/O write cycle. Only the port number is on the address bus: 0050h. Because M/IO' = 0, the memory chips ignore this cycle; WR' = 0 and the data bus holds 5Ah. The device attached to port 50h takes the value.

`in al, 51h` is an I/O read cycle: address 0051h, M/IO' = 0, RD' = 0. The device puts 3Ch on the data bus and AL = 3Ch. The last data instruction, `mov [bx+1], al`, writes to memory. The effective address is BX + 1 = 0201h, and the physical address is 10000h + 0201h = 10201h. In the cycle M/IO' = 1, WR' = 0, and the data bus holds 3Ch. `ret` ends the program and returns control to the operating system.

Instruction	AL	Address bus	Data bus	M/IO'	RD'	WR'
`mov ax,1000h` / `mov ds,ax`	00h	—	—	—	—	—
`mov bx,0200h`	00h	—	—	—	—	—
`mov al,[bx]`	5Ah	10200h	5Ah (from memory)	1	0	1
`out 50h,al`	5Ah	0050h	5Ah (out)	0	1	0

Instruction	AL	Address bus	Data bus	M/IO'	RD'	WR'
`in al,51h`	3Ch	0051h	3Ch (from device)	0	0	1
`mov [bx+1],al`	3Ch	10201h	3Ch (out)	1	1	0

Table 2.1.4: Trace of the buses program. "—" shows that there is no bus cycle for data in that instruction.

Figure 2.1.4: On the left a memory read (`mov al,[bx]`): address 10200h, M/IO' = 1, RD' = 0. On the right an I/O write (`out 50h,al`): address 0050h, M/IO' = 0, WR' = 0.

Figure 2.1.5: On the left an I/O read (`in al,51h`): the device puts 3Ch on the data bus. On the right a memory write (`mov [bx+1],al`): 3Ch is written to address 10201h.

Table 2.1.4, Figure 2.1.4, and Figure 2.1.5 show what the four accesses have in common: they all use the same address and data lines. M/IO' separates memory from I/O, and RD' and WR' separate reads from writes. In a memory access the address bus holds the 20-bit physical address, and in an I/O access the 16-bit port number. When the program is run in emu8086, the final value of AL is 3Ch, the byte at 10201h is 3Ch, and the value written to port 50h is 5Ah.

Common mistake

It is wrong to think that `out 50h, al` writes to the memory cell at address 50h. A port number and a memory address are in different address spaces. Because M/IO' = 0, the memory chips do not respond to this cycle at all; the memory byte at 00050h does not change. On the 8088 this signal has the opposite name and polarity (IO/M'), but the principle of the distinction is the same.

2.1.3 Internal structure: BIU and EU

A simple processor does the same three steps in turn for every instruction: it fetches the instruction from memory, decodes what to do, and executes it. While it fetches, the processor's computation units wait idle; while it executes, the bus sits idle. To close this gap, the 8086 splits its inside into two independent units. One does all the bus-related work and the other executes instructions. Because the two units can work at the same time, the bytes of the next instruction can be fetched while one instruction is being executed.

2.1.3.1 Two basic units

Definition 2.1.5

The **bus interface unit** (BIU) is the unit that carries out all of the 8086's communication with the outside world: it fetches instructions from memory, computes the 20-bit physical address, reads operands, and writes results. The **execution unit** (EU) is the unit that decodes and executes instructions: it contains the ALU, the general-purpose registers, and the flags, and is not connected directly to the external buses.

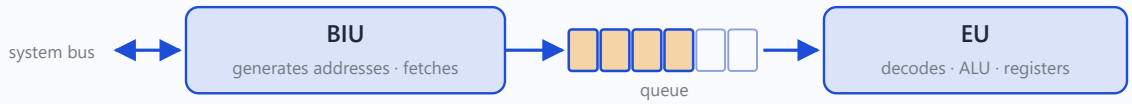


Figure 2.1.6: The BIU is connected to the system bus, generates addresses, and writes instruction bytes into the queue. The EU takes the bytes from the queue, decodes them, and executes them on the ALU and registers.

The bridge between the two units is the prefetch queue. The BIU appends the instruction bytes it fetches from memory to the end of the queue; the EU takes the bytes of the next instruction from the front of the queue. If the EU needs an operand in memory, for example while executing `ADD AX, [BX]`, it computes the offset of the address and asks the BIU to do the read. The BIU runs the bus cycle and hands the data to the EU. The EU never puts anything on the address bus itself.

Thanks to this division of labor the two units work **in parallel**. While the EU does an addition, the BIU fetches the bytes of the next instruction. The overlap of fetching and execution in time is called a **pipeline**. The 8086's pipeline has only two stages, but it gave a significant speed gain for 1978.

2.1.3.2 Block diagram

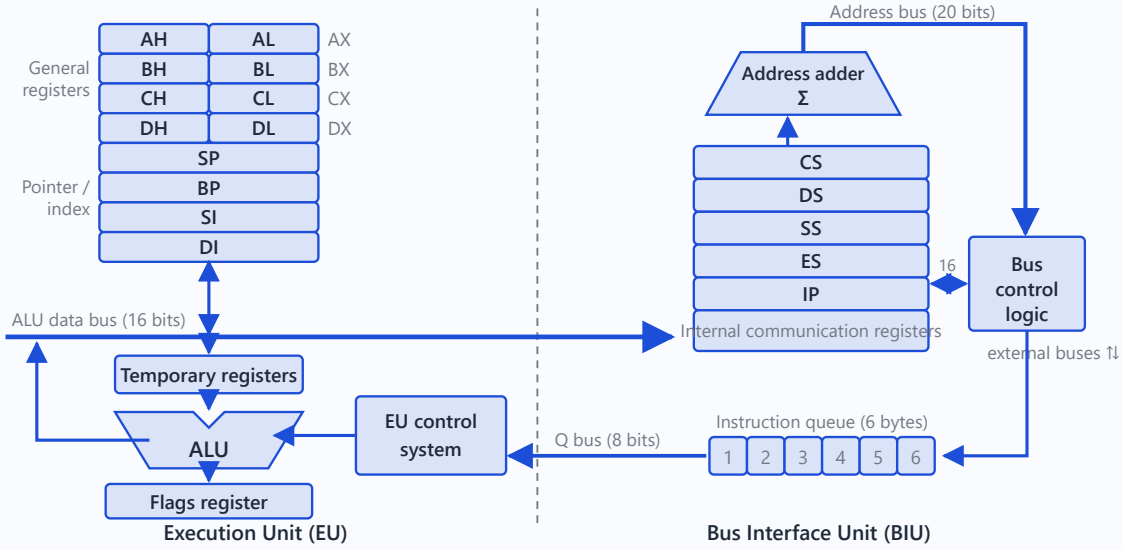


Figure 2.1.7: The internal block diagram of the 8086. On the left the EU (general registers, pointer and index registers, ALU, flags register, control system); on the right the BIU (address adder, segment registers, IP, the 6-byte instruction queue, bus control logic).

Figure 2.1.7 divides the inside of the processor into two halves. In the EU on the left half, the four general-purpose registers sit at the top: AX, BX, CX, DX. Each can be split into two 8-bit halves (AH-AL, BH-BL, CH-CL, DH-DL). Below them are the pointer and index registers: SP, BP, SI, DI. All these registers are connected to the 16-bit **ALU data bus**. The ALU takes its operands from temporary registers; the programmer cannot see these temporary registers. The ALU's output returns to the data bus and carries the result to a register. At the same time the flags register is updated according to the result of the operation. The EU's control system decodes the opcode coming from the queue and produces the signals that run all these parts in sequence.

The most prominent part of the BIU on the right half is the **address adder** (Σ). It multiplies the value in a segment register by 16 and adds an offset; the result is the 20-bit physical address and goes out on the address bus. Below the adder are the four segment registers (CS, DS, SS, ES) and the instruction pointer IP. The internal communication registers temporarily hold data moving between the EU and the BIU. The bus control logic connects to the external buses and runs the bus cycles. Instruction bytes coming from memory are written into the 6-byte **instruction queue**; the EU takes these bytes over the 8-bit **Q bus**.

Which unit a register sits in also says something about its role. The segment registers and IP are in the BIU, because both are for producing addresses. The registers that operate on data and the flags are in the EU, because they are part of execution.

2.1.3.3 The journey of an instruction

To see how the two units work together, let us follow a single instruction, `ADD AX, BX`, from its fetch from memory to the writing of the result.

1. The BIU computes the physical address of the instruction to be fetched in the address adder: **CS × 10h + IP**.
2. The 20-bit address is put on the external address bus; the bus control logic starts a memory read cycle.
3. The incoming instruction bytes are written into the 6-byte queue. IP advances by the number of bytes fetched.
4. When this instruction's turn comes, the EU takes the opcode over the Q bus. The control system decodes the instruction: the operation is addition, the operands are AX and BX.
5. Control signals carry AX and BX over the ALU data bus to the temporary registers and from there to the ALU. The ALU adds.
6. The result is written to AX; the flags (CF, ZF, SF, OF, PF, AF) are updated according to the result. Meanwhile the BIU keeps fetching the next bytes.

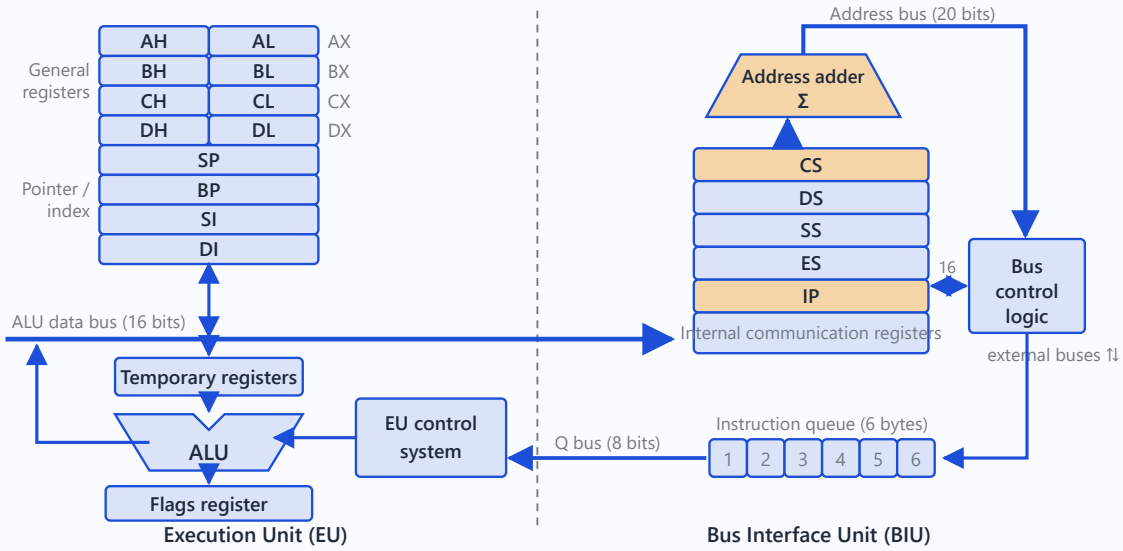


Figure 2.1.8: Step one: the address adder in the BIU computes the instruction's physical address from CS and IP. The highlighted parts are the units working in that step.

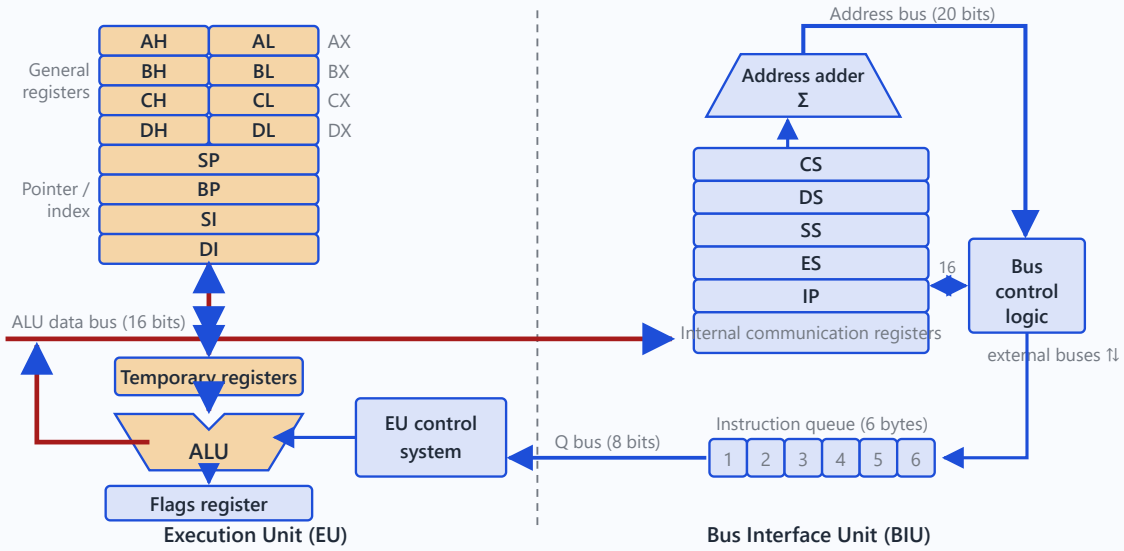


Figure 2.1.9: Step five: AX and BX come to the ALU over the ALU data bus and the temporary registers; the addition is done.

The first three of the steps take place in the BIU and the last three in the EU (Figure 2.1.8, Figure 2.1.9). This boundary is the basis of the pipeline. The moment the EU takes an instruction from the queue in the fourth step, room opens up in the queue. The BIU starts a new fetch cycle without waiting for the EU's fifth and sixth steps. Because `ADD AX, BX` uses no memory operand, the EU needs the bus not at all during execution, and the bus is given entirely to prefetching.

2.1.3.4 The bus interface unit (BIU)

Listing the BIU's duties together explains why it is a separate unit:

- It interfaces with memory and I/O devices over the system bus; it runs all the bus cycles.
- It produces the 20-bit physical address from a segment register and an offset in the address adder.
- It fetches instructions from memory and writes them into the queue.
- It reads from memory or I/O the operands the EU asks for and writes the results.
- It manages the 6-byte prefetch queue; this queue is what makes the pipeline possible.

Inside the BIU are the segment registers (CS, DS, SS, ES), the instruction pointer (IP), the address adder, the 6-byte instruction queue, and the bus control logic. When the EU asks for a memory operand, the BIU finishes the fetch cycle in progress and then carries out the EU's request. The EU's request takes priority over prefetching, because the EU cannot proceed until that data arrives, while the queue can wait a little.

2.1.4 The prefetch queue and the pipeline

The BIU and EU can work in parallel only because there is a buffer between them that accumulates bytes. Without this buffer the BIU would have to hold the byte it fetched until the EU took it. The queue separates the two units: the BIU fills the queue, the EU empties it, and each works at its own pace.

2.1.4.1 How the queue works

Definition 2.1.6

Prefetching is taking instruction bytes from memory and holding them before they are executed. The 8086's **prefetch queue** is a 6-byte FIFO (first in, first out) buffer that holds the instruction bytes the BIU fetches: bytes are consumed in the order they were fetched.

Figure 2.1.10: The BIU writes the bytes it fetches from memory to the end of the queue, and the EU takes them from the front. In the figure the bytes of the instructions MOV AX,1234h (B8 34 12) and ADD AX,BX (01 D8) are waiting in the queue.

Four rules determine the queue's behavior:

1. **The fetch rule.** When at least 2 bytes of room open up in the queue and the EU does not want the bus, the BIU fetches the next word (2 bytes). Because the 16-bit data bus carries two bytes in one cycle, the unit of fetching is the word.
2. **The consume rule.** The EU takes the bytes of the next instruction from the head of the queue. If the queue is empty, the EU waits until the first byte arrives.
3. **The full queue.** If the queue is full and the EU does not want a memory operand, the BIU stays idle. No bus cycle is made in this case.
4. **The branch rule.** When a branch (JMP, CALL, RET, conditional jump, interrupt) executes, the queue is **flushed**. The prefetched bytes came from the following addresses, while the program now continues from another address. The BIU starts fetching from the new address.

The queue being 6 bytes is not arbitrary. 8086 instructions without prefixes are between 1 and 6 bytes. A 6-byte queue can hold even the longest instruction completely. Because the 8088's external data bus is 8 bits, only one byte arrives per cycle; its queue is 4 bytes and it starts fetching when 1 byte is freed.

The gain from the queue is that the EU usually finds an instruction without waiting. The memory access time passes in the background while the EU executes and is invisible to the programmer. The loss appears at a branch: the queue is flushed and the EU waits until it fills again. In programs full of loops and conditional jumps this loss is a significant share.

2.1.4.2 The pipeline

Definition 2.1.7

A **pipeline** is an arrangement in which different stages of consecutive instructions are processed at the same time. The 8086 has two stages: fetch (F) in the BIU and decode and execute (E) in the EU. While the EU executes one instruction, the BIU fetches the next.

To see the gain of the pipeline, let us use a simple model: let each fetch take 2 time units, and each decode and execute also 2 time units. In reality instruction times differ, but the model shows the idea clearly.

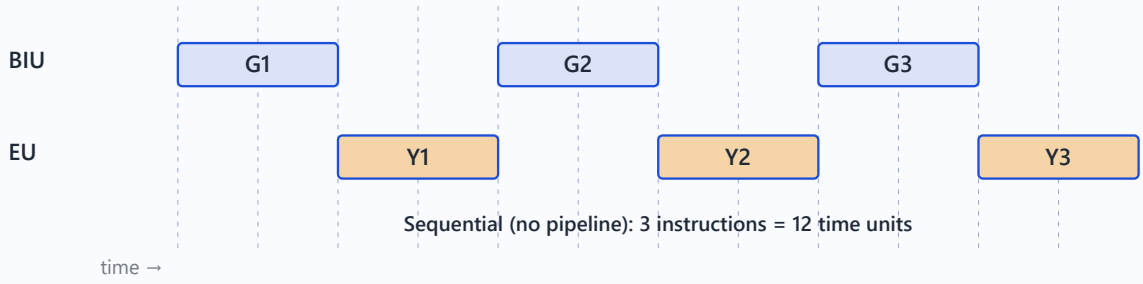


Figure 2.1.11: Without a pipeline: first fetch, then execute. The BIU and EU wait idle in turn; 3 instructions take 12 time units.

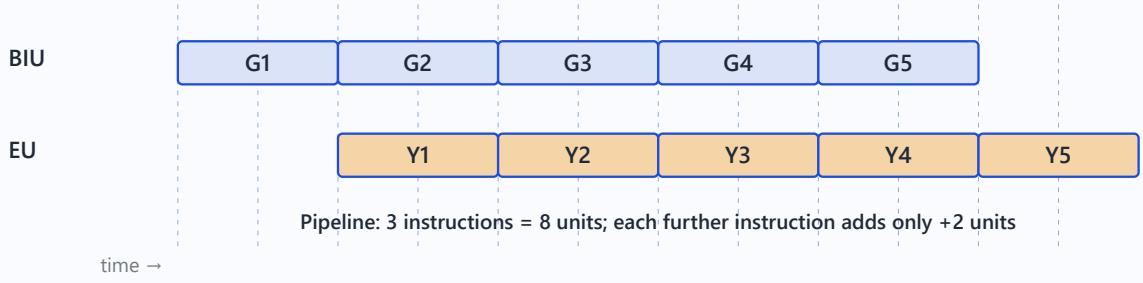


Figure 2.1.12: The 8086's pipeline: while the EU executes E1, the BIU fetches F2. 3 instructions finish in 8 units; each further instruction adds only 2 units.

In the sequential arrangement (Figure 2.1.11) each instruction takes 2 + 2 = 4 units and *n* instructions take *4n* units. In the pipeline (Figure 2.1.12) the first fetch takes 2 units; after that one instruction finishes every 2 units. *n* instructions take *2 + 2n* units. As the number of instructions grows, the ratio 4n / (2n + 2) approaches 2: the throughput roughly doubles.

Example 2.1.3

Let us compute the time of 5 instructions in three cases with the same model.

1. No pipeline: 5 × (2 + 2) = 20 units.
2. With the pipeline: 2 + 5 × 2 = 12 units. F1 0–2, E1 2–4 (F2 at the same time), E2 4–6, E3 6–8, E4 8–10, E5 10–12.
3. The third instruction is a JMP: during E3 (6–8) the prefetched fourth instruction is on the wrong path and is discarded. The target can be fetched only when E3 ends: F4 8–10, E4 10–12, E5 12–14. Total 14 units.

The branch takes back 2 units of the pipeline's gain. In a program of 100 instructions the pipeline reduces 400 units to 202 units; each jump adds 2 units to this figure again.

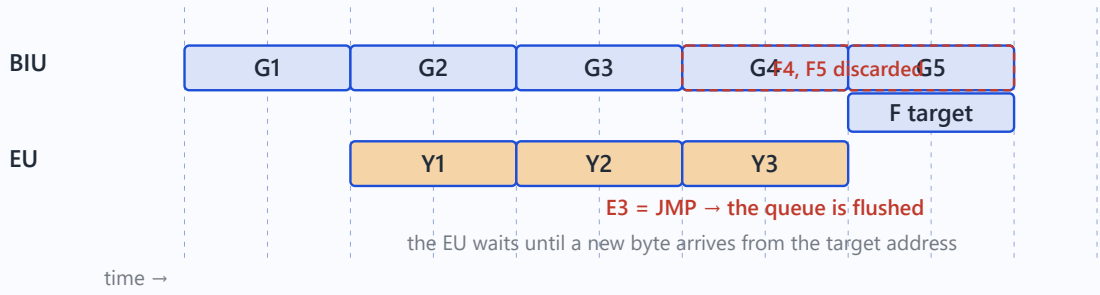


Figure 2.1.13: A branch: if E3 is a JMP, the prefetched F4 and F5 are invalid. The queue is flushed, and the EU waits until a new byte arrives from the target address.

Figure 2.1.13 shows this loss on a timeline. Modern processors ease the same problem with branch prediction: they guess the outcome of a jump in advance and start fetching from the target early. The 8086 has no such mechanism; every taken branch resets the queue.

2.1.4.3 The queue step by step

Following the rules of the queue on a program shows what happens at the byte level. The program starts at CS:0100h and holds these bytes in memory:

```
0100: B8 34 12     mov ax, 1234h
0103: 01 D8        add ax, bx
0105: 41           inc cx
0106: EB 05        jmp short 010Dh
0108: 90 90 90 90 90   ; five NOPs
010D: B0 ...       mov al, ...
```

The number at the start of each line is the offset of the instruction in the code segment. `B8` is the opcode "load a 16-bit constant into AX," and the constant follows with the low byte first (34 12). `01 D8` is the machine code of `ADD AX, BX`. `41` is the one-byte `INC CX`. `EB 05` is the short jump: 5 is added to the address of the instruction after the jump. `90` is the `NOP` instruction, which does nothing.

Step	Queue	EU	IP of the next instruction	Event
1	(empty)	waiting	0100h	start or after a branch
2	B8 34 12 01 D8 41	MOV AX,1234h being decoded	0100h	BIU fetched three words (B8 34, 12 01, D8 41)
3	01 D8 41	MOV AX,1234h	0103h	EU pulled 3 bytes; 3 free slots
4	01 D8 41 EB 05	MOV AX,1234h	0103h	BIU fetched EB 05; EU did not wait
5	41 EB 05 90 90	ADD AX,BX	0105h	EU took 01 D8; BIU fetched 90 90
6	EB 05 90 90 90 90	INC CX	0106h	EU took 41; BIU fetched two more NOPs, queue full
7	(flushed)	JMP SHORT 010Dh	010Dh	IP = 0108h + 5 = 010Dh; four 90s discarded
8	B0	waiting	010Dh	BIU fetches from 010Dh; because it is an odd address, 1 byte first

Table 2.1.5: The step-by-step state of the queue. In the queue column the leftmost byte is the head of the queue.

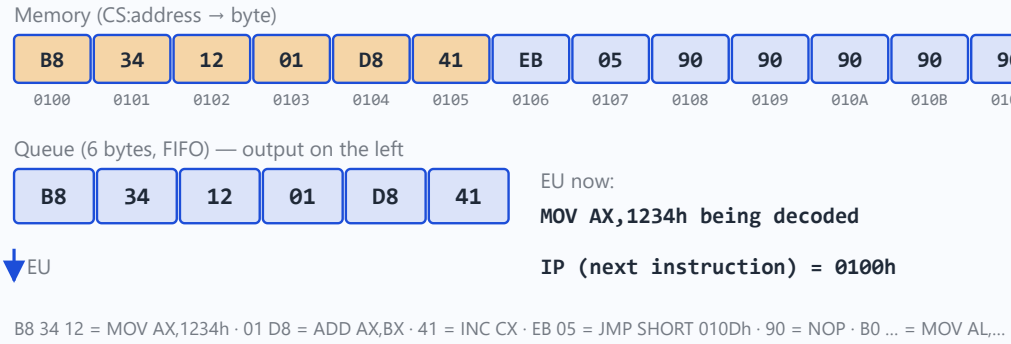


Figure 2.1.14: Step two: the BIU has fetched three words and filled the queue; the EU started decoding MOV AX,1234h as soon as the first byte arrived.

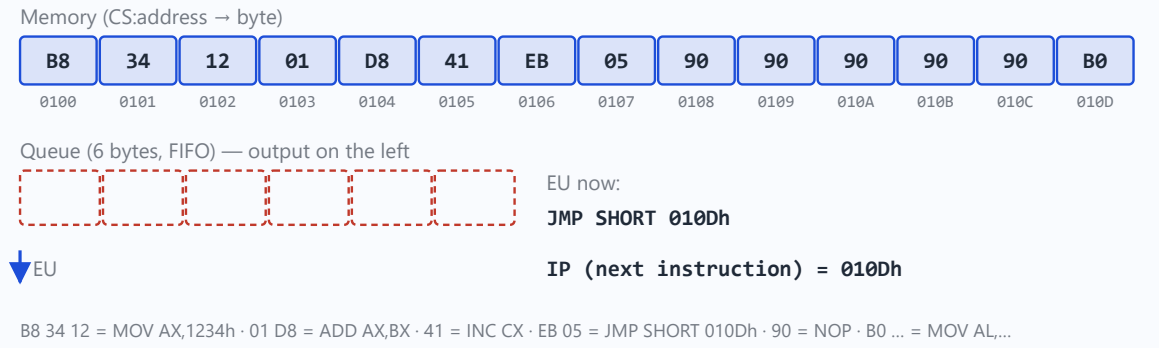


Figure 2.1.15: Step seven: the EU executes the JMP and IP becomes 010Dh. The NOP bytes in the queue are on the wrong path and the queue is flushed.

Table 2.1.5 reveals a few details. In the second step the EU starts working before the queue is completely full; decoding can begin once the first byte arrives. In the third step, when the EU pulls the 3-byte instruction from the queue, 3 free slots open up. Because the rule is "at least 2 free slots," the BIU immediately fetches a word. From then on the BIU stays ahead of the EU: in the fourth and sixth steps the EU never waits.

In the seventh step the jump is executed (Figure 2.1.15). The displacement of the short jump is computed relative to the address of the instruction after the jump. Since the JMP starts at 0106h and takes two bytes, the next address is 0108h; 0108h + 5 = 010Dh. The four 90 bytes waiting in the queue came from addresses 0108h–010Bh and will no longer be executed. The queue is flushed. In the eighth step the BIU starts fetching from 010Dh. 010Dh is an odd address; because the 8086 can read a word in a single cycle only from an even address, it first fetches just one byte (B0), and the later fetches again proceed word by word from even addresses. The EU waits during this time; this wait is the price of the branch.

Note that `ADD AX, BX` is encoded here as `01 D8`. On the 8086 an operation between two registers can be encoded in two ways. The opcode `01` means "add the register to the r/m field," and the opcode `03` means "add the r/m field to the register." When both destination and source are registers, the two do the same job: `01 D8` and `03 C3` are equivalent. Most assemblers, emu8086 included, produce the second form.

2.1.4.4 Prefetch and decode functions

Prefetching and decoding are treated as two separate functions in processor design. The **prefetch function** fetches instructions from memory and keeps them in a buffer. A program counter holds the address of the next byte to be fetched. On the 8086 this is done with IP and a counter inside the BIU: IP is the "next instruction" address the programmer sees, while the BIU's counter points to the next byte to enter the queue. When the queue is full this counter is a few bytes ahead of IP. Because the EU takes instructions from the buffer instead of memory, the memory access time is hidden and more instructions are executed per unit time.

The **decode function** turns machine code into control signals and micro-operations. It determines which operation, which operands, and which addressing mode an opcode asks for. It directs transfers from register to register and from memory to register step by step, as well as complex instructions such as conditional jumps, CALL, and RET. Decoding also works in parallel with prefetching.

On the 8086 these functions are not separate units. Prefetching is done in the BIU and decoding in the EU's control system. The control system is **microcode**-based: each machine instruction corresponds to a short sequence of micro-operations stored in a ROM inside the chip. A complex instruction, such as divide, runs a long sequence of micro-operations; that is why it takes hundreds of clock cycles. Separate prefetch and decode units appeared with the 80286; in the 80286 address computation and instruction decoding moved to their own units and the pipeline grew to four stages.

2.1.5 The execution unit

The EU is where instructions are really "done." It decodes the bytes coming from the queue, takes the operands from registers, has the ALU do the operation, and writes the result back. This section covers the parts of the EU, the operations the ALU performs, and the control unit that directs the EU.

2.1.5.1 Components of the EU

The main components of the EU are these:

- **General-purpose registers** (AX, BX, CX, DX): they store data and take part in arithmetic and logic operations.
- **Arithmetic logic unit** (ALU): it does 8-bit and 16-bit computations.
- **Pointer and index registers** (SP, BP, SI, DI): they hold offsets in stack and memory addressing.
- **Instruction register and instruction decoder**: they hold and decode the opcode taken from the queue.
- **Flags register** (FLAGS): it holds the status bits that summarize the result of the last operation and the control bits that determine the processor's behavior.
- **Control system**: it produces, in sequence, the signals that run all these parts.

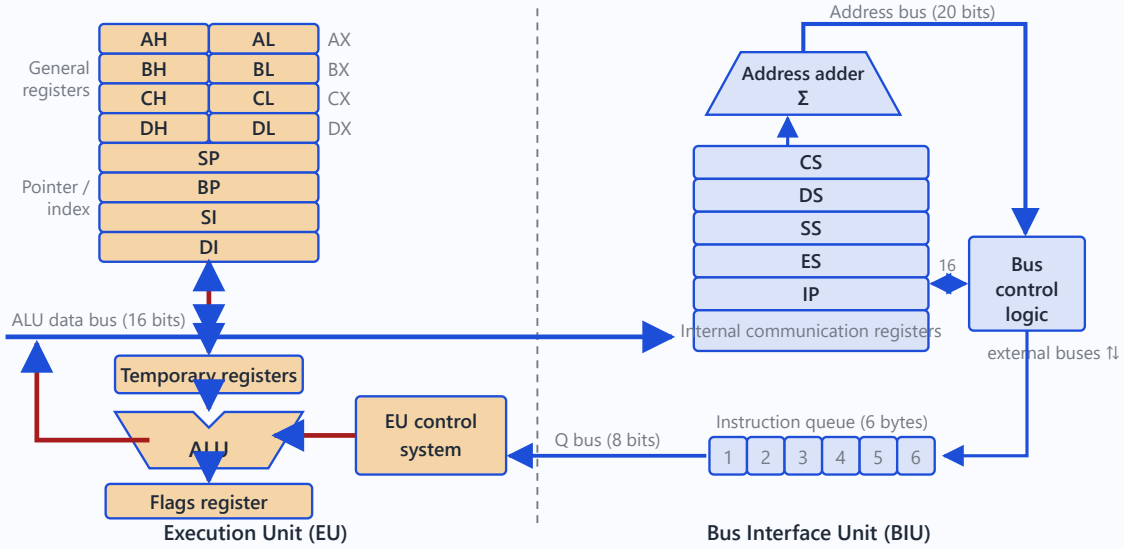


Figure 2.1.16: The EU's components are highlighted on the block diagram: general registers, pointer and index registers, temporary registers, ALU, flags register, and control system.

2.1.5.2 The arithmetic logic unit

Definition 2.1.8

The **arithmetic logic unit** (ALU) is the unit that performs an arithmetic or logic operation on two operands and produces the result. The 8086's ALU works with 8-bit and 16-bit operands and updates the status flags according to the result after every operation.

The ALU's work falls into two groups. The arithmetic operations are addition, subtraction, multiplication, division, increment, decrement, and comparison. The logic operations are AND, OR, XOR, and NOT, and shifting and rotating bits left or right. Comparison (`CMP`) is really a subtraction; its result is stored nowhere, only the flags are set.

Instruction	Operation	Size
`ADD AX, BX`	AX ← AX + BX	16 bit
`SUB AL, 5`	AL ← AL − 5	8 bits
`MUL BL`	AX ← AL × BL	8 → 16 bits
`DIV CX`	AX ← DX:AX ÷ CX, DX ← remainder	32 → 16 bits
`AND AL, 0Fh`	the upper 4 bits of AL are cleared	8 bits
`XOR AX, AX`	AX ← 0 (ZF = 1)	16 bit

Table 2.1.6: Examples of ALU operations.

The size column for multiplication and division needs attention. The product of two 8-bit numbers can grow to 16 bits: 255 × 255 = 65,025. This is why `MUL BL` writes its result not to AL but to the whole of AX. In division the reverse holds: a 16-bit division starts with a 32-bit dividend. The upper half of the dividend is in DX and the lower half in AX, and this pair is written DX:AX. The quotient goes to AX and the remainder to DX.

Flags are the trace the ALU leaves after each operation. There are six status flags:

- **CF** (carry flag): 1 if a carry or borrow came out of the top bit in an unsigned operation.
- **ZF** (zero flag): 1 if the result is zero.
- **SF** (sign flag): the most significant bit of the result; 1 if negative in the signed interpretation.
- **OF** (overflow flag): 1 if the result of a signed operation did not fit in the destination.
- **PF** (parity flag): 1 if the low byte of the result has an even number of 1s.
- **AF** (auxiliary carry flag): 1 if there was a carry or borrow from bit 3 to bit 4.

In addition there are three control flags (TF, IF, DF) set by the programmer; they are not affected by the ALU's result. Not every instruction changes all the flags: `MOV` touches none, `INC` and `DEC` preserve CF, and after `MUL` and `DIV` some flags are **undefined**. An undefined flag can be 0 or 1 in the processor, but its value cannot be relied on; a program must not make decisions by looking at it.

2.1.5.3 Code trace: what does the ALU do?

The following program runs the operations in Table 2.1.6 in turn. At each step we follow which register changes and what the flags are.

```
org 100h
mov ax, 0010h
mov bx, 0030h
add ax, bx      ; AX ← AX + BX
sub al, 5       ; AL ← AL − 5
mov bl, 4
mul bl          ; AX ← AL × BL
mov dx, 0
mov cx, 3
div cx          ; AX ← DX:AX ÷ CX
and al, 0Fh     ; clear upper 4 bits
xor ax, ax      ; AX ← 0
ret
```

The first two `MOV`s load the operands into registers: AX = 0010h, BX = 0030h. `MOV` does not use the ALU; the value is just copied from one place to another and the flags do not change. `add ax, bx` is a 16-bit addition: 0010h + 0030h = 0040h. In decimal that is 16 + 48 = 64. No carry comes out of the top bit (CF = 0), the result is not zero (ZF = 0), and the top bit is 0 (SF = 0).

`sub al, 5` is an 8-bit subtraction and affects only AL. AL = 40h; 40h − 05h = 3Bh (64 − 5 = 59). AH does not change, so AX = 003Bh. Since the result fits as unsigned, no borrow is needed and CF stays 0. In the low nibble, 0 − 5 borrows from bit 4, so AF = 1.

`mov bl, 4` changes only the low half of BX: since BH is already 00h, BX = 0004h. `mul bl` multiplies AL by BL: 3Bh × 04h = ECh, that is, 59 × 4 = 236. The result is written to AX as 16 bits: AX = 00ECh. Because the upper half of the result (AH) is zero, CF = OF = 0; these two flags carry the information "the result fit in AL." SF, ZF, AF, and PF, on the other hand, are undefined after `MUL`.

Preparing for the division takes two lines. `mov dx, 0` clears the upper half of the dividend; so DX:AX = 0000 00ECh = 236. `mov cx, 3` loads the divisor. `div cx` divides 236 by 3: the quotient 78 = 004Eh is written to AX and the remainder 2 = 0002h to DX. Check: 78 × 3 + 2 = 236. After `DIV` all six status flags are undefined.

`and al, 0Fh` is a masking operation. 4Eh = 0100 1110 and 0Fh = 0000 1111; AND multiplies the bits one by one: 0000 1110 = 0Eh. The upper four bits where the mask is 0 are cleared, and the lower four bits where it is 1 are kept. AND always makes CF = OF = 0. The last instruction, `xor ax, ax`, XORs every bit with itself. Since the XOR of two equal bits is 0, AX = 0000h and ZF = 1. This is a shorter way to clear a register than `mov ax, 0`: `33 C0` is two bytes and `B8 00 00` is three. There is a difference as well: XOR changes the flags, MOV does not.

Row	AX	BX	CX	DX	CF	ZF	SF	OF
2–3 `mov ax / mov bx`	0010h	0030h	0000h	0000h	0	0	0	0
4 `add ax, bx`	0040h	0030h	0000h	0000h	0	0	0	0
5 `sub al, 5`	003Bh	0030h	0000h	0000h	0	0	0	0
6–7 `mov bl / mul bl`	00ECh	0004h	0000h	0000h	0	?	?	0
8–9 `mov dx / mov cx`	00ECh	0004h	0003h	0000h	0	?	?	0
10 `div cx`	004Eh	0004h	0003h	0002h	?	?	?	?
11 `and al, 0Fh`	000Eh	0004h	0003h	0002h	0	0	0	0
12 `xor ax, ax`	0000h	0004h	0003h	0002h	0	1	0	0

Table 2.1.7: Trace of the ALU program. "?" is an undefined flag.

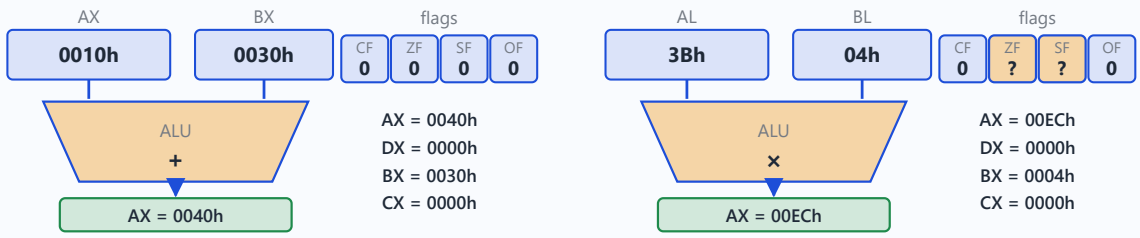


Figure 2.1.17: On the left `add ax, bx`: 16-bit addition, AX = 0040h. On the right `mul bl`: 16-bit product of 8-bit operands, AX = 00ECh; ZF and SF undefined.

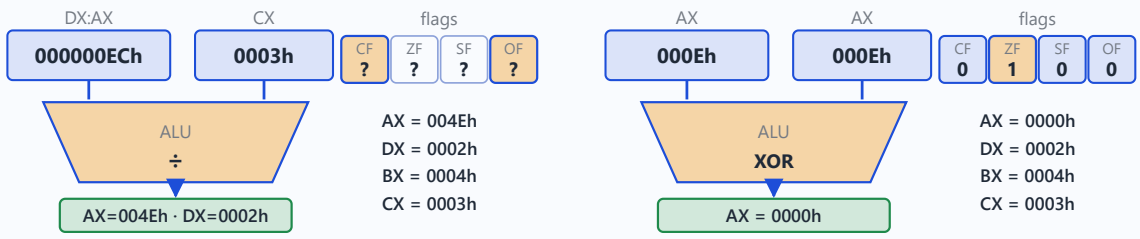


Figure 2.1.18: On the left `div cx` : DX:AX = 236, quotient AX = 004Eh, remainder DX = 0002h; all status flags undefined. On the right `xor ax, ax` : AX = 0000h, ZF = 1.

Common mistake

Forgetting to clear DX before `div cx` is common. A 16-bit `DIV` always divides the DX:AX pair. If DX has been left with, say, 0005h, the dividend becomes 5 × 65,536 + 236 = 327,916. The quotient of dividing this number by 3 is 109,305, which does not fit in 16 bits. In this case the processor raises a divide-error interrupt (INT 0) and the program stops.

The times of these instructions are also very different. On the 8086 `ADD AX, BX` takes 3 clock cycles. `MUL BL` takes 70–77 cycles and `DIV CX` takes 144–162 cycles, because the microcode does multiplication and division by shifting and adding bit by bit. At 5 MHz a single 16-bit division comes to about 30 µs. When the divisor is a power of 2, the shift instructions are much faster than division.

2.1.5.4 The instruction register, decoder, and control unit

Inside the EU there are three parts the programmer never sees but that work on every instruction.

- **The instruction register**: the EU takes the instruction to be executed from the queue; while the opcode is decoded it is held here temporarily. The programmer cannot access this register.
- **The instruction decoder**: it decodes the opcode. It determines which operation is to be done, where the operands are, and which addressing mode is used, and sends this information to the control circuit.
- **The control unit**: it manages the flow of instructions and data inside the processor. It sequences the fetch–decode–execute cycle and determines what the other components do and when.

The control unit's duties can be summarized as a cycle: fetching the instruction through the BIU, decoding it, executing it, updating the processor's state (registers, flags), and, at the end of each instruction, checking for **interrupt requests**. A request can come on two lines: a maskable one from INTR (accepted only when IF = 1) and a non-maskable one from NMI. If there is a request, the processor branches to the interrupt service routine before moving on to the next instruction. Some error conditions also produce interrupts; division by zero or a quotient that does not fit triggers INT 0.

The 8086's control unit is microcode-based. Features such as power management are absent from the 8086; they came with much later processors, for example the 386SL. The 8086 either runs or stops with the `HLT` instruction until an interrupt arrives.

2.1.6 Instruction lengths and machine code

A program sitting in memory is not assembly lines but a sequence of bytes. Each instruction is translated into one or more bytes that the processor decodes. On the 8086 this byte count varies from instruction to instruction. Knowing the length is the key to understanding the behavior of the queue and of IP: IP advances after each instruction by exactly the length of that instruction.

2.1.6.1 Instruction format

Definition 2.1.9

An 8086 machine instruction consists of these fields: the **opcode** (1 byte; it specifies which operation to perform), a **ModR/M byte** if present (it specifies the operands and the addressing mode), an 8- or 16-bit **displacement** if present (a constant added to the memory address), and an 8- or 16-bit **immediate** if present (a constant operand inside the instruction). An instruction without prefixes is between 1 and 6 bytes.

Instruction	Machine code	Bytes	Structure
`INC CX`	`41`	1	opcode only
`MOV AX, BX`	`8B C3`	2	opcode + ModR/M
`ADD AL, 7Ch`	`04 7C`	2	short AL-specific code + 8-bit immediate
`MOV BX, 4050h`	`BB 50 40`	3	opcode + 16-bit immediate
`MOV WORD PTR [BX+1234h], 5678h`	`C7 87 34 12 78 56`	6	opcode + ModR/M + 16-bit displacement + 16-bit immediate

Table 2.1.8: Instructions of different lengths.

The shortest instruction in Table 2.1.8 is `INC CX`. On the 8086 the opcodes of the instructions that increment a 16-bit register by one start at 40h, and the register number is added to the opcode: 40 for AX, 41 for CX, 42 for DX, 43 for BX. Because the register is inside the opcode, no separate byte is needed. In the same way, in `MOV BX, 4050h` the opcode BB means "load a 16-bit constant into BX" (B8 + 3, the number of BX). The constant follows.

Immediate values are written to memory in **little-endian** order: low byte first, then the high byte. 4050h sits in memory in the order `50 40`. The same rule holds for displacements: the displacement 1234h is written `34 12` and the immediate 5678h is written `78 56`.

`ADD AL, 7Ch` uses one of the short codes the 8086 reserves for AL and AX. The general form would take three bytes, `80 C0 7C`; the AL-specific opcode `04` needs no ModR/M byte, and the instruction drops to two bytes. This privilege of the accumulator (AX/AL) is meant to shorten frequent operations.

When one of the operands is a general register or memory, the ModR/M byte comes into play. This byte is split into three fields:

> ModR/M = mod (2 bits) · reg (3 bits) · r/m (3 bits)

- **mod** says how to interpret the r/m field: if 11, r/m is a register; if 00, 01, or 10, it is a memory operand, and there is no displacement, an 8-bit displacement, or a 16-bit displacement, respectively.
- **reg** gives the register of one of the operands: 000 AX, 001 CX, 010 DX, 011 BX, 100 SP, 101 BP, 110 SI, 111 DI (in 8-bit instructions AL, CL, DL, BL, AH, CH, DH, BH).
- **r/m** gives the second register if mod = 11, otherwise the address form: 000 [BX+SI], 001 [BX+DI], 010 [BP+SI], 011 [BP+DI], 100 [SI], 101 [DI], 110 [BP] (a direct address when mod = 00), 111 [BX].

Example 2.1.4

Let us decode two ModR/M bytes.

1. `8B C3`: the opcode 8B means “move r/m16 to reg16” (MOV r16, r/m16). C3h = 1100 0011 = 11 000 011. mod = 11 (register), reg = 000 (AX), r/m = 011 (BX). The instruction is `MOV AX, BX`.
2. `C7 87 …`: the opcode C7 means “write a 16-bit constant to r/m16”; the reg field here is not a register but a subtype of the operation and must be 000. 87h = 1000 0111 = 10 000 111. mod = 10 (16-bit displacement), r/m = 111 ([BX]). The displacement `34 12` = 1234h and the immediate `78 56` = 5678h follow. The instruction is `MOV WORD PTR [BX+1234h], 5678h`.

If `89 D8` is decoded the same way: 89 means “move reg16 to r/m16,” D8h = 11 011 000, reg = BX, r/m = AX. This is also `MOV AX, BX`. A register-to-register MOV has two equivalent encodings.

2.1.6.2 Code trace: bytes and IP

The following program runs the instructions in Table 2.1.8 in turn. The purpose of the trace is to see how many bytes each instruction takes in memory and how IP advances accordingly.

```
org 100h
inc cx
mov bx, 4050h
mov ax, bx
add al, 7Ch
mov word ptr [bx+1234h], 5678h
ret
```

The program starts at 0100h and occupies the range 0100h–010Eh of the code segment, 15 bytes in total: `41 BB 50 40 8B C3 04 7C C7 87 34 12 78 56 C3`. `inc cx` consists of the opcode alone; CX = 1 and IP advances to 0101h. `mov bx, 4050h` is three bytes (BB 50 40); IP = 0104h. `mov ax, bx` is two bytes (8B C3); AX = 4050h, IP = 0106h.

`add al, 7Ch` is the AL-specific short code (04 7C). AL = 50h + 7Ch = CCh; in decimal 80 + 124 = 204. Since AH has not changed, AX = 40CCh. IP = 0108h. The six-byte `mov word ptr [bx+1234h], 5678h` computes the effective address BX + 1234h = 4050h + 1234h = 5284h and writes 5678h there. In memory DS:5284h holds 78 and DS:5285h holds 56. IP = 010Eh. The last instruction, `ret`, is a single byte (C3).

Instruction	Bytes	Length	IP	CX	BX	AX	Memory
`inc cx`	41	1	0101h	0001h	0000h	0000h	—
`mov bx, 4050h`	BB 50 40	3	0104h	0001h	4050h	0000h	—
`mov ax, bx`	8B C3	2	0106h	0001h	4050h	4050h	—
`add al, 7Ch`	04 7C	2	0108h	0001h	4050h	40CCh	—
`mov word ptr [bx+1234h], 5678h`	C7 87 34 12 78 56	6	010Eh	0001h	4050h	40CCh	DS:5284h = 78, DS:5285h = 56
`ret`	C3	1	0000h	0001h	4050h	40CCh	—

Table 2.1.9: Trace of the instruction-length program. IP is the value after the instruction has executed.

Figure 2.1.19: On the left `mov bx, 4050h` : 3 bytes, IP from 0101h to 0104h. On the right the six-byte `mov word ptr [bx+1234h], 5678h` : IP from 0108h to 010Eh; 5678h is written to DS:5284h in little-endian order.

The IP column of Table 2.1.9 is obtained by adding the length of each instruction to the previous IP: 0100h + 1 = 0101h, + 3 = 0104h, + 2 = 0106h, + 2 = 0108h, + 6 = 010Eh. In the last row IP becomes 0000h. `RET` is a branch: into IP it loads not a sequential value but the word at the top of the stack. When a .COM program starts, DOS puts 0000h on the stack; at offset 0000h of the segment, in front of the program, is the 256-byte PSP header, whose first instruction terminates the program. So `RET` returns control to the operating system.

Six instructions took 15 bytes: the lengths varied between 1 and 6 bytes. This is why a 6-byte queue is enough. The machine-code listing and trace emu8086 produces give the bytes and values in the table exactly.

Common mistake

When reading machine code, interpreting `BB 50 40` as "load 5040h into BX" is wrong. The 8086 is little-endian: the first byte is the low byte. The correct reading is 4050h. The same mistake is made with memory dumps: the bytes `78 56` starting at DS:5284h are the word 5678h.

2.1.7 The register set

The most important tool an assembly programmer has is the registers. Every instruction either reads from a register, writes to a register, or accesses the memory location a register points to. This section groups the 8086's registers and explains the meaning of word size. The two groups that play a role in address generation, the segment registers and IP, are treated in detail.

2.1.7.1 The 14 registers

Definition 2.1.10

A **register** is a very fast, small memory cell inside the processor. All the registers a processor offers the programmer are called the **register set**. The 8086's register set has 14 registers of 16 bits.

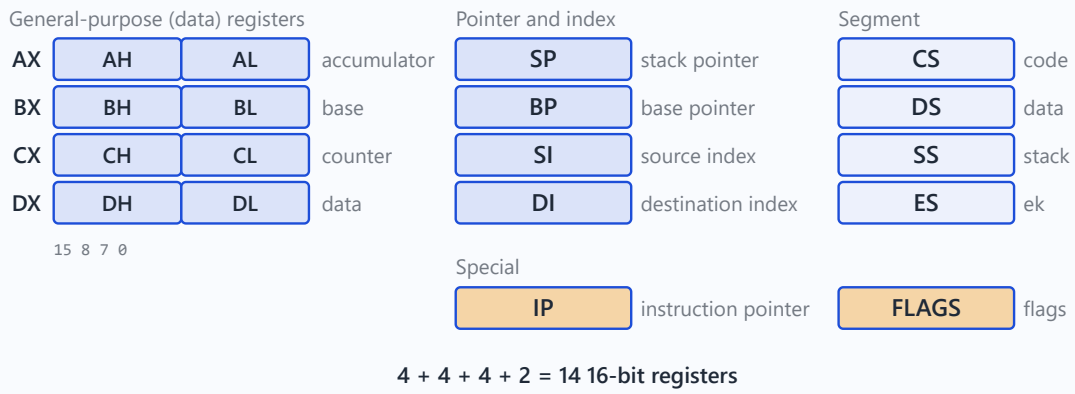


Figure 2.1.20: The 8086's 14 registers: four general-purpose (data) registers, four pointer and index registers, four segment registers, and two special registers (IP, FLAGS).

Figure 2.1.20 divides the registers into four groups: 4 + 4 + 4 + 2 = 14.

- **The general-purpose registers** (AX, BX, CX, DX) store data and take part in arithmetic and logic operations. Each can be split into two 8-bit halves: the high half of AX is AH and the low half is AL. AX is called the accumulator; multiply, divide, and I/O instructions use it. BX is the base register and can hold an offset in memory addressing. CX is the counter; loop and repeat instructions decrement it. DX is the data register; in multiply and divide it becomes the upper half of AX, and in I/O it holds the port number.
- **The pointer and index registers** (SP, BP, SI, DI) point to locations in memory. SP is the stack pointer; it holds the offset of the top of the stack. BP is the base pointer; it accesses parameters and local variables on the stack. SI is the source index and DI the destination index; they are used in array and string operations. These registers cannot be split in two.
- **The segment registers** (CS, DS, SS, ES) point to the starts of the four active segments in memory.
- **The special registers**: IP points to the next instruction, and FLAGS holds the result of the last operation and the processor's control bits.

That the general-purpose registers can be split in two is an important detail. A value written to AL changes the low byte of AX and does not change AH. This behavior was seen in the earlier code traces: `sub al, 5` changed only AL, and `mov bl, 4` changed only BL. The register names describe this structure too: H points to the high byte and L to the low byte.

2.1.7.2 Word size

The 8086 is a 16-bit **integer** processor. Each instruction works on a single data element; there are no vector instructions that process several data at once. Floating-point arithmetic is not in hardware either; a separate coprocessor, the 8087, can be added to the system for this job. Without an 8087, floating-point operations are done in software with long subroutines built from integer instructions.

The width of the registers inside the chip determines the amount of data the processor can work on at once, that is, the **word size**. On the 8086 the word is 16 bits. This width also determines the value ranges:

Width	Example	Unsigned range	Signed range
8 bits	AL, BH	0 … 255 (FFh)	−128 … +127
16 bit	AX, SI	0 … 65,535 (FFFFh)	−32,768 … +32,767
32 bits	DX:AX	0 … 4,294,967,295	−2,147,483,648 … +2,147,483,647

Table 2.1.10: Value ranges by register width.

A 32-bit value does not fit in a single register; it is held in a two-register pair such as DX:AX. Multiply and divide use this pair automatically. In addition and subtraction it is done in two steps: first the low words are added with `ADD`, then the high words are added with carry using `ADC`.

Registers are the fastest storage inside the processor. Accessing a register needs no bus cycle; the value is immediately ready on the internal data bus. Accessing memory, however, takes at least one bus cycle, that is, at least four clock cycles. This difference is the reason a good assembly program keeps frequently used values in registers.

2.1.7.3 The segment registers

> **Definition 2.1.11**
>
> A **segment** is a block of 8086 memory that starts on a 16-byte boundary and is at most 64 KB long. A **segment register** holds the starting address of a segment divided by 16; the physical start of the segment, that is, the **segment base**, is the value in the register multiplied by 10h.

Segments are in memory; segment registers are inside the processor. This distinction matters: a segment register “points to” a region of memory and does not contain it. When the register is changed, the region it points to shifts, but the bytes in memory stay where they are. Each of the four segment registers has a specific role:

Register	Name	Segment it points to	Offset used with it
CS	code segment	the program’s instructions	IP (CS:IP is the next instruction)
DS	data segment	variables and data	BX, SI, DI, direct address
SS	stack segment	stack	SP, BP (SS:SP, SS:BP)
ES	extra segment	extra data; string destination	DI (ES:DI)

Table 2.1.11: The segment registers and their roles.

There are three rules for using the segment registers. These rules come from the segment registers being tied to the BIU and not the ALU, and from the narrow room reserved for them in instruction encoding.

1. **An immediate value cannot be loaded.** `MOV DS, 2000h` is invalid; the 8086 instruction set has no opcode that loads a constant into a segment register. The value goes through a general register first: `MOV AX, 2000h`, then `MOV DS, AX`. The value can also be taken from memory: `MOV DS, [BX]` is valid.
2. **CS is not changed with MOV.** When CS changes, the next instruction is taken from somewhere else; that is a branch. CS changes only with far JMP, far CALL, RETF, INT, and IRET; these instructions set IP at the same time. The assembler does not accept the line `MOV CS, AX`.
3. **There is no direct MOV from segment to segment.** `MOV ES, DS` is invalid. The value goes either through a general register (`MOV AX, DS` / `MOV ES, AX`) or through the stack (`PUSH DS` / `POP ES`).

Common mistake

Writing `MOV DS, 1000h` to set the data segment does not assemble. The assembler gives an error on this line. The correct notation is two lines: `MOV AX, 1000h` and `MOV DS, AX`. This pattern appears in every segment setup and is the reason that in all the code traces the value goes to the segment registers through AX.

2.1.7.4 The instruction pointer (IP)

Definition 2.1.12

The **instruction pointer** (IP) is the 16-bit register that holds the offset of the next instruction to be executed in the code segment. The physical address of the next instruction is CS × 10h + IP.

IP advances as each instruction is fetched; by the time an instruction executes, IP has already grown by the length of that instruction. This behavior was seen in the instruction-length trace: the one-byte `INC CX` advanced IP by 1, and the six-byte `MOV` by 6. Branch instructions (JMP, conditional jumps, CALL, RET, LOOP, INT), on the other hand, load IP with a new, non-sequential value. The flow of the program changes with this value.

The programmer cannot read or write IP directly; there is no instruction like `MOV IP, …`. The only way to change IP is a branch. To find out the value of IP, a small trick is used: the `CALL` instruction pushes the return address, that is, the IP of the next instruction, onto the stack; right afterward this value can be taken into a register with `POP`.

Example 2.1.5

If CS = 4321h and IP = 1000h, at which physical address is the next instruction?

1. CS × 10h: multiplying a hexadecimal number by 10h is appending a 0 on its right. 4321h × 10h = 43210h.
2. Add IP, right-aligned: 43210h + 01000h.
3. The digits are added from the right: 0 + 0 = 0, 1 + 0 = 1, 2 + 0 = 2, 3 + 1 = 4, 4 + 0 = 4. The result is 44210h; no carry occurred.

The next instruction is at physical address 44210h. This address is 20 bits and goes straight onto the address bus.

2.1.8 Segmented memory

What sets the 8086 apart from many other processors is that it addresses memory through segments. A programmer never writes a 20-bit address directly; every memory access arises from the combination of a segment register and an offset. This section explains how this combination is made, why the same byte can be named in many ways, what happens at the 1 MB limit, and which access uses which segment.

2.1.8.1 Logical and physical addresses

The 8086 addresses memory in a **segmented** way. Every memory address is given by two 16-bit parts: the segment and the offset. In writing, the two parts are separated by a colon: `2000h:0010h`. The segment register points to the start of the segment; the offset is the distance of the wanted byte from the start of the segment and ranges from 0000h to FFFFh. The two parts are combined in the address adder in the BIU to give the 20-bit physical address.

Definition 2.1.13

A **logical address** is the `segment:offset` pair the program uses. The **physical address** (PA) is the real 20-bit address that goes onto the address bus and is computed as:

> PA = segment × 10h + offset

If the sum exceeds 20 bits, the 8086 uses only the low 20 bits.

Figure 2.1.21: The address adder: the segment value 2000h is shifted left 4 bits to become the segment base 20000h; the offset 0010h is added; the physical address 20010h goes out on lines A19–A0.

Multiplying by 10h equals shifting a binary number left 4 bits; no multiplier is needed in hardware. The bits of the segment value are connected to the adder's input shifted four positions, and four 0s are appended on the right. In this way the 16-bit segment value becomes a 20-bit base. The offset is added unshifted, right-aligned. Figure 2.1.21 shows this arrangement.

As a result of the shift, the last hexadecimal digit of a segment base is always 0. Segment bases are therefore multiples of 16. These 16-byte boundaries in memory are called **paragraphs**. Each of the 65,536 values a segment register can take points to a paragraph boundary: 1,048,576 / 16 = 65,536.

2.1.8.2 The physical address computation step by step

The computation is always done in three steps: a 0 is appended to the right of the segment value, the offset is written below it right-aligned, and the two numbers are added in hexadecimal.

Example 2.1.6

Let the segment register be 1000h and the offset F000h.

1. Segment base: 1000h × 10h = 10000h.
2. The offset F000h, that is, 0F000h.
3. Sum: 10000h + 0F000h = 1F000h.

Since the offset can be at most FFFFh, this segment covers the range 10000h–1FFFFh. The length of the range is 10000h bytes, that is, 64 KB.

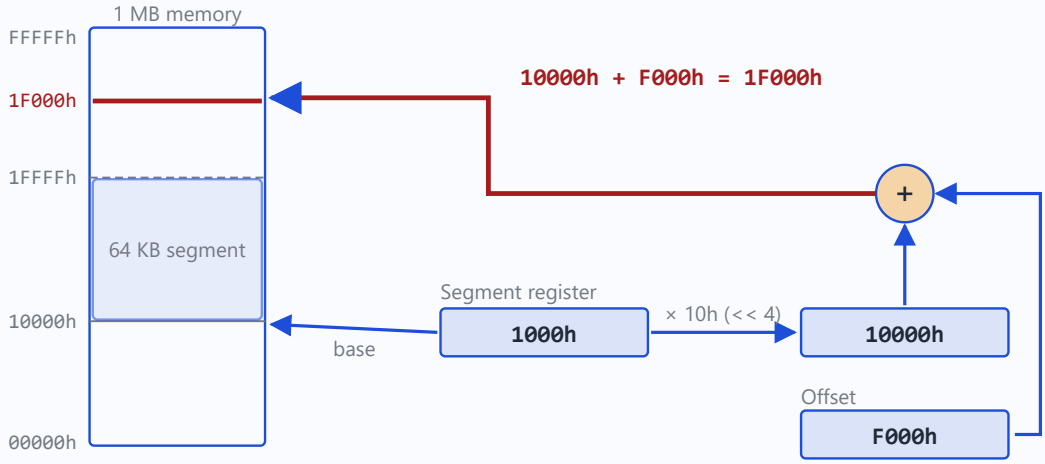


Figure 2.1.22: The 1000h in the segment register is shifted left 4 bits to become the base 10000h; adding the offset F000h gives the physical address 1F000h. The 64 KB segment occupies the range 10000h–1FFFFh.

Example 2.1.7

If CS = F000h and IP = FFF0h, what is the physical address of the instruction?

1. F000h × 10h = F0000h.
2. IP = FFF0h is right-aligned: 0FFF0h.
3. F0000h + 0FFF0h = FFFF0h.

FFFF0h is the start of the last 16 bytes of the 1 MB. When the 8086 is reset it loads CS = FFFFh and IP = 0000h: FFFF0h + 0000h = FFFF0h. Two different pairs point to the same byte; the processor fetches its first instruction from here in every case. In computers this address is in the BIOS ROM, and there is a far jump to the BIOS startup code.

Figure 2.1.23: Digit-by-digit addition. On the left 43210h + 1000h = 44210h, on the right F0000h + FFF0h = FFFF0h. A 0 is appended to the right of the segment; the offset is right-aligned.

Common mistake

Two mistakes are common. The first is to forget to multiply the segment value by 10h and get 4321h + 1000h = 5321h; this is a 16-bit number and cannot be a physical address. The second is to apply the shift to the wrong part: shifting the offset and getting 4321h + 10000h =

14321h. The rule is single: the **segment** is shifted, and the offset is added as it is.

2.1.8.3 The physical address calculator

The interactive calculator in the slides produces the same three rows for any segment and offset entered: the segment, segment × 10h, and the sum. It also draws the 64 KB range of the segment and the position of the physical address on a 1 MB memory map and gives the normalized form of the address. For 4321h:1000h, for example, the segment range is 43210h–5320Fh, the physical address is 44210h, and the normalized form is 4421h:0000h. If the sum exceeds 1 MB, the calculator drops the 21st bit and shows that the result wraps around.

2.1.8.4 Memory segmentation

The 20-bit address bus addresses 1 MB; a segment is at most 64 KB, because the offset is 16 bits. At any moment four segments are active: those CS, DS, SS, and ES point to. Without changing the segment registers, at most 4 × 64 KB = 256 KB can be accessed directly. To access another region of memory, a new value is loaded into a segment register.

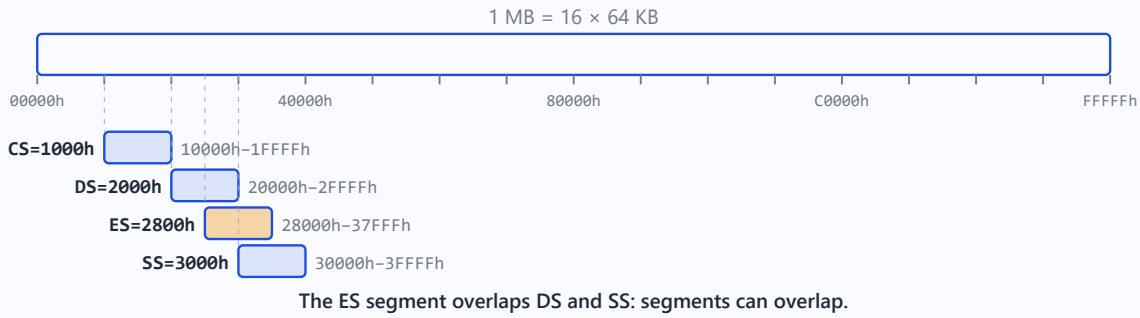


Figure 2.1.24: Four active segments in 1 MB of memory: CS = 1000h (10000h–1FFFFh), DS = 2000h (20000h–2FFFFh), ES = 2800h (28000h–37FFFh), SS = 3000h (30000h–3FFFFh). The ES segment overlaps DS and SS.

Segments do not divide memory into 16 fixed parts. 1 MB / 64 KB = 16 is only arithmetic; on the 8086 segment starts are not fixed. A segment can start at every paragraph boundary; there are 65,536 possible starts. So segments can be disjoint, and they can also overlap. In Figure 2.1.24 the segment ES = 2800h starts at 28000h and ends at 37FFFh. The first half of this range is the same bytes as the second half of the DS segment (28000h–2FFFFh), and the second half is the same bytes as the first half of the SS segment (30000h–37FFFh). The same byte can be accessed through both DS and ES with different offsets: physical 2A000h is read with DS:A000h and also with ES:2000h.

Example 2.1.8

Let us find the range of the segment ES = 2800h and see whether address 30010h can be accessed with ES.

1. Segment base: 2800h × 10h = 28000h.
2. Last byte: 28000h + FFFFh = 37FFFh.
3. 30010h is in this range. The offset: 30010h − 28000h = 8010h. Since it is less than FFFFh, it is valid: `ES:8010h`.

The same byte is also written as `SS:0010h` with SS = 3000h. With DS = 2000h it cannot be accessed: 30010h − 20000h = 10010h, which does not fit in 16 bits.

2.1.8.5 Why segmentation?

Segmentation was a cheap way to address 1 MB in a 16-bit processor. But alongside the conveniences it brought, it has burdens.

Pros	Cons
20-bit (1 MB) memory is addressed with 16-bit registers.	A single array or structure cannot exceed 64 KB; if it does, the segment has to be changed.
Code, data, and stack are separated from one another; the program layout becomes clear.	The same physical address has many segment:offset notations; pointer comparison becomes harder.
Relocatability: a program can be loaded elsewhere in memory by changing only the segment registers.	There is no protection between segments; a faulty write can corrupt another segment.
Instructions stay short: most addresses are written with a 16-bit offset.	Address computation adds an extra mental load for the programmer.

Table 2.1.12: Pros and cons of segmentation.

Relocatability is especially valuable. A .COM program uses all the addresses inside it as offsets from the start of the segment. Whichever paragraph the operating system loads the program into, it is enough to set CS, DS, ES, and SS to that paragraph; no byte of the program needs to change. emu8086 loads .COM programs at segment 0700h: CS = DS = ES = SS = 0700h and the program starts at 0700h:0100h.

The cons became heavier with time. For data structures larger than 64 KB, programmers had to change segments by hand; compilers defined “near” and “far” pointer types. The lack of protection made multitasking operating systems hard. The 80286’s protected mode and the 80386’s 32-bit flat address space came to solve these problems.

2.1.8.6 The same address, different notations

Because segments can overlap, a physical address is not indicated by a single logical address. If the segment is increased by one paragraph the base moves 16 bytes forward; if the offset is decreased by 16 the sum does not change. Table 2.1.13 shows this with numbers.

Logical address	Calculation	Physical address
`1234h:0005h`	12340h + 0005h	**12345h**
`1000h:2345h`	10000h + 2345h	**12345h**
`1200h:0345h`	12000h + 0345h	**12345h**
`0234h:FFF5h`	02340h + FFF5h	**12335h** (not 12345h)
`FFFFh:0010h`	FFFF0h + 0010h = 100000h	**00000h** (wraps to 20 bits on the 8086)

Table 2.1.13: Different logical addresses and their physical counterparts.

The first three rows point to the same byte. The fourth row is misleading: the segment 0234h is 1000h smaller than 1234h, that is, the base is 10000h bytes behind. To reach the same byte the offset would have to be 10005h, but that does not fit in 16 bits. With FFF5h one falls 16 bytes behind. In the last row the sum exceeds 1 MB and wraps to 20 bits.

Definition 2.1.14

A **normalized address** is a logical address whose offset is between 0h and Fh. The first four digits of the five-digit physical address give the segment and the last digit gives the offset: 12345h → `1234h:0005h`.

A physical address has at most 4096 different notations. While the offset is decreased by 16 each time, the segment is increased by one; as long as the offset stays in the range 0000h–FFFFh, every step gives a valid notation. Because the last digit of the offset stays fixed, the number of values that can be chosen is 65,536 / 16 = 4096. For 12345h the notations extend from `0235h:FFF5h` to `1234h:0005h`: 1234h − 0235h + 1 = 1000h = 4096. For addresses near the very start of memory the number of notations is smaller, because the segment cannot go below 0.

Example 2.1.9

Let us normalize the physical address 3A2C5h and find the required offset if DS = 3000h.

1. The first four digits are the segment, the last digit the offset: `3A2Ch:0005h`. Check: 3A2C0h + 5h = 3A2C5h.
2. If DS = 3000h the segment base is 30000h. Offset = 3A2C5h − 30000h = A2C5h.
3. A2C5h ≤ FFFFh, so it is valid: `3000h:A2C5h`.

The same question can be asked in reverse for CS = 2F00h and IP = 01A4h: PA = 2F000h + 01A4h = 2F1A4h; the normalized notation is `2F1Ah:0004h`. In emu8086 a byte written to address 2F00h:01A4h is read back exactly with `ES:[0004h]` when ES = 2F1Ah.

2.1.8.7 Code trace: the address wraps around at 1 MB

The sum segment × 10h + offset can exceed FFFFFh: the largest case is FFFF0h + FFFFh = 10FFEFh, which takes 21 bits. The 8086 has only 20 address lines (A19–A0). The 21st bit does not go onto the address bus, that is, it is dropped. As a result the address wraps around to the start of memory. The following program shows this behavior.

```
org 100h
mov ax, 0
mov ds, ax
mov byte ptr [0500h], 77h
mov ax, 0FFFFh
mov ds, ax
mov bl, [0510h]    ; FFFF:0510
ret
```

The first two instructions make DS = 0000h; the segment is at the very start of memory. `mov byte ptr [0500h], 77h` writes 77h to address 0000h:0500h, that is, physical 00500h. The `byte ptr` specifier announces that the data to be written is one byte; the constant 77h by itself does not determine this. Then DS = FFFFh is set. The base of this segment is FFFF0h, the last 16 bytes of the 1 MB. `mov bl, [0510h]` computes the physical address as FFFF0h + 0510h = 100500h. This number is 21 bits. The 8086's 20 address lines carry only 00500h, and the processor reads the very same byte to which 77h was written in the first step: BL = 77h.

Row	DS	BL	Calculation	byte at 00500h
2–3 `mov ax,0 / mov ds,ax`	0000h	00h	—	00h
4 `mov byte ptr [0500h], 77h`	0000h	00h	00000h + 0500h = 00500h	77h
5–6 `mov ax,0FFFFh / mov ds,ax`	FFFFh	00h	base FFFF0h	77h
7 `mov bl, [0510h]`	FFFFh	77h	FFFF0h + 0510h = 100500h → 00500h	77h

Table 2.1.14: Trace of the wraparound program.

Figure 2.1.25: `mov bl, [0510h]` : FFFF0h + 0510h = 100500h. The 21st bit is dropped, the address wraps to 00500h, and BL = 77h.

This rule means that in the segment FFFFh, all addresses from FFFFh:0010h to FFFFh:FFFFh fall in the region 00000h–0FFEFh at the start of memory. Wraparound on the 8086 is not a bug but the natural consequence of the 20-bit address bus. Some old programs were written relying on this behavior.

The situation changed with the 80286. The 80286 has 24 address lines, and even in real mode the 21st line (A20) can be enabled. In that case FFFFh:0510h really reaches 100500h, just above 1 MB. To keep compatibility with old programs, a circuit that turns the A20 line off and on was added to IBM PC/AT computers. With A20 enabled, the 65,520 bytes above 1 MB (FFFF0h + FFFFh − FFFFFh = FFF0h) can be accessed in real mode through the segment FFFFh.

Common mistake

Writing 100500h as the physical address for FFFFh:0510h is wrong on the 8086; this number does not fit in 20 bits. If the sum exceeds 1 MB the top bit is dropped and 100000h is subtracted: 100500h → 00500h. The same check must be made at the end of every physical address computation.

2.1.8.8 Default segments and the segment prefix

When an instruction writes only an offset, for example `mov al, [bx]`, the processor itself decides which segment register to use. This decision depends on the type of access and on the register used in the address computation.

Definition 2.1.15

The **default segment** is the segment the processor uses in a memory access when no segment is specified. The **segment override prefix** is a one-byte prefix added to the start of an instruction that selects another segment register instead of the default: 26h for ES, 2Eh for CS, 36h for SS, 3Eh for DS.

Access type	Segment	Offset source	Can it be overridden with a prefix?
Instruction fetch	CS	IP	no
Stack operations (PUSH, POP, CALL, RET)	SS	SP	no
Addressing with BP `[BP+…]`	SS	effective address	yes: `DS:[BP]`
General data `[BX]`, `[SI]`, `[DI]`, `[1234h]`	DS	effective address	yes: `ES:[BX]`
String source (MOVSB, LODSB)	DS	SI	yes
String destination (MOVSB, STOSB)	ES	DI	no

Table 2.1.15: Default segments by access type.

The logic of the rules in Table 2.1.15 comes from the roles of the registers. Instructions are always in the code segment. The stack is always in the stack segment. Because BP is designed to reach parameters and local variables on the stack, every address that contains BP uses SS by default: `[BP]`, `[BP+SI]`, `[BP+DI+4]`. All the remaining data accesses use DS. String instructions read the source from DS:SI and write the destination through ES:DI; so a block can be copied between two different segments.

To change the default, a segment prefix is written: `MOV AL, ES:[BX]` computes the physical address as ES × 10h + BX. The prefix adds a byte to the instruction (26h here) and lengthens execution by two clock cycles. Instruction fetch, stack operations, and the string destination cannot be changed with a prefix; in these accesses the processor behaves as if there were no prefix.

Example 2.1.10

With SS = 2000h, DS = 3000h, BP = 0100h, and SI = 0020h, to which address does `mov byte ptr [bp+si+4], 0AAh` write? What changes if the same instruction is written with the `ds:` prefix?

1. Effective address: BP + SI + 4 = 0100h + 0020h + 0004h = 0124h.
2. The address contains BP; the default segment is SS. PA = 20000h + 0124h = 20124h.
3. With the `ds:` prefix, PA = 30000h + 0124h = 30124h. The machine code grows from `C6 42 04 AA` to `3E C6 42 04 AA`.

emu8086 shows that the two instructions write to addresses 20124h and 30124h and that the instruction with the prefix takes 5 bytes.

Common mistake

Thinking that `[BX]` and `[BP]` use the same segment is a common mistake. Even if BX and BP hold the same value, `[BX]` is computed relative to DS and `[BP]` relative to SS, and the two instructions access different bytes. In a .COM program the difference is invisible because DS = SS; the mistake shows up when the segments are set separately.

2.1.8.9 Code trace: the same offset, three segments

The following program uses the same offset, 0010h, with three different segments and accesses three different bytes. Because the program starts as a .COM program, initially CS = DS = ES = SS = 0700h.

```
org 100h
mov ax, 2000h
mov ds, ax
mov ax, 3000h
mov es, ax
mov bx, 0010h
mov bp, 0010h
mov byte ptr [bx], 11h    ; DS
mov byte ptr [bp], 22h    ; SS
mov byte ptr es:[bx], 33h ; ES
mov al, [bx]              ; DS
mov ah, [bp]              ; SS
ret
```

The first four instructions set DS = 2000h and ES = 3000h; the values pass through AX. SS stays at 0700h, untouched. This creates three different segment bases: 07000h (SS), 20000h (DS), 30000h (ES). BX and BP hold the same offset, 0010h. Which byte is accessed will be determined not by the offset but by the segment used.

`mov byte ptr [bx], 11h` uses DS by default: it writes 11h to address 20000h + 0010h = 20010h. `mov byte ptr [bp], 22h` uses SS because it contains BP: it writes 22h to address 07000h + 0010h = 07010h. The `es:` prefix in `mov byte ptr es:[bx], 33h` selects ES instead of DS: it writes 33h to address 30000h + 0010h = 30010h; the machine code has the 26h prefix at its start. The last two instructions read: `mov al, [bx]` again reads DS:0010h and AL = 11h, not the 33h at ES:0010h. `mov ah, [bp]` reads SS:0010h and AH = 22h. The result is AX = 2211h.

Row	Segment	Physical address	Operation	AX
2–5	—	—	DS = 2000h, ES = 3000h, SS = 0700h	3000h
6–7	—	—	BX = BP = 0010h	3000h
8 `[bx] ← 11h`	DS	20010h	write 11h	3000h
9 `[bp] ← 22h`	SS	07010h	write 22h	3000h
10 `es:[bx] ← 33h`	ES	30010h	write 33h	3000h
11 `al ← [bx]`	DS	20010h	read 11h → AL	3011h
12 `ah ← [bp]`	SS	07010h	read 22h → AH	2211h

Table 2.1.16: Trace of the three-segment program.

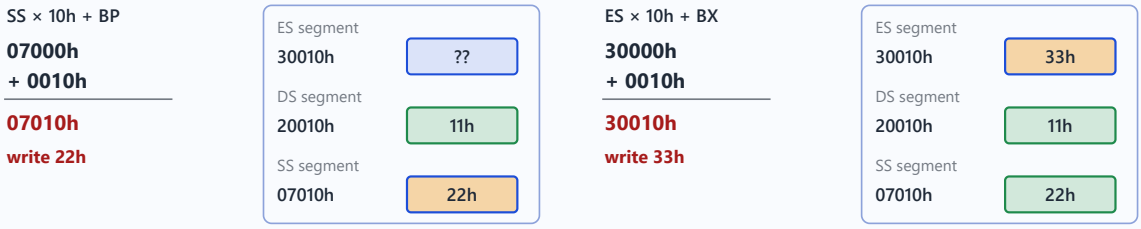


Figure 2.1.26: On the left `[bp]` : SS × 10h + BP = 22h is written to 07010h. On the right `es:[bx]` : the prefix selects ES, and 33h is written to 30010h. The same offset corresponds to three different bytes.

After row 11, AX = 3011h: AL took the new value, while AH kept the 30h left over from the earlier `mov ax, 3000h`. Row 12 makes AH 22h. This detail shows once more that writing to an 8-bit register does not touch the other half of the 16-bit register.

2.1.8.10 Code trace: access with segment:offset

The last trace shows that the same byte can be read with two different logical addresses and where the last byte of a segment is.

```
mov ax, 2000h
mov ds, ax          ; DS = 2000h
mov bx, 0010h
mov byte ptr [bx], 41h
mov ax, 2001h
mov ds, ax          ; +16 bytes
mov bx, 0000h
mov al, [bx]        ; same byte
mov si, 0FFFFh
mov ah, [si]        ; last byte
```

With DS = 2000h the data segment is the range 20000h–2FFFFh. `mov byte ptr [bx], 41h` writes 41h to 20000h + 0010h = 20010h; 41h is the ASCII code of the character 'A'. Then DS = 2001h is set. The segment value grew by 1, but the base moved forward 16 bytes, that is, one paragraph: the new segment is the range 20010h–3000Fh. With BX = 0000h, `mov al, [bx]` reads 20010h + 0000h = 20010h and AL = 41h. `2000h:0010h` and `2001h:0000h` are the same byte. The last two instructions use the largest offset of a segment with SI = FFFFh: 20010h + FFFFh = 3000Fh. This is the last byte of the segment DS = 2001h, and AH takes the value there.

Row	AX	DS	BX	SI	FA	[20010h]
1 `mov ax, 2000h`	2000h	?	0000h	0000h	—	?
2 `mov ds, ax`	2000h	2000h	0000h	0000h	—	?
3 `mov bx, 0010h`	2000h	2000h	0010h	0000h	—	?
4 `mov byte ptr [bx], 41h`	2000h	2000h	0010h	0000h	20010h	41h

Row	AX	DS	BX	SI	FA	[20010h]
5 `mov ax, 2001h`	2001h	2000h	0010h	0000h	—	41h
6 `mov ds, ax`	2001h	2001h	0010h	0000h	—	41h
7 `mov bx, 0000h`	2001h	2001h	0000h	0000h	—	41h
8 `mov al, [bx]`	2041h	2001h	0000h	0000h	20010h	41h
9 `mov si, 0FFFFh`	2041h	2001h	0000h	FFFFh	—	41h
10 `mov ah, [si]`	??41h	2001h	0000h	FFFFh	3000Fh	41h

Table 2.1.17: Trace of the segment:offset program. The final value of AH depends on the byte at 3000Fh.

In the checker, when 99h is placed in advance at 3000Fh, the program ends with AX = 9941h; AH really takes the byte at 3000Fh. That AX is 2041h in the eighth row shows the same principle: AL became 41h and AH kept the 20h left over from the fifth row.

2.1.9 Summary

- The **8086** is a 16-bit microprocessor that came out in 1978 and is the first member of the x86 family. Its registers, ALU, and data bus are 16 bits and its address bus is 20 bits: it addresses 2^{20} bytes = 1 MB of memory. It is a CISC design; its instructions are 1–6 bytes long. All later x86 processors start in **real mode** at power-on, behaving like an 8086.
- There is no memory inside the microprocessor; to work it needs the 8284A clock generator, address latches (8282), data bus transceivers (8286), memory, and I/O units. A **microcontroller** puts these on a single chip.
- The **system bus** consists of three buses: the unidirectional 20-bit address bus, the bidirectional 16-bit data bus, and the control bus (RD', WR', M/IO', ALE, INTR, READY). M/IO' separates memory from I/O; I/O ports are in a separate 16-bit address space.
- A **bus cycle** takes at least four clock cycles (T1–T4); 800 ns at 5 MHz. Memory is much slower than registers.
- The 8086 consists of two units. The **BIU** generates addresses, fetches instructions, and runs all the bus cycles; the segment registers, IP, the address adder, and the queue are in it. The **EU** decodes and executes instructions; the general registers, ALU, flags, and control system are in it.
- The 6-byte **prefetch queue** is a FIFO. When at least 2 bytes are freed, the BIU fetches a word. The overlap of fetching and execution is a two-stage **pipeline** and roughly doubles throughput. A branch flushes the queue and takes back part of this gain.
- The **ALU** does 8- and 16-bit arithmetic and logic operations and updates the status flags (CF, ZF, SF, OF, PF, AF). `MUL` widens the result (AL × BL → AX), `DIV` divides DX:AX; `MOV` does not change the flags.

- A machine instruction consists of an opcode, a ModR/M byte (mod · reg · r/m), a displacement, and an immediate. Multibyte constants are written in **little-endian** order. IP advances after each instruction by the length of that instruction; branches load IP with a new value.
- The 8086 has **14 registers**: AX, BX, CX, DX · SP, BP, SI, DI · CS, DS, SS, ES · IP, FLAGS. An immediate value cannot be loaded into a segment register, CS cannot be changed with MOV, and there is no direct MOV from segment to segment.
- **Physical address** = segment × 10h + offset. A segment is at most 64 KB, starts on a paragraph (16-byte) boundary, and segments can overlap. The same address has at most 4096 notations; in the **normalized** notation the offset is between 0h and Fh.
- If the sum exceeds 1 MB, on the 8086 the 21st bit is dropped and the address **wraps around** (FFFFh:0510h → 00500h). On an 80286 with the A20 line enabled, the same address reaches above 1 MB.
- **Default segments**: instruction fetch CS:IP, stack SS:SP, BP-based addresses SS, other data accesses DS, string destination ES:DI. A **segment prefix** (26h, 2Eh, 36h, 3Eh) changes the default in data accesses.

2.1.10 Key terms

microprocessor microcontroller (MCU) CISC (complex instruction set computer) real mode clock cycle clock generator latch buffer (bus transceiver) bus system bus address bus data bus control bus bus cycle bus interface unit (BIU) execution unit (EU) prefetch prefetch queue pipeline arithmetic logic unit (ALU) flag status flag instruction register instruction decoder control unit opcode ModR/M byte displacement immediate value effective address (EA) little-endian register register set general-purpose register word segment segment register code / data / stack / extra segment offset segment:offset physical address paragraph segmented memory segmentation instruction pointer (IP) default segment segment override prefix

---

2.1.11 Test Yourself

1. The 80286's address bus is 24 bits. How many bytes does it address, and how is its highest address written in hexadecimal? How many times the 8086's address space is it?

   Answer

   2^{24} = 16,777,216 bytes = 16 MB. The highest address is FFFFFFh, with six hexadecimal digits. $2^{24} / 2^{20} = 2^4 = 16$ times.

2. In how many nanoseconds does an 8 MHz 8086-2 complete a bus cycle with no wait states? At this speed, how long does it take at least to read 50 words that start at odd addresses?

   Answer

   The clock cycle is 1 / 8 MHz = 125 ns; the bus cycle is 4 × 125 = 500 ns. A word starting at an odd address needs two cycles: 50 × 2 × 500 ns = 50,000 ns = 50 µs. Had the words been at even addresses the time would have been 25 µs.

3. In the model where fetch takes 2 time units and decode-and-execute takes 2, 8 instructions run and the fourth instruction is a JMP. What are the total times without and with the pipeline?

   Answer

   Without the pipeline, 8 × 4 = 32 units. With the pipeline and no branch it would have been 2 + 8 × 2 = 18 units. The JMP invalidates the fifth instruction fetched during E4 (8–10); the target is fetched at 10–12, and E5 is 12–14, E6 14–16, E7 16–18, E8 18–20. Total 20 units.

4. The machine code of the instruction `ADD CX, [BX+SI+10h]` is `03 48 10`. Split the ModR/M byte into its fields and give the meaning of each field.

   Answer

   48h = 0100 1000 = 01 001 000. mod = 01: a memory operand, with an 8-bit displacement (10h). reg = 001: CX. r/m = 000: [BX+SI]. Since the opcode 03 means “add r/m16 to reg16,” the instruction is CX ← CX + [BX+SI+10h]; since BP is not in the address, the segment is DS. The instruction is 3 bytes.

5. `PUSH AX` executes with SS = 9000h and SP = FFFEh (AX = 1234h). What is the new value of SP, and to which physical addresses are the bytes 34h and 12h written?

   Answer

   PUSH first decrements SP by 2: SP = FFFCh. The word is written to SS:FFFCh in little-endian order. Since the base is 90000h, 34h → 9FFFCh and 12h → 9FFFDh.

6. Find the physical address of the logical address `0ABCDh:1234h` and normalize it.

   Answer

   ABCD0h + 01234h = ACF04h. The sum does not exceed FFFFFh. Normalized notation: the first four digits are the segment and the last digit is the offset → `ACF0h:0004h`.

7. To which physical address does the address `F800h:9000h` correspond on the 8086? What changes

on an 80286 with the A20 line enabled?

> **Answer**
>
> F8000h + 09000h = 101000h; 21 bits. On the 8086 the 21st bit is dropped and the address wraps to 01000h. On an 80286 with A20 enabled, the address really is 101000h, that is, 4 KB above 1 MB.

8. SS = 2000h, DS = 3000h, BP = 0100h, SI = 0020h. Which physical address does `MOV AL, [BP+SI+4]` read? How is the instruction written to read the same data from the DS segment, and how many bytes longer does it become?

> **Answer**
>
> The effective address is 0124h; since it contains BP, the default segment is SS: 20000h + 0124h = 20124h. To read from DS, a segment prefix is written: `MOV AL, DS:[BP+SI+4]` → 30124h. The prefix (3Eh) adds one byte to the instruction.

9. We want to swap the values of DS and ES without disturbing the general-purpose registers. Why can't `MOV ES, DS` be used? Write a solution that works with the stack.

> **Answer**
>
> On the 8086 there is no direct MOV from segment to segment. With the stack: `PUSH DS`, `PUSH ES`, `POP DS`, `POP ES`. Because the stack is LIFO, first ES's value comes off into DS, then DS's old value into ES. Starting with DS = 1111h and ES = 2222h, the result is DS = 2222h and ES = 1111h.

2.1.12 Further reading

- Intel, *The 8086 Family User's Manual* (1979), Chapter 2: the BIU and EU, the queue, segmentation, and instruction encoding are presented in the designer's own words, with timing tables.
- B. B. Brey, *The Intel Microprocessors* (8th edition), Chapters 2 and 3: real-mode memory addressing, default segments, and addressing modes are treated with plenty of examples.
- M. A. Mazidi, J. G. Mazidi, and D. Causey, *The x86 PC: Assembly Language, Design, and Interfacing* (5th edition), Chapter 1: the 8086's internal structure, the pipeline, and the segment:offset computation are explained with DEBUG examples.

2.1.13 References

1. Intel, *8086 16-Bit HMOS Microprocessor*, datasheet (231455).

2. Intel, *The 8086 Family User's Manual*, 1979.

3. Barry B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, 2009.

4. Muhammad Ali Mazidi, Janice Gillispie Mazidi, Danny Causey, *The x86 PC: Assembly Language, Design, and Interfacing*, 5th edition, Pearson, 2010.

5. emu8086 documentation and instruction set reference.

PART I · CHAPTER 2.2

8086 Registers

Most of what an assembly program does happens in a few small memory cells inside the processor. These cells are called registers. The 8086 has fourteen registers, and the name of each describes a job: AX is the accumulator, CX is the counter, SP points to the top of the stack, and IP holds the place of the next instruction. This chapter first explains what a register is and why it is so much faster than memory. It then examines the 8086's register set group by group: the 8-bit and 16-bit uses of the four data registers, how instructions use these registers implicitly, access to the stack and to strings through the pointer and index registers, and the segment registers and IP. The second half of the chapter is devoted to the flags register. It shows with examples in which results the six status flags are set, how the ALU produces these bits in hardware, and how the three control flags change the way the processor works. Every piece of code is traced line by line with the values of the registers, flags, and memory at each step.

LEARNING OUTCOMES

Recognize the register set
Count the fourteen registers of the 8086 by group; explain which ones can be split into two 8-bit halves and why writing to one half does not change the other.

Know the implicit roles
State which registers MUL, DIV, LOOP, REP, IN, OUT, PUSH, POP, and the string instructions use without the registers being written in the instruction.

Use register pairs
Know in which access each of the pairs CS:IP, SS:SP, SS:BP, DS:BX, DS:SI, and ES:DI is used, and compute the physical address.

Compute the status flags
Find the CF, PF, AF, ZF, SF, and OF flags by hand after an addition or subtraction; explain the difference between CF and OF in the unsigned and signed interpretations.

Manage the control flags
Explain how DF, IF, and TF change the processor's behavior; read and write the flags with CLD/STD, CLI/STI, LAHF/SAHF, and PUSHF/POPF.

Trace code
Trace a short 8086 program by writing the state of the registers, flags, memory, and stack at each step in table form.

2.2.1 The register concept

2.2.1.1 What is a register?

In a digital design course, when D flip-flops are lined up side by side and connected to a common clock signal, a **register** results. At the rising edge of the clock the bits at the input are loaded into the flip-flops and stay there until the next load. The registers inside a microprocessor are built this way. The difference is in their number and roles: the processor uses a few dozen registers at the same time and reaches each of them through instructions.

Definition 2.2.1

A **register** is a very fast, small memory cell inside the processor. The 8086's registers are 16 bits wide. All the registers the programmer can read and write directly with instructions are called the **register set**; the 8086's register set has fourteen registers.

There are also registers inside the processor that the programmer cannot see. The instruction register that holds the instruction being decoded, the temporary registers that feed the ALU's inputs, and the prefetch queue are among them. In this chapter "register" means the fourteen registers the programmer can use by name.

While a program runs, temporary data, memory addresses, loop counters, and results of operations are kept in registers. Figure 2.2.1 shows the relationship between the processor and memory. The registers are inside the processor and connected to the ALU by short internal paths. Memory is outside the processor, at the other end of the address and data buses.

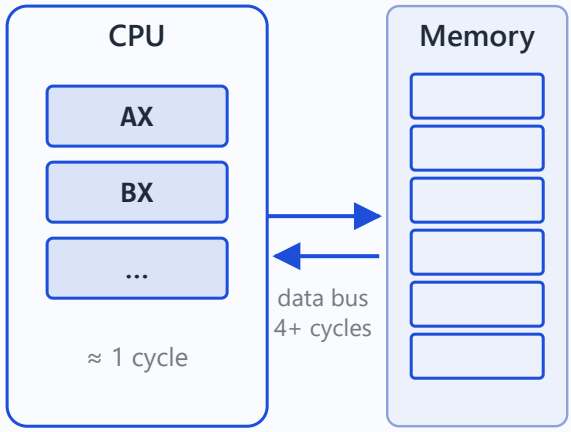


Figure 2.2.1: The registers are inside the processor and the memory is outside. A register access ends on the processor's internal path; a memory access takes at least four clock cycles over the external data bus.

2.2.1.2 Registers and memory: the speed gap

The 8086 talks to external memory in **bus cycles**. A bus cycle consists of at least four clock cycles (T1, T2, T3, T4). In T1 the address is put on the bus, in T2 the read or write signal is issued, in T3 the data is ready on the bus, and in T4 the cycle closes. If the memory is slow, wait cycles are added in between. On a 5 MHz 8086 one clock cycle takes 200 ns; so a single memory read is at least 800 ns.

A register access never goes out on the external bus. The contents of the register reach the ALU within the same clock cycle. Table 2.2.1 gives a few examples from the clock-cycle counts in Intel's 8086 manual. EA is the extra cycles spent computing the effective address. Its value depends on the address form: 5 for a single-register address like `[BX]`, 6 for a direct address like `[0500h]`, and 9 for register plus constant like `[BX+2]`.

Instruction	Operands	Clock cycles	Time at 5 MHz
`MOV AX, BX`	register ← register	2	0.4 µs
`MOV AX, [BX]`	register ← memory	8 + 5 = 13	2.6 µs
`ADD AX, BX`	register ← register + register	3	0.6 µs
`ADD AX, [BX]`	register ← register + memory	9 + 5 = 14	2.8 µs
`ADD [BX], AX`	memory ← memory + register	16 + 5 = 21	4.2 µs

Table 2.2.1: Clock-cycle counts of doing the same job with registers and with memory (8086 manual).

The table gives a clear result: the same addition takes about five times as long when one of the operands is in memory, and seven times as long when the result is written to memory. The `ADD [BX], AX` instruction reads memory first and then writes; it spends two bus cycles. That is why a well-

written assembly program keeps frequently used values in registers and goes to memory only when necessary. Compilers for high-level languages work on the same principle: they place a loop's counter and intermediate results in registers as much as possible.

The number of registers, however, is small. Every register takes space on the processor chip. Inside an instruction there is also a field that selects the register: in 8086 instructions this field is 3 bits and selects one of $2^3 = 8$ registers. Increasing the number of registers both enlarges the chip and lengthens the instructions. With 1978 technology the 8086's designers made do with eight general-purpose registers.

2.2.1.3 There is no memory-to-memory operation

In the 8086's two-operand instructions (MOV, ADD, SUB, AND, CMP ...) at least one of the operands must be a register or a constant. There is no MOV or ADD that copies one memory cell directly to another or adds two memory cells. This rule comes from the instruction format: inside an instruction there is room to describe only one memory address.

```
x  DW 1
y  DW 2
   MOV x, y        ; ERROR: no memory-to-memory MOV

   MOV AX, y       ; right: first into a register
   MOV x, AX       ; then into memory
```

The assembler does not accept the first line and gives an error message. The correct way is to carry the value through a register. The register works here like a relay station. The only exception to this rule is the string instructions. `MOVSB` moves a byte directly from DS:SI to ES:DI. It takes both addresses implicitly from the SI and DI registers without writing them in the instruction.

> **Common mistake**
>
> Thinking of variables as in a high-level language and writing `MOV total, number` or `ADD result, value`. Because both operands are in memory, the assembler does not accept these lines. One of the values is first taken into a register.

2.2.2 The 8086 register set

2.2.2.1 Fourteen registers and their groups

The 8086's fourteen registers are gathered into four groups by role (Figure 2.2.2):

- **Data registers**: AX, BX, CX, DX. Each is 16 bits and can be split into two 8-bit halves: AH/AL, BH/BL, CH/CL, DH/DL.
- **Pointer and index registers**: SP (stack pointer), BP (base pointer), SI (source index), DI (destination index). These point to a location in memory; they have no 8-bit parts.
- **Segment registers**: CS (code), DS (data), SS (stack), ES (extra). Each determines the start of a segment in memory.
- **Special registers**: IP (instruction pointer) and FLAGS (the flags register).

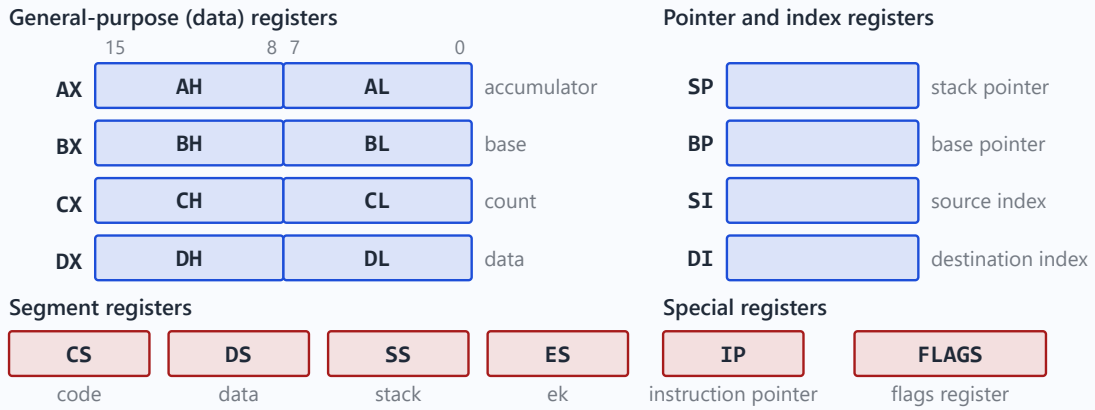


Figure 2.2.2: The 8086 register set. Four data registers with their 8-bit halves, four pointer and index registers, four segment registers, IP, and FLAGS. All are 16 bits; only AX, BX, CX, and DX can be split in two.

The count is as follows: 4 data + 4 pointer/index + 4 segment + IP + FLAGS = 14. In this course the first eight registers, that is, AX, BX, CX, DX, SP, BP, SI, and DI, are counted as **general-purpose registers**. What these eight have in common is that they can be used freely as operands in instructions like MOV, ADD, and AND. `ADD SI, 4` and `MOV BP, AX` are valid instructions. The segment registers, IP, and FLAGS, however, cannot be operands in arithmetic instructions.

The letter X in the register names stands for “extended.” The 8080 processor before the 8086 had 8-bit registers such as A, B, C, and D. The 8086 extended them to 16 bits and named them AX, BX, CX, and DX; the 8-bit halves were called with the letters H (high) and L (low). This naming was chosen so that 8080 programs could be ported easily to the 8086. It continues in today’s x86 processors: in 32-bit processors the lowest 16 bits of the register EAX, and in 64-bit processors of RAX, is still AX.

2.2.2.2 Where the registers sit in the BIU and EU

The 8086 consists of two units: the **bus interface unit** (BIU), which accesses memory and I/O, and the **execution unit** (EU), which decodes and executes instructions. The fourteen registers are shared between these two units, and this sharing makes it easier to understand the roles of the registers.

Unit	Registers	Role in address generation
EU	AX, BX, CX, DX	data; BX can also hold an offset

Unit	Registers	Role in address generation
EU	SP, BP, SI, DI	16-bit offset
EU	FLAGS	none; it holds the ALU result and the control bits
BIU	CS, DS, SS, ES	segment start (paragraph number)
BIU	IP	offset within the code segment

Table 2.2.2: The unit where the registers sit and their role in address generation.

The EU works only with 16-bit numbers. When a memory operand is needed, it computes the effective address, that is, the offset within the segment, from BX, BP, SI, DI, and the constant in the instruction, and gives it to the BIU. It is the BIU that combines the 20-bit physical address with the segment register. That is the reason the segment registers and IP sit in the BIU: independently of the EU, the BIU prefetches the next instruction bytes from CS:IP and puts them in the queue. This arrangement also settles the question “which register gives which part of the address?” The segment register gives the upper part of the address; the pointer and index registers give the position within the segment.

2.2.2.3 General-purpose registers and implicit roles

As their name suggests, general-purpose registers can be used interchangeably in most instructions. It is possible to write `ADD SI, DX` in place of `ADD AX, BX`. Even so, some instructions use a particular register without its name appearing in the instruction.

Definition 2.2.2

An **implicit operand** is an operand that does not appear in the way the instruction is written but is read or written by the definition of the instruction. In the `MUL BX` instruction only BX is written; AX, which is multiplied, and DX, which receives the high half of the result, are implicit operands.

Table 2.2.3 gives the names of the eight general-purpose registers and their implicit roles together. The English names of the registers are abbreviations of their roles, and this makes the table easier to remember.

Register	Ad	8-bit halves	Implicit use (examples)
`AX`	accumulator	AH, AL	MUL, DIV, IN, OUT, LODS/STOS
`BX`	base	BH, BL	memory addressing `[BX]`, XLAT
`CX`	count	CH, CL	LOOP, REP, shift count CL
`DX`	data	DH, DL	upper half of MUL/DIV, port address
`SP`	stack pointer	—	PUSH, POP, CALL, RET
`BP`	base pointer	—	stack frame `[BP+n]`

Register	Ad	8-bit halves	Implicit use (examples)
`SI`	source index	—	string source DS:SI
`DI`	destination index	—	string destination ES:DI

Table 2.2.3: General-purpose registers and their implicit uses.

Implicit roles were chosen to shorten the 8086's instruction encoding. If the LOOP instruction had to say which register holds the counter, a register field would have to be added to every LOOP. Because the counter is always CX, LOOP is only two bytes: the opcode and the jump displacement. The price is flexibility. If CX is being used for another job in a program, it must be saved before LOOP. A programmer who does not know the implicit roles may not notice that a MUL instruction silently erased the value in DX.

2.2.2.4 8-bit and 16-bit use: AX = AH : AL

AX, BX, CX, and DX can be used both as a single 16-bit register and as two separate 8-bit registers. The high-order 8 bits of AX (bits 15–8) are called **AH** and the low-order 8 bits (bits 7–0) **AL**. These are not separate hardware; they are the two halves of the same 16 flip-flops. That is why the following equality holds at every moment:

> AX = AH × 256 + AL (in hexadecimal: AX = AH : AL)

When an instruction writes to AL, only the lower 8 flip-flops are loaded; the upper half stays in place. The best way to see this behavior is to trace a short program. The code is the same as the program in the slides:

```
MOV AX, 1234h   ; AH=12h, AL=34h
MOV AL, 0FFh    ; low half only
ADD AL, 1       ; 8-bit add
MOV AX, 12FFh
ADD AX, 1       ; 16-bit add
```

Row	Instruction	AX	AH	AL	CF	ZF
1	`MOV AX, 1234h`	1234h	12h	34h	0	0
2	`MOV AL, 0FFh`	12FFh	12h	**FFh**	0	0
3	`ADD AL, 1`	1200h	12h	**00h**	**1**	**1**
4	`MOV AX, 12FFh`	12FFh	12h	FFh	1	1
5	`ADD AX, 1`	1300h	**13h**	00h	**0**	**0**

Table 2.2.4: Trace of the AX = AH : AL program (the state after each row has executed).

The first row writes 1234h into AX. In hexadecimal notation every two digits are a byte; so AH = 12h and AL = 34h. The second row changes only AL: AL = FFh, AH = 12h stays, and AX = 12FFh. The third row is the most instructive. `ADD AL, 1` is an 8-bit addition: FFh + 01h = 100h. The result is 9 bits but the destination is 8 bits. The lower 8 bits, 00h, are written to AL. The overflowing 1 is not added to AH; it goes to the **carry flag** (CF). Because the result is zero, the zero flag is also set (ZF = 1). AX is now 1200h (Figure 2.2.3).

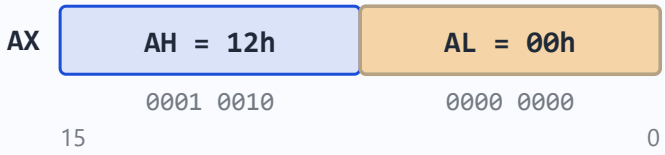


Figure 2.2.3: AX after `ADD AL, 1`. AL = FFh + 1 = 00h; the carry does not pass to AH but goes to CF. AH = 12h stays in place.

The fourth row makes AX 12FFh again. Because MOV does not change the flags, CF = 1 and ZF = 1 are preserved. The fifth row does the same "+1" operation, this time on 16 bits: 12FFh + 0001h = 1300h. This time the carry out of the low byte passes to AH, and 12h becomes 13h. Because the 16-bit result fits in 16 bits, CF = 0. Since the result is not zero, ZF is 0 too.

The difference between these two additions comes from the width at which the ALU works. The processor understands the operation width from the name of the operands: AL is an 8-bit register and AX a 16-bit one. In the instruction code this information is carried by a single bit, the W bit: W = 0 means a byte operation and W = 1 a word operation. `ADD AL, 1` is encoded as `04 01` and `ADD AX, 1` as `05 01 00`; the opcodes differ only in the lowest bit.

Example 2.2.1

If `INC BL` executes with BX = 12FFh, what does BX become?

INC works on the 8-bit BL: FFh + 1 = 100h, and 00h is written to BL. The carry does not pass to BH; BH = 12h stays and BX = 1200h. ZF = 1. The INC instruction never changes the carry flag; CF keeps its previous value. If CF had been set to 1 beforehand, it stays 1 after INC.

The 8-bit halves are also related to the low-byte-first layout in memory. If `MOV [0300h], AX` executes with AX = 1234h, the 34h in AL is written to address 0300h and the 12h in AH to address 0301h. The low byte goes to the low address. The same rule holds when reading from memory: `MOV AX, [0300h]` fills AL from 0300h and AH from 0301h.

Common mistake

Thinking that when AL is written the whole of AX changes. `MOV AL, 5` does not touch AH. If AX was 1234h before, it becomes 1205h after the instruction, not 0005h. If the whole of AX is wanted to be 5, `MOV AX, 5` is written.

Only the four data registers have halves. A notation like `MOV SL, 5` or `MOV BPL, 0` does not exist on the 8086. A program that wants to reach the low byte of SI first copies SI to AX and works on AL.

2.2.3 The data registers

2.2.3.1 AX: the accumulator

AX is the most used register of the 8086. It takes its name from the **accumulator** of older processors. In a processor with an accumulator every arithmetic operation takes one operand from the accumulator and writes the result back there; the results "accumulate" there. The 8086 has dropped this requirement: `ADD BX, CX` works without touching the accumulator. Still, many instructions use AX or AL implicitly:

- `MUL`, `IMUL`, `DIV`, `IDIV`: the multiplicand and dividend sit in AX (or AL, or DX:AX), and the result is written to AX.
- `IN` and `OUT`: the data read from a port comes into AL or AX; the data to be written to a port is in AL or AX.
- `CBW`: extends the signed byte in AL into AX. `DAA` and `DAS`: correct the BCD result in AL.
- `LODSB` loads a byte into AL, and `STOSB` writes the byte in AL to memory.

The privilege of AX shows in instruction length too. Many 8086 instructions have a short form specific to the accumulator. Table 2.2.5 puts the machine codes the assembler produces side by side.

Instruction	Machine code	Length
`ADD AX, 1234h`	`05 34 12`	3 bytes
`ADD BX, 1234h`	`81 C3 34 12`	4 bytes
`MOV AX, [0300h]`	`A1 00 03`	3 bytes
`MOV BX, [0300h]`	`8B 1E 00 03`	4 bytes

Table 2.2.5: Machine code of instructions with and without the accumulator.

In the general form, after the opcode comes a byte that describes the operands (the ModR/M byte: `C3`, `1E`). In the accumulator form the fact that the destination is AX is hidden inside the opcode; this byte is not needed. One byte less is a gain both in memory space and in a bus cycle in the prefetch

queue. That is why compilers and experienced programmers build frequent computations on AX.

The short example in the slides adds AX to itself:

```
ADD AX, AX      ; AX = AX + AX
MOV AX, 0005h
ADD AX, AX      ; AX = 000Ah
MOV AH, 4Ch     ; DOS: exit
INT 21h
```

Instruction	AX	Note
`ADD AX, AX`	0000h	0 + 0 = 0; ZF = 1, PF = 1
`MOV AX, 0005h`	0005h	5 is written to AX
`ADD AX, AX`	000Ah	5 + 5 = 10 = 000Ah; ZF = 0
`MOV AH, 4Ch`	4C0Ah	only AH changes; AL = 0Ah stays
`INT 21h`	4C0Ah	DOS service 4Ch: the program ends, exit code AL = 10

Table 2.2.6: Trace of the program that adds AX to itself (a .COM program, AX = 0 at the start).

Adding a value to itself is multiplying by 2. In a binary number this is a shift of all the bits one place left: 0101b + 0101b = 1010b. The last two rows are interesting too. DOS service 4Ch expects the service number in AH and the program's exit code in AL. Because `MOV AH, 4Ch` does not touch AL, the program ends with exit code 10. If the exit code is to be 0, `MOV AX, 4C00h` is written; this single instruction fills both AH and AL.

2.2.3.2 BX: the base register

BX is 16 bits like AX and is split into BH and BL. What sets it apart from the other data registers is that it can be used in **memory addressing**. A register inside square brackets says that the register's value will be used as an offset. `[BX]` means "the memory cell BX points to."

Definition 2.2.3

A **base register** is a register that holds the base part of the address in indirect addressing. On the 8086 the base registers are BX and BP. The registers that can be used inside square brackets are only BX, BP, SI, and DI; `[AX]`, `[CX]`, and `[DX]` cannot be written.

This restriction too comes from instruction encoding. The 8086's addressing field (the R/M field) selects one of eight address forms with a 3-bit code: `[BX+SI]`, `[BX+DI]`, `[BP+SI]`, `[BP+DI]`, `[SI]`, `[DI]`, `[BP]` (or a direct address), and `[BX]`. AX, CX, and DX are not in the list. The line `MOV AX, [CX]` therefore gets an error from the assembler. If an address in CX has to be used, it is first copied to BX: `MOV BX, CX` and `MOV AX, [BX]`.

The meaning of the square brackets is the point beginners confuse most often. Figure 2.2.4 shows the difference.

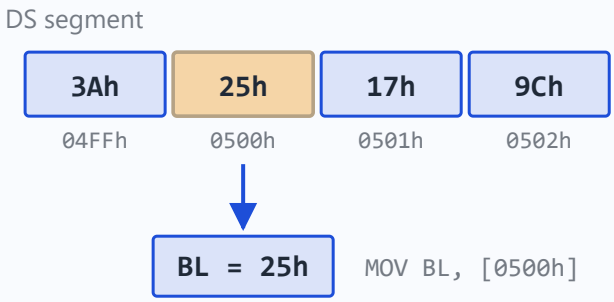


Figure 2.2.4: The instruction `MOV BL, [0500h]` copies the byte 25h at address DS:0500h into BL. The number 500h is not written to BL; the square brackets mean "the value at this address."

`MOV BL, [0500h]` reads from memory: if address DS:0500h holds 25h, BL = 25h. `MOV BX, 0500h`, on the other hand, never goes to memory; it writes the number 0500h into BX. The first instruction is 4 bytes and spends a bus cycle. The second is 3 bytes and the number is inside the instruction. This distinction is the basic distinction between a constant and memory contents. To put a number in a register, square brackets are not used.

The trace program in the slides uses BX as a pointer. Suppose the bytes 25h, 17h, 00h, and 9Ch are at memory addresses 0500h–0503h.

```
MOV BX, 0500h     ; BX = address
MOV AL, [BX]      ; AL = [0500h]
MOV DL, [0501h]   ; direct address
ADD AL, DL        ; AL = 25h + 17h
MOV [BX+2], AL    ; [0502h] = AL
```

Row	Instruction	BX	AL	DL	0500h–0503h
1	`MOV BX, 0500h`	**0500h**	00h	00h	25 17 00 9C
2	`MOV AL, [BX]`	0500h	**25h**	00h	25 17 00 9C
3	`MOV DL, [0501h]`	0500h	25h	**17h**	25 17 00 9C
4	`ADD AL, DL`	0500h	**3Ch**	17h	25 17 00 9C
5	`MOV [BX+2], AL`	0500h	3Ch	17h	25 17 **3C** 9C

Table 2.2.7: Trace of indirect access with BX. The memory column is the bytes 0500h–0503h.

The first row writes a number to BX; from now on this number will be used as an address. In the second row `[BX]` is the byte BX points to: AL = 25h. The third row does the same kind of read by writing the address inside the instruction, and DL = 17h. The fourth row adds two registers: 25h +

17h = 3Ch. In the fifth row the effective address of `[BX+2]` is 0500h + 2 = 0502h; AL is written to this address. BX does not change; what changes is memory (Figure 2.2.5).

Figure 2.2.5: Memory at the end of the program. BX still points to 0500h; 3Ch has been written to address 0502h with `[BX+2]`.

The addresses in the program are offsets. The processor looks them up within the DS segment, because the default segment for accesses made with BX is DS. emu8086 loads a .COM program with DS = 0700h. In this case the physical address of the offset 0500h is 0700h × 10h + 0500h = 07000h + 0500h = 07500h. The byte written in the fifth row appears in emu8086's memory window at physical address 07502h.

That BX can hold an address makes it indispensable in array and table operations. While BX points to the start of an array, the form `[BX+SI]` reaches the SI-th element of the array; the array is walked by incrementing SI on each pass. The XLAT instruction also uses BX implicitly: it treats AL as an index into the table BX points to and does `AL ← [BX+AL]` in a single instruction.

2.2.3.3 CX: the count register

CX is a 16-bit register split into CH and CL. It is used in loops, repetition, and counting; its name comes from this (count register). Three kinds of instructions use CX implicitly:

- `LOOP label`: first does CX ← CX − 1, then jumps to the label if CX ≠ 0. It does not affect the flags.
- The `REP` prefix: repeats the string instruction after it CX times; CX decreases by one each pass.
- Shift and rotate instructions: if the count is greater than 1 it is given in CL: `SHL AX, CL`.

Definition 2.2.4

The `LOOP label` instruction does the following two steps in a single instruction: (1) CX ← CX − 1; (2) if CX ≠ 0, IP ← the address of the label, otherwise move on to the next instruction. The decrement is done **before** the test.

The program in the slides computes the sum 5 + 4 + 3 + 2 + 1. CX is both the counter and the value being added.

```
MOV CX, 5          ; counter
MOV AX, 0          ; sum
AGAIN: ADD AX, CX
LOOP AGAIN         ; CX--, jump if ≠0
MOV DX, AX         ; result
```

Step	Instruction	CX	AX	DX	Next
1	`MOV CX, 5`	0005h	0000h	0000h	row 2
2	`MOV AX, 0`	0005h	0000h	0000h	AGAIN
3	`ADD AX, CX`	0005h	**0005h**	0000h	LOOP
4	`LOOP AGAIN`	**0004h**	0005h	0000h	jumps
5	`ADD AX, CX`	0004h	**0009h**	0000h	LOOP
6	`LOOP AGAIN`	**0003h**	0009h	0000h	jumps
7	`ADD AX, CX`	0003h	**000Ch**	0000h	LOOP
8	`LOOP AGAIN`	**0002h**	000Ch	0000h	jumps
9	`ADD AX, CX`	0002h	**000Eh**	0000h	LOOP
10	`LOOP AGAIN`	**0001h**	000Eh	0000h	jumps
11	`ADD AX, CX`	0001h	**000Fh**	0000h	LOOP
12	`LOOP AGAIN`	**0000h**	000Fh	0000h	falls through
13	`MOV DX, AX`	0000h	000Fh	**000Fh**	—

Table 2.2.8: Trace of the sum 5 + 4 + 3 + 2 + 1 with CX and LOOP.

The loop body runs five times with CX = 5, 4, 3, 2, 1. The fifth LOOP lowers CX from 1 to 0 and does not jump; the program leaves the loop. The body does **not** run at all with CX = 0. The total is 15 = 000Fh. Check with Gauss's formula: 5 × 6 / 2 = 15.

The time of the loop is also easy to compute. When LOOP jumps it takes 17 clock cycles and when it does not jump it takes 5; `ADD AX, CX` is 3 cycles. For five passes 5 × 3 + 4 × 17 + 5 = 88 cycles are spent. This value also shows how the share of loop control in the total time shrinks as the body grows.

The CX = 0 trap

Because LOOP decrements before it tests, entering the loop with CX = 0 gives an unexpected result. The first LOOP computes 0000h − 1 = FFFFh. FFFFh is not zero; the processor jumps. The loop then runs with FFFFh, FFFEh, …, 0001h, and the total number of passes is 65,536.

```
    MOV CX, 0
    MOV BX, 0
    MOV DX, 0
D:  ADD BX, 1      ; pass counter (low word)
    ADC DX, 0      ; high word
    LOOP D
```

This program counts the number of passes in the 32-bit pair DX:BX. When the program ends, DX:BX = 0001:0000h = 65,536. Note that the pass count does not fit in 16 bits; had the counter been BX alone, it would have returned to 0 after 65,536 passes and one would think the loop never ran.

The way to protect against this trap is the `JCXZ` instruction: "jump if CX is zero." It is placed right before the loop and skips the whole loop if the counter is zero.

```
     MOV CX, 0
     MOV AX, 0
     JCXZ skip        ; if CX = 0, do not enter the loop
D:   ADD AX, CX
     LOOP D
skip: MOV DX, AX
```

In this program JCXZ jumps straight to the label skip; the loop body does not run even once and AX = DX = 0 remains. JCXZ, like LOOP, does not look at the flags; it tests CX directly.

Shifting with CL

On the 8086, shift and rotate instructions take the count either as 1 or in CL. The notation `SHL AX, 4` does not exist on the 8086; it came with the 80186. For the 8086 the same job is written like this:

```
MOV AX, 0003h
MOV CL, 4
SHL AX, CL       ; AX = 0030h
```

A four-bit left shift is multiplication by $2^4 = 16$: $3 \times 16 = 48 = 0030$h. In hexadecimal notation this means appending a 0 to the number. The same operation appears when a segment register is converted to a physical address.

2.2.3.4 DX: the data register

DX is split into DH and DL. It has two important implicit roles: holding the high half of AX in wide multiplication and division, and carrying the I/O port number.

Multiplication. The product of two 16-bit numbers can grow to 32 bits: FFFFh × FFFFh = FFFE0001h. The result does not fit in a single 16-bit register. The 8086 therefore splits the result of a 16-bit multiplication across two registers: the high 16 bits are written to DX and the low 16 bits to AX. This pair is written **DX:AX** (Figure 2.2.6). In an 8-bit multiplication, however, the result is 16 bits and fits entirely in AX: `MUL BL` → AX = AL × BL. In this case DX does not change.

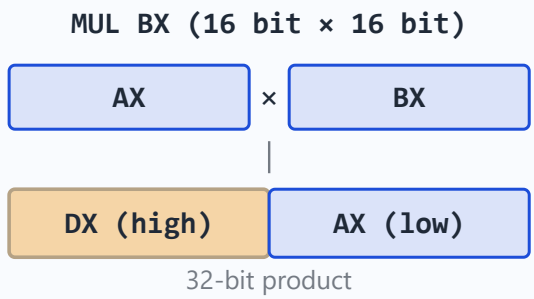


Figure 2.2.6: `MUL BX` : AX is multiplied by BX, and the 32-bit result is written to the pair DX:AX. DX is the high half and AX the low half.

Division. In a 16-bit division the dividend is the 32-bit DX:AX. The quotient is written to AX and the remainder to DX. For example, with DX = 0012h, AX = 3405h, and BX = 0100h, `DIV BX` computes 00123405h ÷ 0100h. The quotient is 1234h and the remainder 0005h: AX = 1234h, DX = 0005h. If the quotient does not fit in 16 bits, the processor raises a type 0 interrupt (divide error). That is why DX is cleared before dividing a 16-bit number; if a leftover value remains in DX, the dividend becomes much larger than expected.

Port number. The IN and OUT instructions take the port number in two ways. Ports between 0 and 255 can be written into the instruction as a constant: `IN AL, 60h`. For ports above 255 the number is put in DX: `OUT DX, AL`. On the PC the first serial port (COM1) starts at 03F8h and the printer port at 0378h; these are reached only with DX. DX being 16 bits covers the whole 65,536-port I/O address space.

DOS calls. Many INT 21h services expect a memory address in DX. Service 09h, which writes a string to the screen, takes the address of the string from DS:DX: `LEA DX, message`.

The program in the slides shows two of these roles together:

```
MOV AX, 1234h
MOV BX, 0100h
MUL BX          ; DX:AX = AX*BX
MOV AX, 00FFh
MOV BL, 2
MUL BL          ; AX = AL*BL
MOV DX, 03F8h   ; port address
OUT DX, AL      ; write to port
```

Row	Instruction	AX	BX	DX	CF	OF	SF ZF AF PF
1	`MOV AX, 1234h`	1234h	0000h	0000h	0	0	0 0 0 0
2	`MOV BX, 0100h`	1234h	0100h	0000h	0	0	0 0 0 0
3	`MUL BX`	**3400h**	0100h	**0012h**	**1**	**1**	? ? ? ?
4	`MOV AX, 00FFh`	00FFh	0100h	0012h	1	1	? ? ? ?
5	`MOV BL, 2`	00FFh	**0102h**	0012h	1	1	? ? ? ?
6	`MUL BL`	**01FEh**	0102h	0012h	1	1	? ? ? ?
7	`MOV DX, 03F8h`	01FEh	0102h	**03F8h**	1	1	? ? ? ?
8	`OUT DX, AL`	01FEh	0102h	03F8h	1	1	? ? ? ?

Table 2.2.9: Trace of the MUL and OUT program. “?” shows the flags that are undefined after MUL.

In the third row 1234h × 0100h = 00123400h is computed. Multiplying by 0100h shifts the hexadecimal number two digits to the left. The 32-bit result is split in two: DX = 0012h, AX = 3400h. MUL sets CF = OF = 1 if the high half of the result is not zero. These two flags mean “the result did not fit in the lower half; DX must be looked at too.” After MUL, SF, ZF, AF, and PF are **undefined**: Intel has not specified the value of these flags and a program must not rely on them. The “?” mark in the table shows this.

The fifth row changes only BL. Because BX was 0100h beforehand, BH = 01h stays and BX = 0102h. The sixth row is an 8-bit multiplication: AL × BL = FFh × 02h = 255 × 2 = 510 = 01FEh. The result is written to the whole of AX; the 0012h in DX stays as it is. Because the high half AH = 01h is not zero, CF = OF = 1 again.

In the seventh row DX holds not data this time but a port number: 03F8h is the data register of the first serial port. The eighth row sends the FEh in AL to this port. OUT uses only the low byte of AX; the 01h in AH does not go to the port. A single byte, FEh, is written to the port.

Common mistake

Thinking the result of MUL is only in AX and going on using DX for another job. A 16-bit MUL always erases the value in DX; DX is written even if the result is small. If there is a value that must be kept in DX, it is pushed onto the stack before the MUL.

Example 2.2.2

With AL = 10h and BL = 03h, what are AX, CF, and OF after `MUL BL` ?

10h × 03h = 16 × 3 = 48 = 0030h. AX = 0030h. Because the high half AH = 00h, the result fit in AL and CF = OF = 0. In this case the program can go on by looking only at AL. Had CF been 1, the result would have to be sought in the whole of AX.

2.2.4 The pointer and index registers

The SP, BP, SI, and DI registers are 16 bits and have no 8-bit halves. All four mostly hold not data but an **offset**: the distance of a cell in memory from the start of the segment it belongs to. SP and BP work in the stack segment (SS), and SI and DI in the data segments (DS, ES). They can also be used in arithmetic instructions: `ADD SI, 2` is the usual way to advance an index while walking an array.

2.2.4.1 SP: the stack pointer

Definition 2.2.5

The **stack** is a region of memory that works in last-in, first-out (LIFO) order. The **stack pointer** (SP) holds the offset of the topmost item of the stack. The physical address of the stack is always given by the pair `SS:SP`.

The 8086's stack is recognized by three properties. First, items are always **16 bits**: `PUSH AX` is valid, `PUSH AL` is not. The assembler counts the line `PUSH AL` as an error. Second, the stack grows **downward** in memory; each PUSH writes to a lower address. Third, the order of operation is fixed:

- `PUSH source`: first SP ← SP − 2, then [SS:SP] ← source. The low byte is written to SP and the high byte to SP + 1.
- `POP destination`: first destination ← [SS:SP], then SP ← SP + 2.

SP always points to the last valid item on the stack. When a new item is to be put on the stack, room is made first and then it is written. POP does not erase the cell it reads; the value stays in memory, but since SP has moved above it, it is no longer considered part of the stack. The next PUSH writes over that cell.

Tracing the following program makes these rules concrete. Assume the stack is empty and SP starts from 0000h.

```
MOV  AX, 1111h
PUSH AX
MOV  BX, 2222h
PUSH BX
POP  CX
POP  DX
```

Instruction	SP	Stack (address: value)	CX	DX
`MOV AX, 1111h`	0000h	empty	0000h	0000h
`PUSH AX`	**FFFEh**	FFFEh: 1111h	0000h	0000h

Instruction	SP	Stack (address: value)	CX	DX
`MOV BX, 2222h`	FFFEh	FFFEh: 1111h	0000h	0000h
`PUSH BX`	**FFFCh**	FFFCh: 2222h, FFFEh: 1111h	0000h	0000h
`POP CX`	**FFFEh**	FFFEh: 1111h	**2222h**	0000h
`POP DX`	**0000h**	empty	2222h	**1111h**

Table 2.2.10: Trace of PUSH and POP (SP = 0000h at the start). The stack column gives the items from SP upward.

The first PUSH makes SP FFFEh by computing 0000h − 2. In 16-bit arithmetic, going below 0 means wrapping to the top; so SP = 0000h is a start from which the whole 64 KB stack segment can be used. The value 1111h is written to the bytes FFFEh–FFFFh. The second PUSH lowers SP to FFFCh and puts 2222h there. Figure 2.2.7 shows the stack at this moment. SP points to the topmost item, that is, the most recently pushed 2222h.

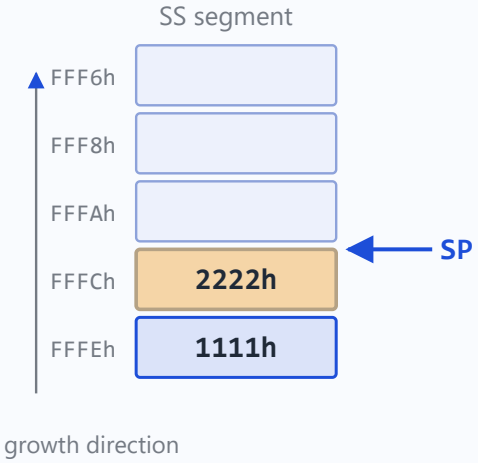


Figure 2.2.7: The stack segment after two PUSHes: 1111h at FFFEh and 2222h at FFFCh. SP = FFFCh points to the top. The stack grows toward low addresses.

The POPs run in the reverse order. `POP CX` takes the topmost 2222h and SP rises to FFFEh. `POP DX` takes 1111h and SP becomes 0000h again. In the end CX = 2222h and DX = 1111h: the values have come back in the reverse of the order they were pushed. That is what LIFO order means. To save and restore AX and BX in a program, `PUSH AX`, `PUSH BX` and then `POP BX`, `POP AX` are written; the POP order is the reverse of the PUSH order.

The fact “SP = FFFEh when the stack is empty” is not a general rule. When DOS loads a .COM program it makes SP FFFEh and puts the word 0000h at address FFFEh. So the `RET` at the end of the program takes this zero into IP and returns to the DOS exit code at the start of the program. emu8086 also starts .COM programs this way. If the same program were run as a .COM file, SP would drop first to FFFCh and then to FFFAh; the values and order would not change.

The implicit users of SP are not only PUSH and POP. `CALL` pushes the return address onto the stack and `RET` takes it from the stack. An interrupt pushes FLAGS, CS, and IP; `IRET` takes all three back. `PUSHF` and `POPF` move the flags register. All these instructions update SP themselves. If the programmer writes to SP directly (`MOV SP, …`), the position of the stack is changed; this is done only at the start of a program or when changing the stack segment.

> **Common mistake**
>
> Not balancing the number of PUSHes and POPs. If an extra PUSH is left inside a procedure, RET takes that value into IP instead of the return address and the program jumps to a random address. Skipping a POP in a conditional branch gives the same result. On every exit path the stack must return to the state it had at entry.

2.2.4.2 BP: the base pointer

BP (base pointer) is used to point to a **fixed location** on the stack. Memory accesses made with BP use the SS segment by default: `[BP+4]` is the address SS:BP+4. `[BX+4]`, written with BX, goes to the DS segment. The only difference between the two base registers is this default segment.

SP points to the top of the stack but keeps changing inside a procedure: every PUSH and POP moves it. Reaching parameters relative to SP is therefore laborious; after a PUSH the offset of the same parameter grows by 2 bytes. BP solves this problem. At the start of the procedure the current value of SP is copied to BP. From then on, however much SP changes, BP stays fixed and each parameter is always at the same distance from BP.

```
      MOV  BX, 1234h
      PUSH BX          ; parameter
      CALL ADDP
      MOV  DX, AX      ; result
      RET

ADDP PROC
  PUSH BP
  MOV  BP, SP     ; frame base
  MOV  AX, [BP+4] ; 1st param
  POP  BP
  RET  2
ADDP ENDP
```

The procedure body is the same as the one in the slides. The calling part has been added with three lines that push the parameter onto the stack and take the result into DX. The program starts as a .COM program with SP = FFFEh.

Instruction	SP	BP	AX	Stack (address: value)
`MOV BX, 1234h`	FFFEh	0000h	0000h	FFFE: 0000
`PUSH BX`	**FFFCh**	0000h	0000h	FFFC: 1234, FFFE: 0000
`CALL ADDP`	**FFFAh**	0000h	0000h	FFFA: 0107, FFFC: 1234, ...
`PUSH BP`	**FFF8h**	0000h	0000h	FFF8: 0000, FFFA: 0107, FFFC: 1234, ...
`MOV BP, SP`	FFF8h	**FFF8h**	0000h	same
`MOV AX, [BP+4]`	FFF8h	FFF8h	**1234h**	same
`POP BP`	**FFFAh**	**0000h**	1234h	FFFA: 0107, FFFC: 1234, ...
`RET 2`	**FFFEh**	0000h	1234h	FFFE: 0000
`MOV DX, AX`	FFFEh	0000h	1234h	FFFE: 0000

Table 2.2.11: Trace of the stack frame. The stack column goes upward from SP.

CALL pushes the return address 0107h onto the stack. This is the address of the instruction after the CALL, that is, of `MOV DX, AX`: the program starts at 0100h, `MOV BX, 1234h` is 3 bytes, `PUSH BX` is 1 byte, `CALL` is 3 bytes, and 0100h + 7 = 0107h. The `PUSH BP` at procedure entry saves the caller's BP value. With `MOV BP, SP`, BP = FFF8h and the base of the frame is fixed. Now the frame reads like this:

- `[BP]` = FFF8h: the saved old BP;
- `[BP+2]` = FFFAh: the return address 0107h;
- `[BP+4]` = FFFCh: the first parameter 1234h.

`MOV AX, [BP+4]` takes the parameter into AX. `POP BP` restores the caller's BP. `RET 2` does two jobs: it takes the return address into IP and also adds 2 to SP, discarding the parameter on the stack. SP returns to the value FFFEh it had at entry; the stack is in the state it had before the call. Even if an extra PUSH had been done in the procedure, `[BP+4]` would not have changed; that is the worth of BP.

Example 2.2.3

Let ADDP be called with two parameters: first `PUSH a`, then `PUSH b`. At which addresses are a and b inside the procedure, and how many bytes must RET discard on return?

The last pushed, b, is right above the return address: `[BP+4]` = b, and a, pushed before it, is at `[BP+6]`. Two parameters take 4 bytes; the return is done with `RET 4`. Each time the number of parameters grows, the offset of the first parameter grows by 2 bytes, but the last parameter is always at `[BP+4]`.

2.2.4.3 SI and DI: the index registers

SI (source index) and DI (destination index) take their names from their roles in the string instructions. String instructions work on consecutive bytes or words in memory: copy (MOVS), compare (CMPS), scan (SCAS), load (LODS), and store (STOS). The operands of these instructions are

not written in the instruction; they are implicit:

- The source is always **DS:SI**: MOVSB, CMPSB, and LODSB read from here.
- The destination is always **ES:DI**: MOVSB, STOSB, and SCASB write or compare here.
- After each step SI and DI increase or decrease by 1 (byte) or 2 (word) according to the DF flag.

Outside the string instructions, SI and DI are ordinary index registers. When `MOV AL, [SI]` or `MOV [DI], AL` is written, both use the **DS** segment by default. The ES:DI rule is only for the destination of string instructions. This distinction matters when DS and ES point to different segments.

Definition 2.2.6

An **index register** is a register that holds a position within an array or string. The 8086's index registers are SI and DI. When used together with a base register (`[BX+SI]`, `[BP+DI+5]`), the effective address is the sum of the two registers and the constant.

The program in the slides copies the string "ABC" from 0200h to 0300h. In a .COM program DS = ES, so the two areas are in the same segment.

```
CLD             ; DF=0: addresses go up
MOV SI, 0200h   ; source
MOV DI, 0300h   ; destination
MOV CX, 3       ; byte count
REP MOVSB       ; ES:[DI] ← DS:[SI]
```

Step	Instruction	SI	DI	CX	DF	0300h–0302h
1	`CLD`	0000h	0000h	0000h	0	— — —
2	`MOV SI, 0200h`	**0200h**	0000h	0000h	0	— — —
3	`MOV DI, 0300h`	0200h	**0300h**	0000h	0	— — —
4	`MOV CX, 3`	0200h	0300h	**0003h**	0	— — —
5	`REP MOVSB` (pass 1)	**0201h**	**0301h**	**0002h**	0	A — —
6	`REP MOVSB` (pass 2)	**0202h**	**0302h**	**0001h**	0	A B —
7	`REP MOVSB` (pass 3)	**0203h**	**0303h**	**0000h**	0	A B C

Table 2.2.12: Trace of REP MOVSB. Each pass of REP is a separate row.

CLD clears DF: SI and DI will increase at every step. SI points to the start of the source and DI to the start of the destination; CX says how many bytes will be copied. On each pass REP MOVSB does four things: it copies the byte at DS:SI to ES:DI, increments SI by 1, increments DI by 1, and decrements CX by 1. When CX reaches zero the repetition ends. After three passes "ABC" has been copied to the destination, and SI and DI point just past the copied area (Figure 2.2.8).

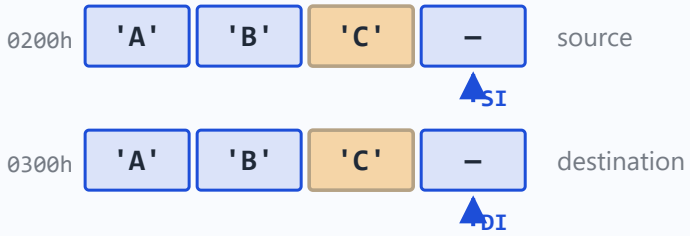


Figure 2.2.8: The source and destination areas after the third pass. The bytes 'A', 'B', 'C' have been written starting at 0300h; SI = 0203h, DI = 0303h, CX = 0.

The REP prefix uses CX and MOVSB uses SI and DI implicitly. A single two-byte instruction (`F3 A4`) does the work of an entire loop. Writing the same copy with `MOV AL, [SI]`, `MOV [DI], AL`, `INC SI`, `INC DI`, and `LOOP` takes five instructions and more clock cycles on every pass. The clearest gain from giving the registers implicit roles in the 8086's design is seen here.

2.2.5 The segment registers and IP

The 8086's address bus is 20 bits and addresses 1 MB of memory. Its registers, however, are 16 bits and by themselves can point to only 64 KB. The two ends are reconciled by dividing memory into **segments**: every memory access consists of a segment register and a 16-bit offset. This section treats segmentation from the registers' point of view: how to write to the segment registers, which register pairs with which segment, and how IP advances.

2.2.5.1 Writing to the segment registers

There are four segment registers: **CS** (code segment), **DS** (data segment), **SS** (stack segment), and **ES** (extra segment). Each holds a 16-bit paragraph number. The physical address where a segment starts is 16 times this number; the segment extends from there for at most 64 KB.

> **Definition 2.2.7**
>
> The **physical address** (20 bits) = segment register × 10h + offset. Multiplying by 10h (16) is appending a 0 to the end of the hexadecimal number. If DS = 2000h and BX = 0500h, the physical address of the access `[BX]` is 20000h + 0500h = 20500h.

The rules for writing to the segment registers are stricter than for the other registers:

- **A constant cannot be loaded.** `MOV DS, 2000h` cannot be written; the assembler rejects the line. The value is first put in a general-purpose register: `MOV AX, 2000h` and `MOV DS, AX`.
- **CS cannot be written with MOV.** A change in CS means the program jumps somewhere else; this is done with far JMP, far CALL, RETF, INT, and IRET. The line `MOV CS, AX` is also rejected.

- **There is no MOV from segment to segment.** `MOV ES, DS` cannot be written. The value is transferred either through a register (`MOV AX, DS` , `MOV ES, AX`) or through the stack (`PUSH DS` , `POP ES`).
- **Arithmetic cannot be done with the segment registers.** `ADD DS, 10h` or `INC ES` do not exist.

The reason for these restrictions is that the segment registers are in the BIU, not the EU. The ALU's output is not directly connected to the segment registers; data reaches the segment registers only through MOV and POP, from a register or a memory word. The following short program sets up and uses a data segment. Suppose the byte 5Ah is at physical address 20500h.

```
MOV AX, 2000h
MOV DS, AX       ; DS = 2000h: segment starts at 20000h
MOV BX, 0500h
MOV AL, [BX]     ; AL ← [20500h]
MOV [BX+1], AL   ; [20501h] ← AL
```

Instruction	AX	DS	BX	Physical address	20500h–20501h
`MOV AX, 2000h`	**2000h**	0700h	0000h	—	5A 00
`MOV DS, AX`	2000h	**2000h**	0000h	—	5A 00
`MOV BX, 0500h`	2000h	2000h	**0500h**	—	5A 00
`MOV AL, [BX]`	**205Ah**	2000h	0500h	20500h (read)	5A 00
`MOV [BX+1], AL`	205Ah	2000h	0500h	20501h (write)	5A **5A**

Table 2.2.13: Setting up and using the DS register.

DS is initially 0700h, given by the .COM loader. After the second row, every access made with BX is relative to the base 20000h. When 5Ah is read into AL, the 20h in AH stays in place, so AX = 205Ah; the independence of the 8-bit halves is seen here too.

2.2.5.2 Register pairs

An offset by itself is not an address; which segment it belongs to must also be known. The 8086 chooses a segment register by default for each type of access. Since the programmer does not write this, these pairs are also part of the implicit roles (Table 2.2.14).

Pair	Use	Example
`CS:IP`	next instruction	every instruction fetch
`SS:SP`	top of the stack	PUSH, POP, CALL, RET, INT
`SS:BP`	stack frame	`MOV AX, [BP+4]`
`DS:BX` , `DS:SI` , `DS:DI`	data	`MOV AL, [BX+SI]`
`DS:SI → ES:DI`	string instructions	`MOVSB` , `CMPSB`

Table 2.2.14: Register pairs and the accesses they are used in.

The rule can be summarized briefly: if the address contains BP the segment is SS, and if not it is DS. `[BP+SI]` uses SS and `[BX+SI]` uses DS. The destination of string instructions is always ES. The default segment can be changed by adding a **segment prefix** in front of the instruction: `MOV AL, ES:[BX]` reads the data from the ES segment. A prefix cannot be used for instruction fetch (CS:IP) and stack operations (SS:SP). The string destination ES:DI cannot be changed either; only the segment of the source DS:SI can be redirected to another segment with a prefix.

Example 2.2.4

With DS = 1000h, SS = 3000h, SI = 0020h, and BP = 0020h, which physical addresses do `MOV AL, [SI]` and `MOV AL, [BP]` read?

`[SI]` does not contain BP; the segment is DS: 1000h × 10h + 0020h = 10000h + 0020h = 10020h. `[BP]`, on the other hand, uses the SS segment: 30000h + 0020h = 30020h. The two instructions have the same offset but their physical addresses are 128 KB apart.

Common mistake

Trying to access an array with BP. If `MOV BP, OFFSET array` and `MOV AL, [BP]` are written, the access goes to the SS segment. In .COM programs the mistake stays hidden because DS = SS; in a .EXE program where DS and SS differ, the wrong memory is read. For data arrays BX, SI, or DI is used.

2.2.5.3 IP: the instruction pointer

Definition 2.2.8

The **instruction pointer** (IP) holds the offset of the next instruction to be executed within the code segment. The physical address of the next instruction is CS × 10h + IP. In other processors this register is called the program counter (PC).

While an instruction executes, IP is advanced by the length of the instruction and points to the next instruction. 8086 instructions are between 1 and 6 bytes; so IP does not grow in regular steps. The addresses in the AX = AH : AL program of Table 2.2.4 are a good example of this (Table 2.2.15).

Address of the instruction	Instruction	Machine code	Length	Next IP
0100h	`MOV AX, 1234h`	`B8 34 12`	3	0103h
0103h	`MOV AL, 0FFh`	`B0 FF`	2	0105h
0105h	`ADD AL, 1`	`04 01`	2	0107h
0107h	`MOV AX, 12FFh`	`B8 FF 12`	3	010Ah

Address of the instruction	Instruction	Machine code	Length	Next IP
010Ah	`ADD AX, 1`	`05 01 00`	3	010Dh

Table 2.2.15: How IP advances according to instruction lengths (.COM program, ORG 100h).

Instructions that carry a 16-bit constant are 3 bytes, and those that carry an 8-bit constant are 2 bytes. The low byte of a constant in machine code comes first: 1234h is written as `34 12`.

The instructions that take IP out of sequence are these: JMP and the conditional jumps, LOOP, CALL, RET, INT, and IRET. IP cannot be written with `MOV` and cannot be read with `MOV`; IP is not an operand the programmer reaches by name. There is an indirect way to learn the value of IP: the `CALL` instruction pushes the return address onto the stack, and with a `POP` this value can be taken into a register.

Because of the BIU's prefetch arrangement, inside the processor the bytes in the queue and the address IP points to may differ for a moment. From the programmer's point of view this difference is invisible; IP always gives the address of the next instruction to be executed. When a jump occurs the queue is flushed and fetching starts again from the new CS:IP.

2.2.6 The flags register and the status flags

2.2.6.1 Structure of the flags register

When an addition is done, the processor produces not only the result but also records a few facts about it. Is the result zero? Negative? Did it fit in 8 bits? These facts are one-bit indicators and are called **flags**. The 8086 gathers its flags in the 16-bit **flags register** (FLAGS). This register is also called the status register.

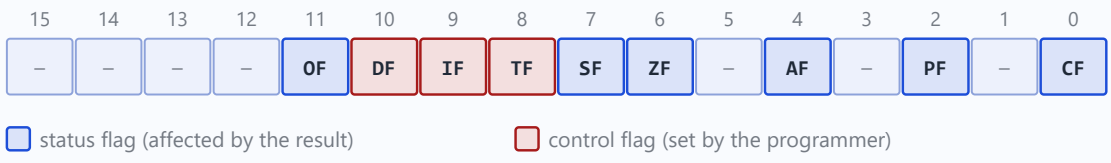


Figure 2.2.9: The 16-bit flags register. Bit 0 CF, 2 PF, 4 AF, 6 ZF, 7 SF, 8 TF, 9 IF, 10 DF, 11 OF; the other bits are unused.

Only 9 of the register's 16 bits are used (Figure 2.2.9). They fall into two groups:

Definition 2.2.9

The **status flags** are CF, PF, AF, ZF, SF, and OF. After arithmetic and logic instructions the processor sets (1) or clears (0) them itself according to the result. The **control flags** are DF, IF, and TF. They are not affected by results; the programmer changes how the processor works by setting them.

The main users of the status flags are the conditional jump instructions. `JZ` looks at ZF, `JC` at CF, `JS` at SF, and `JO` at OF; signed comparisons such as `JG` and `JL` evaluate SF and OF together. Every `if` and every loop condition in a high-level language becomes, in assembly, a comparison followed by a jump that looks at the flags. Knowing when and how the flags are set is therefore fundamental to 8086 programming.

The bit order of the flags is not random. The five flags in the low byte (SF, ZF, AF, PF, CF) are in the same places as in the flags byte of the 8080 processor. This arrangement was kept to ease the translation of 8080 programs, and it is the reason the LAHF/SAHF instructions exist. OF and the control flags were added with the 8086 and put in the high byte.

2.2.6.2 How are flags produced in hardware?

The status flags are produced from the ALU's output with a few simple gates. With digital design knowledge, building this circuit is not hard. In an n-bit adder, let r_i be the i-th bit of the result and c_i the carry into the i-th bit; c_n is the carry out of the top bit. In an 8-bit operation n = 8.

Flag	Logic expression	Circuit
CF	c_8 (borrow in subtraction)	carry out of the last full adder
AF	c_4	carry from bit 3 to bit 4
ZF	$\neg(r_7 + r_6 + \ldots + r_0)$	8-input NOR gate
SF	r_7	copy of the top bit (no gate)
PF	$\neg(r_7 \oplus r_6 \oplus \ldots \oplus r_0)$	XOR tree and a NOT gate at the output
OF	$c_7 \oplus c_8$	XOR of the carries into and out of the top bit

Table 2.2.16: Producing the status flags from the adder output (8-bit operation).

In a 16-bit operation c_{16} is used for CF, r_{15} for SF, and $c_{15} \oplus c_{16}$ for OF; ZF looks at all 16 bits. AF and PF, however, are produced from only the low byte even in a 16-bit operation. The point to notice in the table is that all the flags are produced at the same time and together with the result. The processor does not ask "signed or unsigned?"; after every addition it sets all six flags at once. The program decides which one is meaningful.

Subtraction is done in the ALU as addition of the two's complement: $A - B = A + \neg B + 1$. A carry out of the top bit in this addition means no borrow was taken. The 8086 sets CF in subtraction as the inverse of this carry; thus CF = 1 always means "a borrow was taken." The same inversion is done for

AF.

2.2.6.3 SF: the sign flag

The **sign flag** (SF) is a copy of the most significant bit (MSB) of the result. It is bit 7 in an 8-bit operation and bit 15 in a 16-bit operation. In two's complement the MSB is the sign bit: if 1 the number is negative, if 0 it is positive or zero. SF therefore says whether the result is negative when interpreted as signed.

Result (8 bits)	Binary	Signed value	MSB	SF
`35h`	0011 0101	+53	0	0
`7Fh`	0111 1111	+127	0	0
`80h`	1000 0000	−128	1	1
`C2h`	1100 0010	−62	1	1
`FFh`	1111 1111	−1	1	1

Table 2.2.17: SF in 8-bit results.

The rule is short: for results from 00h to 7Fh, SF = 0, and for those from 80h to FFh, SF = 1. In 16 bits the boundary is between 7FFFh and 8000h. The processor does not know whether a number is signed or unsigned. The byte C2h is 194 as unsigned and −62 as signed; in both interpretations SF = 1. Reading SF as "negative" is the programmer's choice and is meaningful only when working with signed numbers.

> **Common mistake**
>
> Thinking SF gives the sign of the true mathematical result. In 70h + 10h = 80h two positive numbers are added and the true result is +128. Even so SF = 1, because the MSB of the 8-bit result is 1. The result did not fit in the signed range; OF shows this. SF gives the correct sign only when OF = 0.

2.2.6.4 ZF and PF: the zero and parity flags

The **zero flag** (ZF) is 1 if all bits of the result are zero; otherwise it is 0. The value of the flag and the value of the result look opposite: ZF = 1 means "the result is zero." This confusion is cleared by reading the flag as a question: "is the result zero?" If yes, 1.

ZF is used most in equality tests. `CMP AX, BX` computes AX − BX but writes the result nowhere; it only sets the flags. If AX = BX the difference is zero and ZF = 1. The following `JE` (jump if equal, the same instruction as `JZ`) looks at ZF. With AX = BX = 1234h, after CMP ZF = 1, PF = 1, and the other status flags are 0.

Three ways of clearing a register give an interesting comparison in terms of ZF. `SUB AL, AL` and `XOR AL, AL` make the result zero, so they set ZF = 1 and CF = 0. `MOV AL, 0`, however, does not touch the flags at all; the flags left from the previous operation stay. If the register is to be cleared right before a conditional jump, this difference matters.

The **parity flag** (PF) is 1 if the number of 1s in the **low 8 bits** of the result is even, and 0 if it is odd. In 16-bit operations too only the low byte is looked at. Figure 2.2.10 gives two examples.

Figure 2.2.10: The parity flag. The low byte of 03h has two 1s, PF = 1; 07h has three 1s, PF = 0.

The number zero has no 1s at all; since zero is an even number, PF = 1 for 00h. PF's historical use was error checking in serial communication. A parity bit is added to every character sent, and the receiver checks the parity of the byte it receives; a single-bit corruption changes the parity and is noticed. In today's programs PF is rarely used, but every arithmetic and logic operation of the 8086 still sets it. The `JP` and `JNP` instructions look at this flag.

Example 2.2.5

After `ADD AX, 0009h` with AX = 7FF8h, what are ZF and PF?

7FF8h + 0009h = 8001h. The result is not zero: ZF = 0. For PF only the low byte 01h = 0000 0001b is looked at. It has one 1, an odd number: PF = 0. The 1 in the high byte 80h is not counted.

2.2.6.5 AF: the auxiliary carry flag

The **auxiliary carry flag** (AF) is 1 if during the operation there is a carry (in addition) or a borrow (in subtraction) from bit 3 to bit 4, and 0 if not. In other words, it shows whether there was an overflow from the low nibble (a group of 4 bits) to the upper nibble.

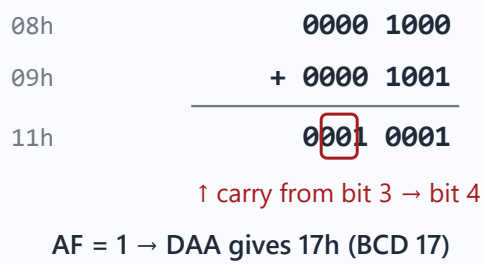


Figure 2.2.11: 08h + 09h = 11h. When the low nibbles are added, 8 + 9 = 17 and a carry comes out from bit 3 to bit 4: AF = 1. DAA uses this information to correct the result to BCD 17.

This flag is specific to **binary-coded decimal** (BCD) arithmetic. In BCD each decimal digit is held in a separate nibble: the number 17 is written 0001 0111 = 17h. When BCD numbers are added with a binary adder, the result is not always valid BCD. In the example in Figure 2.2.11, 08h + 09h gives 11h; the correct BCD answer is 17. Because the low nibble passed 16, the 6 of the decimal system has "been lost." AF = 1 says this loss happened. The `DAA` (decimal adjust after addition) instruction looks at AF and adds 6 to the low nibble: 11h + 06h = 17h. After DAA, AL = 17h.

One property of AF is that the programmer cannot reach it directly. There is no conditional jump instruction that tests AF. There is no special instruction that sets or clears AF either. Only DAA, DAS, AAA, and AAS use it. AF can still be read and written: `LAHF` and `PUSHF` copy it together with the other flags into a register or onto the stack; `SAHF` and `POPF` write it back.

Example 2.2.6

What is AF in the operations `05h - 07h` and `9Ch + 64h` ?

In 05h − 07h the low nibbles are 5 − 7; since 5 is less than 7, a borrow is taken from bit 4: AF = 1. In 9Ch + 64h the low nibbles are C + 4 = 12 + 4 = 16; since it exceeds 15, a carry passes to bit 4: AF = 1.

2.2.6.6 CF: the carry flag

The **carry flag** (CF) is 1 if in an n-bit addition the result does not fit in n bits, that is, a carry comes out beyond the top bit. In subtraction, if a larger number is subtracted from a smaller one and a **borrow** is taken, CF = 1 as well. CF is the overflow indicator for **unsigned** numbers.

Operation (8 bits)	Unsigned interpretation	Result	CF
`FFh + 01h`	255 + 1 = 256 > 255	00h	1
`80h + 80h`	128 + 128 = 256 > 255	00h	1
`70h + 10h`	112 + 16 = 128 ≤ 255	80h	0
`05h - 07h`	5 − 7 < 0 (borrow)	FEh	1

Operation (8 bits)	Unsigned interpretation	Result	CF
`07h - 05h`	7 − 5 = 2	02h	0

Table 2.2.18: CF in 8-bit operations. The unsigned interpretation explains why the result does or does not fit.

In the example 05h − 07h the result is FEh. In the unsigned interpretation this is 254 and is not the true result −2; CF = 1 reports this. The same FEh is −2 in the signed interpretation and is correct. So in a signed subtraction CF = 1 is not an error; whether there is an error is told by OF.

CF has a few properties that differ from the other flags:

- **The programmer can change it.** `STC` makes CF 1, `CLC` makes it 0, and `CMC` flips it. CF is the only status flag that can be set and cleared directly.
- **INC and DEC do not touch CF.** With AL = FFh, `ADD AL, 1` makes CF = 1; `INC AL` produces the same result but leaves CF at its previous value. This design is so that changing a loop counter with INC/DEC does not disturb the carry of a multiword addition.
- In **shift and rotate** instructions the last bit pushed out goes to CF. `SHL AL, 1` with AL = 81h gives AL = 02h and CF = 1.
- **Multiword arithmetic.** `ADC` (add with carry) adds CF to the sum, and `SBB` (subtract with borrow) subtracts CF. In this way a 16-bit processor can add 32- or 64-bit numbers piece by piece.

A 32-bit addition is the best-known use of ADC. Let 1 be added to DX:AX = 0001FFFFh:

```
MOV AX, 0FFFFh   ; low word
MOV DX, 0001h    ; high word
ADD AX, 1        ; AX = 0000h, CF = 1
ADC DX, 0        ; DX = 0001h + 0 + CF = 0002h
```

In the low word FFFFh + 1 = 10000h; AX = 0000h and the overflowing 1 waits in CF. ADC adds both the 0 and CF to the high word: DX = 0002h. The result is DX:AX = 00020000h. Just as in a decimal addition the carry moves to the digit on the left, here CF has been carried from word to word.

2.2.6.7 OF: the overflow flag

The **overflow flag** (OF) is 1 if the result, interpreted as **signed**, does not fit in the number of bits used. In 8 bits the signed range is −128 ... +127, and in 16 bits −32,768 ... +32,767. CF shows unsigned overflow and OF shows signed overflow.

There are two equivalent ways to find OF for addition:

- **The sign rule.** If two positive numbers are added and the result is negative, or two negative numbers are added and the result is positive, OF = 1. The sum of two numbers of different signs never overflows, because the result stays between the two numbers.

- **The carry rule.** If the carry into the MSB and the carry out of the MSB differ, OF = 1 ($c_7 \oplus c_8$ in Table 2.2.16).

For subtraction the sign rule is adapted as follows: if a negative is subtracted from a positive and a negative result is found, or a positive is subtracted from a negative and a positive result is found, OF = 1. For example, in 80h − 01h = 7Fh, −128 − 1 = −129 was meant to be computed and the result came out +127: OF = 1. The difference of two numbers with the same sign never overflows.

Table 2.2.19 gives the four possible combinations of CF and OF with an example each. Each row must be read both as unsigned and as signed.

Operation	Unsigned interpretation	Signed interpretation	Result	CF	OF
`05h + 03h`	5 + 3 = 8	5 + 3 = 8	08h	0	0
`7Fh + 01h`	127 + 1 = 128	127 + 1 ≠ −128	80h	0	**1**
`FFh + 01h`	255 + 1 ≠ 0	−1 + 1 = 0	00h	**1**	0
`80h + 80h`	128 + 128 ≠ 0	−128 + (−128) ≠ 0	00h	**1**	**1**

Table 2.2.19: The four combinations of CF and OF (8-bit addition).

The rows can also be checked with the carry rule. In 7Fh + 01h all of the low seven bits are 1; in the addition a carry goes into bit 7 ($c_7 = 1$) but no carry comes out of bit 7 ($c_8 = 0$). The two differ: OF = 1. In FFh + 01h both c_7 and c_8 are 1: OF = 0, CF = 1. In 80h + 80h no carry goes into bit 7 ($c_7 = 0$), but the sum of the two 1s produces a carry out of bit 7 ($c_8 = 1$): OF = 1, CF = 1.

CF and OF are independent of each other; all four combinations are seen in the table. The processor always computes both. If the program works with unsigned numbers it looks at CF (and jumps such as `JA` and `JB`), and if it works with signed numbers it looks at OF (and jumps such as `JG`, `JL`, and `JO`).

Example 2.2.7

After `MOV AL, 0C8h` and `ADD AL, 64h`, what are AL and the six status flags?

C8h + 64h = 12Ch. The low byte 2Ch is written to AL, and the overflowing 1 goes to CF: CF = 1. In the signed interpretation C8h = −56 and 64h = +100; −56 + 100 = 44 = 2Ch is correct, and the sum of numbers of different signs does not overflow: OF = 0. 2Ch = 0010 1100b; since the MSB is 0, SF = 0, and since the result is not zero, ZF = 0. The low byte has three 1s: PF = 0. The low nibbles are 8 + 4 = 12, which does not exceed 15: AF = 0.

Common mistake

Using the word “overflow” for both flags and counting CF = 1 as an error in a signed computation. In −1 + 1 (FFh + 01h), CF = 1 but the signed result 0 is perfectly correct. Only OF

tells whether a signed computation is correct.

INC and DEC set OF too. With AL = 7Fh, `INC AL` makes AL = 80h; +127 + 1 has exceeded the signed range and OF = 1, SF = 1. In the NEG instruction the only overflow case is the smallest negative number: `NEG AL` with AL = 80h again produces 80h, because +128 cannot be represented in 8 bits; in this case OF = 1.

2.2.7 The control flags

DF, IF, and TF do not describe the result of an operation; they determine how the processor will work from now on. Arithmetic instructions do not touch these flags. The programmer sets them with special instructions or by changing the flags register through the stack.

2.2.7.1 DF: the direction flag

The **direction flag** (DF) determines in which direction the string instructions walk through memory (Figure 2.2.12):

- **DF = 0**: from a low address toward a high one; at each step SI and DI **increase**. The `CLD` instruction makes DF 0.
- **DF = 1**: from a high address toward a low one; at each step SI and DI **decrease**. The `STD` instruction makes DF 1.

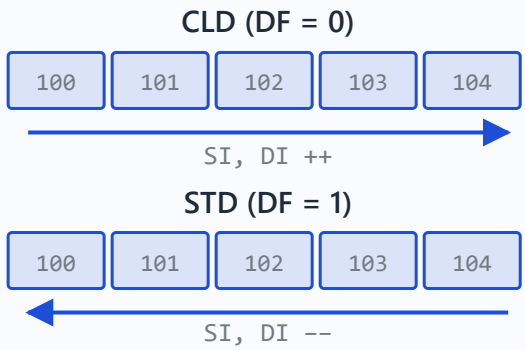


Figure 2.2.12: With DF = 0 the string instructions advance by increasing the addresses, and with DF = 1 by decreasing them.

The step size depends on the width of the instruction: 1 for byte instructions (MOVSB, LODSB ...) and 2 for word instructions (MOVSW, LODSW ...). DF affects only the string instructions. An ordinary instruction such as `MOV AL, [SI]` never changes SI.

DF = 0 is enough to process an array from start to end, and most programs use it. For this reason, instead of trusting the value of DF, it is a good habit to write `CLD` or `STD` before every string operation. If another routine has left DF at 1, a copy loop that does not write CLD walks through memory in the reverse direction.

The real justification for walking backward is **overlapping areas**. If the source and destination overlap in memory, the direction of the copy determines the result. If the destination is ahead of the source (at a higher address), the copy must be done from the end to the start. Otherwise the not-yet-read bytes of the source get overwritten by the bytes written to the destination.

2.2.7.2 Code trace: overlapping copy with DF = 1

The program in the slides shifts the string “KLMN” in memory one byte to the right. The source is arr..arr+3 and the destination is arr+1..arr+4; the two areas overlap in three bytes.

```
org 100h
mov si, offset arr+3 ; source
mov di, offset arr+4 ; destination
mov cx, 4
std                  ; DF = 1
rep movsb
cld
ret
arr db 'KLMN', 0
```

The program starts at 0100h. The instructions take 3 + 3 + 3 + 1 + 2 + 1 + 1 = 14 bytes; so `arr` is placed at address 010Eh. 'K' is at 010Eh, 'L' at 010Fh, 'M' at 0110h, 'N' at 0111h, and the terminating 0 at 0112h. SI is set to point to the last byte of the source (arr+3 = 0111h) and DI to the last byte of the destination (arr+4 = 0112h).

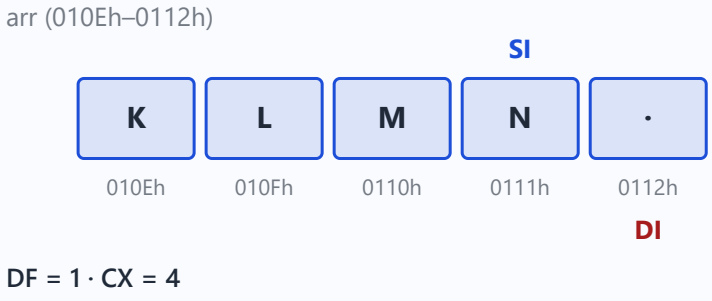


Figure 2.2.13: After `std`, before the copy: SI = 0111h ('N'), DI = 0112h, CX = 4, DF = 1.

Row	Instruction	SI	DI	CX	DF	010Eh … 0112h
2	`mov si, offset arr+3`	**0111h**	0000h	0	0	K L M N ·
3	`mov di, offset arr+4`	0111h	**0112h**	0	0	K L M N ·

Row	Instruction	SI	DI	CX	DF	010Eh … 0112h
4	`mov cx, 4`	0111h	0112h	**4**	0	K L M N ·
5	`std`	0111h	0112h	4	**1**	K L M N ·
6	`rep movsb` (pass 1)	**0110h**	**0111h**	**3**	1	K L M N **N**
6	`rep movsb` (pass 2)	**010Fh**	**0110h**	**2**	1	K L M **M** N
6	`rep movsb` (pass 3)	**010Eh**	**010Fh**	**1**	1	K L **L** M N
6	`rep movsb` (pass 4)	**010Dh**	**010Eh**	**0**	1	K **K** L M N
7	`cld`	010Dh	010Eh	0	**0**	K K L M N

Table 2.2.20: Trace of the backward overlapping copy. The memory column is the bytes 010Eh–0112h (· = 0).

In the first pass the 'N' at 0111h is copied to 0112h. The terminating zero is overwritten, but this byte is not part of the source. In the second pass the 'M' at 0110h is written to 0111h. The 'N' at 0111h had already been read and moved in the previous pass; overwriting it does no harm. Each pass overwrites the byte read in the previous pass, not the byte to be read in the next. After the fourth pass memory holds “KKLMN”: the string has shifted one place right, and the 'K' at the start has stayed in place (Figure 2.2.14). The `cld` on the last line returns DF to the value the rest of the program expects.

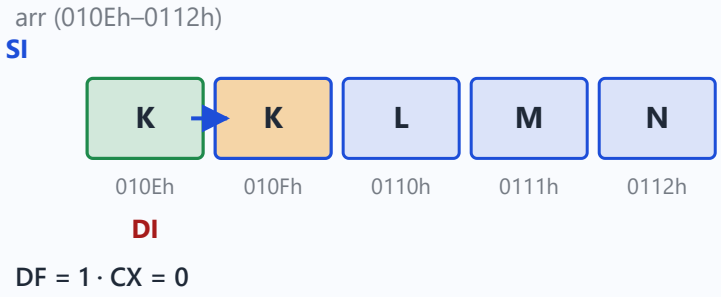


Figure 2.2.14: After four passes memory is “KKLMN.” SI and DI point one byte before the copied area; CX = 0.

Let us try the same shift in the forward direction with CLD: SI = arr, DI = arr+1, CX = 4. The first pass writes 'K' to 010Fh and overwrites 'L'. The second pass reads from 010Fh, but 'K' is there now; 'K' is written to 0110h. Every pass reads the 'K' written by the previous pass. The result is “KKKKK.” The rule is summarized like this: if the destination is ahead of the source, copy backward (STD); if behind it, copy forward (CLD).

Common mistake

Setting SI and DI to the **start** of the areas when copying backward with STD. With DF = 1 the string instruction decrements addresses; SI and DI must point to the **last** byte of the areas. In a word copy (MOVSW) the address of the last word is given, that is, the address one before the last byte.

2.2.7.3 IF: the interrupt flag

The 8086 has an input pin called INTR. When a peripheral such as the keyboard, a timer, or a serial port wants service from the processor, it activates this pin. The processor finishes the current instruction, interrupts the program, and runs an interrupt service routine (ISR). The **interrupt flag** (IF) determines whether these requests are accepted:

- **IF = 1**: maskable interrupt requests coming on INTR are accepted. `STI` makes IF 1.
- **IF = 0**: these requests are ignored, that is, **masked**. `CLI` makes IF 0.

IF does not block everything. The **non-maskable interrupt** coming on the NMI pin is not affected by IF; it is reserved for events that cannot be postponed, such as a power failure or a memory parity error. Software interrupts the program starts itself with the `INT n` instruction do not look at IF either. IF filters only the requests that come from outside, on the INTR pin.

The stack example shows well why a program may want to turn interrupts off. To move the stack, both SS and SP are changed. If an interrupt arrives after SS has changed but before SP has, the processor pushes FLAGS, CS, and IP onto a half-built, wrong stack. These two instructions are a unit that must not be interrupted:

```
CLI             ; disable interrupts
MOV AX, 9000h
MOV SS, AX       ; new stack segment
MOV SP, 0FFFEh   ; new stack top
STI             ; enable interrupts
```

The 8086 itself takes a precaution against this danger: after an instruction that writes to a segment register, it accepts no interrupt until the next instruction completes. If `MOV SP, …` is written right after `MOV SS, AX`, the two instructions run uninterrupted. Even so, the CLI/STI pair shows the intent clearly. Keeping interrupts off for a long time is harmful: keystrokes can be lost and the clock can fall behind. The code between CLI and STI is kept as short as possible.

When the processor accepts an interrupt it makes IF 0 itself; so the ISR is not immediately interrupted by a new interrupt of the same kind. The `IRET` at the end of the routine restores the old FLAGS from the stack, and IF returns to its old value, that is, 1.

2.2.7.4 TF: the trap flag

The **trap flag** (TF) was put in for on-chip debugging. When TF = 1 the processor works in **single-step** mode: after each instruction finishes, it automatically raises a type 1 interrupt (INT 1) (Figure 2.2.15).

Figure 2.2.15: With TF = 1, INT 1 occurs after every instruction: FLAGS, CS, and IP are pushed onto the stack, TF is cleared, the single-step routine runs, and IRET returns to the next instruction.

The flow is as follows. The program executes an instruction. Because TF = 1, the processor pushes FLAGS, CS, and IP onto the stack, clears TF and IF, and jumps to the routine the type 1 interrupt vector points to. This routine is part of a debugger: it prints the registers on the screen and waits for the user to press a key. While the routine runs, TF = 0, so it is not itself single-stepped. The IRET at the end takes FLAGS from the stack; TF becomes 1 again and the program's next instruction is executed. Then the cycle repeats. The `T` (trace) command in DOS's DEBUG program and emu8086's "single step" button use this arrangement.

There is no instruction that sets or clears TF directly. There are STC/CLC for CF, STD/CLD for DF, and STI/CLI for IF; for TF there is none. TF is set by changing the whole flags register through the stack. TF is bit 8 of the flags register and its mask is 0100h; the step-by-step trace of the method is in Table 2.2.26.

Flag	Bit	When 1	When 0	Instructions that set/clear
DF	10	string addresses decrease	string addresses increase	`STD` / `CLD`
IF	9	INTR interrupts are accepted	INTR interrupts are masked	`STI` / `CLI`
TF	8	INT 1 after every instruction	normal operation	none; use PUSHF/POPF

Table 2.2.21: Comparison of the control flags.

2.2.8 Instructions that change the flags

2.2.8.1 The effect of instructions on the flags

The flags change in two ways: some instructions set or clear them directly, while arithmetic and logic instructions update them as a side effect. Table 2.2.22 gives the direct instructions.

Instruction	Effect	Instruction	Effect
`CLC`	CF = 0	`CLD`	DF = 0
`STC`	CF = 1	`STD`	DF = 1
`CMC`	CF = ¬CF	`CLI` / `STI`	IF = 0 / 1
`LAHF`	AH ← SF ZF – AF – PF – CF	`SAHF`	AH → low flags byte
`PUSHF`	push FLAGS onto the stack	`POPF`	load FLAGS from the stack

Table 2.2.22: Instructions that change the flags directly.

The effect of instruction groups on the status flags is collected in Table 2.2.23. The table helps find which instruction last changed the flags before a conditional jump.

Instruction group	Effect on the flags
ADD, ADC, SUB, SBB, CMP, NEG	all of CF, PF, AF, ZF, SF, OF are updated
INC, DEC	all except CF are updated
AND, OR, XOR, TEST	CF = OF = 0; PF, ZF, SF are updated; AF undefined
MUL, IMUL	CF = OF = whether the high half of the result is significant; SF, ZF, AF, PF undefined
DIV, IDIV	all six status flags undefined
SHL, SHR, SAR	CF = the last bit shifted out; PF, ZF, SF are updated
ROL, ROR, RCL, RCR	only CF and OF change
MOV, PUSH, POP, XCHG, LEA, LOOP, JMP, Jcc	do **not** affect the flags

Table 2.2.23: The effect of instruction groups on the status flags.

The word “undefined” means that Intel has not documented the value of the flag after that instruction. The real processor leaves some value, but this value can vary with the processor model. A correct program does not look at an undefined flag. Another subtle point is TEST and CMP: both compute like AND and SUB but do not write the result, only set the flags. For example, `TEST AL, 80h` tests bit 7 of AL. With AL = B5h the result is 80h, giving ZF = 0 and SF = 1; AL does not change.

2.2.8.2 Code trace: LAHF, SAHF, and CMC

LAHF (load AH from flags) copies the low byte of the flags register into AH. **SAHF** (store AH into flags) works in the opposite direction: it writes AH into the low byte of the flags register. The layout of the low byte is as follows:

Bit	7	6	5	4	3	2	1	0
Flag	SF	ZF	–	AF	–	PF	–	CF
Value	SF	ZF	0	AF	0	PF	1	CF

Table 2.2.24: The low byte of the flags register (the LAHF/SAHF layout).

Bit 1 always reads as 1 on the 8086; bits 3 and 5 are 0. Because OF, DF, IF, and TF are in the high byte, LAHF and SAHF cannot reach them. The program in the slides traces these two instructions together with three instructions that change CF:

```
org 100h
mov al, 0FFh
add al, 1       ; CF ZF AF PF = 1
lahf            ; AH ← flags
cmc             ; flip CF
clc             ; CF = 0
stc             ; CF = 1
mov ah, 0C1h    ; 1100 0001b
sahf            ; flags ← AH
ret
```

Row	Instruction	AH	SF	ZF	AF	PF	CF
2	`mov al, 0FFh`	00h	0	0	0	0	0
3	`add al, 1`	00h	0	**1**	**1**	**1**	**1**
4	`lahf`	**57h**	0	1	1	1	1
5	`cmc`	57h	0	1	1	1	**0**
6	`clc`	57h	0	1	1	1	0
7	`stc`	57h	0	1	1	1	**1**
8	`mov ah, 0C1h`	**C1h**	0	1	1	1	1
9	`sahf`	C1h	**1**	1	**0**	**0**	1

Table 2.2.25: Trace of LAHF, SAHF, and CMC.

In the third row FFh + 1 = 00h is computed. Four flags are set at once: the result did not fit in 8 bits (CF = 1), the result is zero (ZF = 1), the low nibble F + 1 overflowed (AF = 1), and 00h has zero 1s and zero is even (PF = 1). SF = 0, because the MSB of 00h is 0.

In the fourth row LAHF copies this state to AH. Written from left to right the bits are SF = 0, ZF = 1, 0, AF = 1, 0, PF = 1, 1, CF = 1: 0101 0111b = 57h (Figure 2.2.16). If the fixed 1 of bit 1 is forgotten, 55h is found; this is a common calculation mistake.

Figure 2.2.16: The flags byte and AH after `lahf`. AH = 57h = 0101 0111b; bit 1 is always 1.

In the fifth row CMC flips CF from 1 to 0. The CLC in the sixth row makes CF 0; since CF is already 0, there is no visible change. In the seventh row STC makes CF 1. These three instructions touch only CF; the table shows clearly that ZF, AF, and PF do not change. These instructions are used, for example, when a routine reports success or failure with CF, and to prepare the carry at the start of an ADC/SBB chain.

The eighth row writes C1h = 1100 0001b to AH. If this bit pattern is read in the layout of the flags byte, bit 7 (SF), bit 6 (ZF), and bit 0 (CF) are 1; AF (bit 4) and PF (bit 2) are 0. In the ninth row SAHF writes this byte into the flags: SF = 1, ZF = 1, CF = 1, AF = 0, PF = 0 (Figure 2.2.17). SF and ZF both being 1 is a situation a normal operation cannot produce: a result cannot be both zero and negative. SAHF brings the flags to the desired value independently of any result. OF, DF, IF, and TF are not affected by this instruction.

Figure 2.2.17: After `sahf`: AH = C1h has been written into the flags byte; SF = ZF = CF = 1, AF = PF = 0.

Example 2.2.8

After `MOV AL, 5`, `CMP AL, 5`, and `LAHF`, what is AH?

CMP computes 5 − 5 = 0: ZF = 1; since the result is 00h, PF = 1; since there is no borrow, CF = 0; since there is no nibble borrow, AF = 0; since the MSB is 0, SF = 0. The flags byte SF ZF 0 AF 0 PF 1 CF = 0 1 0 0 0 1 1 0 = 0100 0110b = 46h.

2.2.8.3 All the flags with PUSHF and POPF

LAHF and SAHF reach only the low byte. All 16 bits of the flags register are pushed onto the stack with **PUSHF** and taken back from the stack with **POPF**. Between the two instructions the word on the stack can be taken into a register and the desired bit changed. This is the only way to set TF, which has no direct instruction. The following program shows the same method on DF; DF is bit 10 and its mask is 0400h.

```
PUSHF            ; FLAGS → stack
POP  AX          ; AX = FLAGS
OR   AX, 0400h   ; bit 10 (DF) ← 1
PUSH AX
POPF             ; FLAGS ← AX
PUSHF            ; check:
POP  BX          ; BX = new FLAGS
```

Instruction	AX	BX	SP	DF	Top of the stack
`PUSHF`	0000h	0000h	**FFFCh**	0	F202h
`POP AX`	**F202h**	0000h	**FFFEh**	0	0000h
`OR AX, 0400h`	**F602h**	0000h	FFFEh	0	0000h
`PUSH AX`	F602h	0000h	**FFFCh**	0	F602h
`POPF`	F602h	0000h	**FFFEh**	**1**	0000h
`PUSHF`	F602h	0000h	**FFFCh**	1	F602h
`POP BX`	F602h	**F602h**	**FFFEh**	1	0000h

Table 2.2.26: Setting a flag with PUSHF/POPF (.COM start: IF = 1, other flags 0).

At the start IF = 1 and the other flags are 0. The word PUSHF pushes is F202h. Its breakdown is instructive: the F000h part is bits 12–15, which always read as 1 on the 8086; 0200h is bit 9, that is, IF = 1; 0002h is bit 1 and is always 1. `OR AX, 0400h` sets only bit 10 to 1 and keeps the other bits: F602h. POPF loads this value into the flags register and DF = 1. The last two rows are the check: when the flags are read again, BX = F602h.

For TF the same code is written with `OR AX, 0100h`. To clear a bit, AND with an inverted mask is used instead of OR: `AND AX, 0FEFFh` clears TF and `AND AX, 0FBFFh` clears DF. One benefit of working through the stack is that it preserves the other flags. When the change is finished, SP returns to the value it had at entry; the numbers of PUSHes and POPs are equal.

Another use of PUSHF/POPF is saving the flags across a routine. If a routine does PUSHF at its start and POPF at its end, the caller's flags are not disturbed, whatever arithmetic is done inside. Interrupts work on the same principle: the processor pushes FLAGS onto the stack on entering an interrupt, and IRET takes it back.

2.2.9 Common mistakes

The assembler cannot catch most mistakes about registers and flags; the program runs but gives a wrong result. The following mistakes are the most common in beginners' programs.

Common mistake

Thinking the carry of an 8-bit operation passes to the neighboring half. With AX = 12FFh, `ADD AL, 1` gives 1200h, not 1300h. The carry of an 8-bit operation goes to CF. If the carry is to be passed to AH, `ADC AH, 0` is written next, or the operation is done on 16 bits from the start.

Common mistake

Forgetting the square brackets or putting them in unnecessarily. `MOV AX, BX` copies the value of BX; `MOV AX, [BX]` reads the memory word BX points to. `MOV SI, [array]` loads into SI not the address of the array but its first word; for the address `MOV SI, OFFSET array` or `LEA SI, array` is written.

Common mistake

Not accounting for implicit operands. Changing CL for `SHL AX, CL` inside a loop while CX is the loop counter, or using a 16-bit MUL or DIV while keeping a value in DX. Which registers instructions silently change is collected in Table 2.2.3; a conflicting register is pushed onto the stack and popped back afterward.

Common mistake

Not clearing DX before DIV. To divide a 16-bit number in AX by BX, if only `DIV BX` is written, the dividend is DX:AX. If DX holds a leftover value from a previous operation, the quotient may not fit in 16 bits and the processor raises a divide-error interrupt. Before an unsigned division `MOV DX, 0` is written, and before a signed one `CWD`.

Common mistake

Thinking MOV changes the flags. Writing `MOV AX, 0` and then `JZ` does not test whether AX is zero; JZ looks at the ZF of the last arithmetic operation before the MOV. Whether a register is zero is tested with `CMP AX, 0`, `TEST AX, AX`, or `OR AX, AX`.

Common mistake

Confusing CF and OF. Testing overflow with OF in a program that works with unsigned numbers, or with CF in one that works with signed numbers. In 7Fh + 01h, CF = 0 but the signed result is wrong; in FFh + 01h, CF = 1 but the signed result is correct.

Common mistake

Forgetting bit 1 in the LAHF byte calculation. In the byte LAHF produces, bit 1 is always 1. With the flags SF = 0, ZF = 1, AF = 1, PF = 1, CF = 1, AH = 57h, not 55h.

Common mistake

Writing a constant to a segment register or confusing the segment with BP. `MOV DS, @data` cannot be written; it is taken into AX first. An access `[BP+SI]` is made in SS, not DS.

2.2.10 Exercises and solutions

The five exercises in the slides are solved here with their intermediate steps. The results can be checked by running them in emu8086.

1. With AX = 3A7Fh, what is AX after `MOV AL, 0` and then `INC AH`?

`MOV AL, 0` changes only the low byte: AH = 3Ah stays, AL = 00h, and AX = 3A00h. `INC AH` increments only the high byte: 3Ah + 1 = 3Bh. The result is AX = **3B00h**. Neither instruction touches the other half.

2. Find AL, CF, ZF, SF, OF, and PF after `MOV AL, 9Ch` / `ADD AL, 64h`.

First the binary addition: 1001 1100 + 0110 0100 = 1 0000 0000. The result is 100h; the low byte 00h is written to AL and the overflowing 1 goes to CF: CF = 1. Since the result is zero, ZF = 1, and since the MSB is 0, SF = 0. In the signed interpretation 9Ch = −100 and 64h = +100; −100 + 100 = 0 is correct, and the sum of two numbers of different signs does not overflow: OF = 0. 00h has zero 1s, an even number: PF = 1. The low nibbles are C + 4 = 16; a carry comes out of bit 3: AF = 1.

3. `MOV AL, 50h` / `ADD AL, 50h`: what are CF and OF? Why do they differ?

50h + 50h = A0h. In the unsigned interpretation 80 + 80 = 160, which is less than 255; the result fits in 8 bits: CF = 0. In the signed interpretation two positive numbers, +80 and +80, were added; +160 exceeds the upper limit of the signed range, +127. Because the MSB of A0h is 1, the result looks like −96. The sum of two positives came out negative: OF = 1 (SF is 1 too). The two flags are answers to different questions: CF reports unsigned overflow and OF signed overflow.

4. What is SP after 3 PUSHes and 1 POP when SP = 0100h?

Each PUSH subtracts 2 from SP and each POP adds 2: 0100h − 3 × 2 + 2 = 0100h − 6 + 2 = 0100h − 4 = **00FCh**. The intermediate values are 00FEh, 00FCh, 00FAh, and after the POP 00FCh. Two items remain on the stack.

5. How many times does a LOOP loop that starts with `MOV CX, 0` run?

LOOP decrements first and then tests. The first LOOP computes 0 − 1 = FFFFh and jumps. As CX goes down from FFFFh to 1 the loop runs 65,535 more times; with the first pass it runs **65,536** times in total. If the pass count is counted in the 32-bit pair DX:BX, 0001:0000h = 65,536 results. To prevent this case, `JCXZ` is put before the loop.

2.2.11 Summary

- A **register** is a very fast, small memory cell inside the processor. A register access does not go out on the external bus; a memory access takes a bus cycle of at least four clock cycles. On the 8086 there is no memory-to-memory MOV or ADD.
- The 8086 has **14 registers**: 4 data registers (AX, BX, CX, DX), 4 pointer and index registers (SP, BP, SI, DI), 4 segment registers (CS, DS, SS, ES), IP, and FLAGS. All are 16 bits; only AX, BX, CX, and DX are split into two 8-bit halves. Writing to one half does not change the other, and the carry of an 8-bit operation goes to CF, not to the neighboring half.
- Some instructions use registers **implicitly**: MUL/DIV → AX and DX; LOOP/REP → CX; shift count → CL; IN/OUT → AL/AX and, for ports above 255, DX; PUSH/POP/CALL/RET → SP; string instructions → DS:SI and ES:DI.
- Inside square brackets only **BX, BP, SI, and DI** can be used. Addresses that contain BP use SS, the others use DS.
- **SP** points to the top of the stack. PUSH first decrements SP by 2 and then writes; POP first reads and then increments SP by 2. **BP**, staying fixed throughout a procedure, allows parameters to be reached at fixed distances such as `[BP+4]`.
- A constant cannot be loaded into a segment register, and CS cannot be written with MOV. **Physical address** = segment × 10h + offset. **IP** is the offset of the next instruction; it advances by the instruction length and cannot be written with MOV.
- 9 bits of the flags register are used. **Status flags**: CF (unsigned overflow, borrow), PF (an even number of 1s in the low byte), AF (carry from bit 3 to bit 4), ZF (the result is zero), SF (the MSB of the result), OF (signed overflow). **Control flags**: DF (string direction), IF (permission for INTR interrupts), TF (single step).
- In hardware CF is the last carry, ZF is a NOR gate, SF is a copy of the MSB, PF is the inverse of an XOR tree, and OF is the XOR of the carries into and out of the MSB.
- CLC/STC/CMC change CF, CLD/STD change DF, and CLI/STI change IF. LAHF copies the low byte of the flags into AH, SAHF writes AH to this byte; PUSHF/POPF move the whole flags register to and from the stack. TF can be changed only with PUSHF/POPF.
- MOV, PUSH, POP, XCHG, LEA, LOOP, and the jump instructions do not affect the flags; INC and DEC do not touch CF; after MUL and DIV some flags are undefined.

2.2.12 Key terms

register register set general-purpose register accumulator base register counter register high / low byte implicit operand pointer register stack pointer (SP) base pointer (BP) index register source / destination index (DI) stack stack frame segment register offset physical address default segment instruction pointer (IP) flag flags register (FLAGS) status flag control flag carry flag (CF) parity flag (PF) auxiliary carry flag (AF) zero flag (ZF) sign flag (SF) overflow flag (OF) direction flag (DF) interrupt flag (IF) trap flag (TF) single-step maskable / non-maskable interrupt string instruction

2.2.13 Test Yourself

1. `INC BL` executes with BX = 12FFh and CF = 1. What do BX, ZF, and CF become?

   Answer

   INC works on the 8-bit BL: FFh + 1 = 00h, BH does not change, and BX = 1200h. Because the result is zero, ZF = 1. Because INC does not touch CF, CF stays 1; the reason it is 1 is not this INC but the earlier state.

2. `MOV AL, 0F0h`, `ADD AL, 10h`, and `ADC AH, 0` execute with AX = 0000h. What are AX and CF?

   Answer

   F0h + 10h = 100h: AL = 00h, CF = 1. ADC adds CF together with 0 to AH: AH = 00h + 0 + 1 = 01h. AX = 0100h. Because ADC's own addition produces no carry, CF = 0 in the end. Together the two instructions have done a 16-bit addition by carrying the 8-bit carry into the upper byte.

3. At the start of a .COM program (SP = FFFEh), `PUSH AX` executes with AX = ABCDh. What does SP become, and which byte is written to which address?

   Answer

   PUSH first decrements SP by 2: SP = FFFCh. The word is written low byte first: SS:FFFCh ← CDh, SS:FFFDh ← ABh. The word 0000h at FFFEh stays in place.

4. Is there a difference between `XOR AX, AX` and `MOV AX, 0` as far as AX is concerned? What is the

difference as far as the flags are concerned?

Answer

Both give AX = 0000h. XOR sets the flags according to the result: ZF = 1, PF = 1, SF = 0, CF = OF = 0 (AF undefined). MOV changes no flag; the flags left from the previous operation are preserved. XOR is also 2 bytes, while `MOV AX, 0` is 3 bytes.

5. After `MOV AX, 0FFFFh`, `MOV BX, 0002h`, `MUL BX`, what are DX, AX, CF, and OF?

Answer

FFFFh × 2 = 1FFFEh (65,535 × 2 = 131,070). The 32-bit result is 0001FFFEh: DX = 0001h, AX = FFFEh. Because the high half is not zero, CF = OF = 1. SF, ZF, AF, and PF are undefined.

6. What are CF and OF in the 8-bit operations `80h - 01h` and `7Fh - FFh`?

Answer

80h − 01h = 7Fh. Unsigned 128 − 1 = 127 is correct, no borrow: CF = 0. Signed −128 − 1 = −129 does not fit; the result came out +127: OF = 1. 7Fh − FFh = 80h. Unsigned 127 − 255 is negative, a borrow is taken: CF = 1. Signed +127 − (−1) = +128 does not fit: OF = 1.

7. With DF = 1 and SI = 0205h, if addresses 0204h and 0205h hold 11h and 22h, what are AL and SI after two `LODSB`s?

Answer

The first LODSB takes the 22h at DS:0205h into AL, and SI = 0204h. The second LODSB takes the 11h at 0204h, and SI = 0203h. The result is AL = 11h, SI = 0203h.

8. `MOV AL, 7Fh` and then `INC AL`: what do AL, SF, OF, and CF become?

Answer

AL = 80h. Because the MSB is 1, SF = 1. +127 + 1 = +128 does not fit in the signed range: OF = 1. INC does not change CF; CF keeps its previous value (0 in this program).

9. Why are interrupts turned off automatically at the start of an interrupt service routine? Is NMI

blocked during this time too?

Answer

When accepting an interrupt, the processor pushes FLAGS onto the stack and then clears IF and TF. Because IF = 0, new requests coming from INTR cannot interrupt the routine right away; if it wants, the routine turns interrupts on with STI. IRET restores the old FLAGS. Because NMI is not affected by IF, it is accepted during this time too.

2.2.14 Further reading

- Intel, *The 8086 Family User's Manual* (1979), Chapter 2: the register model, the full definition of the flags, and the clock-cycle count of every instruction are in this source; the values in Table 2.2.1 are taken from here.
- B. B. Brey, *The Intel Microprocessors* (8th edition), Chapters 2 and 3: the programming model, addressing modes, and default segments, along with the extension from the 8086 to the current x86.
- M. A. Mazidi, J. G. Mazidi, *The x86 PC: Assembly Language, Design, and Interfacing* (5th edition), Chapter 1: registers, the flags register, and many examples in which the flags are computed by hand.
- M. Morris Mano, *Digital Design*: the gate-level design of adders, parity generators, and registers; for the digital-design basis of the flag circuits.

2.2.15 References

1. Intel, *The 8086 Family User's Manual*, 1979.
2. Barry B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, 2009.
3. Muhammad Ali Mazidi, Janice G. Mazidi, *The x86 PC: Assembly Language, Design, and Interfacing*, 5th edition, 2010.
4. emu8086 documentation: the *8086 instructions* and *Addressing modes* sections.
5. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*.

PART I · CHAPTER 3

Assembly Language

Software can be written at different levels of abstraction, from machine language up to high-level languages. The chapter looks at machine language and instruction formats, the roles of the assembler and the linker, assembly languages on different platforms, and compiled versus interpreted languages. The assembly part covers the structure of a source line (label, mnemonic, operands and comment), the difference between instructions and directives, operand types and implicit operands. Instruction timing, how IP advances, the flags and a first program written in emu8086 close the chapter.

NOTES IN THIS CHAPTER

3.1 Programming Languages

3.2 Assembly Language

PART I · CHAPTER 3.1

Programming Languages

A processor recognizes only bits. Every byte it reads from memory is either part of an instruction or a piece of data, and the flow of the program decides which. A person, on the other hand, thinks "add a and b and store the result in x." Programming languages were invented to close the gap between these two worlds. These notes climb that gap layer by layer. At the bottom is machine language, which the processor executes directly; we follow, step by step, how a few bytes of 8086 code are fetched and executed from memory. On a small, made-up instruction format we encode and decode instructions by hand. Next come assembly language, the two passes of the assembler, the linker, and the loader. Writing the same "Welcome" program in six different environments shows how strongly assembly code depends on both the processor and the operating system. The last sections divide high-level languages into compiled, interpreted, and hybrid ones. A program that decides whether a number is odd or even is read in fourteen languages and, finally, in 8086 assembly.

LEARNING OUTCOMES

Distinguish the language levels
Explain the difference between machine language, assembly language, and high-level languages in terms of abstraction, portability, and control over the hardware.

Read and write machine code
Encode and decode binary instructions for a given instruction format; trace how 8086 bytes such as `A1 00 02` are fetched from memory and executed.

Describe the toolchain
Explain the two passes of the assembler, the symbol table, the linker, and the loader; read a listing file.

Show platform dependence
Explain, through system calls and calling conventions, why the same program is written differently for x86, 68000, ARM, PowerPC, and different operating systems.

Compare compilation and interpretation
Compare compilers, linkers, interpreters, intermediate code, and JIT compilation in terms of speed, portability, and error detection.

Read across languages
Trace the same algorithm in several languages and in 8086 assembly; show how `for` and `if` constructs turn into comparisons and conditional jumps.

3.1.1 Software and language levels

3.1.1.1 The role of software and its layers

The hardware of a computer does nothing on its own. The processor, memory, and input/output units are ready to work the moment power is applied, but they do not know what to do or in what order. **Software** supplies that knowledge. Software is the body of instructions and data that makes the hardware work correctly, in coordination, and in the desired way. In an electronic circuit a control unit determines the sequence of signals; in a computer, the program in memory determines that sequence.

The most fundamental piece of software is the **operating system**. The operating system manages the hardware: it shares memory among programs, arbitrates access to the disk and the screen, and keeps concurrently running programs from corrupting one another. Application programs do not touch the hardware directly; when they want to open a file or write text to the screen, they ask the operating system for a service. Such a request is called a **system call**, and we will see its details in the “Welcome” programs. The operating system in turn reaches the hardware through **drivers**, which know the device-specific details.

Figure 3.1.1 shows this arrangement as four layers. The user is at the top and the hardware at the bottom. Each layer uses only the services offered by the layer immediately below it. Applications such as a browser, a compiler, or a game use the operating system, and the operating system uses the processor, memory, and input/output units.

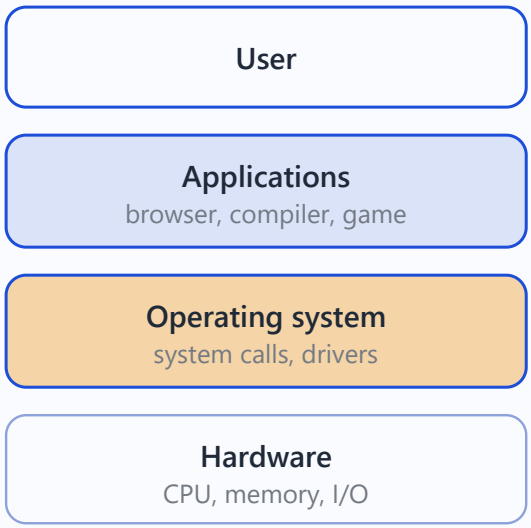


Figure 3.1.1: The layers of a computer system. Applications reach the hardware through the operating system; the operating system serves system calls and manages the hardware with drivers.

Programming languages are used to write applications, operating systems, and drivers. A programming language is an artificial language, with precisely defined rules, for describing a task step by step. Today there are hundreds of them. Some target operating systems, some numerical computation, some web pages. The history of languages also mirrors the history of computer technology. The first computers were programmed with wires and switches. As memories grew and processors got faster, languages appeared that shifted the programmer's burden onto the machine.

3.1.1.2 Three language levels

Programming languages fall into three levels according to how close they are to the hardware. Figure 3.1.2 stacks these levels; as we move up, **abstraction** increases, that is, the programmer sees fewer of the hardware's details.

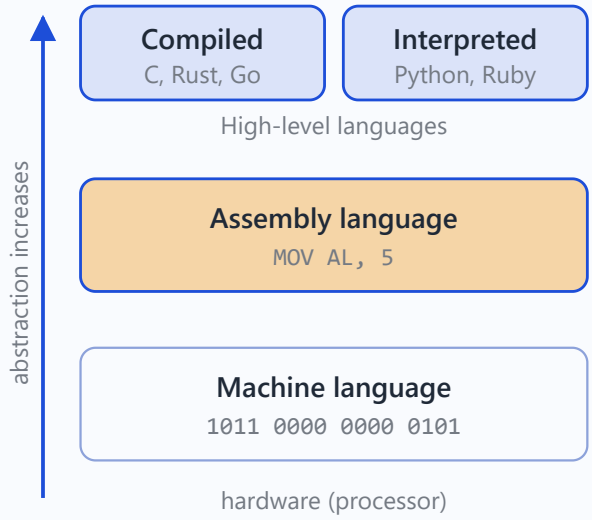


Figure 3.1.2: Language levels. At the bottom is machine language, which the processor executes directly; above it is assembly language, the symbolic notation for machine instructions; at the top are the compiled and interpreted high-level languages.

Definition 3.1.1

Machine language consists of the binary instructions that the processor executes directly, with no translation. **Assembly language** writes each machine instruction with a memorable name such as `MOV` or `ADD`, that is, a **mnemonic**. An assembly line usually corresponds to a single machine instruction. A **high-level language** is written with variables, expressions, loops, and functions, independently of the processor's instructions; one line usually turns into several machine instructions.

High-level languages split further into two groups. In **compiled** languages (C, Rust, Go) the source code is translated entirely into machine language before the program runs. The resulting file runs directly on the processor. In **interpreted** languages (Python, Ruby) the source code is read and processed at run time by a program called an **interpreter**. The examples in the figure show the distinction: the line `MOV AL, 5` is assembly language, and its equivalent, the bits `1011 0000 0000 0101`, is machine language.

As the level rises, three things change together. The first is **readability**. Anyone who reads the line `x = a + b` understands what is being done; decoding the machine code for the same job requires an instruction table. The second is **portability**. A program written in C can be compiled for x86 as well as for ARM, whereas 8086 machine code runs only on the 8086 family. The third is **direct control**, and it moves the opposite way. In a high-level language the compiler decides which register holds a variable and which jump instruction builds a loop. The assembly programmer chooses every register, every flag, and every memory access personally. For hardware-near work such as device drivers, interrupt handlers, or bootloaders, this control is necessary.

For an electrical and electronics engineer, these levels can be thought of through a familiar analogy. Machine language is the bit patterns applied to the control inputs of a digital circuit. Assembly language is a symbol list that gives those patterns names. A high-level language resembles describing behavior in a hardware description language and leaving the details to the synthesis tool. The job is the same at every level; what changes is who fills in the details.

3.1.1.3 The same operation at three levels

The best way to see the difference between the levels is to write the same job at all three. The job is simple: add the variables `a` and `b` and store the result in the variable `x`. All three variables are 16-bit integers.

In C this job is a single line:

```
x = a + b;
```

This line does not say where the variables live in memory or in which register the addition is performed. Those decisions are left to the compiler. In 8086 assembly language the same job takes three lines:

```
MOV AX, [a]     ; load a into AX
ADD AX, [b]     ; AX = AX + b
MOV [x], AX     ; store the result in x
```

Here everything is explicit. The addition is done in the AX register. The square brackets mean the contents of the memory cell at the variable's address. On the 8086, two memory operands cannot be used together in one instruction. For that reason `a` is first brought into a register, then `b` is added, and finally the result is written back to memory. Each line is a single machine instruction.

In machine language only bytes remain. Let the variables' addresses be `a` = 0200h, `b` = 0202h, and `x` = 0204h. The suffix “h” indicates that the number is written in hexadecimal. The assembler translates these three lines into the following bytes:

```
A1 00 02        ; MOV AX, [0200h]
03 06 02 02     ; ADD AX, [0202h]
A3 04 02        ; MOV [0204h], AX
```

The first byte of each instruction is the **opcode**: it tells the processor which operation to perform. `A1` means “load the memory word into AX,” and the address follows immediately. `A3` is its opposite: “store AX to memory.” These two are short forms specific to AX. `03`, by contrast, is a general addition instruction: “add a register or a memory word to a 16-bit register.” Which register and which memory

cell is determined by the second byte, called the ModR/M byte. `06` = `00 000 110` b; the middle `000` encodes AX, while the outer `00` and `110` encode the case "the address is given directly inside the instruction."

The order of the address bytes looks reversed at first glance: the address 0200h is written as `00 02`. The 8086 stores multi-byte numbers in memory **low byte first** (**little-endian**). The low byte of 0200h is 00h and the high byte is 02h; 00h comes first, then 02h. The same rule applies to data: if `a` = 0015h, then address 0200h holds 15h and address 0201h holds 00h.

> **Common mistake**
>
> A common mistake is to read the address bytes in machine code in the order they are written. The address in the instruction `A1 00 02` is 0200h, not 0002h. On the 8086, every 16-bit number inside an instruction, whether an address or a constant, is written low byte first. When you read a word in a memory dump, the second byte is likewise the high byte.

3.1.1.4 Code trace: `x = a + b` in memory

To trace how these three instructions run in memory, let us put them into a small COM program. A COM program is the simplest executable file type, used by emu8086 and DOS. Code, data, and stack live in the same 64 KB segment; the program starts at offset 0100h of the segment.

```
org 100h
mov ax, [0200h]   ; A1 00 02
add ax, [0202h]   ; 03 06 02 02
mov [0204h], ax   ; A3 04 02
ret               ; C3
```

`org 100h` is a **directive**: it generates no machine code and only tells the assembler that the first byte will be placed at address 0100h. The code is the 11 bytes in the range 0100h–010Ah: the three instructions take 3 + 4 + 3 = 10 bytes, and `ret` takes 1 byte. Let `a` = 0015h (21) and `b` = 0027h (39) initially; the value of `x` is unknown.

The processor handles each instruction in two phases. In the **fetch** phase the instruction bytes are read from the address IP points to, and IP advances by the instruction length. In the **execute** phase the opcode is decoded and the work is done. On the 8086 these two phases are divided between two separate units. The **bus interface unit** (BIU) reads instruction bytes from memory and fills a six-byte **instruction queue**. The **execution unit** (EU) takes bytes from this queue, decodes them, and executes them. If the bus is idle while the execution unit is working, the BIU prefetches the next bytes. The trace table shows the two phases on separate rows, but in a real processor the execution of one instruction and the fetching of the next can overlap in time.

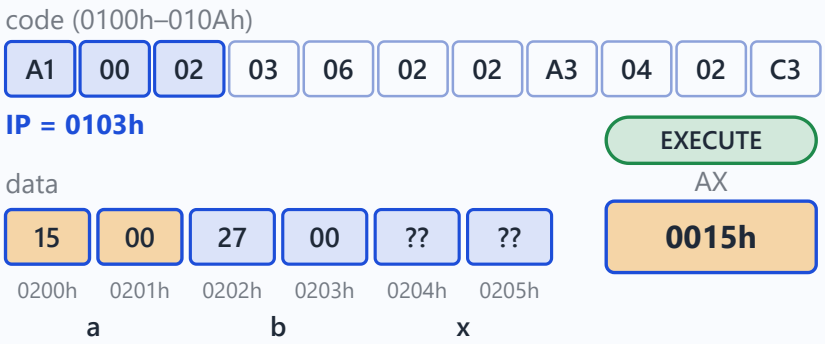
Step	Phase	Instruction bytes	IP	AX	0204h–0205h (x)
1	fetch	`A1 00 02`	0100h → 0103h	????	?? ??
2	execute	read a word from 0200h: `15 00`	0103h	0015h	?? ??
3	fetch	`03 06 02 02`	0103h → 0107h	0015h	?? ??
4	execute	`27 00` from 0202h; 15h + 27h	0107h	003Ch	?? ??
5	fetch	`A3 04 02`	0107h → 010Ah	003Ch	?? ??
6	execute	write AX to 0204h	010Ah	003Ch	3C 00
7	fetch, execute	`C3` (RET)	010Ah → return	003Ch	3C 00

Table 3.1.1: Trace of the machine code for `x = a + b` (a = 0015h, b = 0027h).

In step 1, IP = 0100h. When the processor reads the byte `A1`, it learns that the instruction is three bytes long; the following bytes `00 02` form the address. The address is read little-endian: 0200h. IP advances to 0103h. In step 2 the execution unit asks for a word at address 0200h of the data segment. The BIU reads the 15h at 0200h and the 00h at 0201h; AX becomes 0015h. This read is a real bus cycle. The 8086's data bus is 16 bits wide and 0200h is an even address. The two bytes of the word therefore arrive in a single bus cycle, that is, in at least four clock periods. Had the word been at an odd address, two separate cycles would have been needed.

In step 3 the four-byte `03 06 02 02` is fetched. `03` carries the addition operation, `06` the information “destination AX, source direct address,” and `02 02` the address 0202h. In step 4, 0027h is read from 0202h and added to AX in the arithmetic logic unit: 0015h + 0027h = 003Ch, which is 21 + 39 = 60 in decimal. The addition also updates the flags: since the result is nonzero, ZF = 0; since the most significant bit is 0, SF = 0; and since there is no carry out, CF stays 0.

In step 5, `A3 04 02` is fetched, and in step 6 AX is written to memory. The write is also done low byte first: 3Ch goes to address 0204h and 00h to address 0205h. Now `x` = 003Ch = 60. The byte `C3` in step 7 is the RET instruction; the COM program hands control back to the system that ran it. Figure 3.1.3 shows three moments of this trace.



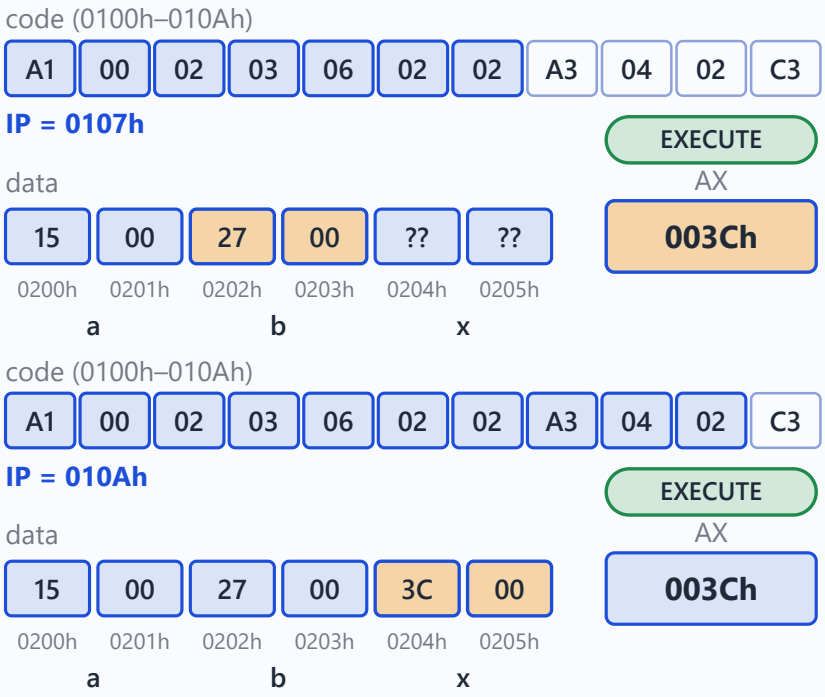


Figure 3.1.3: Three moments of the `x = a + b` machine code. At the top, the first instruction has executed and AX = 0015h; in the middle, the addition is done and AX = 003Ch; at the bottom, AX has been written to addresses 0204h and 0205h as 3C 00. Highlighted boxes are the bytes used in that step.

This trace shows what a single line of C turns into in hardware. The line has become three instructions and ten bytes of machine code. To these are added three data accesses and the bus cycles that fetch the instruction bytes. The timing table in the 8086 user's manual also gives the durations of these instructions. Loading memory into AX takes 10 clock periods. Adding a memory word to a register with direct addressing takes 9 + 6 = 15, and writing AX to memory again takes 10. Since one clock period is 200 ns on a 5 MHz 8086, these three instructions take about 35 × 200 ns = 7 µs. Periods spent waiting for fetches when the instruction queue runs empty may be added to these times.

Example 3.1.1

What happens to `x` and the flags if the same program is run with `a` = 7FFFh and `b` = 0001h?

The addition gives 7FFFh + 0001h = 8000h. Memory receives 0204h ← 00h and 0205h ← 80h. The result is not zero (ZF = 0). Because the most significant bit is 1, SF = 1. In unsigned addition there is no carry out of 16 bits (CF = 0). Viewed as signed numbers, however, the sum of two positives appears negative: +32767 + 1 yields −32768. This is a signed overflow and OF = 1. The processor merely records the overflow in the OF flag and carries on. C does not report signed overflow by itself either; the language standard treats this case as undefined behavior. Checking for overflow is the programmer's job.

3.1.2 Machine language

3.1.2.1 The only language the processor understands

Machine language is the lowest-level language, made of 0s and 1s. The processor's control unit decodes the bit pattern it reads from memory and drives the registers, the arithmetic logic unit, and the data bus accordingly. This decoding is done by decoder circuits: a given opcode activates given control lines. Machine language is therefore entirely **dependent on the processor and the hardware**. Machine code written for one processor family is meaningless to the decoder of another family, or does something completely different.

ENIAC (1945), one of the first general-purpose electronic computers, did not work with a program loaded into memory. The program was set up on the hardware by plugging cables between units and setting switches. Rewiring for a new problem could take days. The idea of placing the program in memory like data solved this problem. The instructions in memory were now numbers, and the language of those numbers was machine language.

Today every processor family has its own machine language, in other words its own **instruction set**. Machine language is impractical to read and write, so programmers do not use it directly. Even so, a microprocessor engineer must be able to decode machine code. Reading a memory dump, making sense of the bytes on a bus with a logic analyzer, or checking what a compiler produced all rest on this skill.

Figure 3.1.4 shows how the job “load 5 into register AL” is encoded in two different machine languages. On the 8086 the job takes two bytes: `B0 05`. The upper four bits of the first byte, `1011`, specify the “load a constant into a register” instruction. The next bit, `w` = 0, says that the operation is 8 bits wide. The last three bits, `000`, select the destination register; among the 8-bit registers, 000 is AL. The second byte is the constant itself: 00000101 = 5. Had the same instruction been written for the 16-bit AX, `w` would be 1 and the constant would take two bytes: `MOV AX, 5` = `B8 05 00`.

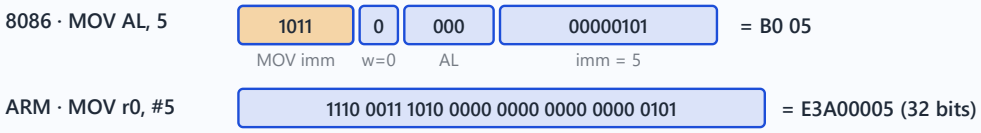


Figure 3.1.4: The same job in two machine languages. On the 8086, `MOV AL, 5` is the two-byte `B0 05`: opcode 1011, the `w` bit, the register field, and the constant. On ARM, `MOV r0, #5` is a single 32-bit word: `E3A00005`.

On ARM processors, by contrast, every instruction is 32 bits long. The instruction `MOV r0, #5` is translated into the word `E3A00005`. The top four bits of this word, `1110`, are the “always execute” condition. The following fields encode that the instruction is a data-processing instruction, that the operation is MOV, that the destination register is r0, and the lowest eight bits hold the constant 5. For

the same job the 8086 uses two bytes and ARM four. The position and meaning of the fields are completely different as well. This difference explains with a single example why 8086 code does not run on an ARM processor.

This dependence of machine language is softened by assembly languages. Assembly language is also tied to the processor, but it spares the programmer from fiddling with bit fields and makes programming friendlier.

3.1.2.2 A sample instruction format

The instruction set of a real processor contains hundreds of instructions and dozens of formats. To see the main idea of machine language, it suffices to design a much smaller, made-up processor. Let this processor have four registers (R0–R3) and let every instruction be 8 bits. Figure 3.1.5 shows the fields of the instruction.

Figure 3.1.5: The 8-bit instruction format of the sample processor. Bits 7–4 are the opcode, bits 3–2 the destination register, and bits 1–0 the source register.

Definition 3.1.2

The **instruction format** is the layout that specifies into which fields the bits of a machine instruction are divided and what each field encodes. The **opcode** field specifies the operation to perform, and the **operand** fields specify which register, memory cell, or constant the operation acts on.

In the sample processor the meaning of an instruction always follows one pattern: **destination ← destination operation source**. The result is written to the destination register; the source register does not change. Registers are encoded with two bits: R0 = `00`, R1 = `01`, R2 = `10`, R3 = `11`. Two bits are exactly enough to tell four registers apart. The opcodes are given in Table 3.1.2.

Opcode	Mnemonic	Meaning
`0000`	HALT	halt the program
`0001`	ADD	dest ← dest + src
`0010`	SUB	dest ← dest − src
`0011`	AND	dest ← dest AND src (bitwise)

Opcode	Mnemonic	Meaning
`0100`	OR	dest ← dest OR src (bitwise)

Table 3.1.2: Opcodes of the sample processor.

Because the opcode field is 4 bits, at most $2^4 = 16$ different operations can be defined. This design uses only five; the remaining eleven codes are free for instructions added later. The width of a field determines how many things it can select: a field of k bits selects one of 2^k things. Conversely, selecting one of n things takes at least $\lceil \log_2 n \rceil$ bits.

An example instruction: `0001 10 11`. The first four bits 0001 = ADD, the next two bits 10 = R2 (destination), and the last two bits 11 = R3 (source). The instruction is ADD R2, R3 and performs R2 ← R2 + R3. In hexadecimal this byte is 1Bh. The convenience of hexadecimal shows here: the first hex digit is directly the opcode, and the second digit carries the destination and source fields together.

3.1.2.3 Encoding and decoding instructions

Translating an assembly line into machine code is called **encoding**, and recovering an assembly line from a byte is called **decoding**. The assembler does the first; the disassembler and the processor's decoder do the second. The way to decode is to split the bits into their fields. Table 3.1.3 splits a four-instruction program into its fields.

Machine code	Hex	Operation	Destination	Source	Meaning
`0001 0001`	11h	ADD	R0	R1	R0 ← R0 + R1
`0011 1011`	3Bh	AND	R2	R3	R2 ← R2 AND R3
`0100 0110`	46h	OR	R1	R2	R1 ← R1 OR R2
`0000 0000`	00h	HALT	—	—	halt the program

Table 3.1.3: Splitting the sample program into fields.

The format defines the first two-bit register field as the destination. Hence the result of each instruction goes to the register written **first**: R0 in the first instruction, R2 in the second, R1 in the third. The register fields are unused in HALT and are left as zero.

Example 3.1.2

Let us encode `SUB R2, R0` and decode the byte 4Dh.

The opcode of SUB is 0010, R2 = 10, R0 = 00. Written side by side, the fields give `0010 1000`, which is 28h in hexadecimal. If R2 = 12 and R0 = 5, the instruction performs R2 ← 12 − 5 = 7, and R0 is unchanged.

4Dh = `0100 1101`. The opcode 0100 = OR. The remaining four bits are 11 01: destination R3, source R1. The byte is the instruction `OR R3, R1` and performs R3 ← R3 OR R1. If R3 = 1010 and R1 = 0011, the result is 1011 = 11.

3.1.2.4 Code trace: the sample machine program

The program in Table 3.1.3 sits at addresses 0, 1, 2, and 3 in memory. The processor's **program counter** (PC) holds the address of the next instruction. On the 8086 this register is called IP. Each time an instruction is fetched, PC increases by one, because every instruction is a single byte.

```
0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
0000 0000   ; HALT
```

Let the initial values be R0 = 5, R1 = 3, R2 = 12, R3 = 10. To follow the logical operations, the values must also be written in binary. Table 3.1.4 gives the state after each instruction; the register that changed is shown in bold.

Step	Instruction	PC	R0	R1	R2	R3
initial	—	0	0000 0101 (5)	0000 0011 (3)	0000 1100 (12)	0000 1010 (10)
1	ADD R0, R1	1	**0000 1000 (8)**	0000 0011 (3)	0000 1100 (12)	0000 1010 (10)
2	AND R2, R3	2	0000 1000 (8)	0000 0011 (3)	**0000 1000 (8)**	0000 1010 (10)
3	OR R1, R2	3	0000 1000 (8)	**0000 1011 (11)**	0000 1000 (8)	0000 1010 (10)
4	HALT	3	0000 1000 (8)	0000 1011 (11)	0000 1000 (8)	0000 1010 (10)

Table 3.1.4: Trace of the sample machine program.

The first instruction is ADD R0, R1: 5 + 3 = 8, and the result is written to R0. The source R1 is unchanged. The second instruction is AND R2, R3. In a bitwise AND, only the bit that is 1 on both sides stays 1: 1100 AND 1010 = 1000, so R2 = 8. The third instruction is OR R1, R2. In OR, a bit that is 1 on either side becomes 1: 0011 OR 1000 = 1011, so R1 = 11. This instruction uses the value of R2 changed in the previous step. Since the instructions execute in order, each instruction sees the state left by the earlier ones. The fourth instruction is HALT and the processor stops. In this design PC is not incremented after HALT is fetched and stays at 3; in real processors this detail varies with the design. Result: R0 = 8, R1 = 11, R2 = 8, R3 = 10.

This small processor also carries the basic working cycle of the 8086: fetch, advance PC, decode, execute, write the result.

3.1.2.5 Pros and cons of machine language

The pros of machine language come from its closeness to the hardware. No translation is needed; the processor runs the code as it is. The programmer has full control over the hardware: which byte is written where and which instruction comes in what order are all their decisions. Written well, this is also how the smallest and fastest code is obtained.

The cons are heavy. Machine code is unreadable, and it is very hard to write and debug. A single wrong bit turns an instruction into another one; if 0011 0001 is written instead of 0001 0001 (ADD R0, R1), the addition becomes an AND. The code is not portable either. The program has to be rewritten from scratch for every processor.

Perhaps the most annoying problem is computing addresses by hand. Jump instructions carry the address of the target. When an instruction is inserted in the middle of the program, the addresses of all the following instructions shift, but the addresses inside the jump instructions stay as they were. Figure 3.1.6 shows this situation.

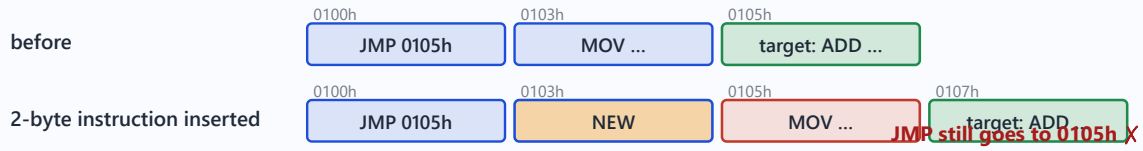


Figure 3.1.6: The consequence of inserting an instruction into machine code. At the top, `JMP 0105h` reaches the right target. At the bottom, a two-byte instruction has been inserted; the target has moved to 0107h, but the jump still goes to 0105h.

The programmer must then find and fix all the jump addresses again. In a large program this takes a long time and is error-prone. To solve these problems, assembly languages (1950s) were developed first, followed by high-level languages such as FORTRAN (1957). In assembly language a jump target is written as a label; the assembler recomputes its address every time.

Common mistake

A common mistake is to mix up the order of the fields in the instruction format. In the sample processor the destination field comes before the source: `0010 11 00` is SUB R3, R0 and performs R3 ← R3 − R0. If the fields are read in reverse, SUB R0, R3 results; because subtraction is not commutative, the result is written to a different register with a different value.

3.1.3 Assembly languages and the assembler

3.1.3.1 Mnemonics

The first gain of assembly language is names. The programmer writes `MOV AL, 5` instead of `10110000 00000101`. `MOV` is a mnemonic: as an abbreviation of the word “move,” it reminds you what the instruction does. `AL` is the name of a register, and `5` is a constant written in decimal. The program that translates this line into machine code is called an **assembler**. Figure 3.1.7 shows what the translation does for a single line.

Figure 3.1.7: The assembler translates the line `MOV AL, 5` into the machine code `B0 05`, that is, the bits `10110000 00000101`.

Assembly language is designed separately for each processor architecture and instruction set. Intel x86, Motorola 68000, ARM, PIC, and AVR processors each have their own assembly language. The register names and instructions valid in one are absent in another. Within a processor family, however, compatibility is preserved. An assembly program written for the 8086 can, with small differences, also be assembled and run on the 80286, 80386, and today's x86 processors. Assembly languages specific to microcontrollers are still widely used in embedded systems and electronics applications.

Assembly language remains important today, when high-level languages are widespread, for three reasons. First, some code can be written only in assembly. The bootloader that runs first when a computer powers on, interrupt handlers, the hardware-touching parts of device drivers, and the context-switching parts of an operating system kernel are of this kind. Second, the way to understand how a processor really works runs through assembly. Why a program crashed, what the code shown by a debugger does, or examining software by reverse engineering all require reading assembly. Third, performance. Special instructions that the compiler cannot generate by itself, such as SIMD or cryptographic instructions, and time-critical loops are still written by hand.

3.1.3.2 From source code to a running program

The text file the programmer writes is called **source code**. Three tools take part on the way from source code to a running program. Figure 3.1.8 shows this toolchain.

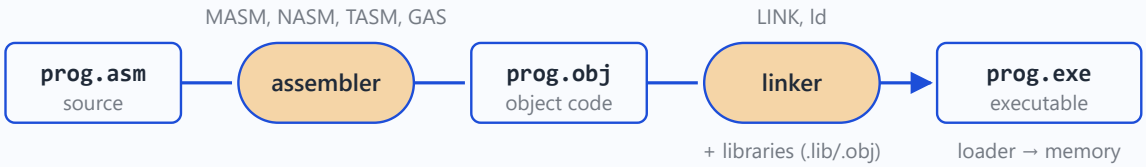


Figure 3.1.8: The source file becomes an object file with the assembler and an executable file with the linker; the loader places this file in memory.

Assembler

: Reads the source file (`prog.asm`) and translates each line into machine code. Its output is an **object file** (`prog.obj` or `prog.o`). The object file contains machine code but is not yet complete. The addresses of functions defined in other files are left blank, and these blanks are recorded in a list. MASM, NASM, TASM, and GNU as are tools that do this job.

Linker

: The **linker** combines one or more object files and libraries (`.lib`, `.obj`). It fills in the blank addresses and produces an **executable file** (`prog.exe`) in a form the operating system understands. Microsoft's LINK and GNU's ld are common linkers.

Loader

: The **loader** is a part of the operating system. It copies the executable file from disk into memory, sets up the segment registers and the stack, adjusts addresses to where the program was loaded if necessary, and hands control to the program's first instruction.

The assembler has one more output: the **listing**. The listing writes, next to each source line, that line's address and the machine code it produced. The listing of a small DOS program that prints “Welcome!” on the screen is as follows:

```
Address Machine code  Source
0100    B4 09         mov ah, 09h
0102    BA 0C 01      mov dx, offset msg
0105    CD 21         int 21h
0107    B8 00 4C      mov ax, 4C00h
010A    CD 21         int 21h
010C    57 65 6C ...  msg db 'Welcome!$'
```

The listing can be read line by line. `mov ah, 09h` is two bytes: `B4` means “load a constant into AH,” and `09` is the constant. `mov dx, offset msg` is three bytes: `BA` means “load a 16-bit constant into DX,” and the constant is 010Ch, the address of the label `msg`, written little-endian as `0C 01`. `int 21h` is two bytes: `CD` is the interrupt instruction and `21` is the interrupt number. `mov ax, 4C00h` loads 4C00h into AX, that is, AH = 4Ch and AL = 00h. The `db` (*define byte*) directive on the last line produces data, not an instruction: the ASCII codes of the characters “Welcome!$” sit one after another in memory. The listing is used both to check what the assembler produced and to match addresses with source lines in a debugger. emu8086 performs all of these steps with a single key and produces the COM or EXE file directly.

3.1.3.3 The assembler's two passes

There is an interesting situation in the listing above. The line `mov dx, offset msg` is the second line, but the label `msg` is defined at the very end. When the assembler reaches this line, it does not yet know the address of `msg`. Referring to a label before its definition is called a **forward reference**. Assemblers solve this problem by reading the source code twice.

> **Definition 3.1.3**
>
> A **two-pass assembler** reads the source code from start to end twice. In the first pass it computes how many bytes each line takes and advances the **location counter** (LC). It also writes each label into the **symbol table** with the current LC value. In the second pass it produces the machine code using the addresses in the symbol table.

The location counter holds the address in memory where the next byte will be placed. In the source it can be read with the `$` sign. There is an important condition for the first pass: the length of every instruction must be computable without knowing the value of the operand. That `mov dx, offset msg` is 3 bytes is known whatever the value of `msg` turns out to be. The program being traced is this:

```
org 100h
mov ah, 09h
mov dx, offset msg
int 21h
mov ax, 4C00h
int 21h
msg db 'Welcome!$'
```

Pass	Line	Length	LC (after the line)	Symbol table	Bytes produced
1	`org 100h`	0	0100h	msg = ?	—
1	`mov ah, 09h`	2	0102h	msg = ?	—
1	`mov dx, offset msg`	3	0105h	msg = ?	—
1	`int 21h`, `mov ax, 4C00h`, `int 21h`	2 + 3 + 2	010Ch	msg = ?	—
1	`msg db 'Welcome!$'`	9	0115h	**msg = 010Ch**	—
2	`mov ah, 09h`	2	0102h	msg = 010Ch	`B4 09`
2	`mov dx, offset msg`	3	0105h	msg = 010Ch	`BA 0C 01`
2	three lines	7	010Ch	msg = 010Ch	`CD 21 B8 00 4C CD 21`

Pass	Line	Length	LC (after the line)	Symbol table	Bytes produced
2	`msg db …`	9	0115h	msg = 010Ch	`4D 65 72 68 61 62 61 21 24`

Table 3.1.5: Trace of a two-pass assembler.

The first pass only counts. `org 100h` sets LC to 0100h; being a directive, it produces no bytes. `mov ah, 09h` takes 2 bytes and LC becomes 0102h. On the third line `msg` is not yet defined, but the instruction is known to be 3 bytes and LC advances to 0105h. The next three instructions are 2, 3, and 2 bytes; LC reaches 010Ch. On the last line the label `msg` is seen and enters the symbol table with the current LC: `msg` = 010Ch. The string is 9 characters ("Welcome!" is eight characters plus the trailing `$`), and LC becomes 0115h.

The second pass restarts LC from 0100h and this time produces the bytes. For `mov ah, 09h`, `B4 09` is written. `offset msg` is now read from the symbol table: 010Ch, little-endian `0C 01`. The remaining instructions are `CD 21`, `B8 00 4C`, and `CD 21`. Finally the ASCII codes of the string are written: W = 57h, e = 65h, l = 6Ch, c = 63h, o = 6Fh, m = 6Dh, e = 65h, ! = 21h, $ = 24h. The output is a COM file of 0115h − 0100h = 15h = 21 bytes. When run, it prints "Welcome!" on the screen. Figure 3.1.9 shows the state at the end of the two passes.

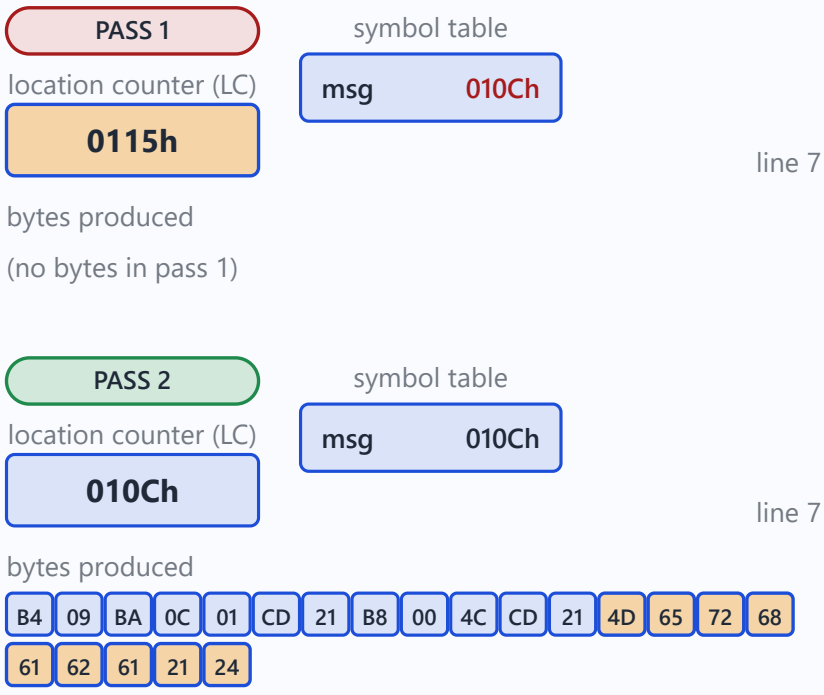


Figure 3.1.9: At the end of the first pass the symbol table is filled in, but no bytes have been produced (top). At the end of the second pass all 21 bytes have been produced (bottom).

The two-pass structure also solves the address-shifting problem of machine language by itself. If the programmer inserts an instruction, the assembler recomputes the addresses of all labels at the next assembly and writes the new addresses into the jump instructions.

> **Common mistake**
>
> A common mistake is to take directives for instructions. `org`, `db`, `equ`, and `offset` are not processor instructions; they are information given to the assembler. `org 100h` produces no bytes and only sets the location counter. `db` produces data, not an instruction. If the processor reaches the bytes of a `db` line as if they were instructions, it tries to decode them as instructions too; this is why data is placed after the code.

3.1.3.4 Assemblers and their syntax

There is more than one assembler for the same x86 processor, and they do not all follow the same rules. The rules that govern how an assembly line is written are called **syntax**. Two main syntaxes are used for x86: Intel and AT&T. Table 3.1.6 summarizes the common assemblers.

Assembler	Syntax	Use
MASM	Intel	Microsoft, Windows
TASM	Intel	Borland, DOS
NASM	Intel	open source, multi-platform
GAS (as)	AT&T	GNU/Linux; the assembler used by gcc
emu8086	Intel (MASM-like)	an 8086 emulator; the tool of this course

Table 3.1.6: Common x86 assemblers.

The most important difference between the two syntaxes is the order of the operands. In Intel syntax the destination is written first and then the source. In AT&T syntax the order is reversed: source first, then destination. The same two instructions look like this in the two syntaxes:

```
; Intel:  destination, source
mov eax, 5
add eax, [ebx+4]

# AT&T:  source, destination
movl $5, %eax
addl 4(%ebx), %eax
```

In AT&T syntax, register names are prefixed with `%` and constants with `$`. The operand size is indicated by the letter at the end of the instruction: `b` means 8, `w` 16, and `l` 32 bits. A memory operand is written as `offset(base)` instead of with square brackets: `4(%ebx)` is the same address

as `[ebx+4]` in Intel syntax. The comment character differs too: assemblers with Intel syntax use `;`, while GNU as uses `#` on x86. Both notations give the processor the same bytes. The difference is only in the text a human reads.

> **Common mistake**
>
> A common mistake is to read AT&T code produced by a compiler with Intel conventions. The instruction `subl %ebx, %eax` does not subtract EAX from EBX; it performs EAX ← EAX − EBX, that is, Intel's `sub eax, ebx`. By default gcc and clang produce AT&T syntax; the `-masm=intel` option switches the output to Intel syntax.

3.1.4 “Welcome” in different environments

A program that prints a short text on the screen is the smallest complete example of a language or an environment. In this section the same job is done in six environments: 32-bit Linux, DOS, Windows, the Motorola 68000-based Amiga, ARM, and PowerPC. All the programs follow the same three steps: define the text in memory, ask the operating system for the service “write this many bytes at this address to the screen,” and terminate the program. The differences lie in how each environment expresses these three steps.

3.1.4.1 Linux, x86 (32-bit)

The first program runs on a 32-bit x86 processor under the Linux operating system and is assembled with NASM. The 16-bit registers AX, BX, CX, and DX of the 8086 were widened to 32 bits starting with the 80386; their 32-bit names are EAX, EBX, ECX, and EDX. AX is the lower half of EAX.

```
section .data
hello  db 'Welcome, Linux!', 10
len    equ $ - hello     ; = 16 bytes
section .text
global _start
_start:
  mov eax, 4     ; sys_write
  mov ebx, 1     ; stdout
  mov ecx, hello ; string address
  mov edx, len   ; byte count
  int 0x80       ; call the kernel
  mov eax, 1     ; sys_exit
  xor ebx, ebx   ; exit code 0
  int 0x80
```

The program consists of two sections. `section .data` opens the section where data lives, and `section .text` the section where code lives. The label `hello` marks the start address of the text; `db` places the bytes of the text and, at the end, 10, the newline character. The line `len equ $ - hello` occupies no memory. `equ` gives a name a constant value; `$` is the current location counter. Subtracting the start address of the text from the location counter gives the length of the text. “Welcome, Linux!” is 15 bytes; with the newline it is 16 bytes. Having the assembler compute the length instead of counting by hand prevents an error when the text changes.

`global _start` makes the label `_start` visible to the linker. On Linux the linker looks for the program's entry point under this name. The code itself is two system calls. Table 3.1.7 gives the registers after each line.

Line	EAX	EBX	ECX	EDX	Screen
`mov eax, 4`	00000004	????????	????????	????????	
`mov ebx, 1`	00000004	00000001	????????	????????	
`mov ecx, hello`	00000004	00000001	hello (address)	????????	
`mov edx, len`	00000004	00000001	hello (address)	00000010	
`int 0x80`	00000010	00000001	hello (address)	00000010	Welcome, Linux!
`mov eax, 1`	00000001	00000001	hello (address)	00000010	Welcome, Linux!
`xor ebx, ebx`	00000001	00000000	hello (address)	00000010	Welcome, Linux!
`int 0x80`	process ends with exit code 0; the call does not return				

Table 3.1.7: Trace of the Linux “Welcome” program.

The parameters of the first call are placed in registers. EAX = 4 is the number of the `sys_write` service in Linux's 32-bit call table. EBX = 1 is the file descriptor to write to; descriptor 1 is standard output, that is, the screen. ECX receives the **address** of the text, not its contents. EDX is the number of bytes to write: 16, which is 10h in hexadecimal. The `int 0x80` instruction hands control to the kernel. The kernel writes the text to the screen and on return places the number of bytes written in EAX; that is why EAX becomes 00000010h.

The second call terminates the program. EAX = 1 is `sys_exit`. EBX carries the exit code, and 0 means success. `xor ebx, ebx` combines a number with itself using exclusive OR. Since every bit equals itself, the result is zero. This way produces shorter machine code than `mov ebx, 0` (2 bytes against 5). XOR also affects the flags: because the result is zero, ZF = 1, and because it is a logical operation, CF = 0 and OF = 0. The last `int 0x80` terminates the process and does not return.

Common mistake

A common mistake is to write the length by hand and then change the text. Had `len` been written as 15, the newline would not be printed and the next output would be appended to the same line. Had the text been shortened and the length left at 16, the memory bytes after

the text would also be printed to the screen. `$ - hello` prevents both errors. Non-ASCII characters need extra care: in UTF-8, letters such as “ü” take 2 bytes; “Dünya” is 5 characters but 6 bytes.

3.1.4.2 The system call

The Linux program did not write to the screen itself; it asked the kernel to do so. In modern operating systems, application programs cannot access the hardware directly. The processor's protection mechanism defines two operating modes. A program running in **user mode** cannot use input/output instructions and some control instructions. **Kernel mode** is privileged and open only to the operating system. When an application needs a hardware service, it crosses into the kernel through a controlled gate.

Definition 3.1.4

A **system call** is the special call by which a program requests a service from the operating system. The program puts the service number and the parameters in specific registers, then hands control to the kernel with a software interrupt or a special instruction. The kernel performs the service, places the result in a register, and returns control to the program.

Figure 3.1.10: The path of a system call. The user program places the parameters in registers and crosses into the kernel with `int 0x80`. The kernel selects the service by the number in EAX, writes to the screen through the driver, and returns the result in EAX.

Figure 3.1.10 shows this path. `int 0x80` is a software interrupt. The processor saves the flags and the return address, jumps to the address stored in entry number 80h of the interrupt table, and switches to kernel mode at the same moment. The entry routine in the kernel looks at EAX and calls the corresponding function from the service table. When the service finishes, a special return instruction both returns to the saved address and switches back to user mode.

The 8086 has no protection mechanism and no user/kernel mode distinction; every program can access everything. The interrupt mechanism is nevertheless the same. When executing `INT n`, the 8086 pushes the flags register, CS, and IP onto the stack and clears the IF and TF flags. It reads the

new CS:IP from the interrupt vector table at the very beginning of memory. Each entry is 4 bytes; for `INT 21h` the entry is at address 21h × 4 = 84h. DOS writes the address of its own service routine into this entry. Table 3.1.8 compares the call paths in different environments.

Environment	Call	Service number	Parameters
Linux 32-bit	`int 0x80`	EAX (4 = write, 1 = exit)	EBX, ECX, EDX
Linux 64-bit	`syscall`	RAX (1 = write, 60 = exit)	RDI, RSI, RDX
MS-DOS	`int 21h`	AH (09h = write, 4Ch = exit)	DX, AL …
BIOS	`int 10h`	AH (0Eh = write character)	AL, BH …

Table 3.1.8: System calls by environment.

The table clearly shows that the numbers are specific to the environment. The “write” service is 4 on 32-bit Linux and 1 on 64-bit Linux. Even two versions of the same operating system use different numbers and different registers. BIOS services sit even below the operating system. These routines, held in the computer's permanent memory, work even before the operating system is loaded.

3.1.4.3 DOS, 8086 (COM program)

The third program is in the environment of this course: a COM program running under DOS, assembled with emu8086.

```
org 100h          ; COM program
mov ah, 09h       ; DOS 09h: print string
lea dx, msg       ; DS:DX = address
int 21h           ; call DOS
mov ax, 4C00h     ; DOS 4Ch: exit
int 21h
msg db 'Welcome, DOS!$'
```

`org 100h` declares that the COM program will be loaded at offset 100h of the segment. When loading the program, DOS places a structure called the **program segment prefix** (PSP) in the first 256 bytes of the segment (0000h–00FFh). The program starts right after this structure. In a COM program all four segment registers point to the same segment: CS = DS = ES = SS. Code, data, and stack are within the same 64 KB.

`mov ah, 09h` selects DOS service number 09h: “print the string starting at DS:DX and ending with `$`” to the screen.” `lea dx, msg` loads the relative position of the label `msg` into DX (*load effective address*). Since DS already points to the program's segment, DS:DX is the full address of the string. `int 21h` calls DOS. DOS prints characters to the screen until it sees `$`; the `$` itself is not printed. `mov ax, 4C00h` loads 4Ch into AH, that is, the “terminate the program” service, and the exit code 00h into AL. The last `int 21h` hands control back to DOS.

Line	Machine code	IP (after)	AH	AL	DX	Screen
`org 100h`	—	0100h	??	??	????	
`mov ah, 09h`	`B4 09`	0102h	09h	??	????	
`lea dx, msg`	`BA 0C 01`	0105h	09h	??	010Ch	
`int 21h`	`CD 21`	0107h	09h	??	010Ch	Welcome, DOS!
`mov ax, 4C00h`	`B8 00 4C`	010Ah	4Ch	00h	010Ch	Welcome, DOS!
`int 21h`	`CD 21`	010Ch	4Ch	00h	010Ch	program ends

Table 3.1.9: Trace of the DOS “Welcome” program.

The addresses in Table 3.1.9 come from the instruction lengths. `lea dx, msg` is three bytes: emu8086 encodes a direct-address LEA in the same form as `mov dx, offset msg`, that is, the opcode `BA` and the address `0C 01` (MASM encodes the same instruction as a real four-byte LEA beginning with `8D 16`). The code takes 2 + 3 + 2 + 3 + 2 = 12 bytes in the range 0100h–010Bh; that is why `msg` is at 010Ch. “Welcome, DOS!” is 13 characters and, together with `$`, occupies 14 bytes.

It is noteworthy that IP shows 010Ch on the second `int 21h` line: this address is the start of `msg`. Because the program terminates, the processor never reaches this address. Because the data is placed **after the code**, the processor does not mistake the data for instructions and execute it. Had the data been placed before the code, the program starting at 0100h would have decoded 57h, the code of the letter “W”, as an instruction. On the 8086, 57h is the `PUSH DI` instruction. The letters that follow would also turn into random instructions, and the program would behave unpredictably.

> **Common mistake**
>
> A common mistake is to forget the `$` at the end of the string in service 09h. DOS has no other way to know where the string ends. Without `$`, it prints whatever lies after the code and data in memory until it hits the first `24h` byte. You will see meaningless characters, beeps, and cursor jumps on the screen.

3.1.4.4 Windows, Win32 API

Windows programs do not make system calls directly. The operating system's services are offered as functions in **DLL** (dynamic-link library) files. The program calls these functions with the `call` instruction like an ordinary subroutine. The following program opens a message box on the screen and then terminates the process.

```
extern MessageBoxA    ; user32.dll
extern ExitProcess    ; kernel32.dll

section .data
msg db 'Welcome, Windows!', 0

section .text
global _start
_start:
    push 0          ; uType = MB_OK
    push msg        ; lpCaption (title)
    push msg        ; lpText (text)
    push 0          ; hWnd = NULL
    call MessageBoxA
    push 0          ; uExitCode = 0
    call ExitProcess
```

The `extern` lines say that the names `MessageBoxA` and `ExitProcess` are defined elsewhere. The assembler does not know the addresses of these names; it leaves a note in the object file: "let the linker fill this in." Using the import information of the libraries `user32.dll` and `kernel32.dll`, the linker directs the calls to these functions. This time the string ends not with `$` but with 0. The Windows API was designed with the C language in mind, and C strings end with a zero byte.

Parameters are passed not in registers but on the stack. In C the call has the form `MessageBoxA(hWnd, lpText, lpCaption, uType)`. In assembly the parameters are pushed **from right to left**: `uType` first, `hWnd` last. That way the first parameter lands at the top of the stack, just above the return address, and the function always finds it in the same place regardless of the number of parameters. This program uses the same string for the title and the text; the two `push msg` instructions push the same address. Table 3.1.10 traces the state of the stack; 0012FF80h is taken as an example initial value of ESP.

Instruction	ESP (after)	Address written ← value	EAX
initial	0012FF80h	—	????????
`push 0` (uType)	0012FF7Ch	0012FF7Ch ← 0 (MB_OK)	????????
`push msg` (lpCaption)	0012FF78h	0012FF78h ← &msg	????????
0012FF78h ← &msg	0012FF74h	0012FF74h ← &msg	????????
`push 0` (hWnd)	0012FF70h	0012FF74h ← &msg	????????
`call MessageBoxA`	0012FF6Ch	0012FF6Ch ← return address	????????
function returns with `ret 16`	0012FF80h	—	00000001h (IDOK)
`push 0` (uExitCode)	0012FF7Ch	0012FF7Ch ← 0	00000001h
`call ExitProcess`	process ends with exit code 0; the call does not return		

Table 3.1.10: Trace of the stack in the Windows program.

The stack grows toward lower addresses. In 32-bit mode every `push` decreases ESP by 4 and writes the value at the address ESP now points to. Four parameters take 16 bytes. `call` also pushes the return address; at the moment of entering the function, ESP = 0012FF80h − 20 = 0012FF6Ch. Figure 3.1.11 shows this moment.

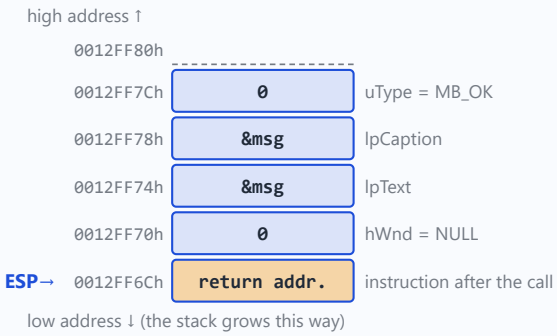


Figure 3.1.11: The stack immediately after `call MessageBoxA` has executed. At the top is the return address, and above it the first parameter `hWnd`.

When the user presses the OK button in the message box, the function returns. The return value is placed in EAX: 1, that is, IDOK, means "OK was pressed." Windows API functions use the **stdcall** calling convention. In this convention the **called** function cleans the parameters off the stack. `MessageBoxA` returns with the `ret 16` instruction: it first takes the return address (ESP + 4), then discards the 16 bytes of parameters (ESP + 16). ESP becomes 0012FF80h again. In the cdecl convention, used by C's own functions, the caller does this cleanup instead.

At link time one more detail may be encountered. In some 32-bit toolchains, stdcall functions are looked up under **decorated** names: `_MessageBoxA@16`. The leading underscore comes from the C naming convention, and the trailing 16 from the number of parameter bytes. At the 8086 level the same stack mechanism can be traced in 16-bit mode, with the only difference that each push takes 2 bytes.

3.1.4.5 Motorola 68000, Amiga

Amiga computers use the Motorola 68000 processor. The 68000 is a processor of the same period (1979) as the 8086, but its architecture is different. It has eight data registers (`d0` – `d7`) and eight address registers (`a0` – `a7`); all are 32 bits. Memory is not divided into segments; addresses are linear.

```
    move.l  4.w,a6          ; a6 = ExecBase
    lea     dosname(pc),a1 ; library name
    moveq   #0,d0           ; any version
    jsr     -552(a6)        ; OpenLibrary()
    tst.l   d0
    beq.s   quit            ; failed → quit
    move.l  d0,a6           ; a6 = DOSBase
    jsr     -60(a6)         ; Output() → d0
    move.l  d0,d1           ; file handle
    move.l  #msg,d2         ; string address
    moveq   #16,d3          ; byte count
    jsr     -48(a6)         ; Write(d1,d2,d3)
    move.l  a6,a1
    move.l  4.w,a6
    jsr     -414(a6)        ; CloseLibrary()
quit moveq  #0,d0           ; return code 0
    rts
dosname dc.b 'dos.library',0
msg     dc.b 'Welcome, Amiga!',10
```

In AmigaOS, system services are offered as libraries, and each library has a **base address**. Functions are called through a jump table that sits at negative offsets from this base. At memory address 4 is the base address of `exec.library`, the core library of the whole system. The first line loads this address into `a6`. `lea dosname(pc),a1` puts the address of the library name into `a1`. `moveq #0,d0` means "any version will do." `jsr -552(a6)` jumps to the entry 552 bytes below `a6`; this entry is `OpenLibrary`.

On return, `d0` holds the base of the opened `dos.library`. `tst.l d0` and `beq.s quit` leave the program if the library could not be opened (`d0` = 0). If it opened, the base is moved into `a6`. `jsr -60(a6)` is the `Output()` function and returns the handle of standard output in `d0`. The parameters of the `Write(d1, d2, d3)` call are passed in registers: the handle goes into `d1`, the address of the text into `d2`, and the length, 16, into `d3`. "Welcome, Amiga!" is 15 bytes; with the newline it is 16. `jsr -48(a6)` does the writing. Finally the library is closed with `CloseLibrary` (offset `-414`) and `rts` is executed with return code `d0` = 0.

In 68000 syntax the operand order is like AT&T: source first, then destination. The size is indicated by the suffix `.b`, `.w`, or `.l` at the end of the instruction. Constants are written with `#`. The library is first opened, used, and then closed. The operating system's resources are managed in this pattern.

3.1.4.6 ARM

ARM processors are used today in smartphones, embedded systems, and many laptops. ARM is a RISC (reduced instruction set computer) architecture: its instructions are few, simple, and of fixed length.

```
    .data
msg:    .ascii "Welcome, ARM!\n"
len = . - msg          @ 14 bytes

    .text
    .global _start
_start:
    mov  r7, #4        @ sys_write
    mov  r0, #1        @ stdout
    ldr  r1, =msg      @ string address
    mov  r2, #len      @ byte count
    svc  #0            @ call the kernel
    mov  r7, #1        @ sys_exit
    mov  r0, #0        @ exit code
    svc  #0
```

The program is assembled with GNU as. In GNU as, data is defined not with `db` but with the `.ascii` directive; `\n` is the newline. The dot in the line `len = . - msg` is the location counter, the counterpart of `$` in NASM. “Welcome, ARM!” is 13 bytes; with the newline it is 14. The comment character for ARM in GNU as is `@`.

On 32-bit ARM Linux the service number goes into `r7` and the parameters into `r0`, `r1`, and `r2`. The write service is again 4 and exit is 1; although the numbers are the same as on 32-bit x86 Linux, the registers differ. `ldr r1, =msg` is a special notation. Since ARM instructions are 32 bits, a 32-bit address does not fit inside a single instruction. The assembler stores the address in a literal pool near the code and generates the `ldr` instruction so that it reads it from there. `svc #0` (formerly named `swi`) hands control to the kernel. On ARM only the `ldr` (load) and `str` (store) instructions work with memory. Operations such as addition are performed only on registers. ARM has no instruction that adds directly from memory like `add ax, [0202h]` on the 8086.

3.1.4.7 PowerPC

PowerPC is a RISC architecture developed jointly by IBM, Motorola, and Apple. It has been used in older Macintosh computers, game consoles, and automotive controllers.

```
    .data
msg:    .ascii "Welcome, PowerPC!\n"
len = . - msg           # 18 bytes

    .text
    .globl _start
_start:
    li   0, 4           # r0 = sys_write
    li   3, 1           # r3 = stdout
    lis  4, msg@ha      # r4 = addr (high)
    addi 4, 4, msg@l    #    + low half
    li   5, len         # r5 = byte count
    sc                  # call the kernel
    li   0, 1           # r0 = sys_exit
    li   3, 0           # r3 = exit code
    sc
```

PowerPC has 32 general-purpose registers. In GNU as these registers are written as plain numbers: the `3` in the instruction `li 3, 1` is register r3. `li` (*load immediate*) loads a constant into a register. The service number goes into `r0` and the parameters into `r3`, `r4`, and `r5`; the switch into the kernel is made with the `sc` instruction. The comment character is `#`. “Welcome, PowerPC!” is 17 bytes; with the newline it is 18.

Loading an address is again a two-step affair. PowerPC instructions are also 32 bits and the constant field is 16 bits. A 32-bit address is split into two 16-bit parts. `lis 4, msg@ha` loads the upper half into the upper 16 bits of the register; `addi 4, 4, msg@l` adds the lower half. `@ha` means “high adjusted.” Because `addi` adds the lower half as a signed number, if the top bit of the lower half is 1, the upper half is increased by 1 beforehand. This detail is left to the assembler.

3.1.4.8 Comparing the six environments

Table 3.1.11 puts the six programs side by side.

Environment	Processor	Switch to the OS	Service no.	Parameters	String end
Linux	x86 (32)	`int 0x80`	EAX	EBX, ECX, EDX	length given
DOS	8086	`int 21h`	AH	DX (address)	`'$'`
Windows	x86 (32)	DLL via `call`	function name	stack	`0`
AmigaOS	68000	`jsr -48(a6)`	library offset	d1, d2, d3	length given
Linux	ARM	`svc #0`	r7	r0, r1, r2	length given
Linux	PowerPC	`sc`	r0	r3, r4, r5	length given

Table 3.1.11: The same “Welcome,” six environments.

The conclusion from the table is this: assembly code depends on two things at once. The first is the **processor**: the instruction set, register names, the way addresses are loaded, and the syntax all change with the processor. Even the comment character changes: `;` on x86, `@` in GNU as for ARM, and `#` on PowerPC. The second is the **operating system**: on the same x86 processor, Linux, DOS, and Windows each demand entirely different call paths, numbers, and string conventions. Moving a program to a new environment means rewriting it at the assembly level. The greatest gain of high-level languages is that they hand this dependence over to the compiler and the interpreter. The line `printf("Welcome\n")` in C is the same in all six environments; each environment's C library and compiler handles the difference.

3.1.5 High-level languages

3.1.5.1 Compiled, interpreted, hybrid

A program written in a high-level language must also end up running in the processor's machine language. Languages differ in who does this translation and when.

Definition 3.1.5

A **compiler** is a program that examines high-level source code as a whole and translates it into machine code before the program runs. An **interpreter** is a program that reads and executes the source code statement by statement at run time; it produces no separate executable file. **Just-in-time (JIT) compilation** translates parts of a program into machine code during execution, as they are needed.

In **compiled languages** the source code is translated into machine language in advance. Because the compiler sees the whole source, most type and syntax errors are caught at compile time. Since there is no translation overhead at run time, the program runs fast. Compilation has two stages: the compiler translates each source file into an object file, and the linker combines the object files with libraries. The output is specific to a particular processor and a particular operating system. An x86 program compiled for Windows does not run on Linux or on an ARM processor.

In **interpreted languages** the source code itself is distributed and processed by the interpreter at run time. Since there is no compilation step, a change can be tried immediately; working line by line in an interactive shell is possible. The source code is platform-independent: the same file runs on any computer that has the interpreter. The price is twofold. Some errors surface only when the relevant line runs. In addition, analyzing every statement as it runs takes time, so the program is usually slower.

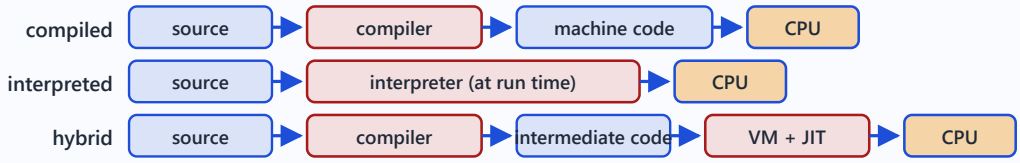


Figure 3.1.12: Three execution models. In a compiled language the source is translated into machine code in advance. In an interpreted language the interpreter processes the source while running. In the hybrid model the source is compiled to intermediate code; the virtual machine translates this intermediate code to machine code with JIT while running.

Between the two models there is also a **hybrid** model (Figure 3.1.12). Java and C# programs are first compiled to a processor-independent **intermediate code**. In Java this code is called bytecode, in C# it is called IL (intermediate language). A **virtual machine** runs the intermediate code: the JVM for Java, the CLR for C#. The virtual machine first interprets the intermediate code; it translates frequently executed parts to real machine code with JIT and caches them. This yields both portability and high speed. Today the Python and JavaScript engines also use bytecode and JIT internally; the boundaries are not sharp.

3.1.5.2 The compilation process

A program written in C becomes an executable file in five stages. Take as an example the program `oddeven.c`, which prints whether a number is odd or even. The gcc compiler runs all the stages with a single command, but it also offers an option to stop at each stage.

1. **Preprocessor**: replaces the `#include` lines with the contents of the corresponding header files and expands the names defined with `#define`. The output is still C code: `oddeven.i` (`gcc -E`). The 15-line program grows to thousands of lines once `stdio.h` is included; with the clang-based compiler used while preparing these notes it came to 1,543 lines.
2. **Compiler**: passes the C code through syntax and type checking, then translates it into **assembly** language: `oddeven.s` (`gcc -S`). Most errors are reported at this stage.
3. **Assembler**: translates the assembly code into machine code and produces an object file: `oddeven.o` (`gcc -c`). In this file the addresses of the `printf` and `scanf` calls are still blank.
4. **Linker**: combines the object file with code such as `printf` and `scanf` from the C library and produces the executable file: `oddeven` on Linux, `oddeven.exe` on Windows.
5. **Loader**: when the program is run, the operating system loads the file into memory, prepares the stack and the startup environment, and finally jumps to the `main` function.

The whole job is done with one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly code the compiler produces, in Intel syntax, use `gcc -S -masm=intel oddeven.c`. Reading this output is the most direct way to see which instructions a high-level line turns into. For example, when the loop of the odd/even program is compiled for x86-64 with optimization (`-O2`), the

comparison `i % 2 == 0` becomes not a division instruction but `test bl, 1`. The compiler tests the lowest bit instead of dividing, because the division instruction is much slower. The same trick can be written by hand in 8086 assembly: `test cl, 1`.

> **Common mistake**
>
> A common mistake is to confuse a compile error with a link error. Misspelling the name `printf` may cause only a warning in the compiler. The real error shows up in the linker as an "undefined reference," because no library contains a function with that name. Looking at which tool an error message came from shows at which stage the problem lies.

3.1.5.3 Code trace: a loop generated by the compiler

To see how a compiler translates a loop, take a small piece of C:

```
sum = 0;
i = 10;
while (i > 0) {
    sum += i;
    i -= 3;
}
```

One equivalent of this fragment that could be generated for the 8086 is as follows:

```
org 100h
mov bx, 0          ; sum = 0
mov cx, 10         ; i = 10
cond: cmp cx, 0    ; i > 0 ?
jle done
add bx, cx         ; sum += i
sub cx, 3          ; i -= 3
jmp cond
done: mov sum, bx
ret
sum dw ?
```

The compiler has placed the variables in registers: `sum` lives in BX and `i` in CX. At the end of the loop the sum is written to the memory variable `sum`. The `while` condition is tested at the top of the loop. There is an important transformation here. The condition in C says "if `i > 0`, enter the body." In assembly the condition is **inverted**: "if `i ≤ 0`, leave the loop." The instruction `cmp cx, 0` performs the subtraction CX − 0 but does not store the result; it only sets the flags. `jle done` (*jump if less or equal*) jumps if the signed comparison says less or equal. This condition holds when ZF = 1 or SF ≠ OF. After the body, `jmp cond` returns unconditionally to the top.

Step	Lines	BX (sum)	CX (i)	ZF	SF	OF	Decision
1	`mov bx, 0` , `mov cx, 10`	0	10	–	–	–	MOV does not change the flags
2	`cmp cx, 0` , `jle`	0	10	0	0	0	10 > 0, not taken
3	body, `jmp`	10	7	0	0	0	back to the condition
4	`cmp` , `jle`	10	7	0	0	0	not taken
5	body, `jmp`	17	4	0	0	0	back to the condition
6	`cmp` , `jle`	17	4	0	0	0	not taken
7	body, `jmp`	21	1	0	0	0	back to the condition
8	`cmp` , `jle`	21	1	0	0	0	not taken
9	body, `jmp`	22	−2 (FFFEh)	0	1	0	back to the condition
10	`cmp` , `jle`	22	−2 (FFFEh)	0	1	0	SF ≠ OF, **taken**
11	`mov sum, bx`	22	−2 (FFFEh)	0	1	0	sum = 22

Table 3.1.12: Trace of the 8086 equivalent of the `while` loop.

The flags in Table 3.1.12 come from the last instruction of the body, `sub cx, 3`; `jmp` does not change the flags. The loop runs four iterations, and `i` takes the values 10, 7, 4, 1 in turn. The sum is 10 + 7 + 4 + 1 = 22. At the end of the fourth iteration `sub cx, 3` produces 1 − 3 = −2. In 16-bit two's-complement representation −2 is FFFEh; since its top bit is 1, SF = 1. This subtraction is also correct as a signed operation, that is, there is no overflow and OF = 0. In step 10, `cmp cx, 0` reproduces the same flags. SF = 1 and OF = 0 means SF ≠ OF; as a signed number CX < 0, and `jle` is taken. Figure 3.1.13 shows this moment next to the C source.

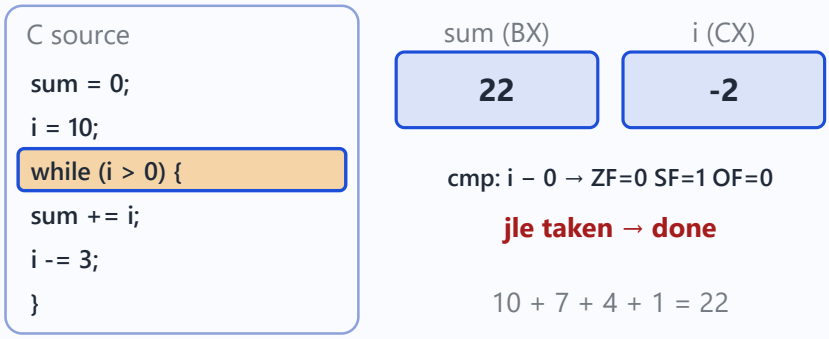


Figure 3.1.13: When i = −2, `cmp cx, 0` produces SF = 1, OF = 0 and `jle done` is taken. The loop has been left; the sum is 22.

In this example it matters that the condition is tested with a signed jump. `i` is an `int` and can be negative. Had the compiler chosen an unsigned jump, for example `jbe`, it would have read FFFEh as 65534 and found it greater than 0; the loop would have continued at the wrong place. The variable type in C decides which conditional jump is chosen in assembly. A single C loop has turned into three

kinds of instructions: comparison, conditional jump, and unconditional jump. When the same C fragment is compiled and run on a computer, `sum` also comes out as 22, and the intermediate values (10, 7), (17, 4), (21, 1), (22, −2) are the same.

Common mistake

A common mistake is to forget to invert the loop condition when translating it to assembly. If the C `while (i > 0)` is written directly as `jg body`, then in the case where the body is not entered, the flow falls through to the instruction just below the body. The correct pattern is an inverted jump that leaps to the end of the loop when the condition is false, plus a `jmp` at the end of the body that returns to the top.

3.1.6 Compiled languages

Compiled languages work with a compiler that examines the source code as a whole before running. Compiled code runs directly on the processor, so it is fast, and type and syntax errors are caught at compile time. Each language was born from a different need:

C and C++
: Systems programming, operating systems, embedded software, and high-performance applications. The great majority of microcontroller software is written in C.

Rust
: Safe and fast systems programming; memory safety is checked by the compiler.

Fortran
: Scientific and numerical computing; the first widespread high-level language (1957).

Go
: Server and network software; fast compilation and concurrency.

Ada
: Systems where safety is critical, such as aviation, defense, and railways.

Java and C#
: Hybrid: compiled to intermediate code, run with JIT in a virtual machine.

3.1.6.1 The common task and C

To see the differences between languages fairly, all of them are given the same task. The program reads a number from the user and prints whether each number from 1 up to that number is odd or even. The task is small, but it contains all the basic parts of a program: input, type conversion, a counted loop, the remainder operation, a condition, and output. The C version is as follows:

```
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

`#include <stdio.h>` adds the declarations of the standard input/output functions. The program starts from the `main` function. `int number;` reserves an integer variable; its value is not yet defined. `printf` prints the prompt. `scanf("%d", &number)` reads a decimal integer from the keyboard. `scanf` is given not the variable itself but its **address** (`&number`). The function writes the value it reads to that address. This is the same idea as giving DOS the address of a string rather than its contents in assembly.

`for (int i = 1; i <= number; i++)` has three parts: the initialization (`i = 1`), the condition tested before each iteration (`i <= number`), and the step done after each iteration (`i++`). `i % 2` is the remainder of dividing i by 2; if the remainder is 0, the number is even. The value of i is printed in place of `%d` in `printf`, and `\n` is the newline. `return 0` returns the success exit code to the operating system. Table 3.1.13 is the trace of the program for input 3.

Step	Line	number	i	i % 2	Output added
1	`printf`	?	–	–	Enter a number:
2	`scanf`	3	–	–	(user types 3)
3	`for`: 1 ≤ 3	3	1	–	
4	`if` → `else`	3	1	1	1 is odd.
5	`i++`; 2 ≤ 3	3	2	–	
6	`if` true	3	2	0	2 is even.
7	`i++`; 3 ≤ 3	3	3	–	
8	`if` → `else`	3	3	1	3 is odd.
9	`i++`; 4 ≤ 3 false	3	4	–	loop ends
10	`return 0`	3	–	–	exit code 0

Table 3.1.13: Trace of the C program (input 3).

The loop variable `i` is defined in the `for` line, so it is valid only inside the loop; it cannot be used after the loop ends. The program was compiled and run with inputs 3 and 5 while these notes were prepared; the output is the same as in the table. The strings use no non-ASCII characters for a practical reason: some terminals do not display UTF-8 characters correctly. Every character of these strings is a single byte.

3.1.6.2 C++

```
#include <iostream>

int main() {
    int number;
    std::cout << "Enter a number: ";
    std::cin >> number;
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            std::cout << i << " is even."
                      << std::endl;
        } else {
            std::cout << i << " is odd."
                      << std::endl;
        }
    }
    return 0;
}
```

C++ (1985, Bjarne Stroustrup) was designed as a superset of C; most valid C code is also valid C++. It adds constructs such as classes, templates, and exceptions. For input and output, **streams** are used instead of `printf` / `scanf`. `std::cout <<` writes and `std::cin >>` reads. The `<<` operator can be chained: first `i`, then the text, then the newline is sent to the stream. No address-of operator is needed when reading; C++'s reference mechanism takes care of that.

`std::endl` adds a newline and also flushes the output buffer. Flushing the buffer on every line is slow for output with many lines; in that case `'\n'` is faster. Compilation is done with `g++ oddeven.cpp -o oddeven`. The output is identical to the C version.

3.1.6.3 C# and Java

```
using System;

class Program {
  static void Main() {
    Console.Write("Enter a number: ");
    int number = int.Parse(Console.ReadLine());
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        Console.WriteLine($"{i} is even.");
      } else {
        Console.WriteLine($"{i} is odd.");
      }
    }
  }
}
```

C# (2000, Microsoft) is the language of the .NET platform. The source code is compiled to the IL intermediate language; .NET's virtual machine, the CLR (common language runtime), translates this code to machine code with JIT while running. The program starts with the `Main` method inside a class. `Console.ReadLine()` reads a line of text and `int.Parse` converts it to an integer. If the input is not a number, `int.Parse` throws an exception and the program stops; `int.TryParse` is safer because it returns false in that case. The notation `$"{i} …"` inserts the value of a variable into the string (*string interpolation*). The program is run with `dotnet run`.

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int number = scanner.nextInt();
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        System.out.println(i + " is even.");
      } else {
        System.out.println(i + " is odd.");
      }
    }
  }
}
```

Java (1995, Sun Microsystems) is also a hybrid language. The command `javac Main.java` produces the file `Main.class`, which contains processor-independent **bytecode**. The command `java Main` starts the JVM; the JVM first interprets the bytecode and compiles frequently executed parts with JIT.

The same `.class` file runs on every platform that has a JVM. Java's slogan “write once, run anywhere” comes from this.

In Java everything is inside a class, and the name of the `public` class must match the file name: the class `Main` lives in the file `Main.java`. Input is read with the `Scanner` class; `nextInt()` returns the next integer. When the `+` operator joins a number to a string, it converts the number to text. The program was compiled and run with JDK 21 while these notes were prepared and gave the same output as C.

3.1.6.4 Go and Rust

```
package main

import "fmt"

func main() {
    var number int
    fmt.Print("Enter a number: ")
    fmt.Scan(&number)
    for i := 1; i <= number; i++ {
        if i%2 == 0 {
            fmt.Printf("%d is even.\n", i)
        } else {
            fmt.Printf("%d is odd.\n", i)
        }
    }
}
```

Go (2009, Google) compiles directly to machine code and produces a single, self-contained executable that includes all its dependencies. Memory is managed by a **garbage collector**; memory that is no longer used is reclaimed automatically. For concurrent work there are **goroutines**, which are very lightweight threads. The syntax resembles C, but conditions and loops use no parentheses. The notation `i := 1` defines the variable and infers its type from the value on the right. `fmt.Scan(&number)` takes an address as in C. The program is run with `go run oddeven.go`.

```
use std::io;

fn main() {
    let mut input = String::new();
    println!("Enter a number:");
    io::stdin().read_line(&mut input)
        .expect("read error");
    let number: i32 = input.trim().parse()
        .expect("parse error");
    for i in 1..=number {
        if i % 2 == 0 {
            println!("{} is even.", i);
        } else {
            println!("{} is odd.", i);
        }
    }
}
```

Rust (2015, Mozilla) provides memory safety without a garbage collector. Every value has a single **owner**; the ownership and borrowing rules are checked at compile time. Errors such as accessing freed memory or modifying the same data from two places at once are caught before the program runs. Variables are immutable by default; the variable `input` is defined with `mut` because it will be filled with the line read.

Input is read into a string with `read_line`. `trim()` discards the newline and `parse()` converts to a number. These operations can fail; Rust does not allow this possibility to be ignored. `.expect("…")` prints the message and stops the program if there is an error. `1..=number` is a range that includes the upper bound. The program is compiled with `rustc oddeven.rs` or `cargo run`.

3.1.6.5 Fortran and Ada

```
program Main
    implicit none
    integer :: number, i
    write(*,*) 'Enter a number:'
    read(*,*) number
    do i = 1, number
        if (mod(i, 2) == 0) then
            write(*,*) i, ' is even.'
        else
            write(*,*) i, ' is odd.'
        end if
    end do
end program Main
```

Fortran (1957, IBM, John Backus) is the first widespread high-level language. Its name comes from **FOR**mula **TRAN**slation, and it aims to let formulas be written directly as code. It is still common today in weather forecasting, fluid dynamics, and physics simulations; most of the large numerical libraries in these fields are written in Fortran.

`implicit none` requires all variables to be declared explicitly. In old Fortran, variables whose names started with the letters i–n were automatically taken to be integers; this rule is turned off because it hides typing errors. The loop `do i = 1, number` includes the upper bound. The remainder is computed not with an operator but with the function `mod(i, 2)`. `write(*,*)` writes in free format; a space may appear before numbers. Compilation is done with `gfortran oddeven.f90 -o oddeven`.

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   Number : Integer;
begin
   Put ("Enter a number: ");
   Get (Number);
   for I in 1 .. Number loop
      Put (I, Width => 0);
      if I mod 2 = 0 then
         Put_Line (" is even.");
      else
         Put_Line (" is odd.");
      end if;
   end loop;
end Main;
```

Ada (1983, U.S. Department of Defense) was designed for large systems where safety is critical. Its type checking is very strong; for example, integers of different types cannot be mixed without an explicit conversion. It is used in aviation, defense, and railway software. The `with` lines name the packages used, and the `use` lines make it possible to use the names in those packages without writing the package name. Reading and writing integers requires the `Ada.Integer_Text_IO` package. `Put (I, Width => 0)` writes the number without leading spaces; `Put_Line` adds a newline at the end of the text. `I mod 2` gives the remainder. Compilation is done with `gnatmake main.adb`.

The Go, Rust, Fortran, and Ada compilers were not installed on the computer on which these notes were prepared. These four programs were not run; they were checked line by line against each language's documentation.

3.1.7 Interpreted languages

3.1.7.1 How an interpreter works

An interpreted language has no separate compilation step. The programmer gives the source file directly to the interpreter: `python oddeven.py`. The interpreter is also an ordinary program running on the processor; it is mostly written in C. Its job is to read the statements in the source code in order and do the work each one asks for. This approach allows rapid development and experimentation. In the interactive shell (REPL, *read–eval–print loop*) of languages such as Python and Ruby, you type a line and see its result immediately.

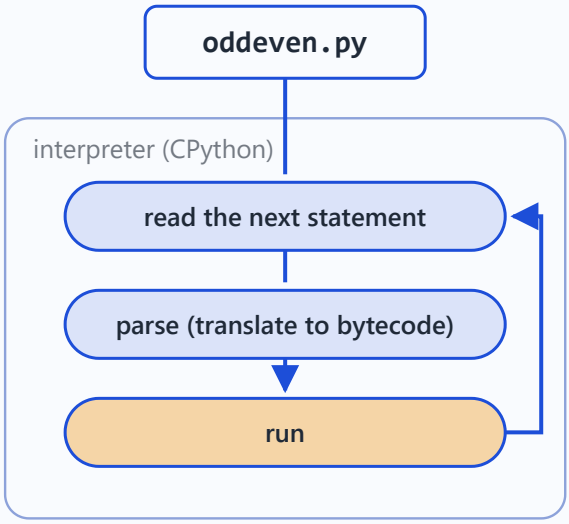


Figure 3.1.14: The interpreter loop. CPython reads the next statement, parses it by translating it to bytecode, runs it, and moves on to the next statement.

The loop in Figure 3.1.14 is the software counterpart of the processor's fetch–decode–execute cycle. The difference is that each step is done by a program instead of hardware. An addition takes a single instruction on the processor but dozens, sometimes hundreds, in an interpreter. The interpreter must first look at the types of the operands, then choose the appropriate addition routine, and then create the result as a new object. This is where the slowness of interpreted languages comes from. Real interpreters do not parse the source code as text each time; they first translate it to bytecode and interpret the bytecode. CPython stores the bytecode it produced in `.pyc` files, so on the next run it skips the translation too.

Each of the common interpreted languages stands out in a particular area. Python is general-purpose; it is widely used for data science, artificial intelligence, and automation. JavaScript is the language of the web; it runs in the browser and, with Node.js, on the server. Ruby is known for its ease of use and flexibility, especially through the Ruby on Rails web framework.

3.1.7.2 Python

```
# Read input from the user
number = int(input("Enter a number: "))

# Check the numbers from 1 to number
for i in range(1, number + 1):
    if i % 2 == 0:
        print(f"{i} is even.")
    else:
        print(f"{i} is odd.")
```

Python (1991, Guido van Rossum) puts readability first. Blocks are determined not by braces but by **indentation**. The indented lines under the `for` line are the body of the loop; when the indentation ends, the body ends. Here indentation is not a stylistic habit but a rule of the language. Wrong indentation either gives a syntax error or changes the meaning of the program.

`input` always returns text. Even if the user types 3, the returned value is the string `"3"`; `int(...)` converts it to an integer. `range(1, number + 1)` produces a sequence of numbers starting at 1, but **does not include the upper bound**. That is why 1 is added to `number`. `f"{i} is …"` is a formatted string (*f-string*): the value of the expression inside the braces is placed into the text. The program is run with `python oddeven.py`.

Python is **dynamically typed**: the type of a variable is determined at run time by the value assigned to it. `number` is not declared beforehand; the assignment creates it. The job that takes 15 lines in the C version is 6 lines of code here. This is the **expressive power** of high-level languages. The price is speed: CPython runs the same loop dozens of times slower than compiled C, because on every `i % 2` it checks the types and creates new integer objects.

3.1.7.3 JavaScript

```
// Read input from the user
let number = Number(prompt("Enter a number:"));

// Check the numbers from 1 to number
for (let i = 1; i <= number; i++) {
    if (i % 2 === 0) {
        console.log(i + " is even.");
    } else {
        console.log(i + " is odd.");
    }
}
```

JavaScript (1995, Brendan Eich) is the language of browsers; it also runs on the server with Node.js. Modern engines such as V8 first translate the code to bytecode and interpret it, and compile frequently running functions to machine code with JIT. JavaScript today therefore sits somewhere between “interpreted” and “hybrid.”

`prompt` opens an input box in the browser and returns what the user typed as **text**. `Number(...)` converts this text to a number. Without the conversion, the comparison `i <= number` would still work, because JavaScript automatically converts the text to a number when comparing a number with text. But an expression such as `number + 1` would put “3” and 1 side by side and produce “31.” An explicit conversion prevents surprises of this kind. The `===` operator tests equality without type conversion: `0 === "0"` is false, whereas `0 == "0"` is true. `console.log` writes the output to the browser's developer console or to the terminal. Since Node.js has no `prompt` function, while these notes were prepared the program was run in Node.js with a small `prompt` definition that supplies the input, and it wrote the same lines as C.

3.1.7.4 Ruby, PHP, and Perl

```
# Read input from the user
print "Enter a number: "
number = gets.chomp.to_i

# Check the numbers from 1 to number
for i in 1..number do
  if i % 2 == 0
    puts "#{i} is even."
  else
    puts "#{i} is odd."
  end
end
```

Ruby (1995, Yukihiro Matsumoto) is fully object-oriented: everything, numbers included, is an object. `gets.chomp.to_i` is three method calls in a row. `gets` reads a line, `chomp` removes the newline at its end, and `to_i` converts to an integer. `1..number` is a range that includes the upper bound. Blocks are closed with `end`. `"#{i} …"` is the counterpart of Python's formatted string: the value of the expression inside `#{…}` is placed into the text. `puts` adds a newline at the end of the text it prints. The program is run with `ruby oddeven.rb`.

```
<?php
// Read input from the user
echo "Enter a number: ";
$number = (int) trim(fgets(STDIN));

// Check the numbers from 1 to number
for ($i = 1; $i <= $number; $i++) {
    if ($i % 2 == 0) {
        echo $i . " is even.\n";
    } else {
        echo $i . " is odd.\n";
    }
}
?>
```

PHP (1995, Rasmus Lerdorf) is a web language that runs on the server side. Code is written between `<?php` and `?>` and can be embedded in HTML pages. Variable names begin with `$`. The `.` operator concatenates strings; `$i . " is …"` first converts the number to text and then concatenates. On the command line, `fgets(STDIN)` reads a line, `trim` removes the newline, and `(int)` converts to an integer. The program is run with `php oddeven.php`.

```
use strict;
use warnings;

# Read input from the user
print "Enter a number: ";
my $number = <STDIN>;
chomp $number;

# Check the numbers from 1 to number
for my $i (1..$number) {
    if ($i % 2 == 0) {
        print "$i is even.\n";
    } else {
        print "$i is odd.\n";
    }
}
```

Perl (1987, Larry Wall) was designed for text processing and system administration and is known for its powerful regular expressions. `<STDIN>` reads a line from standard input; the newline is part of the text read, and `chomp` removes it. In a double-quoted string, `$i` is replaced directly by its value. `use strict;` and `use warnings;` catch common errors such as using an undeclared variable. `my` declares a variable. The program is run with `perl oddeven.pl`.

Of these three languages, only Perl could be run while these notes were prepared; its output is the same as C. The Ruby and PHP interpreters were not installed, so the two programs were checked against the languages' documentation.

3.1.7.5 MATLAB

```
% Read input from the user
number = input('Enter a number: ');

% Check the numbers from 1 to number
for i = 1:number
    if mod(i, 2) == 0
        fprintf('%d is even.\n', i);
    else
        fprintf('%d is odd.\n', i);
    end
end
```

MATLAB (1984, MathWorks) takes its name from **MAT**rix **LAB**oratory and was designed for matrix-based numerical computing. It is very common in engineering, especially in signal processing and control systems. `input` evaluates what the user types as an expression; when 3 is typed, 3 is returned as a number. `1:number` produces a row vector made of the values 1, 2, ..., `number`, and `for` walks through the elements of this vector in turn. The remainder is found with `mod(i, 2)`. A `;` at the end of a line suppresses printing the result on the screen. Comments begin with `%`. MATLAB is a commercial product; a free and largely compatible alternative is GNU Octave. Newer versions of MATLAB also use JIT internally.

3.1.7.6 Comparing the fourteen languages

Table 3.1.14 compares the same program in fourteen languages on three points: the loop header, the remainder operation, and printing.

Language	Model	Loop header	Remainder	Printing
C	compiled	`for (int i = 1; i <= n; i++)`	`i % 2`	`printf`
C++	compiled	`for (int i = 1; i <= n; i++)`	`i % 2`	`std::cout <<`
C#	IL + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`Console.WriteLine`
Java	bytecode + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`System.out.println`
Go	compiled	`for i := 1; i <= n; i++`	`i%2`	`fmt.Printf`
Rust	compiled	`for i in 1..=n`	`i % 2`	`println!`
Fortran	compiled	`do i = 1, n`	`mod(i, 2)`	`write(*,*)`
Ada	compiled	`for I in 1 .. N loop`	`I mod 2`	`Put_Line`
Python	interpreted	`for i in range(1, n + 1):`	`i % 2`	`print`

Language	Model	Loop header	Remainder	Printing
JavaScript	interpreted/JIT	`for (let i = 1; i <= n; i++)`	`i % 2`	`console.log`
Ruby	interpreted	`for i in 1..n do`	`i % 2`	`puts`
PHP	interpreted	`for ($i = 1; $i <= $n; $i++)`	`$i % 2`	`echo`
Perl	interpreted	`for my $i (1..$n)`	`$i % 2`	`print`
MATLAB	interpreted/JIT	`for i = 1:n`	`mod(i, 2)`	`fprintf`

Table 3.1.14: The same program, fourteen languages.

Two families are clearly visible in the table. The languages derived from C (C++, C#, Java, JavaScript, PHP, Go) keep the three-part `for` header and the `%` operator. The other languages run the loop over a range or a vector: `1..=n` in Rust, `range` in Python, `1..n` in Ruby, `(1..$n)` in Perl, `1:n` in MATLAB, and a pair of bounds in Fortran and Ada. In range notation the most frequent source of errors is whether the upper bound is included. Python's `range` leaves the upper bound out; in Rust, `1..n` leaves it out and `1..=n` includes it. In Ruby, `1..n` includes the upper bound and `1...n` does not. For the remainder, Fortran, Ada, and MATLAB use `mod` instead of an operator.

Common mistake

A common mistake is to carry the range rule of one language over to another. If the C loop `for (i = 1; i <= n; i++)` is translated to Python as `for i in range(1, n):`, the last value (`n`) is skipped. The program is syntactically correct and gives no error; the missing line is noticed only when the output is examined.

3.1.8 Compiled or interpreted?

3.1.8.1 Criteria

The three execution models can be compared on five criteria (Table 3.1.15).

Criterion	Compiled	Interpreted	Hybrid (bytecode + JIT)
Translation time	before running, once	at every run	first to intermediate code, then while running
Speed	highest	lowest	high (after warm-up)
Distributed	machine code	source code	intermediate code (`.class`, `.dll`)
Portability	recompiled for each platform	wherever there is an interpreter	wherever there is a virtual machine
Error detection	mostly at compile time	mostly at run time	type errors at compile time

Criterion	Compiled	Interpreted	Hybrid (bytecode + JIT)
Examples	C, C++, Go, Rust, Fortran, Ada	Python, Ruby, PHP, Perl, MATLAB	Java, C#, JavaScript (V8)

Table 3.1.15: Comparison of the compiled, interpreted, and hybrid models.

"After warm-up" describes a property of JIT. When the program starts, the virtual machine interprets the code and counts which parts run often. When a frequently running loop or function crosses a certain threshold, it is compiled to machine code. A hybrid program is therefore slow in the first seconds and then runs at a speed close to compiled code. In some cases a JIT compiler can even produce better code than an ahead-of-time compiler, because it sees which values the program actually runs with.

Definition 3.1.6

Being "compiled" or "interpreted" is a property not of a language but of its **implementation**. Both a compiler and an interpreter can be written for the same language.

C is usually compiled, but C interpreters exist. Python is usually interpreted by CPython, but Cython translates Python-like code to C and compiles it, and PyPy runs it with JIT. Instead of saying "Python is a slow language," it is more accurate to say "the CPython interpreter is slow." The language itself is a definition; its speed is determined by the tool that runs it.

3.1.8.2 Intermediate code and the stack machine

Intermediate code is the machine language of an imaginary processor that belongs to no real processor. Virtual machines such as the JVM and the CLR define this imaginary processor as a **stack machine**. In a stack machine, instructions have no operand fields. The operands sit on an **operand stack**. An instruction takes the top items off the stack, performs the operation, and pushes the result back. Because the number of registers varies from processor to processor, a design without registers makes the intermediate code portable. Mapping this code onto the registers of a real processor is the JIT's job.

For the expression `x = (a + b) * c`, the intermediate code produced by the compiler is as follows:

```
// x = (a + b) * c;
iload a      // push a
iload b      // push b
iadd         // add the top two
iload c
imul         // multiply the top two
istore x     // pop into x
return
```

Let the local variables be a = 6, b = 7, c = 3. Table 3.1.16 writes the operand stack from bottom to top.

Instruction	Operation	Operand stack (bottom → top)	x
`iload a`	push a	[6]	?
`iload b`	push b	[6, 7]	?
`iadd`	pop 7 and 6, push 13	[13]	?
`iload c`	push c	[13, 3]	?
`imul`	pop 3 and 13, push 39	[39]	?
`istore x`	pop the top, store in x	[ ]	39
`return`	return	[ ]	39

Table 3.1.16: Trace of `x = (a + b) * c` on a stack machine (a = 6, b = 7, c = 3).

The `iadd` instruction has no operand name; the instruction knows which values to add from the state of the stack. The `i` at the start of the instructions indicates that the operation is on integers. The parenthesized expression has become an order of operations in the intermediate code: the addition is done before the multiplication because it is written first. This order of the intermediate code is the same as the postfix notation of the expression, `a b + c *`.

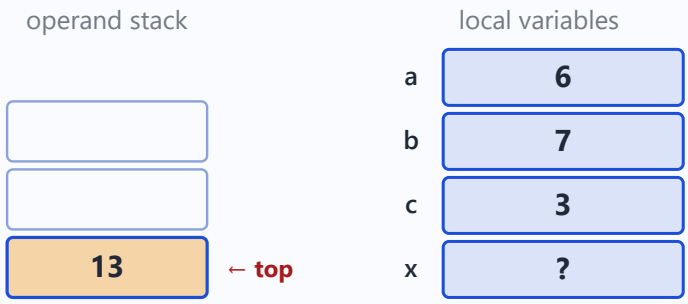


Figure 3.1.15: The stack machine after `iadd`. The top two items (6 and 7) have been popped and their sum, 13, has been pushed; the local variables are unchanged.

Real Java bytecode has the same structure. Only, local variables are referred to not by name but by index. When the following Java method is compiled with JDK 21 and disassembled with `javap -c`, this code appears:

```
static int compute(int a, int b, int c) {
    int x = (a + b) * c;
    return x;
}

 0: iload_0      // a (local 0)
 1: iload_1      // b
 2: iadd
 3: iload_2      // c
 4: imul
 5: istore_3     // x (local 3)
 6: iload_3
 7: ireturn
```

The numbers on the left are byte positions inside the bytecode; each instruction here is a single byte. The call `compute(6, 7, 3)` returns 39. The bytecode produced by CPython has the same form: the line `x = (a + b) * c` is translated into the instructions `LOAD_NAME a`, `LOAD_NAME b`, `BINARY_OP +`, `LOAD_NAME c`, `BINARY_OP *`, `STORE_NAME x`. Both virtual machines are stack machines.

This intermediate code is tied to no processor. The virtual machine either interprets each instruction or translates it with JIT into the machine code of a real processor such as the 8086, x86-64, or ARM. On the 8086 this expression takes a few instructions using registers instead of an operand stack: `mov ax, a`, `add ax, b`, `imul c`, `mov x, ax`. The “push onto the stack” and “pop from the stack” operations of the stack machine turn into loads into registers and stores from registers on a real processor.

3.1.9 The same task in 8086 assembly

3.1.9.1 Program and memory layout

Finally, the odd/even task is written in 8086 assembly. The program is assembled in emu8086 as a COM file. Two simplifications keep the task short. `n` = 3 is written into the code as a constant; it is not read from the keyboard. The numbers are single-digit, so printing a number means printing a single ASCII character.

```
org 100h
       mov  cx, 1       ; i = 1
again: cmp  cx, 3       ; compare i with n
       ja   done        ; done if i > n
       mov  dl, cl
       add  dl, '0'     ; digit → ASCII
       mov  ah, 02h
       int  21h         ; print DL
       test cl, 1       ; lowest bit
       jnz  odd         ; 1 means odd
       lea  dx, even_m
       jmp  print
odd:   lea  dx, odd_m
print: mov  ah, 09h
       int  21h         ; print string
       inc  cx          ; i++
       jmp  again
done:  ret
odd_m  db ' is odd.',0Dh,0Ah,'$'
even_m db ' is even.',0Dh,0Ah,'$'
```

The CX register is the loop counter and stands in for `i` in C. The `cmp cx, 3` and `ja done` at the label `again` are the inverted form of the C condition `i <= n`: leave the loop if `i > n`. `ja` (*jump if above*) jumps on “greater” in an unsigned comparison; the jump is taken when CF = 0 and ZF = 0. Since the counter is never negative, an unsigned jump is correct here.

The first four lines of the body print the number. `mov dl, cl` copies the low byte of the counter into DL. `add dl, '0'` adds to it the ASCII code of the character '0', that is, 30h. In the ASCII table the digits run in order from 30h to 39h; 1 + 30h = 31h is the character '1'. `mov ah, 02h` and `int 21h` are DOS's “print the character in DL” service.

The next two lines are the counterpart of the C test `i % 2 == 0`. The remainder of division by 2 is the lowest bit of the number. `test cl, 1` performs a bitwise AND of CL and 1, does not store the result, and only sets the flags. If the lowest bit is 1, the result is 1 and ZF = 0; if the bit is 0, the result is 0 and ZF = 1. `jnz odd` (*jump if not zero*) jumps when ZF = 0. The division instruction (`div`) can take more than 80 clock periods on the 8086; `test` takes only a few. Compilers choose the same route.

For an even number the flow continues without jumping: `lea dx, even_m` loads the address of the even message into DX, and `jmp print` goes to the common printing code. For an odd number the jump goes to the label `odd` and the address of the odd message is loaded into DX. The two paths merge at the label `print`: service 09h prints from the address in DX up to `$`. The `0Dh, 0Ah` at the end of the messages are the carriage return and line feed characters; in DOS both are needed to move to a new line. `inc cx` increments the counter, and `jmp again` returns to the condition. `ret` returns the COM program to DOS.

The program is laid out in memory as follows. The lengths and addresses of the instructions are given in Table 3.1.17; the code is 38 bytes, and the messages begin right after the code.

Address	Bytes	Line
0100h	3	`mov cx, 1`
0103h	3	`again: cmp cx, 3`
0106h	2	`ja done`
0108h	2	`mov dl, cl`
010Ah	3	`add dl, '0'`
010Dh	2 + 2	`mov ah, 02h` , `int 21h`
0111h	3	`test cl, 1`
0114h	2	`jnz odd`
0116h	3	`lea dx, even_m`
0119h	2	`jmp print`
011Bh	3	`odd: lea dx, odd_m`
011Eh	2 + 2	`print: mov ah, 09h` , `int 21h`
0122h	1	`inc cx`
0123h	2	`jmp again`
0125h	1	`done: ret`
0126h	11	`odd_m db …` (8 characters + 0Dh + 0Ah + '$')
0131h	12	`even_m db …` (9 characters + 0Dh + 0Ah + '$')

Table 3.1.17: Memory layout of the 8086 odd/even program.

All the conditional and unconditional jumps are two bytes. The first byte is the opcode and the second is a signed displacement: the target is found by adding this displacement to the address of the next instruction. For example, `ja done` is at 0106h; the next instruction starts at 0108h and the target is 0125h. The displacement is 0125h − 0108h = 1Dh, and the instruction is encoded as `77 1D`. Thanks to this **relative** addressing, the jumps work wherever in memory the program is loaded.

3.1.9.2 Code trace

Table 3.1.18 traces the whole program. The flags are the values after the last flag-affecting instruction of the row group. MOV, LEA, the return from INT 21h, and the jump instructions do not change the flags.

Step	Lines	CX	DL / DX	ZF	CF	Event
1	`mov cx, 1`	0001h	–	–	–	i = 1
2	`cmp cx, 3` , `ja`	0001h	–	0	1	1 − 3 borrows; not taken
3	`mov dl, cl` … `int 21h`	0001h	DL = 31h	0	0	“1” is printed
4	`test cl, 1` , `jnz`	0001h	DL = 31h	0	0	1 AND 1 = 1; taken (odd)
5	`lea dx, odd_m` … `int 21h`	0001h	DX = 0126h	0	0	“ is odd.”
6	`inc cx` , `jmp`	0002h	0126h	0	0	i = 2

Step	Lines	CX	DL / DX	ZF	CF	Event
7	`cmp cx, 3`, `ja`	0002h	0126h	0	1	not taken
8	`mov dl, cl` … `int 21h`	0002h	DL = 32h	0	0	“2” is printed
9	`test cl, 1`, `jnz`	0002h	DL = 32h	1	0	2 AND 1 = 0; not taken (even)
10	`lea dx, even_m`, `jmp print`	0002h	DX = 0131h	1	0	go to the common printing code
11	`mov ah, 09h`, `int 21h`	0002h	DX = 0131h	1	0	“ is even.”
12	`inc cx`, `jmp`	0003h	0131h	0	0	i = 3
13	`cmp cx, 3`, `ja`	0003h	0131h	1	0	3 − 3 = 0; ZF = 1, not taken
14	`mov dl, cl` … `jnz`	0003h	DL = 33h	0	0	“3” is printed; odd
15	`lea dx, odd_m` … `int 21h`	0003h	DX = 0126h	0	0	“ is odd.”
16	`inc cx`, `jmp`	0004h	0126h	0	0	i = 4
17	`cmp cx, 3`, `ja`	0004h	0126h	0	0	4 − 3 = 1; **taken**
18	`done: ret`	0004h	0126h	0	0	return to DOS

Table 3.1.18: Trace of the 8086 odd/even program (n = 3).

In the second step, `cmp cx, 3` performs the subtraction 1 − 3. As unsigned numbers 1 is less than 3; the subtraction borrows and CF = 1. `ja` jumps only when CF = 0 and ZF = 0, so it is not taken. In the third step `add dl, '0'` produces 01h + 30h = 31h; there is no carry, so CF = 0. In the fourth step `test` leaves ZF = 0 and `jnz` is taken. The `test` instruction always clears CF and OF.

The `inc cx` in the sixth step carries a subtlety: INC updates ZF, SF, and OF but **does not change CF**. Here CF is already 0, but in loop counters this property matters; the carry of a previous addition can be preserved through an INC. In the thirteenth step 3 − 3 = 0: ZF = 1, CF = 0. Because the condition of `ja` also requires ZF = 0, the jump is not taken; equality does not count as “above.” In the seventeenth step 4 − 3 = 1: ZF = 0, CF = 0 and `ja done` is taken. In the end the screen shows these three lines:

```
1 is odd.
2 is even.
3 is odd.
```

The output is the same as the output of the C program with input 3, apart from the prompt line.

3.1.9.3 Comparison with C

The `for` and `if` constructs of C do not exist on the 8086. All of them are built from **compare + conditional jump** pairs. Table 3.1.19 gives this mapping line by line.

C	8086
`int i = 1`	`mov cx, 1` (a register instead of a variable)
`i <= number`	`cmp cx, 3` + `ja done` (inverted condition)

C	8086
`i++`	`inc cx` + `jmp again`
`i % 2 == 0`	`test cl, 1` + `jnz odd` (lowest bit)
`if … else`	two branches: two `lea` instructions that merge with `jmp print`
`printf("%d …", i)`	`add dl, '0'` + service 02h, then service 09h
`return 0`	`ret`

Table 3.1.19: 8086 equivalents of C constructs.

The C version is 15 lines and can print any number. The 8086 version is 20 lines, but it prints only single-digit numbers and does not read `n`. Printing a multi-digit number needs a loop that repeatedly divides the number by 10 and converts the remainders to digits, and reading from the keyboard needs a loop that assembles the digits. That is the work `printf` and `scanf` do in C. One line of a high-level language often takes a whole routine in assembly. In return, the assembly programmer controls every byte and every clock period: the 8086 program is 61 bytes and needs no library.

3.1.10 Common mistakes

Common mistake

Confusing an address with its contents. Every service that prints a string asks for the *address* of the string: DX in DOS, ECX in Linux, r1 on ARM, a value pushed on the stack in Windows. In assembly, the notation `mov dx, msg` in some assemblers loads not the address but the word at that address. For the address, use `lea dx, msg` or `mov dx, offset msg`. Forgetting the `&` in the C call `scanf("%d", &number)` is the high-level form of the same mistake.

Common mistake

Not choosing the string-end rule for the environment. DOS service 09h expects `$`; the Windows API and the C library expect a 0 byte; Linux's `write` call looks at no terminator and takes the length as a parameter. If `$` is put at the end of the string in a Linux program, this character is printed to the screen; in a DOS program, a string ending in 0 runs on until a `$` is found.

Common mistake

Putting the data before the code. A COM program starts running at its first byte. If a `db` line is put right after `org 100h`, the processor decodes the bytes of the string as instructions. Data is either put after the code, or a `jmp` at the start of the code jumps over the data.

Common mistake

Mixing up signed and unsigned jumps. After the same `cmp` instruction, `ja` / `jb` make an unsigned comparison and `jg` / `jl` a signed one. If `ja` is used with a counter that can be negative, −1 (FFFFh) is seen as the largest unsigned number and the loop continues on the wrong branch.

Common mistake

Thinking "an interpreted language has no errors." In an interpreted language the program starts to run, but a syntax or type error in a branch that never runs shows up only when that branch runs. During testing, inputs that run every branch at least once must be chosen.

Common mistake

Not following the calling convention. If, after calling a stdcall function, the caller also cleans the stack with `add esp, 16`, the stack is cleaned twice. ESP moves up by an extra 16 bytes and the next `ret` takes the wrong return address. Which convention applies must be read from the function's documentation.

3.1.11 Exercises and solutions

The exercises in the slides are solved here with their intermediate steps. The first three exercises use the sample instruction format of Figure 3.1.5: 4-bit opcode, 2-bit destination, 2-bit source; HALT = 0000, ADD = 0001, SUB = 0010, AND = 0011, OR = 0100; R0 = 00, R1 = 01, R2 = 10, R3 = 11.

1. What are the binary and hexadecimal codes of the instructions `SUB R3, R0` and `OR R0, R2`?

The code of SUB is 0010. The destination is R3 = 11 and the source is R0 = 00. Written in order, the fields give `0010 1100`. The four-bit groups convert to hexadecimal digits: 0010 = 2, 1100 = C. The result is 2Ch. The code of OR is 0100; the destination is R0 = 00 and the source is R2 = 10. The instruction is `0100 0010` = 42h.

2. Which instruction is `0010 0111`? What is the result if R1 = 9 and R3 = 4?

The first four bits 0010 = SUB. The remaining four bits are 01 11: destination R1, source R3. The instruction is `SUB R1, R3` and performs R1 ← R1 − R3. R1 = 9 − 4 = 5; R3 stays 4. Binary check: 1001 − 0100 = 0101.

3. How many different opcodes can be defined in this design? How must the instruction format change to raise the number of registers to 8?

The opcode field is 4 bits: $2^4 = 16$ opcodes. Selecting one of eight registers takes $\log_2 8 = 3$ bits. The destination and source fields grow to 3 bits each, and the instruction becomes 4 + 3 + 3 = 10 bits. If the instruction is to stay at 8 bits, the register fields take 6 bits and 2 bits remain for the opcode. With 2 bits only $2^2 = 4$ operations can be defined; the five operations of the design would not fit. This is the basic trade-off among instruction length, number of registers, and number of instructions.

4. What happens in a DOS program if the `'$'` at the end of the string is forgotten?

Service 09h prints every byte starting at DS:DX and stops when it sees a byte with the value 24h. If there is no terminator, the memory bytes after the string are printed too. These bytes may be the program's own code, other data, or random values. Among them, a 07h byte produces a beep, and 0Dh and 0Ah produce cursor movement. Printing continues until the first `$` character in memory is met.

5. What would appear on the screen if `len` were written by hand as 15 in the Linux example?

`sys_write` writes exactly as many bytes as EDX. The first 15 bytes of the string are the text "Welcome, Linux!"; the 16th byte is the newline. If EDX = 15, the text is printed but the newline is not. The terminal's prompt line comes right after the text, on the same line. On return EAX = 15.

6. Why are the parameters pushed in reverse order in the Windows example? What is ESP right after the `call`?

In C calling conventions the first parameter must be at the top of the stack, just above the return address. Because the stack grows downward, the value pushed last stays on top; that is why the parameters are pushed from last to first. The function always finds the first parameter at [ESP + 4] and the second at [ESP + 8]. Four `push` instructions and the return address pushed by `call` take 5 × 4 = 20 bytes. ESP = 0012FF80h − 14h = 0012FF6Ch.

7. Why can Java not be classified in a single word as "compiled" or "interpreted"?

Java source code is compiled with `javac`, but the output is bytecode, not machine code. This bytecode is first interpreted in the JVM; its frequently running parts are compiled to machine code with JIT at run time. The program is both compiled in advance and interpreted and recompiled while running. That is why Java is an example of the hybrid model. Moreover, the compiled/interpreted distinction is a property of the implementation, not of the language: there are also tools that compile Java ahead of time directly to machine code.

3.1.12 Summary

- **Software** manages the hardware. The **operating system** is the fundamental software; applications reach the hardware through its system calls. Programming languages are used to develop software.
- Languages are at three levels: **machine language**, **assembly language**, and **high-level languages**. As the level rises, readability and portability increase and direct control over the hardware decreases.
- **Machine language** is the only language the processor understands: binary instructions made of opcode and operand fields. It depends entirely on the hardware; `B0 05` on the 8086 and `E3A00005` on ARM do the same job. The 8086 stores multi-byte numbers low byte first.
- In an instruction format, a field of k bits encodes one of 2^k alternatives. Decoding an instruction means splitting the bits into their fields and reading each field from a table.
- **Assembly language** writes machine instructions with mnemonics. The **assembler** translates source code into an object file, the **linker** turns object files and libraries into an executable file, and the **loader** places the program in memory. A two-pass assembler fills the symbol table in the first pass and produces the bytes in the second.
- Assembly code depends on both the processor and the operating system: `int 0x80`, `int 21h`, a DLL call, `jsr`, `svc`, `sc`. Service numbers, parameter registers, and string-end rules change with the environment.
- **Compiled languages** (C, C++, Go, Rust, Fortran, Ada) are translated to machine code in advance: they are fast, but the output is platform-specific. **Interpreted languages** (Python, Ruby, PHP, Perl, MATLAB) are processed while running: they give fast development and portable source, but run more slowly. Java and C# sit between the two with **intermediate code + JIT**; JavaScript engines use JIT too.
- Intermediate code is the language of a **stack machine** that uses no registers; operands are taken from the stack and results are put on the stack.
- High-level `while`, `for`, and `if` constructs are realized in assembly as **compare + conditional jump**; the condition is usually inverted. Whether a variable is signed or unsigned determines which jump instruction is chosen.

3.1.13 Key terms

software operating system high-level / low-level language machine language machine code assembly language mnemonic instruction set instruction format opcode operand encoder / decoder source code assembler object file linker loader executable file listing file directive location counter ($) symbol table syntax (Intel / AT&T) system call little-endian string calling convention compiler interpreter bytecode / intermediate language virtual machine

just-in-time (JIT) compilation stack machine implementation

3.1.14 Test Yourself

1. In the sample instruction format, addresses 0, 1, and 2 hold the bytes 13h, 2Ch, and 00h in that order. Initially R0 = 2, R1 = 0, R2 = 5, R3 = 9. What are the registers at HALT?

   Answer

   13h = 0001 00 11: ADD R0, R3 → R0 = 2 + 9 = 11. 2Ch = 0010 11 00: SUB R3, R0 → R3 = 9 − 11 = −2; in an 8-bit register, in two's complement, 1111 1110 = FEh. 00h: HALT. Result: R0 = 11, R1 = 0, R2 = 5, R3 = FEh (−2). The second instruction uses the value of R0 left by the first.

2. On the 8086 the instruction `mov ax, [0300h]` is encoded as `A1 00 03`. If address 0300h holds `34` and address 0301h holds `12`, what is AX? What would change on the data bus if the instruction read a word from address 0301h?

   Answer

   Because the low byte is stored first, AX = 1234h. 0301h is an odd address. The 8086's 16-bit data bus can read in one cycle only a word that starts at an even address. A word at an odd address arrives with two separate byte reads, that is, two bus cycles. The result is the same, but the access takes at least 4 clock periods longer.

3. Why does the first pass of a two-pass assembler not produce the bytes of the instructions? If all labels were defined before they were used, would a single pass be enough?

   Answer

   In the first pass the addresses of forward references are not yet known; an operand such as `offset msg` cannot be encoded. This pass only adds up the lengths and fills the symbol table. If all labels were defined before use, every address would be known when needed and a single pass would be enough. But most jump instructions go forward, for example to the end of a loop; this restriction cannot be applied in practice.

4. In a DOS COM program the line `msg db 'Hello$'` starts at address 0110h. What is the machine code of the instruction `lea dx, msg`? How many characters does service 09h print?

   Answer

   emu8086 encodes a direct-address `lea dx, …` like `mov dx, offset …`: `BA` and the little-endian address, that is, `BA 10 01` (3 bytes). In MASM the same instruction would be `8D 16 10 01`. Service 09h prints the 5 characters of the word “Hello”; the `$` is not printed.

5. What does the sequence `mov eax, 1` / `mov ebx, 7` / `int 0x80` do on 32-bit Linux? With which registers and which instruction is the same job done on 64-bit Linux?

   Answer

   EAX = 1 is `sys_exit`; the process terminates with exit code 7. On 64-bit Linux the number of the exit service is 60 and it goes in RAX, the exit code in RDI; the switch into the kernel is made with the `syscall` instruction.

6. Before a function with two parameters is called under the stdcall convention, ESP = 0012FF40h. What is ESP at the moment of entering the function, and what does [ESP + 8] hold? With which instruction must the function return?

   Answer

   Two `push` instructions and the return address take 12 bytes: ESP = 0012FF40h − 0Ch = 0012FF34h. [ESP] is the return address, [ESP + 4] the first parameter, and [ESP + 8] the second parameter. To discard the 8 bytes of parameters itself, the function returns with `ret 8`; on return ESP is 0012FF40h again.

7. What is the intermediate code of the expression `a * (b - c)` on a stack machine? For a = 4, b = 10, c = 3, what is the maximum depth of the stack?

   Answer

   `iload a`, `iload b`, `iload c`, `isub`, `imul`, `istore x`. The stack goes through [4], [4, 10], [4, 10, 3], [4, 7], [28], [ ]. The maximum depth is 3; the result is x = 28.

8. In the 8086 odd/even program, `cmp cx, 5` is written instead of `cmp cx, 3` and `jae` is mistakenly used instead of `ja`. Which numbers does the program print?

   Answer

   `jae` jumps when CF = 0, that is, when CX ≥ 5. For CX = 1, 2, 3, 4, CF = 1 and the body runs; at CX = 5, 5 − 5 = 0 and CF = 0, and the loop is left. The program prints 1 through 4; 5 is missing. The correct choice is `ja`: the body must also run on equality.

9. When a Python program is written with `for i in range(1, n):`, over which values does it iterate for n = 4? What happens if the same loop is written in Ruby as `1..n` and as `1...n`?

   Answer

   Python iterates over 1, 2, 3; the upper bound is not included. In Ruby, `1..4` iterates over 1, 2, 3, 4 and `1...4` over 1, 2, 3. Two dots include the upper bound; three dots leave it out.

3.1.15 Further reading

- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), Chapters 1–3: language levels, the assembler and linker, listing files, and first programs on x86; it is the source closest to the assembly sections of these notes.
- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapters 1 and 4: machine language, encoding of 8086 instructions, the ModR/M byte, and the structure of COM/EXE programs.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (3rd ed.), Section 1.6: how system calls work, user and kernel modes.
- J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann: describes object files, symbol resolution, relocation, and the loading process in detail.

3.1.16 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 3rd edition, Pearson.
2. B. B. Brey, *The Intel Microprocessors*, 8th edition, Pearson.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7th edition, Pearson.
4. NASM documentation (nasm.us) and the GNU Binutils *as* manual.
5. Ralf Brown's Interrupt List (DOS INT 21h services); Microsoft Win32 API documentation (`MessageBoxA`, `ExitProcess`).
6. emu8086 documentation and the 8086 instruction set reference.
7. J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann, 1999.

PART I · CHAPTER 3.2

Assembly Language

Assembly language writes a processor's machine instructions as human-readable abbreviations. Every instruction line corresponds to a single machine instruction. For that reason, anyone writing assembly manages registers, memory, and flags directly; every detail a compiler would hide becomes visible. These notes treat 8086 assembly in the emu8086 environment. First come the fields of a line, the difference between an instruction and a directive, and the naming rules. Then we examine operand types, the implicit operands that instructions do not spell out, and the cost of a memory access in clock periods. We trace step by step how IP advances on every instruction and which instructions change the flags. The last sections write a first .COM program, read the bytes it occupies in memory one by one, run it in the emu8086 window, and classify the most common mistakes together with their causes.

LEARNING OUTCOMES

Explain where assembly fits

Show the one-to-one relationship between assembly language and machine language, and how it differs from high-level languages, through the three views of a single instruction.

Read the structure of a line

Separate the label, instruction, operand, and comment fields; tell an instruction from a directive and a macro; recognize valid and invalid names.

Recognize operands

Identify register, immediate, and memory operands, and the implicit operands of instructions such as MUL, DIV, LOOP, CBW, PUSH, and XLAT.

Compute instruction timing

Find the time difference between instructions with register and memory operands using clock periods and effective address calculation; explain the difference through bus cycles.

Trace IP and the flags

Trace step by step how IP advances by the instruction length and what instructions that change or leave the flags alone do.

Write the first program
Write a .COM program in emu8086, read its bytes, run it one step at a time; tell a syntax error from a logic error.

3.2.1 Assembly language

3.2.1.1 Mnemonics and the machine instruction

A microprocessor decodes the bytes it reads from memory and turns each byte sequence into an operation. The language made of these bytes is called **machine language**. Machine language is the only language the processor understands directly, but it is hard for people to read. Someone looking at the bytes `B8 05 00` can work out that they mean "load 5 into register AX" only by consulting a table. When a program reaches hundreds of instructions, this method breaks down.

Assembly language was born to solve this problem. Every machine instruction is given a short, memorable name: `MOV` for moving, `ADD` for addition, `JMP` for jumping. These short names are called **mnemonics**. Operands are also written by name instead of by number: registers as AX and BX, and memory locations by names the programmer chooses.

Definition 3.2.1

Assembly language is a low-level programming language that writes a processor's machine instructions with mnemonics and symbolic names. Each instruction line in assembly source corresponds to a single machine instruction of the processor. This one-to-one relationship makes the language processor-dependent: every processor family has its own assembly language.

The easiest way to see the one-to-one relationship is to write a single line in three different forms. Table 3.2.1 shows the line `MOV AX, 5` side by side as the programmer, the assembler, and the processor see it.

View	Notation	Who uses it?
Assembly	`MOV AX, 5`	programmer
Hexadecimal machine code	`B8 05 00`	listing, emulator, debugger
Binary machine code	`10111000 00000101 00000000`	processor (voltage levels on the data bus)

Table 3.2.1: One line, three views.

The first byte `B8` is the **opcode** of the operation "load an immediate value into a 16-bit register." In this instruction family the number of the destination register is added to the opcode. Since AX is number 0, the code is B8h + 0 = B8h. BX is number 3; that is why the first byte of `MOV BX, 3` is BBh.

The remaining two bytes of the instruction carry the 16-bit constant. The value 0005h is stored in memory **low byte first**: 05 first, then 00. This byte order of the 8086 is called *little-endian*.

The binary line is what the processor actually sees. The 8086 reads instruction bytes over the address/data lines (AD0–AD15). Each bit is a high or low voltage on one line. Seen through the eyes of an electronic circuit, `MOV AX, 5` is nothing but a 24-bit pattern arriving at the input of the instruction decoder. The decoder turns this pattern into control signals that drive the ALU, the write input of the register file, and the internal buses.

This closeness is also what separates assembly from high-level languages. A programmer who writes `x = a * b;` in C does not know in which register the multiplication will be done; the compiler chooses. In assembly, it is the programmer's responsibility that the multiplicand sits in AL, that the result is written to AX, and that the old value in AH is wiped out. The power of the language and its difficulty both come from here. Writing assembly requires knowing the registers, the memory, and the system bus. In return, how a program runs in hardware can be seen in every detail.

Example 3.2.1

What is the machine code of `MOV BX, 1234h` ? How many bytes does it occupy in memory?

The instruction family is the same: an immediate value into a 16-bit register. The opcode is B8h + the register number. Since BX is number 3, the first byte is BBh. The constant 1234h is written low byte first: 34h, then 12h. The instruction is `BB 34 12` and occupies three bytes. There is a separate family for 8-bit registers: `MOV AL, 5` is encoded in two bytes as `B0 05`, because the constant is a single byte.

3.2.1.2 The assembler and emu8086

Assembly source is a plain text file. The processor cannot run this text; it must first be translated into machine code. The program that does this translation is called an **assembler**. The assembler translates mnemonics into opcodes, labels into addresses, and variable names into relative locations. The translation is one-to-one: one instruction line yields one machine instruction. MASM, TASM, and NASM are common assemblers for the 8086 family.

The translated code is written to a file. For simple programs this file is directly a runnable **.COM** file. Multi-segment programs are first translated into an **object file** (.obj); then the **linker** combines the object files and produces an **.EXE** file. To run the program, the operating system's **loader** copies the file into RAM, sets the segment registers, and hands control to the program's first instruction. Figure 3.2.1 shows this chain.

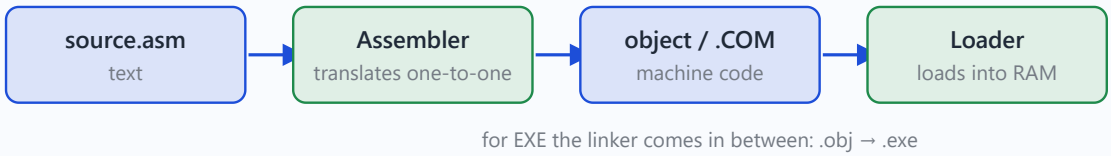


Figure 3.2.1: From source file to memory. The assembler translates the text one-to-one into machine code, and the loader places the result in RAM. For .EXE files the linker comes in between.

emu8086, used in this course, gathers all the links of the chain in a single window. It contains a text editor, an assembler, and an 8086 **emulator**. An emulator is a program that imitates the 8086 in software. In a real processor the insides of the registers are not visible from outside; the emulator, by contrast, shows the registers, memory, and flags on the screen after every instruction. A program can be advanced one instruction at a time, stepped back, and stopped wherever you like. While learning assembly this is the most valuable tool, because the effect of every instruction can be predicted and tested immediately.

The difference between the .COM and .EXE formats also determines the structure of the program. Table 3.2.2 compares the two formats.

Feature	.COM	.EXE
Number of segments	a single segment; code, data, and stack within the same 64 KB	multiple segments (separate code, data, and stack)
Size limit	64 KB minus 256 bytes (PSP) and room for the stack	the 64 KB limit applies to each segment separately
File header	none; the file is copied into memory as it is	present; contains a relocation table and initial values
Start address	`ORG 100h`, IP = 0100h	the label given in the `END` directive
Segment registers	CS = DS = ES = SS	the program loads DS itself

Table 3.2.2: .COM and .EXE programs.

All the examples in these notes are in the .COM format and are written in emu8086 syntax. Most of the examples also assemble unchanged with MASM and TASM. The byte values of the instructions and the IP and register traces were obtained by running them on an 8086 emulator.

3.2.1.3 When is assembly used?

A program written in assembly is generally smaller and faster than a high-level program that does the same job. The reason is simple: the program contains only the instructions the programmer wrote. There is no startup code added by a compiler, no library calls, and no general-purpose helper routines. Why every instruction is there is known.

A more important advantage than speed is direct access to the hardware. Some 8086 instructions have no counterpart in standard C. `OUT` writes a byte to an I/O port, `CLI` disables interrupts, and `PUSHF` pushes the flags register onto the stack. Bootloaders, device drivers, interrupt service routines, and small embedded systems are therefore written wholly or partly in assembly. The first code that runs when a computer is powered on is assembly anyway, since the stack and memory layout have not yet been set up.

The price of the language is also heavy. Table 3.2.3 summarizes both sides.

Pros	Cons
Takes less space; there is no unnecessary code.	Not portable: 8086 code does not run on an ARM processor.
Fast, because it performs only the necessary operations.	Laborious to write, read, and debug.
Gives port I/O, interrupt handling, and direct access to the flags.	Low productivity in large projects.
Indispensable in bootloaders, drivers, and embedded systems.	Modern compilers often produce code that is fast enough.

Table 3.2.3: Pros and cons of programming in assembly.

In practice the two sides are used together. Most of the program is written in a language such as C. Small time-critical pieces, such as the entry of an interrupt routine or the inside of a frequently called loop, are written in assembly. This **mixed** approach preserves both development speed and run speed. Another benefit of knowing assembly is reading high-level code. An engineer who can look at the assembly output a compiler produced can find out why the code runs slowly or why a variable took an unexpected value.

> **Common mistake**
>
> **Believing "assembly is always faster."** Hand-written assembly is fast only if the programmer knows the processor better than the compiler does. A loop counter kept in memory, memory reads repeated needlessly, or a badly chosen instruction yield a program slower than the code a good compiler would produce. Speed comes less from the language than from choosing register and memory use correctly.

3.2.2 The structure of an instruction line

3.2.2.1 The fields of a line

Assembly source is read line by line. Each line carries at most one instruction or one directive. The general form of an instruction line is as follows; the fields in square brackets are optional:

```
[label:]  instruction  [destination], [source]  [; comment]
```

Figure 3.2.2 shows these fields on the line `start: MOV AX, 5 ; AX ← 5`. The **label** is the name given to the address of the line. The **instruction** field carries the mnemonic. After the instruction come at most two **operands** separated by a comma. The part from the semicolon to the end of the line is the **comment**.

Figure 3.2.2: The fields of an instruction line. The label `start`, the instruction `MOV`, `AX` the destination operand, `5` the source operand, and everything after the semicolon is a comment. The effect of the line is the assignment AX ← 5.

Definition 3.2.2

An **operand** is the value, register, or memory location an instruction operates on. In two-operand instructions in Intel syntax, the left operand is the **destination** and the right one is the **source**. The instruction reads the source and writes the result to the destination. The source does not change; the old value of the destination is lost.

The destination–source order is where everyone learning assembly first stumbles. `MOV AX, BX` is read not as "move AX to BX" but as "AX ← BX." This order is the same as assignment in C: `ax = bx;`. Two-operand arithmetic instructions follow the same rule. When `SUB CX, AX` runs, CX ← CX − AX; AX does not change. Thus the destination operand is both the first input and the place where the result is written.

Not every instruction takes two operands. Table 3.2.4 splits three lines with different numbers of operands into their fields.

Line	Label	Instruction	Operands	Comment
`start: MOV AX, 5 ; AX ← 5`	`start`	`MOV`	`AX`, `5`	`; AX ← 5`
`INC AX`	—	`INC`	`AX`	—
`RET`	—	`RET`	none	—

Table 3.2.4: The fields of three lines.

`INC AX` has one operand: AX is both source and destination, and AX ← AX + 1. `RET` writes no operand at all, but it still does work: it reads a word from the stack and puts it into IP. These unwritten operands are treated in detail under implicit operands.

The assembler is case-insensitive. `mov ax, bx`, `MOV AX, BX`, and `Mov Ax, Bx` give the same machine code. The number of spaces and tabs is also unimportant; aligning the fields one under another is only for readability. Comments never enter the machine code. When the assembler sees a semicolon, it skips the rest of the line. A comment can therefore be as long as you like without affecting the size or speed of the program.

Comments matter more in assembly than in high-level languages. The line `ADD AX, BX` already says what it does; it does not say why. Whether AX holds a running total, an address, or a counter cannot be told from the code. A good comment does not repeat the instruction as "AX ← AX + BX"; it states the purpose, such as "add this student's grade to the total."

3.2.2.2 Instruction, directive, and macro

Not every line in an assembly source produces machine code. Lines fall into three kinds.

Definition 3.2.3

An **instruction** is an operation the processor executes at run time and is translated into a single machine instruction. A **directive** is information given to the assembler: it states where the code will be placed, how much space to reserve, and what a name will stand for. A directive produces no machine instruction; data-definition directives produce only data bytes. A **macro** is a named text pattern that is replaced by several lines at assembly time.

The difference between an instruction and a directive is in timing. The instruction is executed by the processor while the program runs. A directive carries meaning only at assembly time; when the program starts to run, no trace of it remains. Table 3.2.5 gives frequently used lines with their kinds and what they produce in memory.

Line	Kind	What does it produce in memory?
`MOV AX, 5`	instruction	machine code: `B8 05 00`
`ORG 100h`	directive	nothing; sets the location counter to 0100h
`num DB 0`	directive (data definition)	1 byte of data: `00`
`k EQU 5`	directive	nothing; the name `k` takes the value 5
`END`	directive	nothing; the source ends here
`PUTC 'A'`	macro call	the code of the instructions in the macro body

Table 3.2.5: Line kinds and what they produce in memory.

To understand how directives work, one must know the assembler's **location counter**. While reading the source from top to bottom, the assembler keeps in a counter the relative location where the next byte will be placed. Each instruction or data definition advances the counter by the number of bytes it produces. `ORG 100h` sets the counter directly to 0100h; it produces no bytes itself. `EQU`, on the other

hand, never touches the counter: it only binds a name to a constant. The assembler writes 5 wherever it sees the name `k`. The line `num DB 0` binds the current value of the counter to the name `num`, reserves one byte, and puts 0 into it.

The small program below makes this difference concrete. The comments on the right show the addresses and bytes in the assembler's listing.

```
ORG 100h            ; no bytes; location counter = 0100h
k EQU 5             ; no bytes; k = 5
MOV AX, 5           ; 0100h: B8 05 00
MOV BX, k           ; 0103h: BB 05 00  (5 in place of k)
MOV AL, num         ; 0106h: A0 0A 01  (num = 010Ah)
RET                 ; 0109h: C3
num DB 0            ; 010Ah: 00
END                 ; no bytes
```

Of the eight lines only five produce bytes, and the program is 11 bytes. The codes of `MOV AX, 5` and `MOV BX, k` are the same apart from the register number, because the assembler put 5 in place of `k`. A constant defined with `EQU` occupies no memory. The situation is different in the line `MOV AL, num`. `num` is a memory location; the instruction reads the byte at that location. The assembler writes the location's address into the instruction: 010Ah, low byte first, `0A 01`. This address is not known until the assembler reaches the `num` line. How the assembler resolves this forward reference is explained by two-pass assembly; what matters here is that the address is computed at assembly time and embedded in the instruction.

The macro is the third kind. `PUTC 'A'`, from the `emu8086.inc` library that ships with emu8086, is the name of a pattern of a few instructions that prints one character on the screen. When the assembler sees this line, it copies the body of the macro in place of the line and translates the copy. A macro call is not a subroutine call: the body is rewritten at every use and the program grows accordingly. Macros are a separate topic; they are mentioned here only to show that one line can produce more than one machine instruction.

Common mistake

Treating an EQU constant as a variable. After the definition `k EQU 5`, writing `MOV k, 7` means “5 ← 7” and the assembler reports an error. An EQU name occupies no memory; its value is fixed at assembly time and cannot be changed at run time. A quantity whose value will change is defined with `DB` or `DW`.

3.2.2.3 Labels and names

In an assembly program, addresses are written as names instead of numbers. The name given to the address of an instruction is called a **code label**, and the name given to a data area is called a **variable name**. Both are bound to an address in the assembler's symbol table, but they are used differently.

```
start:  MOV CX, 3   ; code label
again:  INC AX
        LOOP again  ; again = address
        RET
num     DB 7        ; data name
```

A code label ends with a colon. The label itself produces no machine code; it only stands for the address of the instruction that follows it, that is, its relative location within CS. If this program is assumed to start with `ORG 100h`, `start` = 0100h. `MOV CX, 3` is three bytes (`B9 03 00`); `again` = 0103h. `INC AX` is a single byte (`40`), and `LOOP again` is two bytes (`E2 FD`). `RET` is at 0106h and `num` at 0107h.

The second byte of `LOOP again`, FDh, is noteworthy. The instruction stores not the address but the **displacement**. When LOOP executes, IP already points to the next instruction, that is, 0106h. Since the target is 0103h, the displacement is 0103h − 0106h = −3. −3 is written as FDh in 8-bit two's complement. The label spares the programmer this calculation: the programmer writes `again` and the assembler computes the displacement. When a new instruction is added to the program and addresses shift, the calculation is renewed automatically.

A variable name is written without a colon. For a variable name, the assembler keeps three pieces of information: its address, its segment, and its **type**. A name defined with `DB` has type byte, and one defined with `DW` has type word. The type information lets the assembler check sizes. If `MOV AX, num` is written after the definition `num DB 7`, the assembler notices that an 8-bit variable is being moved into a 16-bit register and reports an error.

The rules for names are as follows:

- A name begins with a letter or an underscore; it cannot begin with a digit. Any string that begins with a digit is a number to the assembler.
- The rest of the name may contain letters, digits, and underscores.
- Instruction, register, and directive names (MOV, AX, DB, END ...) cannot be used as names; these are **reserved words**.
- Case is not distinguished. `again` and `AGAIN` are the same label; defining both gives a “defined twice” error.

Name	Valid?	Why?
`_count`	**yes**	begins with an underscore
`total2`	**yes**	begins with a letter, the digit comes later

Name	Valid?	Why?
`2total`	**no**	begins with a digit; the assembler takes it for an invalid number
`cx`	**no**	it is a register name (same as CX)
`last-value`	**no**	– is an operator; the assembler reads `last` minus `value`

Table 3.2.6: Examples of names.

The last row is instructive. `last-value` is parsed as an expression. If the names `last` and `value` do not exist in the program, the assembler reports an "undefined symbol" error; if they do, it computes the difference of the two addresses and produces a number the programmer never wanted. Underscores are therefore used in names made of several words: `last_value`.

Common mistake

Thinking `Num` and `num` are two different variables. C and Java distinguish case; 8086 assemblers do not. A programmer who writes `MOV Num, 3` after the definition `num DB 7` may think a new variable has been created, but the instruction writes 3 to the same byte and the 7 is lost. The assembler does not count this as an error, because the line is valid. Different quantities are given different names.

3.2.3 Operands

3.2.3.1 Register, immediate value, and memory

The operand of an instruction can come from three places: a register inside the processor, the instruction's own bytes, or memory. These three kinds behave differently both in syntax and in hardware.

Type	Example	Where does the value come from?	Restriction
Register	`MOV AX, BX`	from the register file inside the processor	the fastest type
Immediate value	`MOV AX, 5`	from within the bytes of the instruction	can only be a source
Memory	`MOV AX, [BX]`	from memory, with a bus cycle	at most one per instruction

Table 3.2.7: Three operand types.

A **register operand** is inside the processor. The 8086's execution unit (EU) has eight general-purpose 16-bit registers: AX, BX, CX, DX, SP, BP, SI, and DI. The first four can also be used through their 8-bit halves: AH and AL, BH and BL, CH and CL, DH and DL. An instruction with register operands never goes out onto the external bus. The value travels over the internal bus to the ALU, and the result returns to the register over the internal bus.

An **immediate value** is a constant embedded in the instruction. The bytes `05 00` of `MOV AX, 5` are this constant. Because the value is fetched along with the instruction, it need not be read separately; it is already waiting in the instruction queue. An immediate value can only be a source, because writing to a constant is meaningless: `ADD 5, AX` would mean "let the new value of 5 be 5 + AX."

A **memory operand** is given by an address. In `MOV AX, [BX]` the square brackets mean "the number in BX is an address; take the word at that address." The processor first computes the address, then gives the bus interface unit (BIU) a read request. The BIU puts the address on the address bus, reads the data from memory, and passes it to the execution unit. This external access adds at least four clock periods compared with a register operand.

The basic rules among the three kinds are these. The two operands must be the same size: a 16-bit register cannot be moved into an 8-bit register. At most one memory operand can appear in an instruction; `MOV [BX], [SI]` is invalid. An immediate value cannot be loaded into a segment register (CS, DS, ES, SS). All these rules come from the 8086's instruction encoding. The instruction format has a single field that specifies a memory address; there is no room to encode a second memory operand. No opcode that loads a constant into a segment register was designed either.

Example 3.2.2

Identify the operand types of the following instructions: `ADD CL, 10`, `MOV [BX], AX`, `SUB DX, counter`.

`ADD CL, 10`: the destination is a register (CL), the source is an immediate value (10). CL ← CL + 10. There is no memory access.

`MOV [BX], AX`: the destination is memory, the source is a register. The two bytes of AX are written to the address BX points to, low byte first. One bus cycle (write) is needed.

`SUB DX, counter`: the destination is a register, the source is memory (`counter` is a variable name, which the assembler turns directly into an address). If `counter` is a `DW` variable, DX ← DX − [counter]; if it is a `DB` variable, a size-mismatch error results.

3.2.3.2 Implicit operands

Some instructions, besides the operands that are written, also use or change registers that do not appear in the instruction line at all. Which registers these are is built into the definition of the instruction.

Definition 3.2.4

An **implicit operand** is a register or memory location that an instruction reads or changes although it is not written in the instruction line. Implicit operands are fixed in the instruction's opcode; the programmer cannot choose them and only has to know them.

Instruction	Written operand	Implicit operand(s)	Operation
`MUL BL`	BL	AL (multiplicand), AX (result)	AX ← AL × BL
`DIV BL`	BL	AX (dividend), AL (quotient), AH (remainder)	AL ← AX ÷ BL, AH ← AX mod BL
`LOOP don`	again	CX (counter)	CX ← CX − 1; jump if CX ≠ 0
`CBW`	—	AL → AX	the sign bit of AL is copied to all bits of AH
`PUSH AX`	AX	SS:SP (top of the stack)	SP ← SP − 2; [SS:SP] ← AX
`XLAT`	—	AL (index and result), BX (table address)	AL ← [DS:BX + AL]

Table 3.2.8: Implicit operands of some instructions.

The implicit uses in the table are also the source of the names of the 8086 registers. AX is the **accumulator**: the result of multiplication and division is always written there. CX is the **count** register: LOOP decrements it automatically, and multi-bit shifts read the count from its lower half CL. BX is the **base** register: XLAT looks for the start of the table there. SP points to the top of the stack and changes automatically on every PUSH and POP. The “special duties” of registers are really hardwired connections of these instructions in the hardware. One input of the multiplier circuit is connected directly to AL and its output directly to AX.

The practical consequence of implicit-operand knowledge is this: an instruction can also change a register that is not written. In the line `MUL BL`, AH is never mentioned, but the instruction wipes AH out, because the 16-bit result is written to the whole of AX. A value saved in AH is lost after this line. A program that wants to preserve a register's value must know the implicit operands of the instructions it uses.

Example 3.2.3

`DIV BL` runs with AX = 100 (0064h) and BL = 7. What are AL, AH, and AX?

In 8-bit division the dividend is implicitly the whole of AX, and the divisor is the written BL. The quotient of 100 ÷ 7 is 14 and the remainder 2 (7 × 14 = 98, 100 − 98 = 2). The quotient is written to AL and the remainder to AH: AL = 0Eh, AH = 02h. Read together, the two halves give AX = **020Eh**. The value AX had before the division is lost. Had the quotient not fit in 8 bits, for example with AX = 1000 and BL = 2, the processor would have generated a divide-error interrupt (INT 0).

Example 3.2.4

`CBW` runs with AL = F6h. What is AX? What would the difference have been if `MOV AH, 0` had done the same job?

In two's complement F6h is −10; its sign bit (bit 7) is 1. CBW copies this bit into the eight bits of AH: AH = FFh, AX = **FFF6h**. FFF6h is also −10 in 16 bits; the signed value is preserved. `MOV AH, 0`, however, gives AX = 00F6h = 246. This result is correct if AL is an unsigned number and wrong if it is signed. CBW has no written operand at all; both its source (AL) and its destination (AX) are implicit.

3.2.3.3 Code trace: MUL, SHR, and LOOP

The following program uses three instructions with implicit operands in a row. In the program CX is written only once, and AX appears to be written only in the ADD line. The trace shows how many times these registers actually change.

```
ORG 100h
MOV AL, 12
MOV BL, 10
MUL BL          ; AX = AL × BL
MOV CL, 2
SHR AX, CL      ; count in CL
MOV CX, 3       ; LOOP counter
again:
ADD AX, 1
LOOP again      ; CX decrements
RET
```

At the start of the program the initial state of .COM programs applies: the general registers are 0 and the status flags are 0. Table 3.2.9 gives the state after each instruction. The address column is the instruction's relative location and the bytes column is the machine code.

Address	Bytes	Instruction	AX	BX	CX	CF	ZF	Comment
0100h	`B0 0C`	`MOV AL, 12`	000Ch	0000h	0000h	0	0	AL = 12
0102h	`B3 0A`	`MOV BL, 10`	000Ch	000Ah	0000h	0	0	BL = 10
0104h	`F6 E3`	`MUL BL`	0078h	000Ah	0000h	0	?	AX ← AL × BL = 120
0106h	`B1 02`	`MOV CL, 2`	0078h	000Ah	0002h	0	?	shift count
0108h	`D3 E8`	`SHR AX, CL`	001Eh	000Ah	0002h	0	0	120 ÷ 4 = 30
010Ah	`B9 03 00`	`MOV CX, 3`	001Eh	000Ah	0003h	0	0	the 2 in CL is overwritten
010Dh	`05 01 00`	`ADD AX, 1`	001Fh	000Ah	0003h	0	0	iteration 1
0110h	`E2 FB`	`LOOP again`	001Fh	000Ah	0002h	0	0	CX ≠ 0 → again
010Dh		`ADD AX, 1`	0020h	000Ah	0002h	0	0	iteration 2
0110h		`LOOP again`	0020h	000Ah	0001h	0	0	CX ≠ 0 → again
010Dh		`ADD AX, 1`	0021h	000Ah	0001h	0	0	iteration 3
0110h		`LOOP again`	0021h	000Ah	0000h	0	0	CX = 0 → falls through

Address	Bytes	Instruction	AX	BX	CX	CF	ZF	Comment
0112h	`C3`	`RET`	0021h	000Ah	0000h	0	0	result 33

Table 3.2.9: Trace of the implicit-operand program (each row is the state after the instruction executes).

The MUL step. In the line `MUL BL` only BL is written. The multiplicand is implicitly AL (12), and the result is implicitly AX. 12 × 10 = 120 = 0078h. Since the result fits in 8 bits, AH = 00h and CF = OF = 0 remain. On the 8086, MUL sets these two flags according to the question "is the upper half of the result significant?" SF, ZF, AF, and PF, on the other hand, are **undefined** after MUL: the processor writes some value into them, but Intel does not document that value. This is why a question mark is placed for ZF in the table. For a program to rely on ZF, an instruction that sets the flag in a defined way must follow.

The SHR step. To shift a register by more than one bit on the 8086, the shift count can be given only in CL. The form `SHR AX, 2` arrived with the 80186. The program therefore first writes `MOV CL, 2`. A right shift halves the value at every bit: 0078h = 0111 1000b shifted two bits to the right becomes 0001 1110b = 001Eh = 30. The last bit shifted out is written into CF. Since the two bits shifted out are 0 and 0, CF = 0. Since the result is nonzero, ZF = 0. With this instruction ZF has become defined again.

The MOV CX, 3 step. This line writes 3 into CX. CL is the lower half of CX; the 2 written to CL in the previous step is thereby erased. The loss is unimportant because the shifting is finished, but had the order been reversed, that is, had CX been set first and then 2 written to CL, the loop would have run 2 times instead of 3. One physical register being known under two names becomes a frequent source of errors when combined with implicit operands.

The LOOP steps. The instruction `LOOP again` does two jobs in one instruction: it decrements CX by 1 and jumps to the label `again` if CX is not zero. No line in the program decrements CX, yet the trace shows CX changing three times, 3 → 2 → 1 → 0. LOOP's decrement does not affect the flags; LOOP is an instruction that does not change flags. On each iteration `ADD AX, 1` increments AX by one: 30 → 31 → 32 → 33. At the third LOOP, CX drops to zero, the jump is not taken, and the flow falls through to RET.

At the end of the program AX = 0021h = 33. Read line by line, the result is not surprising, but someone who thinks AX is written only in the ADD line would miss that MUL and SHR changed AX. Figure 3.2.3 shows three moments of the trace with drawings from the slides.

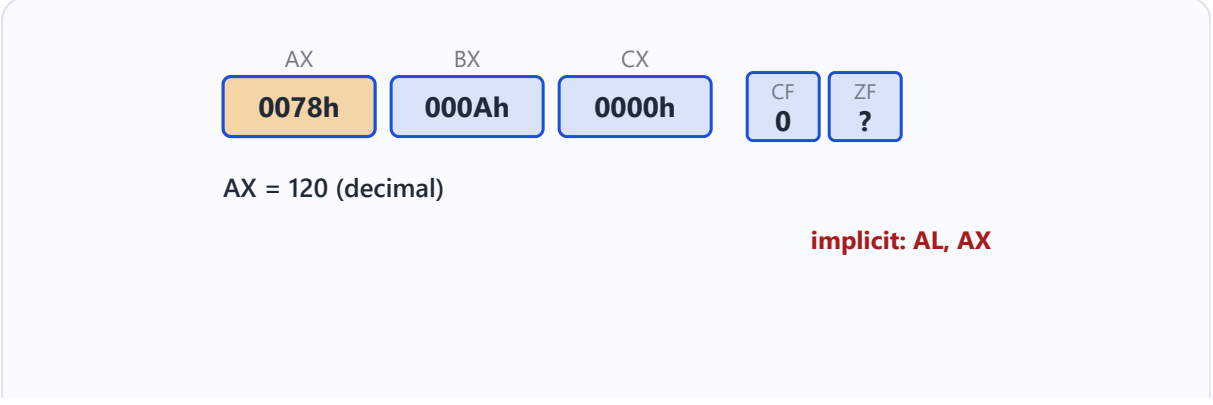


Figure 3.2.3: Three moments of the implicit-operand trace. At the top, after `MUL BL`: AX = 0078h, ZF undefined. In the middle, after `SHR AX, CL`: AX = 001Eh, CX = 0002h. At the bottom, at RET: AX = 0021h, CX = 0000h. Highlighted boxes are the registers that changed in that step.

Common mistake

Assuming MUL preserves AH. A program that keeps a counter or an intermediate result in AH loses that value when it runs `MUL BL`. The 16-bit `MUL BX` changes an even wider area: the result is written to the pair DX:AX, and the old value of DX is erased too. A register that must be preserved is moved to another register or to the stack before the MUL.

3.2.3.4 Register or memory?

A value is held either in a register or in memory. The choice between the two is the decision that affects a program's speed the most. Figure 3.2.4 shows how far the two places are from the processor.

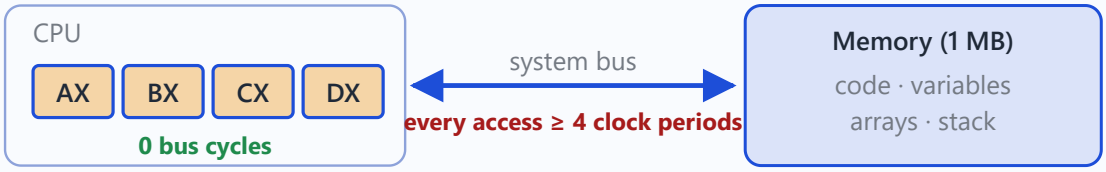


Figure 3.2.4: Registers are inside the CPU; memory is at the other end of the system bus. Accessing a register needs no bus cycle; every memory access is a bus cycle that lasts at least four clock periods.

The 8086 has **14 registers** visible to the programmer: 8 general-purpose registers (AX, BX, CX, DX, SP, BP, SI, DI), 4 segment registers (CS, DS, ES, SS), the instruction pointer IP, and the FLAGS register. All are 16 bits. Registers are small, very fast memory cells built from flip-flops. The value of a register reaches the ALU directly over the internal bus; it never goes out onto the external lines.

Memory, on the other hand, is outside the CPU and on the 8086 can grow up to 1 MB. Program code, variables, arrays, and the stack live here. To read a word from memory, the processor has to execute a **bus cycle**.

Definition 3.2.5

A **bus cycle** is the operation in which the processor reads or writes a memory or I/O location over the system bus. On the 8086 a bus cycle consists of at least four **T states** (T1–T4); each T state is one clock period. If the memory is slow, wait states (Tw) are inserted between T3 and T4.

In a read cycle, at T1 the processor puts the 20-bit address on the lines AD0–AD15 and A16–A19 and has the address captured by external latches with the ALE signal. At T2 the same lines are released for data and the RD signal becomes active. At T3 the memory puts the data on the data bus. At T4 the processor takes the data and the signals are withdrawn. With a register operand none of these four steps exists. A memory access is therefore many times longer than a register access.

A memory access has one more cost. The 8086's data bus is 16 bits wide, and in one bus cycle it can read as a whole a word that starts at an even address. If the word starts at an odd address, two separate bus cycles are needed: one for the low byte and one for the high byte. In this case the access is four clock periods longer.

This difference sets the rule of good assembly code: frequently used values are kept in registers, and memory is visited only when necessary. Keeping a loop counter in CX instead of in memory makes the loop faster. The elements of an array have to stay in memory, but the sum of the elements can be accumulated in a register during the loop and written to memory once at the end.

3.2.3.5 Instruction timing

The Intel 8086 data sheet gives how many clock periods each instruction takes. For instructions with memory operands, an **effective address** (EA) time is added to this for computing the address. The effective address is the memory operand's relative location within the segment; it can be a direct constant, a register, or the sum of a register and a constant.

Instruction	Operands	Clock periods
`MOV AX, BX`	register, register	2
`MOV AX, 5`	register, immediate	4
`MOV AX, num`	accumulator, direct memory	10
`MOV CX, [BX]`	register, memory	8 + 5 = 13
`ADD AX, BX`	register, register	3
`ADD AX, num`	register, memory	9 + 6 = 15
`INC CX`	16-bit register	2

Instruction	Operands	Clock periods
`INC counter`	memory (word)	15 + 6 = 21

Table 3.2.10: Timings of some 8086 instructions (best case, clock periods).

Address form	Example	EA time (periods)
base or index only	`[BX]`, `[SI]`	5
direct address	`num`, `[0200h]`	6
base + index	`[BX+SI]`, `[BP+DI]`	7
base + index	`[BX+DI]`, `[BP+SI]`	8
base or index + constant	`[BX+4]`	9

Table 3.2.11: Time of the effective address calculation.

The differences in the table are due to the hardware. `MOV AX, BX` is an internal transfer: 2 periods. `MOV AX, 5` takes 4 periods, because the 16-bit constant must be pulled from the instruction queue. `MOV CX, [BX]` first computes the effective address (5 periods), then executes a bus cycle. `MOV AX, num` is a special case. Loading AX from a direct address uses the short **accumulator form** (opcode `A1`), which carries the address inside the instruction and needs no EA calculation: 10 periods. Instructions that write to memory are the longest. `INC counter` first reads the word, increments it in the ALU, then writes it back; it contains two bus cycles and takes 21 periods.

The values in the table are best-case timings. A word access at an odd address adds 4 periods to each access. If the instruction queue is empty after a jump, the processor waits for the bytes to arrive. If a segment prefix is used, 2 more periods are added. The real time may therefore be longer than the table's. The ratio does not change, though: an instruction with a memory operand is several times slower than its register counterpart.

Example 3.2.5

On a 5 MHz 8086, in a loop that runs 1000 times, a counter is incremented once per iteration. How long do the increments take in total when the counter is kept in CX and when it is kept in memory?

At 5 MHz one clock period is $1 / (5 \times 10^6)$ s = 0.2 µs. If the counter is in CX, each increment with `INC CX` takes 2 periods: 1000 × 2 = 2000 periods = 2000 × 0.2 µs = **0.4 ms**. If the counter is in memory, each increment with `INC counter` takes 21 periods: 1000 × 21 = 21,000 periods = **4.2 ms**. Only the counter's location changed, and the time grew more than tenfold. Since the other instructions of the loop remain the same, the difference in total time is also 3.8 ms.

Example 3.2.6

How many clock periods does the sequence `MOV AX, 5` , `MOV BX, 3` , `ADD AX, BX` , `MOV CL, AL` take? What would change if the second line were `MOV BX, num` (direct address, word)?

An immediate value into a register takes 4, register-to-register addition 3, and an 8-bit register-to-register MOV 2 periods: 4 + 4 + 3 + 2 = **13 periods**; at 5 MHz, 2.6 µs. If the second line reads from memory, the accumulator form cannot be used because the destination is BX, not AX; the time becomes 8 + 6 = 14 periods. The total is 4 + 14 + 3 + 2 = **23 periods**, that is, 4.6 µs. A single memory operand has roughly doubled the time of the sequence.

Common mistake

Confusing instruction length with instruction time. A short instruction is not necessarily a fast one. `MUL BL` is only two bytes (`F6 E3`) but takes 70–77 clock periods; the three-byte `ADD AX, 1234h` (`05 34 12`) takes 4. The number of bytes determines the space the instruction occupies in memory and how far IP will advance. The time is determined by the instruction's internal operation and the number of bus cycles.

3.2.4 IP and the flags

3.2.4.1 How does IP advance?

IP is never written in programs, yet every instruction changes it. The **instruction pointer** (IP) holds the relative location, within the code segment, of the next instruction. The physical address is computed from CS and IP: CS × 10h + IP. This pair is called **CS:IP**.

Definition 3.2.6

After an ordinary (non-jumping) instruction executes, IP advances by the instruction's length in bytes: $IP_{new} = IP_{old}$ + instruction length. Jump, call, return, and interrupt instructions, on the other hand, write a new value into IP. IP cannot be written with `MOV` .

8086 instructions range in length from 1 to 6 bytes. To know how far IP will advance, you must know each instruction's length. In the program below the lengths are written as comments.

```
ORG 100h
MOV AX, 5     ; 3 bytes
MOV BX, 2     ; 3 bytes
INC AX        ; 1 byte
ADD AX, BX    ; 2 bytes
RET           ; 1 byte
```

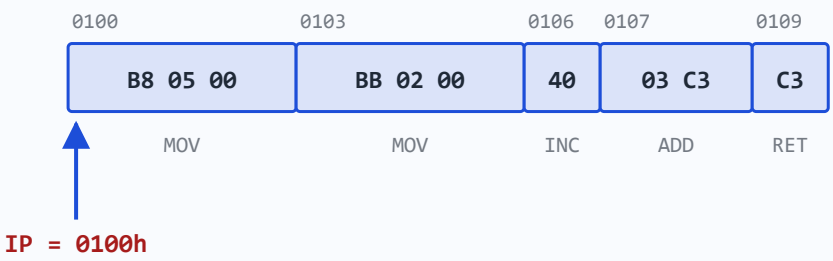
Step	Instruction address	Bytes	Instruction	IP	AX	BX	SP
0	—	—	(loaded)	0100h	0000h	0000h	FFFEh
1	0100h	`B8 05 00`	`MOV AX, 5`	0103h	0005h	0000h	FFFEh
2	0103h	`BB 02 00`	`MOV BX, 2`	0106h	0005h	0002h	FFFEh
3	0106h	`40`	`INC AX`	0107h	0006h	0002h	FFFEh
4	0107h	`03 C3`	`ADD AX, BX`	0109h	0008h	0002h	FFFEh
5	0109h	`C3`	`RET`	0000h	0008h	0002h	0000h

Table 3.2.12: IP trace (each row is the state after the instruction executes).

In step 0 the program has been loaded at offset 0100h of the segment, and IP = 0100h points to the first instruction. In step 1 the processor reads the byte `B8`, understands that a two-byte constant follows this opcode, and takes all three bytes together. IP advances to 0103h; when the instruction executes, AX = 0005h. In step 2 the same happens for BX: 3 more bytes, IP = 0106h.

The `INC AX` in step 3 is a single byte. The code of the 16-bit register increment instructions is 40h + register number; for AX, 40h. IP advances by only 1: 0107h. In step 4, `ADD AX, BX` is two bytes: the opcode 03h and a byte that defines the operands (C3h). IP = 0109h, and AX = 0006h + 0002h = 0008h.

Step 5 is not an ordinary instruction. `RET` takes the word at the top of the stack into IP. When a .COM program starts, the top of the stack (SS:FFFEh) holds the 0000h that DOS placed there. RET writes this value into IP and increases SP by 2: FFFEh + 2 = 10000h, which does not fit in 16 bits, so SP = 0000h. IP = 0000h now points not to the program but to the first byte of the PSP at the start of the segment. There sits the `INT 20h` instruction (`CD 20`), which ends the program.



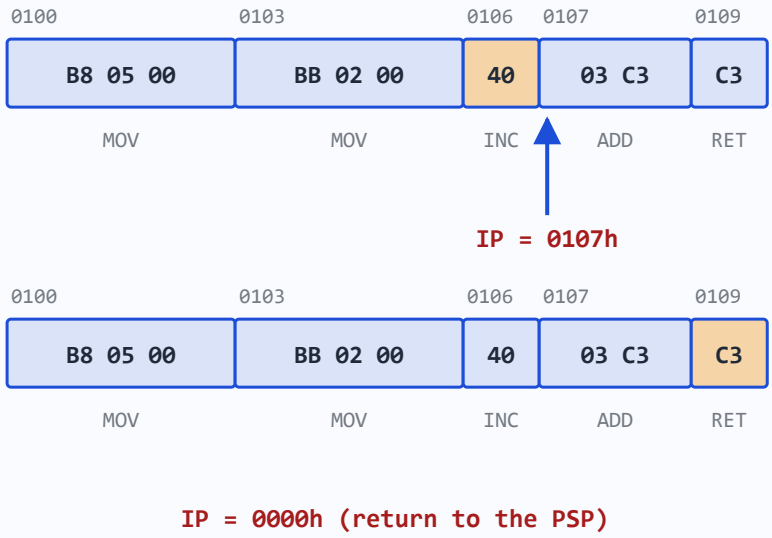


Figure 3.2.5: The instruction bytes in memory and IP. At the top the program has been loaded, IP = 0100h. In the middle three instructions have executed, IP = 0107h. At the bottom RET has taken 0000h from the stack and IP has returned to the start of the PSP. The width of each box shows the instruction's byte count.

The relationship between IP and the hardware hides a subtlety here. The 8086's bus interface unit prefetches instruction bytes and holds them in a 6-byte queue. The address the BIU reads from memory is a few bytes ahead of the address of the instruction being executed. The IP the programmer sees, however, is always the address of the next instruction **to be executed**. When a jump occurs, the queue is flushed and the BIU starts fetching from the new IP. The queue is invisible to the programmer; it shows up only in the timings.

Example 3.2.7

A program starts at 0100h and executes in order four instructions of lengths 3, 2, 1, and 4 bytes. None of the instructions is a jump. What is IP after the fourth instruction? What is the address of the third instruction?

IP advances by the length on every instruction: 0100h → 0103h → 0105h → 0106h → 010Ah. The four instructions total 3 + 2 + 1 + 4 = 10 bytes, and IP = 0100h + 0Ah = **010Ah**. The third instruction starts where the first two end, at address **0105h**.

Common mistake

Advancing IP by the number of instructions. The thought "three instructions have executed, so IP is 0103h" is wrong. IP counts bytes, not instructions. Because instruction lengths on the 8086 vary, computing IP requires knowing the length of every instruction. The lengths can be read from emu8086's memory view or from the assembler's listing.

3.2.4.2 Which instructions change the flags?

The flags register (FLAGS) carries information about the result of the last arithmetic or logic operation. The status flags are CF (carry), ZF (zero), SF (sign), OF (overflow), AF (auxiliary carry), and PF (parity). Conditional jump instructions decide by looking at these flags. Which instruction changes which flag therefore directly determines the flow of the program.

The basic rule is that the flags are written only by instructions that *compute* a result. Instructions that move data compute no result; they do not touch the flags. Table 3.2.13 groups instructions by their effect on the flags.

Group	Instructions	Effect
Data movement	`MOV`, `XCHG`, `LEA`, `PUSH`, `POP`, `CBW`	no flags change
Loops and jumps	`JMP`, `Jcc`, `LOOP`, `CALL`, `RET`	no flags change
NOT	`NOT`	no flags change
Increment, decrement	`INC`, `DEC`	ZF, SF, OF, AF, PF are rewritten; **CF is unchanged**
Addition, subtraction, comparison	`ADD`, `ADC`, `SUB`, `SBB`, `CMP`, `NEG`	all six status flags are rewritten
Logic	`AND`, `OR`, `XOR`, `TEST`	CF = OF = 0; ZF, SF, PF according to the result
Multiplication	`MUL`, `IMUL`	CF, OF defined; ZF, SF, AF, PF undefined
Direct setting	`STC`, `CLC`, `CMC`	CF only

Table 3.2.13: Effect of instructions on the status flags.

Two rows of the table are surprising. The first is NOT. NOT is a logic operation and computes a new value, but on the 8086 it changes no flag. A program that wants to test the result after NOT must follow it with `TEST AL, AL` or `CMP`. The second is INC and DEC. These two instructions do not touch CF. The aim of the design is loops working on multi-byte numbers: while the loop counter or index is changed with INC/DEC, the carry carried by ADC is not lost. To see that INC has taken a number from FFFFh to 0000h, look not at CF but at ZF.

Definition 3.2.7

The flag a conditional jump looks at shows the result of **the last instruction that wrote** that flag before it. Flag-preserving instructions in between, such as MOV, NOT, and LOOP, do not disturb this relationship; an INC, ADD, or CMP in between does.

3.2.4.3 Code trace: the flags

The following program tests the flag rules one by one. The comment on each line tells in advance what that line does to the flags; the trace confirms these predictions.

```
ORG 100h
STC              ; CF = 1
MOV AL, 0FFh     ; flags kept
INC AL           ; CF untouched
MOV CX, 0        ; ZF kept
DEC AL
ADD AL, 1        ; new CF
CLC
NOT AL           ; flags kept
RET
```

Address	Bytes	Instruction	AL	CF	ZF	SF	OF	AF	PF	Which flags were written?
0100h	`F9`	`STC`	00h	1	0	0	0	0	0	CF only
0101h	`B0 FF`	`MOV AL, 0FFh`	FFh	1	0	0	0	0	0	none
0103h	`FE C0`	`INC AL`	00h	1	1	0	0	1	1	all except CF
0105h	`B9 00 00`	`MOV CX, 0`	00h	1	1	0	0	1	1	none
0108h	`FE C8`	`DEC AL`	FFh	1	0	1	0	1	1	all except CF
010Ah	`04 01`	`ADD AL, 1`	00h	1	1	0	0	1	1	all
010Ch	`F8`	`CLC`	00h	0	1	0	0	1	1	CF only
010Dh	`F6 D0`	`NOT AL`	FFh	0	1	0	0	1	1	none

Table 3.2.14: Flag trace (each row is the state after the instruction executes).

STC and MOV. `STC` (*set carry*) sets CF directly to 1. It is one of the few instructions that change a flag without being the result of an operation. `MOV AL, 0FFh` writes 255 to AL; none of the flags changes. MOV computes no result; it only copies.

INC AL. FFh + 1 = 100h does not fit in 8 bits and AL = 00h. Had the addition been done with ADD, this carry would have been written to CF. INC does not touch CF; CF is the 1 left over from STC. Since the result is zero, ZF = 1, and since a carry passes from bit 3 to bit 4, AF = 1. The number of one bits in 00h is zero, that is, even; PF = 1. In the signed interpretation −1 + 1 = 0 is a correct result, with no overflow: OF = 0.

MOV CX, 0. Zero is written to CX. Although the result is zero, ZF is not recomputed; the 1 in ZF is left over from the previous INC. Had a `JZ` followed this line, the jump would have been taken, but the reason would be AL, not CX.

DEC AL. 00h − 1 = FFh. Since the top bit is 1, SF = 1, and since the result is not zero, ZF = 0. Because a borrow comes from bit 4, AF = 1, and because the eight one bits of FFh are an even count, PF = 1. In the unsigned interpretation, subtracting 1 from 0 causes a borrow. SUB would write this to CF; DEC does not, and CF stays 1.

ADD AL, 1. FFh + 1 = 100h, AL = 00h. This time ADD does the addition, and ADD rewrites all six status flags. CF = 1; the value in the table looks the same as in the previous row, but it is no longer the 1 left from STC; it is the carry ADD produced. ZF = 1, SF = 0, AF = 1, PF = 1, OF = 0.

CLC and NOT. `CLC` sets CF to 0. `NOT AL` inverts all the bits of AL: 00h → FFh. Although AL is no longer zero, ZF is still 1, because NOT writes no flags. To test after this line whether AL is zero, `TEST AL, AL` is needed.

Figure 3.2.6: Three moments of the flag trace. At the top, after `INC AL`: AL = 00h, ZF = 1, CF preserved. In the middle, after `DEC AL`: AL = FFh, SF = 1, CF again preserved. At the bottom, after `NOT AL`: although AL = FFh, ZF = 1 remains.

Common mistake

Not checking who set the flag before a conditional jump. Inserting a `MOV AX, [BX]` between `DEC CX` and `JZ done` does not disturb the flow; MOV writes no flags. Inserting an `ADD SI, 2` does: JZ now looks not at CX but at the sum in SI. Before a jump, the question "which instruction wrote this flag last?" should be asked.

3.2.5 The first program

3.2.5.1 The .COM program template

When a new file is opened in emu8086 with **COM template** selected, the editor opens with an empty .COM template. The skeleton of a .COM program has only three parts: `ORG 100h`, which states the starting location, the instructions, and the `RET` that ends the program. Variables, if any, come after the RET.

```
ORG 100h          ; .COM: code starts at 0100h

MOV AX, 5         ; AX = 5
MOV BX, 3         ; BX = 3
ADD AX, BX        ; AX = AX + BX = 8
MOV CL, AL        ; CL = 08h

RET               ; back to DOS

num DB 0          ; variables go after RET
```

ORG 100h. When DOS loads a .COM program, it first picks a segment and writes a header called the **PSP** (program segment prefix) into the first 256 bytes of this segment (0000h–00FFh). The PSP holds the program's command line, its end address, and other information DOS uses. The program's bytes are copied right after the PSP, starting at 0100h. `ORG 100h` tells the assembler this fact. Without the directive, the assembler would assume the first byte is at 0000h and compute all variable addresses 256 bytes too low.

Sequential execution. Instructions execute from top to bottom in order. If there is no jump instruction, after each instruction the processor moves on to the bytes right after it in memory. Blank lines and comments add nothing to this order.

RET. When the program ends, control must return to the operating system. RET does this indirectly: it takes 0000h from the stack into IP, the processor executes the `INT 20h` instruction at the start of the PSP, and DOS terminates the program. If RET is forgotten, the processor cannot know that the program has ended. It goes on decoding the bytes after the instructions, that is, the variables and the random memory contents after them, as instructions.

Where variables go. That is why variables are placed at the end of the code, after the RET. The processor that reaches RET returns and never touches the bytes of the variables. If the variables are to be placed before the code, the first instruction must be a `JMP` that jumps over the data.

The initial state. A .COM program has a single segment: code, data, and stack are in the same 64 KB. The loader therefore puts the same value into all four segment registers: CS = DS = ES = SS. emu8086 uses 0700h for this value. IP = 0100h and SP = FFFEh; the stack starts at the very end of the segment

and grows downward. At the top of the stack, at address FFFEh, sits the word 0000h that makes RET's return work. The general registers are zero.

Register	Value	Meaning
CS, DS, ES, SS	0700h	single segment; the segment starts at physical address 07000h
IP	0100h	the first instruction after the PSP
SP	FFFEh	top of the stack; [FFFEh] = 0000h
AX, BX, CX, DX, SI, DI, BP	0000h	—
FLAGS	IF = 1, others 0	interrupts enabled

Table 3.2.15: Initial state of a .COM program in emu8086.

The job of the four instructions in the template is simple: 5 is written to AX, 3 to BX, the two are added, and the low byte of the result is copied to CL. The purpose of the program is not a calculation but to show how a .COM program sits in memory and how it runs.

3.2.5.2 Code trace: a .COM program in memory

Table 3.2.16 gives the state of the template program after each instruction. The physical address column is 07000h + the relative location for segment 0700h.

Step	Instruction (address)	Physical address	AX	BX	CL	IP	SP
0	(loaded)	—	0000h	0000h	00h	0100h	FFFEh
1	`MOV AX, 5` (0100h)	07100h	0005h	0000h	00h	0103h	FFFEh
2	`MOV BX, 3` (0103h)	07103h	0005h	0003h	00h	0106h	FFFEh
3	`ADD AX, BX` (0106h)	07106h	0008h	0003h	00h	0108h	FFFEh
4	`MOV CL, AL` (0108h)	07108h	0008h	0003h	08h	010Ah	FFFEh
5	`RET` (010Ah)	0710Ah	0008h	0003h	08h	0000h	0000h
6	`INT 20h` (PSP, 0000h)	07000h	program ends				

Table 3.2.16: Trace of the .COM program (each row is the state after the instruction executes).

Steps 1–4 are ordinary instructions, and IP advances on each by the instruction length: 3, 3, 2, and 2 bytes. The addition in step 3 also writes the flags: 5 + 3 = 8, the result is nonzero (ZF = 0), the sign bit is 0 (SF = 0), and there is no carry or overflow (CF = OF = 0). Since 08h has a single 1 bit, the parity is odd and PF = 0. The `MOV CL, AL` in step 4 copies the low byte of AX into CL; CH is unchanged and CX = 0008h.

In step 5, RET reads a word from the stack. SS:SP = 0700:FFFEh, that is, the word at physical address 16FFEh is 0000h. This value is written to IP, and SP returns to 0000h by increasing 2. In step 6 the processor executes the two bytes at 0700:0000, `CD 20`, that is, `INT 20h`. DOS's terminate service removes the program from memory. The `num` byte at 010Bh was not executed in any step; it is only data.

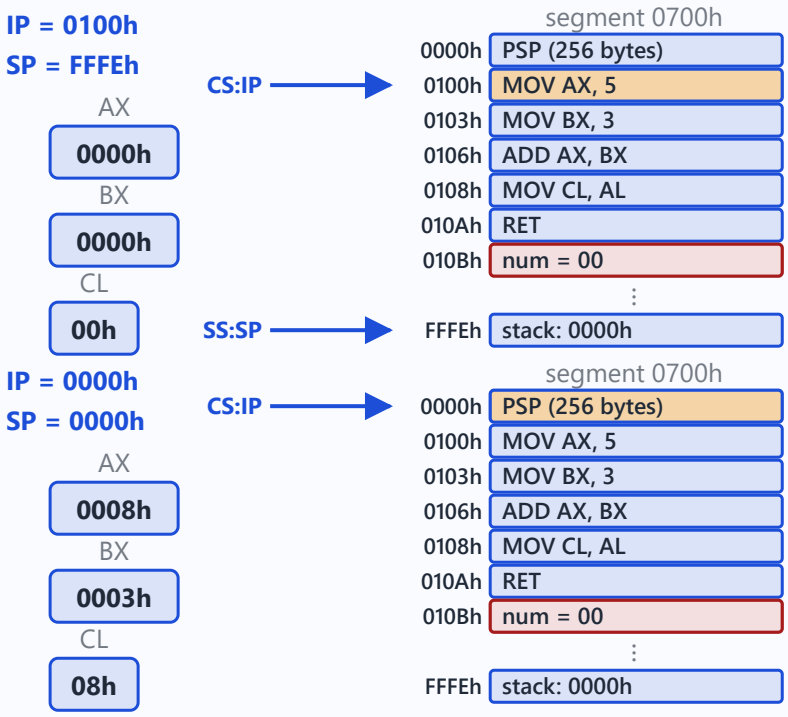


Figure 3.2.7: The layout of the .COM program within the segment. At the top the program has been loaded: PSP at 0000h–00FFh, code from 0100h, `num` at 010Bh, the top of the stack at FFFEh; CS:IP points to the first instruction and SS:SP to the top of the stack. At the bottom RET has executed and control has returned to the PSP with IP = 0000h.

This trace can also be read from the point of view of the system bus. The BIU fetches the bytes of each instruction from the code segment. The physical address used for fetching is CS × 10h + IP: 07000h + 0100h = 07100h for the first byte. The program's four arithmetic instructions read no data from memory; all their operands are registers or immediate values. A bus cycle for data happens only at RET: a word is read from the stack. RET is also a jump; the instruction queue is flushed and the BIU starts fetching again from 07000h.

3.2.5.3 The program in memory: byte by byte

The bytes the assembler produces from the template program are as in Table 3.2.17. A .COM file consists of exactly these bytes.

Relative location	Bytes	Source line	Decoding
0100h	`B8 05 00`	`MOV AX, 5`	B8 + register no. (AX = 0); constant low byte first
0103h	`BB 03 00`	`MOV BX, 3`	B8 + 3 (BX)
0106h	`03 C3`	`ADD AX, BX`	C3 = 11 000 011: register–register, AX, BX
0108h	`8A C8`	`MOV CL, AL`	C8 = 11 001 000: CL, AL
010Ah	`C3`	`RET`	single-byte instruction

Relative location	Bytes	Source line	Decoding
010Bh	`00`	`num DB 0`	data; not an instruction

Table 3.2.17: Machine code of the template program.

The first two instructions are from the "immediate value into a register" family, and the opcode contains the register number. The 16-bit register numbers of the 8086 are: AX = 0, CX = 1, DX = 2, BX = 3, SP = 4, BP = 5, SI = 6, DI = 7. Note that the order is not alphabetical; BX is number 3. For the 8-bit registers the same numbers are used in the order AL, CL, DL, BL, AH, CH, DH, BH.

The third and fourth instructions work between two registers. In these instructions the first byte defines the operation and the second byte defines the operands. The second byte is called the **ModR/M byte** and is divided into three fields: the 2-bit **MOD**, the 3-bit **REG**, and the 3-bit **R/M**. MOD = 11 means "both operands are registers." For `ADD AX, BX`, the opcode 03h has the form "REG ← REG + R/M." REG = 000 (AX) gives the destination and R/M = 011 (BX) the source. Combined, the bits give 11 000 011 = C3h. For `MOV CL, AL`, the opcode 8Ah has the form "8-bit REG ← R/M." REG = 001 (CL) and R/M = 000 (AL), and 11 001 000 = C8h results.

Example 3.2.8

What are the machine codes of `ADD DX, CX` and `MOV BL, AH`?

`ADD DX, CX`: opcode 03h (REG ← REG + R/M), MOD = 11, REG = DX = 010, R/M = CX = 001. ModR/M = 11 010 001 = D1h. The instruction is `03 D1`.

`MOV BL, AH`: opcode 8Ah, MOD = 11, REG = BL = 011, R/M = AH = 100. ModR/M = 11 011 100 = DCh. The instruction is `8A DC`. Both instructions are two bytes; IP advances by 2 on each.

Four conclusions from the table matter. First, the .COM file is exactly these 12 bytes. There is no header; DOS copies the file's bytes into memory as they are, starting at 0100h. Second, the line `ORG 100h` has no bytes at all; the directive only sets the location of the first byte.

Third, the same instruction can have more than one encoding. The register–register instructions of the 8086 have a direction bit: 03h means "REG is the destination" and 01h means "R/M is the destination." `ADD AX, BX` can also be written with 01h; then REG = BX, R/M = AX, and the code is `01 D8`. MASM chooses 03 C3; another assembler may choose 01 D8. Both do the same job in the same time. A listing may therefore show different bytes in different assemblers.

Fourth and most important, there is no marker in memory between instruction and data. The `00` byte at 010Bh is a variable, but had the processor arrived there with IP = 010Bh, it would have decoded it as an instruction. On the 8086, 00h is the opcode of an `ADD` instruction, and the byte after it would be read as ModR/M. Whether a byte is an instruction or data is determined only by the flow of the program. This is why variables are placed after the RET.

Common mistake

Writing variables right below `ORG 100h`**.** In high-level languages variables are declared at the start of the program; when this habit is carried into assembly, the program does not work. `num DB 0` lands at 0100h, and the processor executes the byte 00h, the start of an ADD, as its first instruction. The result is a random memory write and shifted instruction boundaries. Data is either put after the RET or jumped over with a JMP at the start of the code.

3.2.5.4 The emu8086 emulator window

After the source is written, the **emulate** button in emu8086 assembles the program and loads it into the emulator window. Figure 3.2.8 shows the main parts of this window in a simplified drawing.

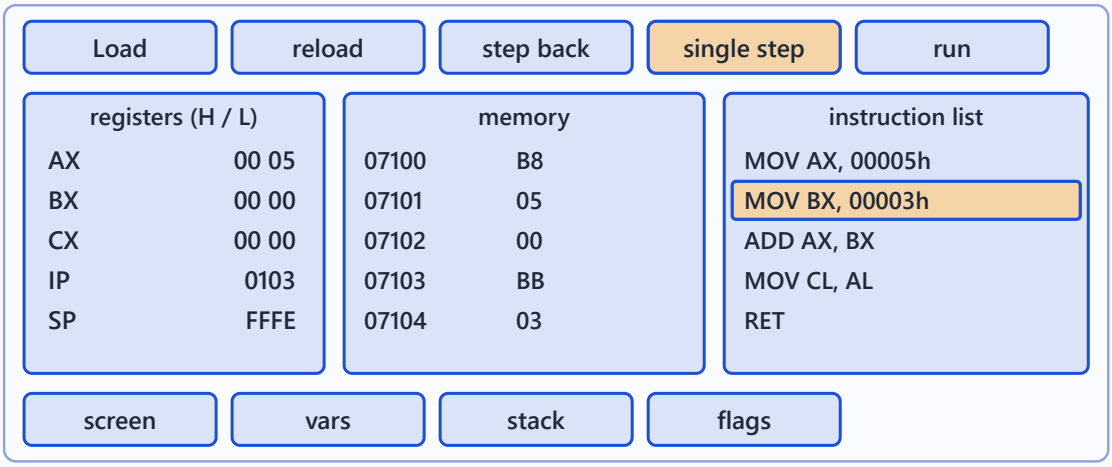


Figure 3.2.8: A simplified drawing of the emu8086 emulator window. At the top are the control buttons; on the left the registers with their high and low bytes; in the middle the memory with its physical addresses; on the right the instruction list. The drawing shows the moment after the first instruction of the template program has executed: AX = 0005h, IP = 0103h, and the highlighted line is the next instruction, `MOV BX, 00003h`.

The parts of the window are as follows:

- **Control buttons.** *single step* executes one instruction and stops; the registers and memory that changed are visible immediately. *step back* undoes the last step, which makes it easy to go back to the cause of an error. *run* runs the program to the end, and *reload* loads the program again from the start.
- **Registers.** Each register is shown with its high (H) and low (L) bytes separately: AX = 0005h appears as “00 05.” This layout makes it clear that AH and AL are the two halves of the same register.
- **Memory.** The address column shows physical addresses: 0700:0100 = 07000h + 0100h = 07100h. The program bytes start at this address: B8, 05, 00, BB, 03 …

- **Instruction list.** This is the bytes in memory translated back into instructions. The highlighted line is the **next** instruction, that is, the one CS:IP points to, not the one just executed. The list shows constants in hexadecimal with 5 digits: `MOV AX, 00005h`.
- **Extra windows.** *screen* shows the screen output, *vars* the variables, *stack* the stack, and *flags* the flags, each in a separate window.

The meaning of the highlight in the instruction list often misleads beginners. In the situation in the figure the highlighted line is `MOV BX, 00003h`, but BX is still 0000h. This is not a mistake: the highlighted instruction has not executed yet. The next single step executes it; BX = 0003h and the highlight moves to the line `ADD AX, BX`. The register values are always the result of the instructions *before* the highlighted line.

The method of examining a program in the emulator is simple. Before each single step, predict how the next instruction will change which register and flag. Take the step and compare the prediction with the value the window shows. The trace tables in these notes are exactly the written-out form of such predictions.

> **Common mistake**
>
> **Confusing the address in the memory column with the relative location.** While the emulator's memory column shows 07100h, the program uses 0100h. 07100h is the physical address; 0100h is the relative location within the segment CS = 0700h. The instruction `MOV AL, [0100h]` reads the byte at 07100h, not the one at 00100h. The relationship between the two is always segment × 10h + relative location.

3.2.6 Writing rules

The assembler recognizes numbers, characters, and names by how they are written. The rules are few, but forgetting one yields either an assembly error or a wrong value that is hard to notice.

Number bases. The base of a number is indicated by a letter added at its end. If there is no letter, the number is decimal. Table 3.2.18 gives the same value, 25, in four bases. The assembler converts all of them to the same byte, 19h; every one of the `MOV AL, …` lines becomes `B0 19`.

Base	Suffix	Written as	`MOV AL, …` code
decimal	none or `d`	`25`, `25d`	`B0 19`
hexadecimal	`h`	`19h`	`B0 19`
binary	`b`	`00011001b`	`B0 19`
octal	`o`	`31o`	`B0 19`

Table 3.2.18: Writing number bases (all 25).

The choice of base is only for readability; the processor sees no difference. When a port mask or flag pattern is written in binary, which bit is set is clear at once: `00100000b` is bit 5. An address is written in hexadecimal, because segment and relative-location calculations are done in hexadecimal. Loop counts and everyday quantities are written in decimal.

Hexadecimal numbers that begin with a letter. Hexadecimal digits include the letters A–F. When `FFh` is written, the assembler cannot tell whether it is a number or a label named `FFh`; by the rule, any string that begins with a letter is a name. That is why a `0` is put in front of hexadecimal numbers that begin with a letter: `0FFh`, `0B800h`. The leading zero does not change the value; it only declares that the string is a number. `MOV AX, 0B800h` is encoded as `B8 00 B8`.

Characters. A character in quotes stands for its ASCII code. Since `'A'` = 41h, `MOV CL, 'A'` and `MOV CL, 41h` produce the same machine code: `B1 41`. Character notation removes the burden of memorizing codes and makes clear what the program does. When a program that prints a digit on the screen writes `ADD DL, '0'`, it is immediately clear that it converts a value between 0 and 9 into the character '0'–'9'.

Signed numbers. A constant with a minus sign is stored in two's complement. `-1` becomes FFh in 8 bits and FFFFh in 16 bits: `MOV AL, -1` = `B0 FF`, `MOV AX, -1` = `B8 FF FF`. The constants that can be written into an 8-bit register lie between −128 and 255: negative values fit under the signed interpretation and large positive values under the unsigned one. `MOV AL, 256` or `MOV AL, -129` gives an error.

Case. The assembler is case-insensitive: `mov`, `MOV`, and `Mov` are the same; `var1` and `VAR1` are the same name. A consistent style is chosen for readability. In these notes, instructions and registers are written in upper case, and variable and label names in lower case.

Comments. Everything after the `;` sign is a comment. In assembly almost every line gets a comment, because the instructions themselves do not explain the purpose. A good habit is to write in the comment not “what” the instruction does but “why” it is done. `INC SI ; SI = SI + 1` gives no information; `INC SI ; next element of the array` does.

> **Common mistake**
>
> **Forgetting the suffix.** `MOV AL, 10` and `MOV AL, 10h` are different values: 10 and 16. Writing a hexadecimal value without the suffix is an error where the assembler gives no complaint but the program works wrongly. A programmer who writes the screen memory segment B800h as `MOV AX, B800` forgets both the suffix and the leading zero; the assembler takes `B800` for an undefined name.

3.2.7 Errors

3.2.7.1 Operand restrictions of MOV

MOV is the most frequently used instruction and the one with which errors are most often made. Almost all the errors come from writing an operand combination that does not exist in the 8086's instruction encoding. Table 3.2.19 gives the six most common.

Wrong	Why?	Correct
`MOV AL, BX`	sizes differ (8 ↔ 16 bits)	`MOV AL, BL` or `MOV AX, BX`
`MOV DS, 1234h`	a constant cannot be assigned to a segment register	`MOV AX, 1234h` · `MOV DS, AX`
`MOV AL, [AX]`	only BX, SI, DI, BP in addressing	`MOV BX, AX` · `MOV AL, [BX]`
`MOV AL, FFh`	`FFh` is taken for a name	`MOV AL, 0FFh`
`MOV IP, 200h`	IP cannot be written with MOV	jump with `JMP`
`MOV AL, 300`	300 > 255, does not fit in 8 bits	`MOV AX, 300`

Table 3.2.19: Common MOV errors.

Size mismatch. The two operands of MOV must be the same width, because the opcode carries a single width bit: either 8 bits or 16 bits. In `MOV AL, BX` it is unclear which half of a 16-bit value is to be taken. A programmer who wants the low byte writes `MOV AL, BL`. The reverse `MOV CX, AL` is wrong for the same reason; to widen an 8-bit value to 16 bits, the upper half is set separately: `MOV CL, AL` and `MOV CH, 0`.

A constant into a segment register. On the 8086, a MOV that writes to a segment register reads only from a general register or from memory. There is no form that loads an immediate value. Setting a segment to reach screen memory therefore takes two lines: `MOV AX, 0B800h` and `MOV DS, AX`.

Address register. On the 8086 only BX, BP, SI, and DI can be used inside square brackets. The reason is the R/M field of the ModR/M byte: in the memory modes these three bits encode only certain combinations of these four registers. There is no code for using AX as an address.

Writing to IP. IP cannot be the operand of any data instruction. The only way to change IP is the flow instructions: JMP, Jcc, CALL, RET, INT, and IRET. The assembler does not recognize `IP` as a register name; it rejects the line with a “destination cannot be a constant” or “undefined name” error.

A constant that does not fit. An 8-bit register holds at most 255 in the unsigned interpretation and −128 to 127 in the signed one. 300 needs nine bits. The assembler rejects the line; the value is written to a 16-bit register.

3.2.7.2 Other syntax errors

Wrong	Why?	Correct
`MOV CS, AX`	CS changes only with JMP, CALL, RET, INT	far jump
`MOV ES, DS`	there is no segment-to-segment MOV	`MOV AX, DS` · `MOV ES, AX`
`MOV [BX], [SI]`	two memory operands	`MOV AL, [SI]` · `MOV [BX], AL`
`INC [BX]`	size is ambiguous	`INC BYTE PTR [BX]`
`MOV AL, [BX+BP]`	two base registers cannot be combined	`[BX+SI]` or `[BP+DI]`
`ADD 5, AX`	the destination cannot be a constant	`ADD AX, 5`
`PUSH 5`	not available before the 80186	`MOV AX, 5` · `PUSH AX`
`SHL AX, 2`	on the 8086 the count can only be 1 or CL	`MOV CL, 2` · `SHL AX, CL`

Table 3.2.20: Other syntax errors.

Writing to CS. Changing CS with MOV would mean switching to another segment in the middle of the code; but IP stays the same and the processor continues from a random address. CS and IP have to change together. Far JMP, far CALL, far RET, INT, and IRET do this.

Segment to segment. There is no direct transfer between two segment registers. The value is carried in two steps through a general register. A memory-to-memory transfer is done the same way: an instruction has at most one memory operand, so the value is first taken into a register.

Ambiguous size. In the line `INC [BX]` it is not clear whether the operand is a byte or a word; no operand is a register from which the size could be inferred. The assembler therefore requires the size to be written explicitly: `INC BYTE PTR [BX]` or `INC WORD PTR [BX]`. The same rule applies to `MOV [BX], 5`. emu8086 also accepts the abbreviations `b.[bx]` and `w.[bx]`.

Address combinations. The address forms the 8086 accepts are combinations of a base register (BX or BP), an index register (SI or DI), and a constant. Two bases (`[BX+BP]`) or two indexes (`[SI+DI]`) cannot be used together; the ModR/M byte has no code for them.

Instructions of later processors. `PUSH 5` and `SHL AX, 2` arrived with the 80186. When assembling for the 8086, MASM reports an error on these lines. emu8086 accepts these forms; the assembled code runs in the emulator, but on a real 8086 the processor does not recognize these bytes. In a program written for the 8086, a constant is pushed onto the stack through a register, and for a multi-bit shift the count is put in CL.

Example 3.2.9

`wvar DW 1234h` and `bvar DB 5` are defined. Why is the line `MOV wvar, bvar` wrong, and how is it fixed?

The line has two errors at once. Both operands are memory, and their sizes differ (16 ↔ 8 bits). The value is carried through a register and the upper byte is zeroed separately:

```
MOV AL, bvar      ; AL = 05h
MOV AH, 0         ; AX = 0005h
MOV wvar, AX      ; wvar = 0005h
```

If `bvar` were a signed number, `CBW` would be used instead of the second line.

3.2.7.3 Assembly errors and logic errors

An assembly program has two kinds of errors. Knowing the difference determines how to look for the error.

Definition 3.2.8

An **assembly error** (syntax error) is an error the assembler finds while translating the source. The assembler reports the line number and the reason; the program is never produced. A **logic error**, on the other hand, is the case where the program assembles and runs but gives a wrong result. The assembler cannot know this, because each line is valid on its own.

Assembly error	Logic error
`MOV AL, BX` : size mismatch	forgetting `RET` : data is taken for instructions
`JMP strat` : undefined label	using `JA` for signed numbers
`MOV AL, 300` : a constant that does not fit	`ADD` instead of `ADC` : the carry is lost
`MOV DS, 1234h` : a constant into a segment	not loading DS in an .EXE program

Table 3.2.21: Examples of assembly and logic errors.

An assembly error is the easy kind: the assembler shows where, and the fix is usually a single line. A logic error is silent. The program runs, perhaps gives the right result for most inputs, and goes wrong only in some cases. The examples in the table are typical of this kind. `JA` is the jump of an unsigned comparison; when comparing −1 with 1, it counts −1 as FFFFh, a large number. A multi-byte addition done with `ADD` does not carry the carry from the low bytes into the high bytes. If DS is not loaded in an .EXE program, the variables are looked for in the segment where the PSP is, and entirely different bytes are read.

The way to find a logic error is single-step execution. The program is advanced instruction by instruction in the emulator. Before each step, the expected register and flag values are computed; after the step, they are compared with what the window shows. The first instruction where the values diverge is where the error lies. The error may be in this instruction itself. The instruction may also be correct; then an earlier line has left it a wrong value. emu8086's *step back* button eases this search: on seeing the wrong value, go back a few steps and trace where the value came from.

Example 3.2.10

A student wants to multiply the number in AL by 3 and get the result in AX, and writes this code: `MOV BL, 3 · MUL BL` . For AL = 7 the program gives AX = 0015h, that is, 21. When the same code runs elsewhere with AH = 12h, the student asks why AH was erased. Is this a bug?

The code is correct; the misunderstanding is in the expectation. The result of MUL is implicitly written to the whole of AX: 7 × 3 = 21 = 0015h, AH = 00h. The expectation that the 12h in AH is preserved is a logic error, but the error is not in the code; it is in the assumption that the program will keep AH. If the value in AH is needed, it is moved to another register before the MUL. Single-step execution shows this at once: after the MUL step, the AH column changes from 12 to 00.

Common mistake

Running the program as a whole and looking only at the result. Running a program that gives a wrong result with *run* and starting to rewrite it just moves the error somewhere else. First go step by step and find the first deviation. If single-stepping a long loop is tedious, compare the state before and after the loop, narrowing the error first to a region and then to a line.

3.2.8 Exercises and solutions

The six exercises in the slides are solved below with their intermediate steps. All register and flag values were checked by running them in the emulator.

Exercise 1

With AX = 0ABCDh, what are AX, AL, and BX after `MOV AH, 0` ?

In AX = ABCDh the high half is AH = ABh and the low half AL = CDh. `MOV AH, 0` writes only to AH: AH = 00h. AL is left untouched. The result is AX = **00CDh**, AL = **CDh**. BX is neither written nor implicit in the instruction; its value stays whatever it was. This instruction is the shortest way to widen an unsigned 8-bit value to 16 bits.

Exercise 2

With AX = 1234h and BL = 03h, what is AX after `MUL BL` ? What happens to the 12h in AH?

In 8-bit MUL the multiplicand is implicitly not AX but AL: AL = 34h = 52. 52 × 3 = 156 = **009Ch**. The result is implicitly written to the whole of AX; AH = 00h, AL = 9Ch. The 12h in AH is erased. Since the upper half is zero, CF = OF = 0. Someone who thinks all of AX is multiplied by 3 finds 1234h × 3 = 369Ch; that result needs `MUL BX` and is written to DX:AX.

Exercise 3

On an 8 MHz 8086, in a loop that runs 2000 times, if the counter is incremented with `INC CX` (2 periods) instead of `INC counter` (21 periods), how many clock periods and how many milliseconds are saved?

The saving per iteration is 21 − 2 = 19 clock periods. Over 2000 iterations, 2000 × 19 = **38,000** clock periods are saved. At 8 MHz one clock period is 1 / (8 × 10^6) s = 0.125 µs. The saving is 38,000 × 0.125 µs = 4750 µs = **4.75 ms**.

Exercise 4

With AL = 80h, what are AL, CF, ZF, SF, and OF after `ADD AL, 80h` ?

80h + 80h = 100h. The result needs nine bits; the low eight bits are written to AL: AL = **00h**. The ninth bit goes to CF: CF = 1. The result is zero: ZF = 1. The top bit is 0: SF = 0. In the signed interpretation 80h = −128, and −128 + (−128) = −256 does not fit in the 8-bit range (−128 … 127): OF = 1. In addition, PF = 1 because the number of one bits in 00h is even, and AF = 0 because there is no carry out of bit 3.

Exercise 5

The program starts at 0100h: what is IP after four instructions of 3 + 2 + 1 + 4 bytes have executed?

Since there are no jumps, IP advances by the length on every instruction: 0100h → 0103h → 0105h → 0106h → 010Ah. The total is 10 bytes = 0Ah, so IP = 0100h + 0Ah = **010Ah**.

Exercise 6

Why are `MOV CX, AL` and `MOV ES, 0B800h` wrong? What are their correct forms?

`MOV CX, AL` : the operands are of different sizes (16 ↔ 8 bits). If the aim is only to copy the low byte, write `MOV CL, AL` . If the value of AL is wanted in the whole of CX as an unsigned number, the upper half is also zeroed: `MOV CL, AL · MOV CH, 0` .

`MOV ES, 0B800h` : an immediate value cannot be loaded into a segment register. The value is carried through a general register: `MOV AX, 0B800h · MOV ES, AX` .

3.2.9 Summary

- **Assembly language** is a low-level language that maps one-to-one onto machine code. Each instruction line is translated into a single machine instruction: `MOV AX, 5` = `B8 05 00`. It produces small and fast code and accesses the hardware directly, but it is not portable.
- The **assembler** translates mnemonics into opcodes and names into addresses. emu8086 combines an editor, an assembler, and an emulator; it shows the registers, memory, and flags step by step.
- The line structure is `[label:] instruction destination, source ; comment`. The destination is on the left; the source does not change. The assembler is case-insensitive; comments do not enter the machine code.
- **Directives** (ORG, EQU, END) produce no machine instruction; they give information to the assembler. DB and DW produce only data bytes. A macro call stands for several instructions.
- A name begins with a letter or an underscore; reserved words cannot be names. A code label ends with a colon and denotes an address; a variable name carries an address and a type together.
- Operand types: **register** (fastest), **immediate value** (source only), and **memory** (needs a bus cycle, at most one per instruction). The two operands must be the same size.
- **Implicit operands** are not written in the instruction but are changed: MUL writes AX, DIV writes AL and AH; LOOP decrements CX; CBW fills AH; PUSH changes SP.
- A memory access is a bus cycle of at least four T states. Instructions with memory operands, with the effective address time, take several times longer than their register counterparts: `INC CX` takes 2 and `INC counter` 21 periods.
- **IP** is the relative location of the next instruction and advances by the instruction length. Only flow instructions write a new value into IP.
- MOV, XCHG, LEA, PUSH, POP, NOT, and LOOP do not change the flags. INC and DEC do not touch CF. ADD, SUB, and CMP rewrite all six status flags. After MUL, ZF, SF, AF, and PF are undefined.
- A .COM program starts at 0100h with `ORG 100h` ; the first 256 bytes are the PSP. Initially CS = DS = ES = SS, IP = 0100h, SP = FFFEh. RET returns, through the 0000h on the stack, to the `INT 20h` in the PSP. Data goes after the RET.
- There is no marker in memory between instruction and data; the flow of the program determines the meaning of a byte. The same instruction can have more than one valid encoding.
- The assembler finds **assembly errors**; a **logic error** can be found only by single-stepping and comparing the expected values with those in the emulator.

3.2.10 Key terms

assembly language mnemonic instruction opcode operand implicit operand immediate value comment label variable directive constant (EQU) location counter ($) macro assembler emulator .COM program program segment prefix (PSP) accumulator counter register bus cycle T state clock cycle effective address (EA) instruction pointer (IP) flags register (FLAGS) ModR/M byte little-endian single-step syntax error / logic error

3.2.11 Test Yourself

1. The instructions `STC`, `MOV AX, 0FFFFh`, `INC AX` run in order. What are AX, CF, and ZF at the end?

   Answer

   FFFFh + 1 = 10000h does not fit in 16 bits, and AX = 0000h. The result is zero: ZF = 1. INC does not touch CF; CF is the 1 left from STC. Since MOV also changes no flags, the CF = 1 was never disturbed. Had the same addition been done with `ADD AX, 1`, CF would also be 1, but this time because of the carry out.

2. In a program that begins with `ORG 100h` there are, in order, the lines `MOV AL, 5`, `MOV BX, 1234h`, `INC BX`, `RET`, and `data DB 9`. What are the bytes of the program, and what is the relative location of `data`?

   Answer

   `MOV AL, 5` = `B0 05` (0100h), `MOV BX, 1234h` = `BB 34 12` (0102h), `INC BX` = `43` (0105h; 40h + 3), `RET` = `C3` (0106h), `data` = `09` (0107h). The instructions are 2 + 3 + 1 + 1 = 7 bytes; `data` is at 0100h + 7 = **0107h**. The program is 8 bytes in total, and at the end BX = 1235h.

3. In the best case, how many clock periods and how many microseconds do the instructions `MOV AX, [BX+SI]` and `ADD [BX], AX` take on a 5 MHz 8086? (A read from memory into a register takes 8 + EA, and an addition into memory 16 + EA periods.)

   Answer

   For `[BX+SI]`, EA = 7 periods: MOV takes 8 + 7 = 15 periods. For `[BX]`, EA = 5 periods: ADD takes 16 + 5 = 21 periods. The total is 36 periods; at 5 MHz, 36 × 0.2 µs = **7.2 µs**. The reason ADD takes long is that it both reads and writes memory: two bus cycles.

4. With AX = 25, `MOV BL, 4` and `DIV BL` run. What are AL, AH, and AX?

> **Answer**
>
> The dividend is implicitly AX = 25. 25 ÷ 4 = 6, remainder 1. The quotient is written to AL and the remainder to AH: AL = 06h, AH = 01h, AX = **0106h**. Someone who reads AX as a single number sees 262; after DIV the two halves of AX carry separate meanings.

5. How many lines of the following program produce bytes, and how many bytes is the program? `ORG 100h` · `k EQU 5` · `MOV AX, 5` · `MOV BX, k` · `MOV AL, num` · `RET` · `num DB 0` · `END`. What are the bytes of the line `MOV AL, num`?

> **Answer**
>
> ORG, EQU, and END produce no bytes. Three MOVs of 3 bytes each, RET of 1 byte, DB of 1 byte: five lines, **11 bytes**. `num` is at 0100h + 10 = 010Ah. `MOV AL, num` is the accumulator form that reads from a direct address into AL: `A0 0A 01`.

6. `MOV AL, 0F0h`, `ADD AL, 20h`, `MOV BL, AL`, `NOT BL` run. What are AL, BL, CF, and ZF at the end?

> **Answer**
>
> F0h + 20h = 110h: AL = 10h, CF = 1, ZF = 0. With MOV, BL = 10h; the flags do not change. NOT BL = EFh; NOT changes no flags either. The result is AL = 10h, BL = EFh, CF = 1, ZF = 0. The flags are as ADD left them.

7. In emu8086, single step is pressed three times in the template program (`MOV AX, 5`, `MOV BX, 3`, `ADD AX, BX`, `MOV CL, AL`, `RET`). Which line is highlighted in the instruction list, and what are IP and AX? At which physical address does the first byte of this instruction appear in the memory column?

> **Answer**
>
> Three instructions have executed: AX = 0008h, BX = 0003h. IP = 0100h + 3 + 3 + 2 = **0108h**. The highlighted line is the next instruction, `MOV CL, AL`; CL is still 00h. The physical address is 07000h + 0108h = **07108h**; the byte `8A` appears there.

8. What are the machine codes of `MOV DX, 0Ah` and `INC CX`? If the two are consecutive and start at 0103h, at which address does the next instruction start?

> **Answer**
>
> The register number of DX is 2. `MOV DX, 0Ah` = B8h + 2 = BAh with the constant low byte first: `BA 0A 00` (3 bytes). `INC CX` = 40h + 1 = `41` (1 byte). The two instructions are 4 bytes; the next instruction starts at 0103h + 4 = **0107h**.

9. A program sets up a shift count with `MOV CL, 3`, then sets up a loop counter with `MOV CX, 10`, and at the end writes `SHL AX, CL`. Does the program assemble? By how many bits does AX shift?

   Answer

   The program assembles; every line is valid. But CL is the lower half of CX. `MOV CX, 10` makes CL 0Ah, and the shift is **10 bits**, not 3. This is a logic error. The shift count must be written to CL right before the shift.

3.2.12 Further reading

- The emu8086 documentation, *8086 assembler tutorial for beginners*: number systems, memory access, variables, and the instruction set reference; it is the source closest to emu8086 syntax.
- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapters 4 and 6: data movement instructions, ModR/M encoding, and the structure of .COM and .EXE programs.
- M. A. Mazidi, J. G. Mazidi, *The 80x86 IBM PC and Compatible Computers*, Chapters 1–2: the structure of an assembly program, directives, sample programs, and working with a debugger.
- Intel, *8086 16-Bit HMOS Microprocessor* data sheet: the T states of the bus cycle and the table of instruction timings.

3.2.13 References

1. The emu8086 documentation, *8086 assembler tutorial for beginners* (part 1: numbering systems, part 2: memory access).
2. Intel, *8086 16-Bit HMOS Microprocessor* data sheet.
3. B. B. Brey, *The Intel Microprocessors*, 8th edition, Pearson.
4. M. A. Mazidi, J. G. Mazidi, *The 80x86 IBM PC and Compatible Computers*, Pearson.

PART II

Memory and System Services

Chapters 4–5

PART II · CHAPTER 4

Memory Access and Variables

The 8086 reaches memory by combining a segment value with an offset. This chapter presents the addressing modes built from the BX, SI, DI and BP registers, effective and physical address calculation, specifying data size with byte ptr and word ptr, and the valid operand combinations of MOV and XCHG. The variables part covers DB, DW and DD definitions, memory layout, ORG 100h and .COM programs, arrays with DUP, taking addresses with LEA and OFFSET, and EQU constants. Sample programs that move, add and compare variables complete the chapter.

NOTES IN THIS CHAPTER

4.1 Memory Access

4.2 Variables

PART II · CHAPTER 4.1

Memory Access

Most of what an 8086 program does is to bring a value from memory into a register or to put a value in a register back into memory. Addition, comparison, printing text on the screen, and walking through an array cannot be done without these moves. Which byte in memory the processor reaches is decided by the square brackets inside the instruction. These notes describe what may be written inside the square brackets, that is, the addressing modes. From the base and index registers and a displacement, an effective address is computed. The effective address combines with a segment register and becomes the 20-bit physical address put on the address bus. Which segment is chosen can send the same relative location to an entirely different byte. The notes do this calculation in both directions and show how a segment override changes the default segment. Next come the size of an access (byte or word), byte order, and the rules of the MOV and XCHG instructions. Example programs are traced step by step at the register and memory level; the last example writes a letter directly into the memory of the color text screen.

LEARNING OUTCOMES

Build a valid memory operand

Form the 24 memory addressing forms of the 8086 from BX, BP, SI, DI and a displacement; tell invalid combinations apart and explain why they are invalid.

Name the addressing modes

Recognize the immediate, register, direct, register indirect, based, indexed, and based-indexed modes, as well as the string, I/O, and relative modes, with examples; choose the mode that suits a data structure.

Compute addresses

Compute the effective address with 16-bit wraparound; go from segment:offset to the physical address and from the physical address to the possible segment:offset pairs; find the memory capacity from the number of address lines.

Choose the segment correctly
Apply the default-segment rule (SS if BP is present, otherwise DS); explain the effect of a segment override on the machine code and the trap of accessing data through BP.

Size and byte order
Specify the access size with byte ptr and word ptr; read the layout of words in memory in little-endian order; convert an index into a byte offset in a word array.

Trace programs with MOV and XCHG
Tell valid operand combinations apart; trace copy, swap, and video-memory programs step by step at the register, memory, and IP level.

4.1.1 Memory operands

4.1.1.1 Square brackets and address registers

The 8086 has fourteen registers, and only eight of them are used for general work. A program's data, on the other hand, mostly sits in memory: variables, arrays, tables, the contents of the screen, the stack. To work with this data the processor goes to memory every time, reads the byte or word it wants, and writes the result back to memory when it is done. So the first step in learning an instruction set is learning how an instruction says which memory cell it means.

In assembly language a memory cell is denoted with **square brackets**. The instruction `MOV AX, BX` copies the number in BX into AX. The instruction `MOV AX, [BX]`, on the other hand, uses the number in BX as an address; it reads the word at that address from memory and puts it into AX. The square brackets mean “the place pointed to by what is inside.” Compared with pointer notation in C, `BX` is the pointer itself and `[BX]` is like `*p`.

On the 8086 not every register can be written inside square brackets. Only four registers can hold a memory address: **BX**, **BP**, **SI**, and **DI**. Of these, BX and BP are called **base registers** and SI and DI **index registers**. A base register usually holds the start address of a block of data, for example an array or a record. An index register counts the position within this block. A constant number written in the instruction can also be added to these four registers. The constant is called the **displacement**.

The source of this restriction is the hardware. In the instruction's machine code, only a three-bit field tells how the memory operand is to be computed. Three bits mean eight options, and the designers of the 8086 allotted these eight options to the most useful combinations of BX, BP, SI, and DI. AX, CX, DX, and SP are not on this list. That is why, however natural a notation such as `MOV AL, [CX]` may look, the assembler rejects it; the processor has no circuit to produce such an address.

Definition 4.1.1

The rule that determines where an instruction's operand is found and how its address is computed is called the **addressing mode**. On the 8086 the address of a memory operand is written inside square brackets. The address is the sum of three parts: a base register (BX or BP), an index register (SI or DI), and a displacement. Each of these three parts is optional, but at least one must be present.

4.1.1.2 Twenty-four combinations

All the permitted combinations of the base, index, and displacement parts are collected in Table 4.1.1. The rows of the table show the register part and the columns the size of the displacement. **d8** means an 8-bit and **d16** a 16-bit displacement.

Register(s) only	+ d8	+ d16
`[BX + SI]`	`[BX + SI + d8]`	`[BX + SI + d16]`
`[BX + DI]`	`[BX + DI + d8]`	`[BX + DI + d16]`
`[BP + SI]`	`[BP + SI + d8]`	`[BP + SI + d16]`
`[BP + DI]`	`[BP + DI + d8]`	`[BP + DI + d16]`
`[SI]`	`[SI + d8]`	`[SI + d16]`
`[DI]`	`[DI + d8]`	`[DI + d16]`
`d16` (direct address)	`[BP + d8]`	`[BP + d16]`
`[BX]`	`[BX + d8]`	`[BX + d16]`

Table 4.1.1: The 24 memory addressing forms of the 8086. The row determines the register part and the column the size of the displacement.

The eight rows of the table correspond to the eight values of the three-bit field in the machine code. This field is called **R/M** and sits in the lowest three bits of the instruction's second byte, the **ModR/M byte**. The highest two bits of the same byte are the **MOD** field; it selects which column of the table is used. MOD = 00 means no displacement, MOD = 01 means d8 is present, and MOD = 10 means d16 is present. MOD = 11 shows that the operand is not memory but a register. Three columns times eight rows make 24; all the memory addressing forms of the 8086 are in these 24 cells.

One row looks odd. The first cell of the seventh row does not have `[BP]`; in its place is `d16`, that is, only a constant address. The designers allotted the combination R/M = 110 and MOD = 00 to **direct addressing**: the address of an instruction such as `MOV AL, [1234h]` is written with this code. No code was left for `[BP]` without a displacement. If the programmer writes `MOV AL, [BP]`, the assembler quietly encodes it as `[BP + 0]`, that is, with MOD = 01 and d8 = 00. The instruction grows by a byte but works correctly. Table 4.1.2 shows the machine code the assembler produces for a few instructions.

Instruction	Machine code	MOD	R/M	Length
`MOV AL, [BX+SI]`	`8A 00`	00	000	2 bytes
`MOV AL, [BX+DI]`	`8A 01`	00	001	2 bytes
`MOV AL, [BP+SI]`	`8A 02`	00	010	2 bytes
`MOV AL, [BP+DI]`	`8A 03`	00	011	2 bytes
`MOV AL, [SI]`	`8A 04`	00	100	2 bytes
`MOV AL, [DI]`	`8A 05`	00	101	2 bytes
`MOV CL, [1234h]`	`8A 0E 34 12`	00	110	4 bytes
`MOV AL, [BX]`	`8A 07`	00	111	2 bytes
`MOV AL, [BP]`	`8A 46 00`	01	110	3 bytes
`MOV AL, [BX+5]`	`8A 47 05`	01	111	3 bytes
`MOV AL, [BX+128]`	`8A 87 80 00`	10	111	4 bytes

Table 4.1.2: Machine code of the same `MOV AL, …` instruction with different memory operands. The second byte is the ModR/M byte.

The opcode `8A` in the table means “move a byte from memory or a register to an 8-bit register.” The middle three bits of the ModR/M byte (the REG field) select the destination register: 000 for AL, 001 for CL. For example, the bits of `0E` = 00 001 110 read as MOD = 00, REG = 001 (CL), R/M = 110 (direct address). `MOV CL, [1234h]` is therefore four bytes: the opcode, the ModR/M, and a two-byte address. The address is written low byte first, like all multi-byte numbers on the 8086: `34 12`. If an instruction with a direct address loads into AL or AX, the assembler chooses a shorter special form: `MOV AL, [1234h]` is encoded in three bytes as `A0 34 12`.

4.1.1.3 The combination rule

The structure of the table can be reduced to a few simple rules. Once these rules are memorized, you can tell whether a notation is valid without looking at the table. Figure 4.1.1 shows the rule as a three-column choice: at most one item is taken from each column.

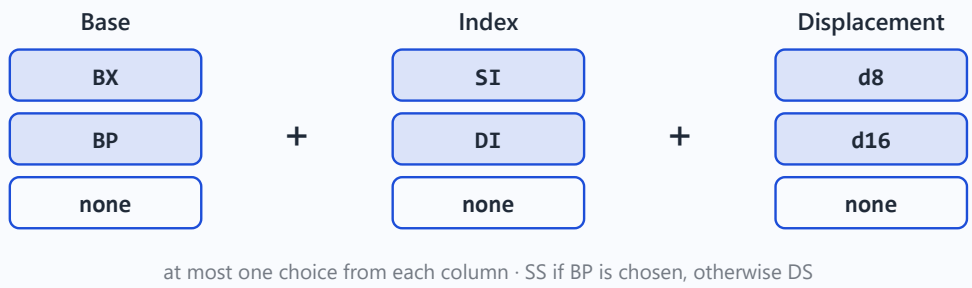


Figure 4.1.1: A valid memory operand is built with at most one choice from each of the columns base (BX, BP, or none), index (SI, DI, or none), and displacement (d8, d16, or none).

Definition 4.1.2

The combination rule. A memory operand contains at most one base register (BX or BP) and at most one index register (SI or DI). At most one displacement is added to them. BX and BP, or SI and DI, cannot be used together in the same address. The order in which the parts are written does not matter.

The default segment rule. If BP is in the address, the access goes to the SS (stack) segment; otherwise to the DS (data) segment.

The essence of the segment concept is this: the 8086 sees memory through 64 KB windows, and a segment register tells on every access which window to use. DS is used for data and SS for the stack. Since BP was designed to access data on the stack, addresses built with it automatically go to the SS window. The four examples in the slides apply these rules:

- `[BX+5]` : base + d8. No BP, segment DS.
- `[BX+SI]` : base + index. Segment DS.
- `[DI+BX-4]` : the assembler reads this as `[BX+DI-4]` . The order does not matter; the machine code is `8A 41 FC` . Here `FC` is the 8-bit two's complement of −4. Segment DS.
- `[BP+SI+300]` : base + index + d16. Since 300 does not fit in the range −128…127, a 16-bit displacement is needed. Since BP is in the address, the segment is SS.

4.1.1.4 Displacement

The displacement is a signed constant added to the address calculation and held inside the instruction's machine code. Its 8-bit form (**d8**) carries values between −128 and +127, and its 16-bit form (**d16**) values between −32,768 and +32,767. Because in 16-bit addition −1 and FFFFh give the same result, d16 actually covers the whole range 0000h–FFFFh. Example values: for d8, 22, 55h, −1; for d16, 300, 5517h, −259.

The source of the displacement is either a number or the **relative location** of a variable. When `[BX + arr]` is written, the assembler puts the address of the name `arr` within the segment, that is, its relative location, as the displacement. If the notation contains more than one constant, the assembler adds them into a single number at assembly time: `[BX + 2 + 3]` is the same instruction as `[BX + 5]` . The constant can be written inside or outside the square brackets; the notations `[BX+5]` , `5[BX]` , and `[BX]+5` all produce the bytes `8A 47 05` . The C-like notation `arr[SI]` comes from this flexibility. The constant can also be negative: the instruction `[SI-1]` accesses the byte one before the one SI points to and is encoded as `8A 44 FF` .

The assembler decides which of the two forms to choose by looking at the value. If the value is in the range −128…127, the short form (d8) is chosen and the instruction is a byte shorter. Otherwise the long form (d16) is needed. `[BX+127]` is three bytes (`8A 47 7F`) and `[BX+128]` is four bytes (`8A 87`

`80 00`). The number 128 fits in 8 bits but would be read as −128 when signed; that is why the 16-bit form is used. Since the relative location of variable names is usually greater than 127, notations such as `arr[SI]` almost always use d16.

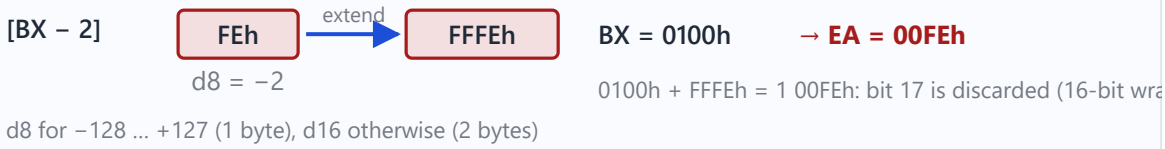


Figure 4.1.2: Before joining the addition, an 8-bit displacement is extended to 16 bits according to its sign: FFh becomes FFFFh, not 00FFh, and acts like −1.

The processor performs **sign extension** on an 8-bit displacement before adding it to 16-bit registers. If the highest bit is 1, the upper byte is filled with FFh, and if 0, with 00h (Figure 4.1.2). `FC` thus becomes FFFCh. Adding FFFCh to a number is the same as subtracting 4 once the carry that overflows the 16-bit limit is discarded.

This last point is a general property of address calculation. The effective address is always computed in 16 bits, and if the result does not fit in 16 bits, the overflowing bit is discarded; the address **wraps around**. With BX = FFFFh, the access `[BX+2]` goes not to 10001h but to 0001h. The 8086 does not carry an address overflowing the end of a segment into the next segment; it returns it to the start of the same segment.

Example 4.1.1

With SI = 0010h, from which relative location does the instruction `MOV AL, [SI+0FFF8h]` read?

The addition gives 0010h + FFF8h = 1 0008h. The 17th bit of the result is discarded and the effective address is 0008h. FFF8h is the 16-bit two's complement of −8; the instruction actually goes to the same place as `[SI-8]`. The assembler notices this too and chooses the short form: both notations produce the bytes `8A 44 F8`. The processor turns the byte `F8` back into FFF8h by sign extension.

4.1.1.5 Valid and invalid notations

The combination rule serves for quickly testing the validity of a notation. Table 4.1.3 collects the examples from the slides together with the reasons the invalid ones are invalid. The assembler reports an error for every invalid line.

Notation	Status	Explanation
`[BX]`	valid	register indirect, DS
`[1234h]`	valid	direct address, DS

Notation	Status	Explanation
`[SI+10]`	valid	indexed, d8, DS
`[BP+DI-2]`	valid	based-indexed, d8, SS
`ES:[BX+SI]`	valid	with a segment override, ES
`[AX]`, `[CX]`, `[DX]`	invalid	these registers cannot be used in address calculation
`[BL]`, `[BH]`	invalid	an 8-bit register cannot hold a 16-bit relative location
`[BX+BP]`	invalid	two base registers
`[SI+DI]`	invalid	two index registers
`[SP]`, `[BX*2]`	invalid	not on the 8086; present in 32-bit addressing on the 80386 and later

Table 4.1.3: Valid and invalid memory operands.

The last row often causes confusion. On today's x86 processors, notations such as `[ESP+4]` or `[EBX+ESI*4]` are valid. They are part of the 32-bit addressing that came with the 80386. In the 8086's 16-bit ModR/M table there is neither SP nor a scale factor. To reach a value on the stack, SP is first copied to BP, and then `[BP+n]` is used.

Common mistake

Falling into the forbidden combination when wanting to use the sum of two registers as an address. `MOV AX, [SI+DI]` is invalid. If one of the two index registers is first copied to a base register, the job is done: `MOV BX, SI` followed by `MOV AX, [BX+DI]`. The same route works for `[BX+BP]`: one of BP or BX is transferred to SI or DI. In the copy, which register is chosen can also change the default segment; an address with BP goes to SS.

4.1.2 Types of addressing modes

4.1.2.1 Where does the operand live?

The twenty-four combinations cover all the cases where the operand is in memory. But an instruction's operand does not have to be in memory. A constant can be written directly into the instruction, or the operand can sit in a register. Books therefore name the addressing modes according to the question “where is the operand?” Table 4.1.4 gives the classification in the slides.

Mode	Example	Where is the operand?
Immediate	`MOV AX, 5`	inside the instruction (constant)
Register	`MOV AX, BX`	in a register
Direct	`MOV AX, [1000h]`	DS:1000h
Register indirect	`MOV AX, [BX]`	DS:BX
Based	`MOV AX, [BP+4]`	SS:BP+4

Mode	Example	Where is the operand?
Indexed	`MOV AL, [SI+3]`	DS:SI+3
Based-indexed	`MOV AL, [BX+DI]`	DS:BX+DI
Based-indexed + displacement	`MOV AL, [BX+SI+8]`	DS:BX+SI+8

Table 4.1.4: Types of addressing modes for data operands. The last column shows that in modes without BP the default segment is DS.

In **immediate mode** the operand is the instruction itself. The machine code of `MOV AX, 5` is `B8 05 00`; the number 5 sits in the last two bytes of the instruction. The processor has already read this value while fetching the instruction and does not go to memory a second time. An immediate value can only be a source. Since nothing can be written into a constant, `MOV 5, AX` is meaningless.

In **register mode** the operand is inside the processor. The instruction `MOV AX, BX` completes inside the execution unit without ever going out onto the data bus. It is therefore the fastest mode; on the 8086 it takes 2 clock periods. It is selected by writing MOD = 11 in the ModR/M byte: `MOV AX, BX` = `8B C3`, that is, the bits `C3` = 11 000 011.

All the remaining modes go to memory. In these the processor first computes an **effective address** (EA). The effective address is the operand's location within the segment, that is, a relative location. Then this relative location is combined with the segment register:

> physical address = segment × 10h + EA

The names of the modes describe how the effective address is built. In **direct mode** the EA is the 16-bit constant inside the instruction. In **register indirect mode** the EA is the contents of a register (BX, BP, SI, or DI). In **based mode** the EA is the sum of BX or BP and a displacement. In **indexed mode** SI or DI is used instead of the base. In **based-indexed mode** a base and an index are added, and an optional displacement is added to that. Naming differs slightly among books. Some call `[BX+4]` “register relative,” others “based.” But the calculation is the same everywhere.

Definition 4.1.3

The **effective address** (EA) is, for a memory operand, the 16-bit sum of the base register, index register, and displacement in square brackets. If the sum exceeds FFFFh, the overflowing bit is discarded. The EA is the relative location within the segment. The **physical address** is the 20-bit address obtained by adding the EA to 16 times the chosen segment register.

4.1.2.2 String, I/O, and relative modes

There are three more modes. They do not use the square-bracket notation. The operand is either not written in the instruction at all, or it lives in an address space separate from memory, or the address is computed relative to IP.

In **string mode** the operands are implicit. The instruction `MOVSB` moves a byte from address DS:SI to address ES:DI with no operand written beside it. The source is always DS:SI and the destination always ES:DI. After each operation SI and DI advance automatically. They increase if the direction flag DF = 0 and decrease if DF = 1. The step is 1 for a byte operation and 2 for a word operation. The details of string instructions are a separate topic. What matters here is that in these instructions the address calculation has the same structure as register indirect mode, and that the destination is looked for in the ES segment, not DS.

```
MOVSB           ; ES:[DI] ← DS:[SI], then SI and DI advance by one
```

In **I/O mode** the operand is an input/output **port**. The 8086's ports live in their own 64 K address space, separate from memory. When the processor accesses a port, it puts the port number on the address bus and makes the M/IO̅ line 0. Thus I/O devices, not memory chips, respond. Data passes only through AL or AX. If the port number is in the range 0–255, it can be written inside the instruction. Larger numbers are put in DX.

```
IN  AL, 45h      ; read a byte from port 45h
OUT DX, AX       ; write AX to the port in DX
```

In **relative mode** the instruction holds not the target itself but a signed displacement to be added to IP. Jump instructions use this mode. The new IP is the sum of the address of the instruction right after the jump and the displacement. In short jumps the displacement is 8 bits and reaches the range −128…+127 bytes. For example, a `JNZ again` at the end of a loop can be encoded as `75 FA`. `FA` is the 8-bit two's complement of −6. Since the instruction is two bytes, the jump goes 4 bytes back from the start of the instruction. Relative addressing has a big advantage: even if the program is moved elsewhere in memory, the displacements do not change.

```
JNZ again       ; 75 FA: IP ← address of the next instruction − 6
```

Common mistake

Carrying the notation of other processors over to the 8086. Instruction names are written joined: `MOVSB` is correct, `MOVS B` is not. In the `OUT` instruction the port comes first and then the data register: `OUT 50h, AL`. The notation `IN A, 45` belongs to the 8085 processor. On

the 8086 the accumulator is called AL or AX, and hexadecimal numbers are written with the `h` suffix.

4.1.2.3 Code trace: six addressing modes

The small COM program below reads values from the same six-byte table with different modes. The result of each step is given in the trace table.

```
ORG 100h
MOV BX, OFFSET table
MOV SI, 2
MOV AH, 7          ; immediate
MOV AL, [table+1]  ; direct
MOV CL, [BX]       ; indirect
MOV CH, [BX+4]     ; based
MOV DL, [BX+SI]    ; base+index
MOV DH, [BX+SI+3]  ; + d8
RET
table DB 11h,22h,33h,44h,55h,66h
```

The line `ORG 100h` tells the assembler that the program will start at relative location 0100h. In COM programs code, data, and stack are in the same 64 KB segment. emu8086 loads this segment at paragraph 0700h: CS = DS = ES = SS = 0700h. Adding up the instruction lengths, `RET` falls at 0115h and `table` at 0116h. The six bytes of the table are at relative locations 0116h–011Bh; their physical addresses are 07116h–0711Bh.

The `OFFSET` operator in the line `MOV BX, OFFSET table` gives not the contents of the variable but its relative location. This value is known at assembly time and is written into the instruction as an immediate value: `BB 16 01`. As a result BX = 0116h. BX is now a pointer to the start of the table. `MOV SI, 2` puts 2 into the index register.

Instruction	Machine code	Mod	EA	Physical	Result
`MOV AH, 7`	`B4 07`	immediate	—	—	AH = 07h
`MOV AL, [tablo+1]`	`A0 17 01`	direct	0117h	07117h	AL = 22h, AX = 0722h
`MOV CL, [BX]`	`8A 0F`	register indirect	0116h	07116h	CL = 11h
`MOV CH, [BX+4]`	`8A 6F 04`	based	011Ah	0711Ah	CH = 55h, CX = 5511h
`MOV DL, [BX+SI]`	`8A 10`	based-indexed	0118h	07118h	DL = 33h
`MOV DH, [BX+SI+3]`	`8A 70 03`	based-indexed + d8	011Bh	0711Bh	DH = 66h, DX = 6633h

Table 4.1.5: Trace of the six addressing modes (DS = 0700h, BX = 0116h, SI = 0002h; table = 11 22 33 44 55 66).

Read one by one, the difference of each mode is plain. `MOV AH, 7` is immediate mode: the number 7 is the second byte of the instruction `B4 07`, and memory is not visited separately. `MOV AL, [table+1]` is direct mode. The assembler computes the expression `table+1` as 0116h + 1 = 0117h and embeds this address in the instruction. The instruction takes its short accumulator form, `A0 17 01`. The byte at 0117h is 22h, the second element of the table. Since AH was set to 07h earlier, AX = 0722h.

`MOV CL, [BX]` is register indirect mode. The address is not written in the instruction; the processor takes it from BX at run time. EA = 0116h and the byte read is 11h. `MOV CH, [BX+4]` is based mode: EA = 0116h + 4 = 011Ah, and the value read is 55h, the fifth element of the table. CX = 5511h. The difference between these two instructions is the addition of a constant offset. When accessing the fields of a record, BX holds the start of the record and the displacement the position of the field within the record.

`MOV DL, [BX+SI]` is based-indexed mode: EA = 0116h + 0002h = 0118h, DL = 33h. The last instruction adds an 8-bit displacement to that: EA = 0116h + 2 + 3 = 011Bh, DH = 66h. Since BP is used in none of these six accesses, the segment is always DS. The physical address is found as DS × 10h + EA = 07000h + EA.

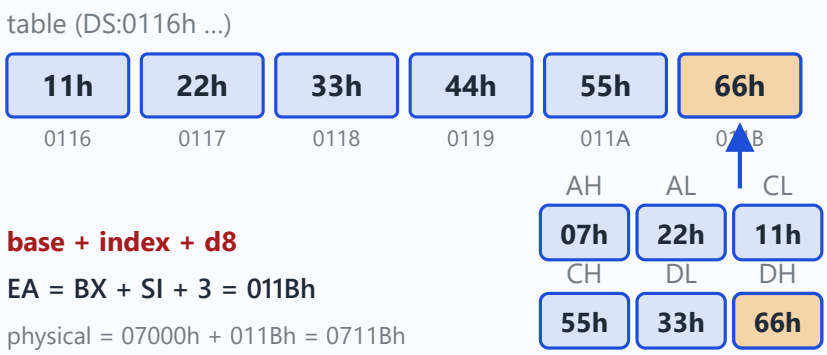


Figure 4.1.3: The last step of the six addressing modes: `MOV DH, [BX+SI+3]` reads the byte whose effective address is 011Bh, that is, the last element of the table (66h).

The final state of the program is AX = 0722h, BX = 0116h, CX = 5511h, DX = 6633h, SI = 0002h. `RET` pulls the value 0000h from the stack and IP = 0000h. At relative location 0000h of the segment where the COM program was loaded sits the `INT 20h` instruction placed by DOS, and the program ends. Since the MOV instructions do not change the flags, the flags stay as at the start throughout the trace: only IF = 1.

4.1.3 Effective address calculation

4.1.3.1 Step by step: with DS

The effective address calculation is done with an addition inside the execution unit (EU). The result is handed to the bus interface unit (BIU) to be combined with the segment register. The BIU's own adder shifts the segment left by 4 bits, adds the EA, and puts the 20-bit address on the lines A19–A0. These two stages can be followed with an example:

```
MOV AL, [BX + SI + 4]   ; DS = 1000h, BX = 0200h, SI = 0030h
```

1. Take the base register BX = 0200h.
2. Add the index register SI = 0030h: 0230h.
3. Add the 8-bit displacement +4.
4. EA = 0200h + 0030h + 0004h = 0234h. The addition is 16-bit; there is no overflow here.
5. Since BP is not in the address, the segment is DS. Physical address = 1000h × 10h + 0234h = 10000h + 0234h = 10234h. AL takes the byte at this address.

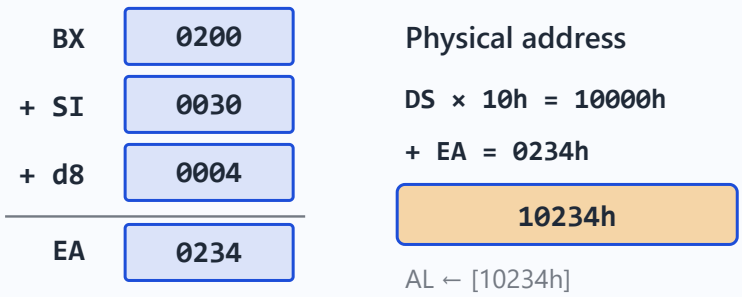


Figure 4.1.4: The effective address and physical address for `MOV AL, [BX + SI + 4]` : BX + SI + 4 = 0234h; with DS = 1000h, 10234h.

Multiplying the segment by 10h means appending a zero on the right in hexadecimal: 1000h → 10000h. In binary, this means shifting the 16-bit number left by 4 bits and putting 0 into the lowest four bits. The physical address is therefore easy to compute: write a 0 after the segment, write the EA right-aligned beneath it, and add the two numbers.

4.1.3.2 Step by step: with BP

If BP is in the address, the same calculation is done with the SS segment. The second example in the slides reads a word and uses a negative displacement:

```
MOV AX, [BP + DI - 2]   ; SS = 2000h, BP = 0100h, DI = 0010h
```

1. Take the base register BP = 0100h.
2. Add the index register DI = 0010h: 0110h.
3. The displacement is −2; it sits in the machine code as the byte `FE` and becomes FFFEh by sign extension.
4. EA = 0100h + 0010h + FFFEh = 1 010Eh → 010Eh. The overflowing bit is discarded; the result is the same as 0110h − 2.
5. Because BP is used, the segment is SS: 20000h + 010Eh = 2010Eh. Since AX is a word, two bytes are read: AL ← [2010Eh], AH ← [2010Fh].

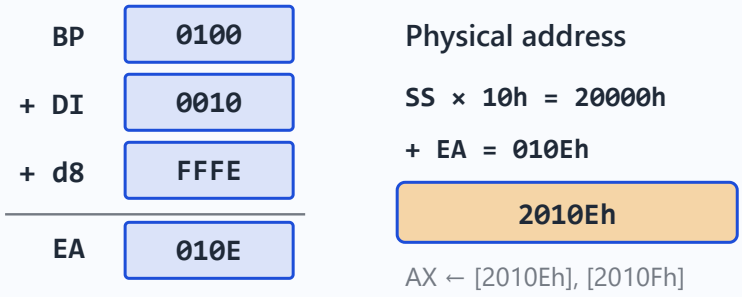


Figure 4.1.5: `MOV AX, [BP + DI - 2]` : EA = 010Eh; because BP is used, the segment is SS and the physical address 2010Eh. The second byte of the word is at 2010Fh.

The machine code of this instruction is `8B 43 FE`. The ModR/M byte `43` = 01 000 011: MOD = 01 (d8 present), REG = 000 (AX), R/M = 011 (`[BP+DI]`). “SS” is not written anywhere in the instruction. The segment choice comes from the R/M field itself: if the R/M value is a combination that contains BP, the hardware selects SS, otherwise DS.

Reading a word has a subtlety on the data-bus side. The 8086's data bus is 16 bits, but memory is addressed byte by byte. The two bytes of a word at an even address arrive together in a single bus cycle. Since 2010Eh in this example is even, the read is a single cycle. Had the word started at an odd address, for example at 2010Fh, the processor would first have read the low byte from the high half of the bus and then, in a separate cycle, the high byte from the low half of the bus. The result is the same, but the access is one bus cycle, that is, at least 4 clock periods, longer. That is why assemblers have the `EVEN` directive, which aligns word variables to even addresses.

4.1.3.3 The effective address calculator

The calculator in the slides performs the same calculation interactively. From the base column you choose BX, BP, or none; from the index column SI, DI, or none; and you enter a displacement. Register and segment values can be written in hexadecimal, and the displacement in decimal (−2) or hexadecimal (1Ah). For a valid combination the tool gives the EA, the default segment, and the physical address. It is suitable for checking a calculation done by hand. On paper the same calculation rests on three rules: do the addition in 16 bits, choose SS if BP is present, and multiply the segment by 10h.

Example 4.1.2

DS = 3000h, SS = 4000h, BX = 0010h, BP = FFFEh, SI = 0005h. What are the effective and physical addresses of the operands `[BX+SI+20h]` and `[BP+4]` ?

The first operand has no BP. EA = 0010h + 0005h + 0020h = 0035h, segment DS: 30000h + 0035h = 30035h.

For the second operand, EA = FFFEh + 0004h = 1 0002h. The 17th bit is discarded and EA = 0002h. Since BP is present, the segment is SS: 40000h + 0002h = 40002h. Without the wraparound the address would have come out as 50002h. This value is outside the SS segment, and the 8086 does not produce such an address.

4.1.3.4 Choosing a mode by data structure

The same byte can be reached with more than one mode. To choose the right mode, look at which part of the data is fixed at assembly time and which part varies at run time. The fixed part goes into the displacement and the varying parts into registers. Table 4.1.6 summarizes the common cases.

Data accessed	Suitable mode	Example
A single variable	direct	`MOV AX, counter`
The element a pointer points to	register indirect	`MOV AL, [SI]`
The i-th element of an array (array fixed, index variable)	indexed + d16	`MOV AL, arr[SI]`
A field of a record (structure)	based + d8	`MOV AX, [BX+2]`
A matrix or an array of records	base + index + d	`MOV AL, mat[BX+SI]`
A parameter or local variable on the stack	based, with BP (SS)	`MOV AX, [BP+4]`
A buffer in another segment	with a segment override	`MOV AL, ES:[DI]`

Table 4.1.6: The suitable addressing mode by the data accessed.

The idea behind the table is this. The base register shows which record or which row is being worked on. When the record changes, only the base register is updated. The index register counts the element inside the record and is incremented in a loop. The displacement is the fixed location of the array or

field; the assembler embeds it in the instruction. For example, in an array of student records of 6 bytes each, the grade field at the 4th byte of a record is read with `[BX+4]`. To move to the next record, `ADD BX, 6` is enough; the read instruction stays the same.

The stack row has a separate importance. When a parameter is passed to a subroutine on the stack, the subroutine copies SP to BP and reaches the parameters with fixed offsets such as `[BP+4]` and `[BP+6]`. BP's default segment being SS was chosen for this use: the stack is in the SS segment.

4.1.3.5 Code trace: accessing a matrix element

The clearest use of the base + index + displacement mode is a two-dimensional array. In the program below, `mat` is a byte matrix of 3 rows and 4 columns. The element values were chosen so that they are recognizable at a glance: the first hexadecimal digit of the value is the row number and the second the column number, counting from 1. For example, 32h is the second element of the third row. Since the program counts rows and columns from 0, this element is called `mat[2][1]`.

```
org 100h
mov bx, 2*4          ; row 2
mov si, 1            ; column 1
mov al, mat[bx+si]   ; mat[2][1]
mov si, 3            ; column 3
mov ah, mat[bx+si]   ; mat[2][3]
sub bx, 4            ; row 1
mov dl, mat[bx+si]   ; mat[1][3]
ret
mat db 11h,12h,13h,14h
    db 21h,22h,23h,24h
    db 31h,32h,33h,34h
```

The matrix is stored in memory in **row-major order**: first the four bytes of row 0, then those of row 1, and finally those of row 2. `mat` starts at 0119h and ends at 0124h. The address of an element is found with this formula:

> address(mat[row][column]) = mat + row × number of columns + column

In the program the term `row × number of columns` is put in BX, the term `column` in SI, and the relative location of `mat` in the displacement. The notation `mat[bx+si]` is the same as `[bx+si+0119h]` and is encoded as `8A 80 19 01`. Since 0119h is greater than 127, d16 is used.

Step	Instruction	BX	SI	EA	AX	DL
1	`mov bx, 2*4`	0008h	0000h	—	0000h	00h
2	`mov si, 1`	0008h	0001h	—	0000h	00h
3	`mov al, mat[bx+si]`	0008h	0001h	0122h	0032h	00h
4	`mov si, 3`	0008h	0003h	—	0032h	00h

Step	Instruction	BX	SI	EA	AX	DL
5	`mov ah, mat[bx+si]`	0008h	0003h	0124h	3432h	00h
6	`sub bx, 4`	0004h	0003h	—	3432h	00h
7	`mov dl, mat[bx+si]`	0004h	0003h	0120h	3432h	24h

Table 4.1.7: Trace of the matrix access (mat = 0119h, row length 4 bytes).

In step 1, BX = 2 × 4 = 8. The assembler does the multiplication at assembly time and writes 0008h into the instruction. Row 2 is 8 bytes past the start of the matrix. In step 2, SI = 1 selects the column. In step 3, EA = 0119h + 8 + 1 = 0122h and AL = 32h is read. This is column 1 of row 2 of the matrix (both counted from 0).

Steps 4 and 5 stay in the same row and change the column. Since BX is left untouched, the row is fixed. EA = 0119h + 8 + 3 = 0124h and AH = 34h; AX = 3432h. The read instruction is the same as before; only the value of the index register has changed. The `sub bx, 4` in step 6 goes up one row: BX = 0004h. Since SI = 3 is kept, the column stays the same. In step 7, EA = 0119h + 4 + 3 = 0120h and DL = 24h. This is column 3 of row 1.

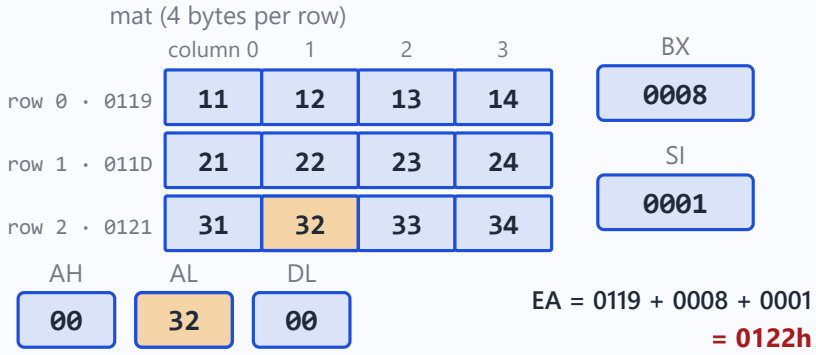


Figure 4.1.6: Step 3 of the matrix access: BX selects the row (8 bytes ahead) and SI the column (1); EA = 0122h and AL = 32h.

Unlike MOV, `sub bx, 4` changes the flags. The result 8 − 4 = 4 is not zero and needs no borrow; ZF = 0, CF = 0, SF = 0. Since 4 = 0000 0100b has an odd number of 1 bits, the parity flag PF = 0. At the end of the program AX = 3432h, BX = 0004h, DL = 24h. Three different elements were read with a single addressing pattern.

In a matrix of words, both terms in the formula are multiplied by 2: address = mat + (row × number of columns + column) × 2. A word occupies 2 bytes, and index registers always count bytes.

4.1.3.6 Effective address calculation time

On the 8086 the effective address is computed by a separate addition inside the EU. On the 80286 and later processors there is a separate address unit for this job. On the 8086 the time of the EA calculation is added to the instruction's own time and varies with the mode. Table 4.1.8 gives the EA

timing table from the Intel 8086 user's manual.

Effective address combination	Saat turu
`[BX]`, `[BP]`, `[SI]`, `[DI]`	5
direct `[d16]`	6
`[BP+DI]`, `[BX+SI]`	7
`[BP+SI]`, `[BX+DI]`	8
single register + d8/d16	9
`[BP+DI+d]`, `[BX+SI+d]`	11
`[BP+SI+d]`, `[BX+DI+d]`	12
segment prefix	+2

Table 4.1.8: Time of the effective address calculation on the 8086 (clock periods).

The general MOV that moves from memory to a register takes 8 periods plus the EA time. Register-to-register MOV takes 2 periods. Accordingly:

- `MOV AX, BX` : 2 periods. Memory is not visited.
- `MOV CL, [BX]` : 8 + 5 = 13 periods.
- `MOV CL, [BX+DI+4]` : 8 + 12 = 20 periods.
- `MOV CL, ES:[BX]` : 8 + 5 + 2 = 15 periods.

An odd detail in the table stands out: `[BX+SI]` takes 7 periods and `[BX+DI]` 8. This difference comes from the 8086's internal organization and can be a small selection criterion for the programmer. A word starting at an odd address needs a second bus cycle and so adds 4 more periods per transfer. On a 5 MHz 8086 one clock period is 200 ns. `MOV CL, [BX+DI+4]` thus takes 20 × 200 ns = 4 µs; the same job register-to-register takes 0.4 µs.

These differences explain why keeping a frequently read value in a register throughout a loop is fast. On each iteration memory is not visited, and the EA calculation disappears entirely. The short direct-address form specific to AL and AX (`A0` / `A1`) is also fast: it contains no EA calculation and takes 10 periods. `MOV CL, [1234h]`, on the other hand, takes 8 + 6 = 14 periods.

4.1.4 Segment and offset

4.1.4.1 From segment:offset to the physical address

The 8086's address bus has 20 lines: A19–A0. With 20 bits, 2^{20} = 1,048,576 different bytes, that is, 1 MB, can be addressed. The processor's registers, however, are 16 bits, and a single register can span only a 64 KB area. The designers solved this mismatch by combining two 16-bit numbers. The first number says which block of memory is used, and the second the position within that block.

Definition 4.1.4

The value in a segment register (CS, DS, SS, ES) is called the **segment**. An address that comes from the effective address calculation or from a register such as IP or SP is called the **offset**. An address is written in the form `segment:offset`. The physical address is found with

$$\text{physical address} = \text{segment} \times 10\text{h} + \text{offset}$$

The segment points to a block of memory starting on a 16-byte boundary, and the offset to the byte within that block.

In the example in the slides, DS = 1234h and SI = 7890h. The address is written `1234:7890`. The calculation is done in four steps (Figure 4.1.7):

1. Take the segment and the offset: 1234h and 7890h.
2. Multiply the segment by 10h, that is, shift it left 4 bits: 12340h. In hexadecimal notation this is appending a 0 on the right.
3. Right-align the offset to enter the addition: 07890h.
4. Add the two numbers: 12340h + 07890h = 19BD0h.

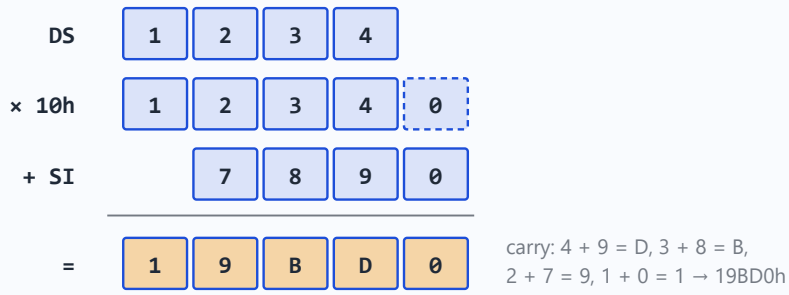


Figure 4.1.7: The physical equivalent of address 1234:7890: the segment is shifted left 4 bits (12340h), the offset (7890h) is added; the result is 19BD0h.

Adding digit by digit gives 0 + 0 = 0, 4 + 9 = 13 = Dh, 3 + 8 = 11 = Bh, 2 + 7 = 9, 1 + 0 = 1. Since no digit produces a carry, the result is easy to read. An example with carries is the pair 1A5D:FFF3. In the calculation 1A5D0h + 0FFF3h, start from the right: 0 + 3 = 3; D + F = 13 + 15 = 28 = 1Ch, write C in the digit and carry 1; 5 + F + 1 = 21 = 15h, write 5 in the digit; A + F + 1 = 26 = 1Ah, write A in the digit; 1 + 0 + 1 = 2. The result is 2A5C3h. In hexadecimal addition, if a digit exceeds 15, 16 is subtracted and a carry of 1 is passed to the next digit.

In hardware this addition is done by a separate 20-bit adder inside the BIU. The 16 bits of the segment register are fed to the upper 16 inputs of the adder, with 0 wired to the lowest 4 inputs. This wiring is the entire circuit for “multiplying by 10h”; there is no separate multiplier. The adder's

output goes to the address latches and from there to the lines A19–A0. When fetching instructions, the BIU uses the same adder with CS and IP.

One consequence of multiplying the segment by 16 is that segments can begin only at addresses that are multiples of 16. This 16-byte unit is called a **paragraph**. The number in a segment register says at which paragraph the segment starts. Since a segment's offset is 16 bits, a segment is at most 2^{16} = 64 KB long. Because two consecutive segment values start only 16 bytes apart, segments overlap one another. The same physical byte can be reached with many different `segment:offset` pairs. For example, the pairs 0700:0100, 0710:0000, and 06F0:0200 all point to address 07100h.

What happens if the sum overflows 20 bits? The sum of the pair FFFF:0010 is FFFF0h + 0010h = 100000h; 21 bits are needed. The 8086 has no 21st line, and the overflowing bit is discarded. The address wraps around to 00000h, the start of memory. Addresses from FFFF:0010 to FFFF:FFFF thus fall in the region 00000h–0FFEFh.

4.1.4.2 Address lines and memory capacity

The relationship between the number of address lines and the memory size is familiar from decoders in digital design. *n* address lines produce 2^n different binary patterns, and each pattern selects one memory location. How many bits a location holds is determined by the number of data lines:

> $$\text{capacity} = 2^n \times \text{data width}$$

The 8086's 20 address lines select 2^{20} = 1,048,576 locations. Since each location is a byte, the capacity is 1 MB and addresses range from 00000h to FFFFFh. The limit of a segment comes from the offset: 16 bits → 2^{16} = 65,536 bytes = 64 KB. Table 4.1.9 gives the capacity for different numbers of lines.

Address / data lines	Number of locations	Capacity
12 / 8	2^{12} = 4096	4096 × 1 B = 4 KB
16 / 4	2^{16} = 64 K	64 K × 4 bit = 256 Kbit = 32 KB
16 / 8	64 K	64 KB (one segment)
20 / 8	1 M	1 MB (8086)
24 / 16	16 M	16 M × 2 B = 32 MB
32 / 8	4 G	4 GB (80386)

Table 4.1.9: Memory capacity by the number of address and data lines.

In a memory chip the number of data lines is the chip's word width. If the capacity is wanted in bytes, the total number of bits is divided by 8. A chip with 16 address lines and 4 data lines holds 65,536 × 4 = 262,144 bits, that is, 32,768 bytes. In an 8086 system, chips of this kind are connected side by side to build an 8- or 16-bit data bus.

The question can also be asked the other way around: how many address lines are needed at minimum for a given capacity? For this, find the first power of 2 equal to or greater than the capacity. If memory is byte-addressed, 2^{32} = 4 GB is not enough for 6 GB; 2^{33} = 8 GB suffices and 33 lines are needed. For 4 KB, 2^{12} = 4096 is exactly enough; 12 lines are needed (A0–A11).

Common mistake

Confusing bits with bytes. A chip with 20 address lines and 16 data lines holds 2^{20} × 16 bits = 16 Mbit. That is 2 MB, not 16 MB. When writing a capacity, it must be stated clearly whether the unit is bits or bytes. The 8086's memory is byte-addressed; there, 20 lines mean 1 MB.

4.1.4.3 The reverse calculation: from the physical address to segment:offset

When a physical address is given and the segment and offset are wanted, there is no single answer. Because segments overlap, many pairs give the same address. The right question is the range of values the segment register can take. In the example in the slides, address 2A5C3h is to be accessed with DS:SI.

The largest DS. This is the case where SI is smallest. DS × 10h must not exceed 2A5C3h. Taking the first four hexadecimal digits of the address gives DS = 2A5Ch. DS × 10h = 2A5C0h, leaving SI = 2A5C3h − 2A5C0h = 0003h. Had DS been 2A5Dh, 2A5D0h would already have been greater than 2A5C3h.

The smallest DS. This is the case where SI is largest. SI can be at most FFFFh. In this case DS × 10h ≥ 2A5C3h − FFFFh = 1A5C4h must hold. Dividing 1A5C4h by 10h gives 1A5C.4h. Since DS is an integer, it is rounded up: DS = 1A5Dh. Then SI = 2A5C3h − 1A5D0h = FFF3h.

How many pairs? Each time DS increases by one, the start of the segment moves ahead by 10h and, for the same address, SI decreases by 10h. DS can take any value from 1A5Dh to 2A5Ch. The number of pairs is 2A5Ch − 1A5Dh + 1 = 0FFFh + 1 = 1000h = 4096.

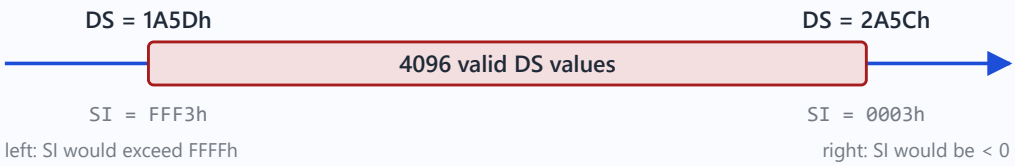


Figure 4.1.8: The values DS can take for address 2A5C3h: smallest 1A5Dh (SI = FFF3h), largest 2A5Ch (SI = 0003h); there are 4096 pairs in the range.

Such a calculation is always checked in the forward direction. 1A5Dh × 10h + FFF3h = 1A5D0h + FFF3h = 2A5C3h. 2A5Ch × 10h + 0003h = 2A5C0h + 3 = 2A5C3h. Both ends give the same physical address.

The length of the range is not the same for every address. For very small addresses, the lower bound of DS runs against 0. For example, for address 00345h, DS is at least 0000h and at most 0034h, and there are only 35h = 53 pairs. For addresses of 0FFF0h and above, the range is always 4096 pairs. This count does not take the wraparound at the 1 MB boundary into account; wrapped pairs such as FFFF:0355 also go to 00345h.

4.1.4.4 The segment override prefix

The default segment rule chooses the right segment in most cases. Sometimes, however, a program wants to reach a byte outside the data segment. For this, the name of a segment register and a colon are written in front of the square brackets. This notation is called a **segment override prefix**.

```
MOV AX, 0B800h
MOV ES, AX          ; ES = video memory
MOV BX, 0
MOV AL, ES:[BX]     ; read ES:0000
MOV AL, [BX]        ; read DS:0000
MOV DL, SS:[SI]     ; read SS:SI
```

In machine code the prefix is a one-byte code placed in front of the instruction: ES: = `26h`, CS: = `2Eh`, SS: = `36h`, DS: = `3Eh`. When the processor reads this byte, it changes the default segment only for the instruction that follows. The prefix does not carry over to the next instruction. `MOV AL, ES:[BX]` thus becomes `26 8A 07`: a `26` has been added in front of the code `8A 07` of the unprefixed `MOV AL, [BX]`. 2 clock periods are also added to the EA calculation.

A prefix is needed in three cases:

- To access data in another segment without changing DS. Video memory is the most frequent example.
- To access data on the stack with a register other than BP: `SS:[BX]`.
- To access a table in the code segment: `CS:[SI]`. Some programs place constant tables next to the code; even if DS points elsewhere, the table is read with CS.

Not every prefix can change every access. IP is always used with CS and SP always with SS; these pairings are wired into the hardware. Instruction fetch is done from CS:IP and stack operations (PUSH, POP, CALL, RET) from SS:SP. ES:DI, the destination of string instructions, cannot be changed either. The string source DS:SI, on the other hand, can be redirected to another segment with a prefix.

Definition 4.1.5

The **default segment** is the segment used in a memory access when there is no prefix. It is CS for instruction fetch (with IP), SS for stack operations (with SP), SS for addresses containing BP, ES for the string destination (with DI), and DS for all other data accesses. A segment prefix changes the default segment of data accesses only for that instruction.

4.1.4.5 Code trace: video memory with ES

On IBM PC-compatible computers the contents of the color text screen sit in the memory starting at physical address B8000h. The display card reflects every byte written to this memory onto the screen at once. Each character cell of the screen is two bytes: first the character's ASCII code, then the **attribute byte** that determines the color. A row is 80 cells, that is, 160 bytes. The program below uses ES to write a yellow K into the first cell of row 1 of the screen and also reads the same offset with DS.

```
ORG 100h
MOV AX, 0B800h
MOV ES, AX          ; ES = video
MOV BX, 00A0h       ; row 1
MOV BYTE PTR ES:[BX], 'K'
MOV BYTE PTR ES:[BX+1], 1Eh
MOV AL, [BX]        ; DS:00A0h
MOV DL, ES:[BX]     ; B800:00A0h
RET
```

IP	Instruction	Machine code	Address accessed	Result
0100h	`MOV AX, 0B800h`	`B8 00 B8`	—	AX = B800h
0103h	`MOV ES, AX`	`8E C0`	—	ES = B800h, DS = 0700h
0105h	`MOV BX, 00A0h`	`BB A0 00`	—	BX = 00A0h
0108h	`MOV BYTE PTR ES:[BX], 'K'`	`26 C6 07 4B`	B80A0h	[B80A0h] = 4Bh
010Ch	`MOV BYTE PTR ES:[BX+1], 1Eh`	`26 C6 47 01 1E`	B80A1h	[B80A1h] = 1Eh
0111h	`MOV AL, [BX]`	`8A 07`	070A0h	AL = 00h
0113h	`MOV DL, ES:[BX]`	`26 8A 17`	B80A0h	DL = 4Bh
0116h	`RET`	`C3`	stack	IP = 0000h

Table 4.1.10: Trace of the segment prefix program (DS = 0700h; the video cell initially holds 20h 07h).

The first two instructions set up ES. On the 8086 a constant cannot be loaded directly into a segment register; B800h is first moved into AX and from there into ES. DS still points to the program's own segment (0700h). This way access to the program's variables is not disturbed. The third instruction puts 00A0h = 160 into BX. This number comes from the calculation (1 × 80 + 0) × 2 and is the offset in video memory of the first cell of row 1 (rows are counted from 0).

The fourth instruction begins with the `ES:` prefix. The `26` byte in the machine code makes the instruction that follows, `C6 07 4B` ("set the byte BX points to to 4Bh"), go to the ES segment. The physical address is B8000h + 00A0h = B80A0h, and the ASCII code of the letter 'K', 4Bh, is written there. The fifth instruction writes 1Eh to the next byte. 1Eh = 0001 1110b is the attribute byte: the background color bits (6–4) 001 = blue, the foreground color bits (3–0) 1110 = yellow. A yellow K appears on the screen on a blue background.

The sixth instruction carries no prefix. `[BX]` uses DS by default and reads address 07000h + 00A0h = 070A0h. This address falls in the 256-byte **PSP** (program segment prefix) area in front of the COM program; in the example there is 00h there and AL = 00h. The same offset has gone to an entirely different byte with a different segment. The seventh instruction again uses the `ES:` prefix and reads the 4Bh written to video memory: DL = 4Bh.

Figure 4.1.9: The same offset (00A0h) in two segments: without a prefix, 070A0h is read through DS (AL = 00h); with the `ES:` prefix, the 'K' (4Bh) at B80A0h is read.

At the end of the program ES = B800h and DS = 0700h. The prefix affected only the instruction it was written on and never touched DS.

4.1.4.6 Code trace: the BP and SS trap

In COM programs CS, DS, SS, and ES carry the same value. Choosing the wrong segment in a COM program therefore often goes unnoticed. In an EXE program with separate segments, however, accessing data through BP fetches the wrong byte. The program below shows this trap.

```
data segment
pad db 3 dup(0)
x db 'D'
data ends
stk segment stack
dw 64 dup(0)
stk ends
code segment
assume cs:code, ds:data, ss:stk
start:
mov ax, data
mov ds, ax
mov bp, offset x
mov al, [bp]         ; reads SS:BP!
mov bl, ds:[bp]      ; DS:BP
mov si, offset x
mov cl, [si]         ; DS:SI
mov ax, 4C00h
int 21h
code ends
end start
```

Three segments are defined in the program. The `data` segment is four bytes: three zero bytes and `x`, which holds the letter 'D'. The relative location of `x` is 0003h. The `stk` segment is a stack of 64 words filled with zeros. The word `stack` tells the linker that this is the program's stack. When opening the program, the loader sets SS to this segment and SP to its end (0080h). The `assume` directive generates no code; it tells the assembler which segment register will point to which segment.

The load layout of the program is as follows. DOS places the PSP at paragraph 0700h; the program's image starts 256 bytes later, at 0710h. The segments are laid out in definition order, aligned to paragraph boundaries. The four-byte `data` takes one paragraph: 0710h. `stk` starts right after it at 0711h, and the code starts at 0719h after the 128-byte stack. These values are one example layout. On a real DOS the paragraph numbers vary with the occupied part of memory; the logic of the calculation stays the same.

Instruction	Machine code	DS	SS	Address accessed	Result
`mov ax, veri`	`B8 10 07`	0700h	0711h	—	AX = 0710h
`mov ds, ax`	`8E D8`	0710h	0711h	—	DS = 0710h
`mov bp, offset x`	`BD 03 00`	0710h	0711h	—	BP = 0003h
`mov al, [bp]`	`8A 46 00`	0710h	0711h	SS:0003 = 07113h	AL = 00h
`mov bl, ds:[bp]`	`3E 8A 5E 00`	0710h	0711h	DS:0003 = 07103h	BL = 44h ('D')
`mov si, offset x`	`BE 03 00`	0710h	0711h	—	SI = 0003h
`mov cl, [si]`	`8A 0C`	0710h	0711h	DS:0003 = 07103h	CL = 44h ('D')

Instruction	Machine code	DS	SS	Address accessed	Result
`mov ax, 4C00h` / `int 21h`	`B8 00 4C` / `CD 21`	0710h	0711h	—	return to DOS, code 0

Table 4.1.11: Trace of accessing data through BP (data = 0710h, stk = 0711h; x = 0710:0003 = 07103h).

When the program starts, DS does not point to the data segment; the loader sets DS to the PSP (0700h). The instruction `mov ax, data` loads the paragraph number of the `data` segment into AX. This number becomes known at load time; the loader writes it into the instruction by consulting the relocation table in the EXE file. After `mov ds, ax`, DS = 0710h. SS = 0711h is a separate block; in this program DS ≠ SS.

`mov bp, offset x` puts the correct offset of `x` (0003h) into BP. The address is right; the problem is in the segment. In `mov al, [bp]` there is no prefix and BP is in the address. The processor chooses SS and reads address 0711h × 10h + 3 = 07113h. That is the fourth byte of the stack and it holds 00h. AL = 00h; the 'D' has not arrived. The assembler gives no error, because the instruction is valid.

In `mov bl, ds:[bp]` the `DS:` prefix changes the default. The byte `3E` is added to the front of the machine code and 07103h is read: BL = 44h ('D'). When the same offset is taken into SI, no prefix is needed. `[si]` uses DS by default and CL = 44h. The last two instructions call DOS's 4Ch service: AH = 4Ch is “terminate the program” and AL = 0 is the return code.

Figure 4.1.10: The `mov al, [bp]` step: BP = 0003h is the correct offset, but because the default segment is SS, the byte at 07113h of the stack is read (AL = 00h). The letter 'D' is at 07103h in the data segment.

The rule from this trace is simple. To access data, choose BX, SI, or DI; to access the stack, choose BP. If accessing the data segment with BP is truly necessary, write the `DS:` prefix. The detail `46 00` in the machine code of `mov al, [bp]` is also noteworthy. Since `[BP]` without a displacement cannot be encoded, the assembler produced `[BP+0]`.

Common mistake

Accessing a variable through BP, getting the right result in a COM program, and then moving the code to an EXE program. In a COM program CS = DS = SS, so `[BP]` and `DS:[BP]` go to the same byte and the error is invisible. In an EXE program with separate segments the

same code reads from the stack. The error is noticed not at assembly time but when wrong values appear.

4.1.4.7 Changing a segment register

The values of segment registers can be changed, but each has its own rules.

```
PUSH DS            ; save DS
MOV  AX, 0B800h
MOV  DS, AX        ; temporary
MOV  BYTE PTR [0], 'Z'
POP  DS            ; variables back

MOV  AX, new_stack
MOV  SS, AX        ; right after it
MOV  SP, 0400h     ; set SP
```

DS. Variable names in assembly are only offsets. If the program changes DS, the same name goes to a byte in another segment. The first four lines of the code temporarily set DS to B800h to write a Z in the top left corner of the screen. `BYTE PTR [0]` thus goes to B8000h. When the job is done, `POP DS` restores the old value. Had PUSH and POP not been used, all subsequent variable accesses would have gone to video memory. A safer way is to use ES without touching DS at all. ES is the default segment for no access other than the string destination.

SS and SP. The two change together. After SS is changed, SP must take a value that suits the new segment. If an interrupt came between the two instructions, the processor would push the return address to a wrong place formed by the new SS and the old SP. The 8086 prevents this in hardware: after an instruction that does a MOV or POP to a segment register, no interrupt is accepted until the next instruction finishes. That is why `MOV SP, …` is written right after `MOV SS, AX`, and the SS:SP pair is never half set.

CS. It cannot be written with MOV. If CS changes, the next instruction is fetched from an entirely different place and the old value of IP becomes meaningless. CS therefore changes together with IP: far JMP, far CALL, far RET, INT, and IRET load both at once. Reads such as `MOV AX, CS`, on the other hand, are free.

4.1.5 Data type and byte order

4.1.5.1 byte ptr and word ptr

A memory address points to a single byte. The 8086, however, can read and write a byte (8 bits) or a word (16 bits, two bytes) starting at the same address. Which one an instruction does is written in the lowest bit of the opcode, the **W bit**: W = 0 means byte and W = 1 means word. This is the difference between the opcodes `8A` and `8B`. To set this bit correctly, the assembler must know the size of the access.

In most instructions the size can be told from the other operand. In `MOV [BX], AL`, AL is 8 bits and a byte is written. In `MOV [BX], AX`, AX is 16 bits and a word is written. But in `MOV [BX], 5` neither operand has a size of its own. The square brackets say only an address. The number 5 fits both in a byte (`05`) and in a word (`05 00`). The assembler cannot decide in this case. Some assemblers report an error, others guess a size. If the guess is wrong, the program silently corrupts the neighboring byte.

Definition 4.1.6

A **data type specifier** is the addition that tells the assembler whether a memory operand is a byte or a word. `byte ptr [BX]` means a one-byte access and `word ptr [BX]` a two-byte access. emu8086 also recognizes their short forms: `b.[SI]` for byte, `w.[DI]` for word.

The specifier is for the assembler only; the processor does not see it. `MOV BYTE PTR [BX], 5` is encoded as `C6 07 05` (3 bytes) and `MOV WORD PTR [BX], 5` as `C7 07 05 00` (4 bytes). The whole difference is the opcode's W bit (`C6` → `C7`) and the length of the constant. Figure 4.1.11 shows reading at the same address with two different sizes.

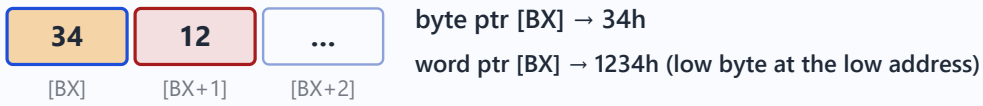


Figure 4.1.11: At the same address `byte ptr` reads one byte and `word ptr` reads two.

Instruction	How is the size known?
`MOV [BX], AL`	AL is 8 bits → a byte is written
`MOV [BX], AX`	AX is 16 bits → a word is written
`MOV [BX], 5`	ambiguous → `MOV BYTE PTR [BX], 5` or `MOV WORD PTR [BX], 5`
`MOV AL, num`	AL is 8 bits; compatible if `num` is defined with DB, a type mismatch if with DW

Table 4.1.12: Where the access size is known from.

The last row shows that a variable name also carries a type. A variable defined as `num DB 7` is of type byte, and one defined with `DW` is of type word. In an instruction containing a variable name, the assembler takes the size from this type and reports an error if it does not match the register. To change the type deliberately, a specifier is again written: `MOV AL, BYTE PTR wvar` reads only the low byte of a word variable.

4.1.5.2 Code trace: byte ptr and word ptr

The program below writes to and reads from the same address in different sizes. The four bytes starting at 0200h hold the values AA BB CC DD beforehand. That way it is clear which bytes change.

```
MOV BX, 0200h
MOV WORD PTR [BX], 1234h
MOV BYTE PTR [BX], 0FFh
MOV WORD PTR [BX], 0FFh
MOV AX, [BX]        ; word
MOV CL, [BX+1]      ; byte
```

Step	Instruction	Machine code	0200h 0201h 0202h 0203h	Register
1	`MOV BX, 0200h`	`BB 00 02`	AA BB CC DD	BX = 0200h
2	`MOV WORD PTR [BX], 1234h`	`C7 07 34 12`	**34 12** CC DD	—
3	`MOV BYTE PTR [BX], 0FFh`	`C6 07 FF`	**FF** 12 CC DD	—
4	`MOV WORD PTR [BX], 0FFh`	`C7 07 FF 00`	**FF 00** CC DD	—
5	`MOV AX, [BX]`	`8B 07`	FF 00 CC DD	AX = 00FFh
6	`MOV CL, [BX+1]`	`8A 4F 01`	FF 00 CC DD	CL = 00h

Table 4.1.13: Trace of byte ptr and word ptr (initially [0200h…0203h] = AA BB CC DD).

In step 2 two bytes are written. The low byte of the constant, 34h, goes to the low address (0200h) and the high byte, 12h, to the next address (0201h). In step 3 only 0200h changes and becomes FFh; the 12h at 0201h stays in place. In step 4 the same constant FFh is this time written as a word. FFh as a word means 00FFh. FFh goes to 0200h and 00h to 0201h. The 12h preserved in the previous step is now erased. The constant itself is the same; only the specifier has changed.

In step 5 no specifier is needed; since AX is 16 bits, a word is read. The FFh at 0200h goes to AL and the 00h at 0201h to AH: AX = 00FFh. In step 6, CL is 8 bits and the single byte at `[BX+1]` = 0201h is read: CL = 00h. The bytes CC DD at 0202h and 0203h are not touched in any step.

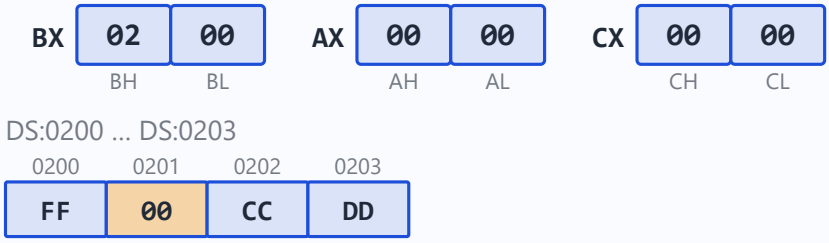


Figure 4.1.12: Step 4: the 0FFh written with `WORD PTR` spreads over two bytes (FF 00); the 12h preserved in the previous step is erased.

4.1.5.3 Little-endian order

In the previous trace we saw that words sit "reversed" in memory: 1234h was stored in memory as `34 12`. This is not a coincidence but the 8086's byte-order rule.

Definition 4.1.7

The order that writes the low byte of a multi-byte value to the low address and the high byte to the next address is called **little-endian** (low byte first). Conversely, the order that writes the high byte first is **big-endian**. The 8086 and the entire x86 family are little-endian.

Figure 4.1.13: Little-endian: the low byte 34h of the word 1234h is written to the low address and the high byte 12h to the next address.

The counterpart of the rule at the register level is very tidy. The instruction `MOV [0200h], AX` writes AL to 0200h and AH to 0201h. `MOV AX, [0200h]` does the opposite: the byte at 0200h goes to AL and the one at 0201h to AH. The low half of a register always matches the low address. One benefit of little-endian arises from this. If a byte is read from the address of a word variable, the low byte arrives. A small number such as 0034h gives the same value at the same address whether read as a word or as a byte.

The emu8086 memory window shows bytes in address order. That is why words look reversed in the window: 1234h → `34 12`. The rule also applies to constants in machine code. `MOV AX, 1234h` is encoded as `B8 34 12` and `MOV AX, 0B800h` as `B8 00 B8`. Addresses obey the same rule: `MOV AX, [1000h]` = `A1 00 10`.

Common mistake

Reading a memory dump left to right and assembling the word that way. If 0200h holds 34h and 0201h holds 12h, then after `MOV AX, [0200h]` AX = 1234h, not 3412h. The low address is the low byte. The same misreading is made when reading machine code: the address in the instruction `A1 00 10` is 1000h, not 0010h.

4.1.5.4 Code trace: index in a word array

When byte order and byte addressing come together, a common mistake appears. To reach the i-th element of a word array, 2i, not i, must be put in the index register. The program below follows first the wrong and then the right way.

```
org 100h
mov si, 1            ; index 1
mov ax, arr[si]      ; BUG!
shl si, 1            ; index × 2
mov bx, arr[si]      ; arr[1]
mov si, 2            ; index 2
shl si, 1            ; SI = 4
mov cx, arr[si]      ; arr[2]
ret
arr dw 1234h, 5678h, 9ABCh
```

Adding up the instruction lengths, `arr` starts at 0117h. The three words are laid out in memory in little-endian as `34 12 78 56 BC 9A` between 0117h and 011Ch. arr[0] = 1234h is at 0117h, arr[1] = 5678h at 0119h, and arr[2] = 9ABCh at 011Bh.

Step	Instruction	SI	EA	Bytes read	Result
1	`mov si, 1`	0001h	—	—	—
2	`mov ax, arr[si]`	0001h	0118h	12 78	AX = 7812h (wrong)
3	`shl si, 1`	0002h	—	—	—
4	`mov bx, arr[si]`	0002h	0119h	78 56	BX = 5678h = arr[1]
5	`mov si, 2`	0002h	—	—	—
6	`shl si, 1`	0004h	—	—	—
7	`mov cx, arr[si]`	0004h	011Bh	BC 9A	CX = 9ABCh = arr[2]

Table 4.1.14: Trace of the index in a word array (arr = 0117h: 34 12 78 56 BC 9A).

In step 1 the programmer gives SI = 1 with the thought “the second element.” But an index register counts bytes, not elements. In step 2, EA = 0117h + 1 = 0118h. The word read from there is a combination of the high byte of arr[0] (12h) and the low byte of arr[1] (78h). By the little-endian rule, the 12h at 0118h goes to AL and the 78h at 0119h to AH: AX = 7812h. This value is not any element of the array. The processor gives no warning; reading a word from an odd address is valid on the 8086, it just takes one extra bus cycle.

The `shl si, 1` in step 3 shifts SI left by one bit, that is, multiplies by 2: SI = 2. In a word array the byte offset is index × 2. In step 4, EA = 0119h and BX = 5678h = arr[1] arrives correctly. Steps 5 and 6 do the same job for index 2: SI = 2, then 4. In step 7, EA = 011Bh and CX = 9ABCh = arr[2]. Shifting is the shortest and fastest way to multiply by 2; `D1 E6` is a two-byte instruction.

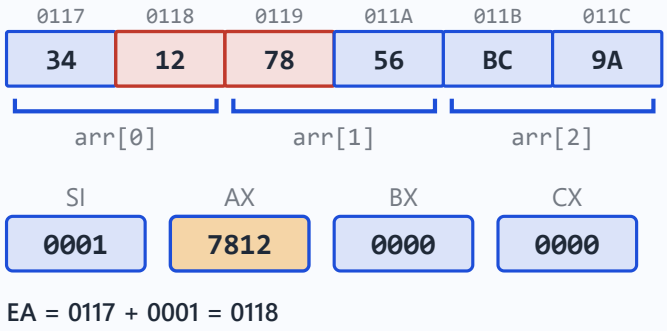


Figure 4.1.14: With SI = 1, `arr[si]` reads from the middle of two elements: the 12h at 0118h and the 78h at 0119h combine, AX = 7812h.

Unlike MOV, the `shl` instruction also updates the flags. Shifting 1 by one bit gives 2; since the bit shifted out is 0, CF = 0, and since the result is nonzero, ZF = 0. The auxiliary carry flag AF is undefined after shift instructions. When walking a word array with a loop, SI is advanced on each iteration with `ADD SI, 2`. `INC SI` is only for byte arrays.

4.1.6 The MOV instruction

4.1.6.1 Definition and rules

MOV, used in all the previous examples, is the 8086's most frequently used instruction. A significant part of the code that compilers produce consists only of MOVs.

Definition 4.1.8

The instruction `MOV destination, source` copies the value in the source to the destination. The source does not change. The source can be an immediate value (a constant), a general-purpose register, a segment register, or a memory location. The destination is a general-purpose register, a segment register (except CS), or a memory location.

The rules of MOV are gathered in five points:

1. No value is assigned to CS and IP with MOV. CS can be read as a source; IP does not appear in any MOV.
2. The two operands must be the same size: both bytes or both words.
3. The two operands cannot both be memory.
4. An immediate value cannot be loaded into a segment register, and one segment register cannot be copied directly to another.
5. MOV does not affect the flags.

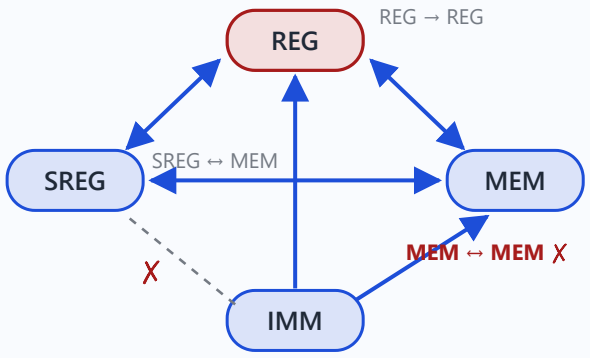


Figure 4.1.15: The allowed source → destination directions for MOV. There is no arrow between memory and memory, between a constant and a segment register, or between two segment registers.

Most of the rules come from the structure of the machine code. A MOV instruction contains at most one ModR/M byte, and this byte can define only one memory operand. The second operand is a register in the REG field or a constant at the end of the instruction. There is no room for two memory operands. Segment registers have separate opcodes: `8E` moves to a segment register and `8C` from one. Their second operand is always a general register or a memory word selected by ModR/M. That is why there is no instruction from segment to segment or from a constant to a segment.

That the flags do not change is also a deliberate choice. After a comparison, a value may need to be brought into a register before the conditional jump. If MOV disturbed the flags, the result of the comparison would be lost.

The two examples in the slides apply the rules. `MOV AX, BX` copies the value in BX to AX; BX does not change. `MOV [SI], 10h`, on the other hand, wants to write 10h to the address DS:SI that SI points to. The destination is a memory location, the source is a constant, and the size is ambiguous. The correct notation is `MOV BYTE PTR [SI], 10h`.

4.1.6.2 Supported operands

The emu8086 documentation lists the valid forms of MOV in two groups. Table 4.1.15 gives this list along with the meanings of the abbreviations used in the notation.

General-purpose	With segment registers
`MOV REG, memory`	`MOV SREG, memory`
`MOV memory, REG`	`MOV memory, SREG`
`MOV REG, REG`	`MOV REG, SREG`
`MOV memory, immediate`	`MOV SREG, REG`
`MOV REG, immediate`	—

Table 4.1.15: Supported operand combinations of the MOV instruction.

- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **SREG**: DS, ES, SS, and CS. CS can be only the second operand, that is, the source.
- **memory**: `[BX]`, `[BX+SI+7]`, a variable name, and any of the 24 forms.
- **immediate**: constants such as 5, −24, 3Fh, 10001101b.

The forms `MOV SREG, immediate` and `MOV SREG, SREG` do not exist. The value is first taken into a general register: `MOV AX, 0B800h` followed by `MOV DS, AX`. There are two ways to copy from segment to segment: `MOV AX, ES` / `MOV DS, AX` or `PUSH ES` / `POP DS`. The second way does not disturb a general register but uses the stack.

4.1.6.3 Valid or not?

The rules can be tested on ten instructions. Table 4.1.16 gives the status and reason for each instruction. The assembler reports an error for every invalid instruction.

No.	Instruction	Status	Reason and correct form
1	`MOV [BX], [SI]`	invalid	there is no memory-to-memory MOV → `MOV AL, [SI]` · `MOV [BX], AL`
2	`MOV DS, ES`	invalid	there is no segment-to-segment → `MOV AX, ES` · `MOV DS, AX` or `PUSH ES` · `POP DS`
3	`MOV DS, 100h`	invalid	a constant cannot be assigned to a segment

No.	Instruction	Status	Reason and correct form
4	`MOV CS, AX`	invalid	CS cannot be the destination (use a far JMP); IP cannot be written either (`JMP AX` instead of `MOV IP, AX`)
5	`MOV [BX], 5`	invalid	the size is ambiguous → `BYTE PTR` / `WORD PTR`
6	`MOV AX, CS`	valid	CS can be a source; its code is `8C C8`
7	`MOV SP, BX`	valid	SP is a general register; but this moves the stack somewhere else
8	`MOV BL, DX`	invalid	8 and 16 bits do not match
9	`MOV AL, [SI+BP]`	valid	same as `[BP+SI]` ; segment SS
10	`MOV 5, AX`	invalid	the destination cannot be a constant

Table 4.1.16: Validity of ten MOV instructions.

Instruction 7 deserves some attention. The instruction obeys the rules and the assembler accepts it. But the moment SP's value changes, the location of the stack changes too. The next PUSH, CALL, or incoming interrupt writes its data to where BX points. If this line was not written on purpose, the program often returns to a wrong address at `RET` . Instruction 9 is a reminder that the writing order does not matter: `[SI+BP]` goes to the SS segment because it contains BP.

4.1.6.4 Code trace: block copy through a register

Since there is no memory-to-memory MOV, to copy a block of memory elsewhere every byte is passed through a register. The program below copies the four-byte string `src` into the buffer `dst` .

```
org 100h
mov si, offset src
mov di, offset dst
mov cx, 4
copy:
mov al, [si]         ; memory → AL
mov [di], al         ; AL → memory
inc si
inc di
loop copy
ret
src db 'ABCD'
dst  db 4 dup('.')
```

The code is 18 bytes in 0100h–0111h. `src` starts at 0112h and `dst` at 0116h. `4 dup('.')` reserves four period characters (2Eh). The instruction `loop copy` decrements CX by one and jumps to the label `copy` if CX is not zero. The machine code is `E2 F8` : LOOP is at 010Fh, the next instruction is at

0111h, and 0111h + (−8) = 0109h, which is the address of `copy`. The jump is in relative mode.

Step	Instruction	SI	DI	CX	AL	dst
1	`mov si, offset src`	0112h	0000h	0000h	00h	`....`
2	`mov di, offset dst`	0112h	0116h	0000h	00h	`....`
3	`mov cx, 4`	0112h	0116h	0004h	00h	`....`
4	`mov al, [si]`	0112h	0116h	0004h	41h	`....`
5	`mov [di], al`	0112h	0116h	0004h	41h	`A...`
6	`inc si`	0113h	0116h	0004h	41h	`A...`
7	`inc di`	0113h	0117h	0004h	41h	`A...`
8	`loop copy`	0113h	0117h	0003h	41h	`A...`
9–13	iteration 2	0114h	0118h	0002h	42h	`AB..`
14–18	iteration 3	0115h	0119h	0001h	43h	`ABC.`
19–23	iteration 4	0116h	011Ah	0000h	44h	`ABCD`
24	`ret`	0116h	011Ah	0000h	44h	`ABCD`

Table 4.1.17: Trace of the block copy (src = 0112h: 'ABCD', dst = 0116h: '....').

The first three steps set up the two pointers and the counter. SI holds the offset of the source and DI that of the destination; the two pointers point to two separate buffers. CX = 4 is the number of bytes to copy. In step 4, `mov al, [si]` moves from memory to a register and AL = 'A' = 41h. In step 5, `mov [di], al` moves from the register to memory and 41h is written to 0116h. AL is a way station here. Since there is no memory-to-memory MOV, there is no other way.

In steps 6 and 7 the two pointers advance by one byte together. In step 8, LOOP lowers CX to 3. Since CX is not zero, IP = 0109h and the loop returns to the start. The second, third, and fourth iterations repeat the same five steps; the table summarizes each iteration in a single row. At the end of the fourth iteration CX = 0 and LOOP does not jump; the program falls through to `ret`. The destination now contains “ABCD.” SI and DI point right past the buffers: SI = 0116h, DI = 011Ah. The final value of SI happens to be the start of `dst`, because the two buffers are adjacent.

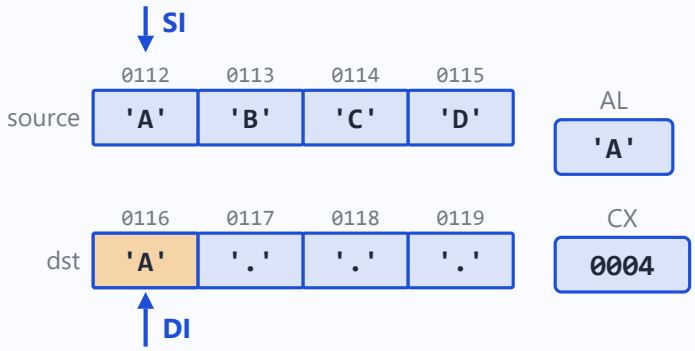


Figure 4.1.16: Step 5 of the block copy: the 'A' (41h) in AL is written to 0116h, which DI points to.

MOV does not change the flags, but INC does. The flag state influences no decision in this program. LOOP does not look at the flags; it looks only at CX. Still, the trace shows the parity flag changing with the INCs. For example, when DI = 0117h after `inc di`, the low byte is 17h = 0001 0111b. Since it has four 1 bits, PF = 1.

Each byte needs two memory accesses, two pointer increments, and one loop instruction. Because this job is done so often, the 8086 has a separate instruction for it: `REP MOVSB` copies CX bytes from DS:SI to ES:DI and advances the pointers itself. The same program shrinks with string instructions to the lines `CLD`, `REP MOVSB`.

4.1.7 Example program: printing an 'A' on the screen

4.1.7.1 The program and its machine code

What has been described so far comes together in a small program. The program writes a character directly into the video memory of the color text screen. No service of the operating system is used; writing to the screen amounts to putting a word at a particular memory address.

```
ORG 100h          ; for a .com program
MOV AX, 0B800h    ; AX = B800h
MOV DS, AX        ; DS = B800h
MOV CL, 'A'       ; CL = 41h
MOV CH, 11011111b ; CH = DFh
MOV BX, 15Eh      ; BX = 015Eh
MOV [BX], CX      ; B800:015E ← CX
RET               ; back to DOS
```

Each line of the program introduces an element. `ORG` is a directive; it determines the program's start address and is not translated into machine code. Everything after the semicolon is a comment; the assembler ignores it. `MOV` copies a value between source and destination. Since a constant cannot be

assigned directly to DS, B800h is first loaded into AX. `'A'` is a character constant and is the same as the ASCII code 41h. `11011111b` is a constant written in binary and equals DFh. `MOV [BX], CX` writes the word in CX to the address DS:BX.

Address	Instruction	Machine code	Explanation
0100h	`MOV AX, 0B800h`	`B8 00 B8`	B8 = “immediate word into AX”; constant little-endian
0103h	`MOV DS, AX`	`8E D8`	8E = to a segment register; D8 = 11 011 000 (DS ← AX)
0105h	`MOV CL, 'A'`	`B1 41`	B0 + 1 (CL); 41h = 'A'
0107h	`MOV CH, 11011111b`	`B5 DF`	B0 + 5 (CH)
0109h	`MOV BX, 15Eh`	`BB 5E 01`	B8 + 3 (BX)
010Ch	`MOV [BX], CX`	`89 0F`	89 = word from register to memory; 0F = 00 001 111
010Eh	`RET`	`C3`	near return

Table 4.1.18: Machine code of the program (15 bytes starting at 0100h).

A regular structure stands out in the codes in the table. The opcode of the MOV that loads an immediate value into a register starts at B0 for byte registers and at B8 for word registers, and the register's three-bit number is added. The numbers are AL/AX = 0, CL/CX = 1, DL/DX = 2, BL/BX = 3, AH/SP = 4, CH/BP = 5, DH/SI = 6, BH/DI = 7. B1 for CL, B5 for CH, and BB for BX are found this way. These instructions have no ModR/M byte; the register is embedded in the opcode.

4.1.7.2 Code trace

The program runs in emu8086 as a COM program. Initially CS = DS = ES = SS = 0700h, IP = 0100h, and SP = FFFEh. All the cells of the screen are empty: every cell holds the bytes 20h (space) and 07h (gray text on a black background).

Step	Instruction	AX	DS	CX	BX	[B800:015E] [015F]
0	(initial)	0000h	0700h	0000h	0000h	20h 07h
1	`MOV AX, 0B800h`	B800h	0700h	0000h	0000h	20h 07h
2	`MOV DS, AX`	B800h	B800h	0000h	0000h	20h 07h
3	`MOV CL, 'A'`	B800h	B800h	0041h	0000h	20h 07h
4	`MOV CH, 11011111b`	B800h	B800h	DF41h	0000h	20h 07h
5	`MOV BX, 15Eh`	B800h	B800h	DF41h	015Eh	20h 07h
6	`MOV [BX], CX`	B800h	B800h	DF41h	015Eh	**41h DFh**
7	`RET`	B800h	B800h	DF41h	015Eh	41h DFh

Table 4.1.19: Trace of the program that prints 'A' on the screen.

In step 1, AX = B800h. B800h is the segment of the color text screen; physically it starts at B8000h. In step 2, DS = B800h. From now on, all unprefixed data accesses made with square brackets go to video memory. After this point the program cannot access its own variables through DS; since this small

program has no variables, this causes no problem.

Steps 3 and 4 fill the two halves of CX separately. CL = 41h is the character's ASCII code and CH = DFh is the color (attribute) byte. Read together, the two halves give CX = DF41h. In step 5, BX = 015Eh = 350. This number is the relative address in video memory of the position on the screen.

Step 6 is the real work of the program. `MOV [BX], CX` writes a word. By the little-endian rule, CL (41h) goes to the low address, B800:015Eh, and CH (DFh) to the high address, B800:015Fh. The physical addresses are B8000h + 015Eh = B815Eh and B815Fh. The character byte coming first and the color byte second fits the layout of video memory exactly. That is why the character was put in the low half and the color in the high half. The display card reads these two bytes on the next scan, and a colored 'A' appears in the cell.

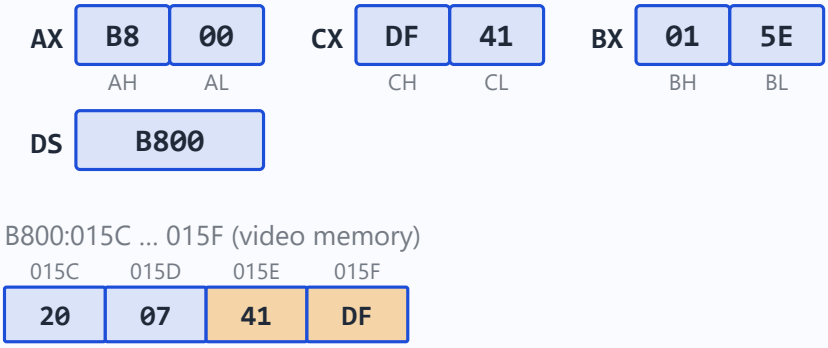


Figure 4.1.17: Step 6: CX = DF41h is written as a word; [015Eh] = 41h ('A'), [015Fh] = DFh (color). Physical address B815Eh.

In step 7, RET takes the 0000h at the top of the stack into IP and SP = 0000h. The `INT 20h` at 0700:0000 ends the program. The letter stays on the screen, because video memory is independent of the program. DS remaining B800h causes no problem either; after the program ends, DOS sets up its own registers again.

4.1.7.3 Text-mode video memory

In the 80 × 25 text mode the screen consists of 25 rows and 80 columns. Each cell holds 2 bytes in video memory: first the character, then the color. A row is 80 × 2 = 160 bytes and the whole screen takes 25 × 160 = 4000 bytes. If rows and columns are counted from 0, the relative address of a cell is found as follows:

> relative address = (row × 80 + column) × 2

The 015Eh in the program comes from this formula: (2 × 80 + 15) × 2 = 175 × 2 = 350 = 15Eh. The letter was written in row 2, column 15 (Figure 4.1.18). The calculation can also be done in reverse. 015Eh = 350 is divided by two: 175. Since 175 = 2 × 80 + 15, it is row 2, column 15.

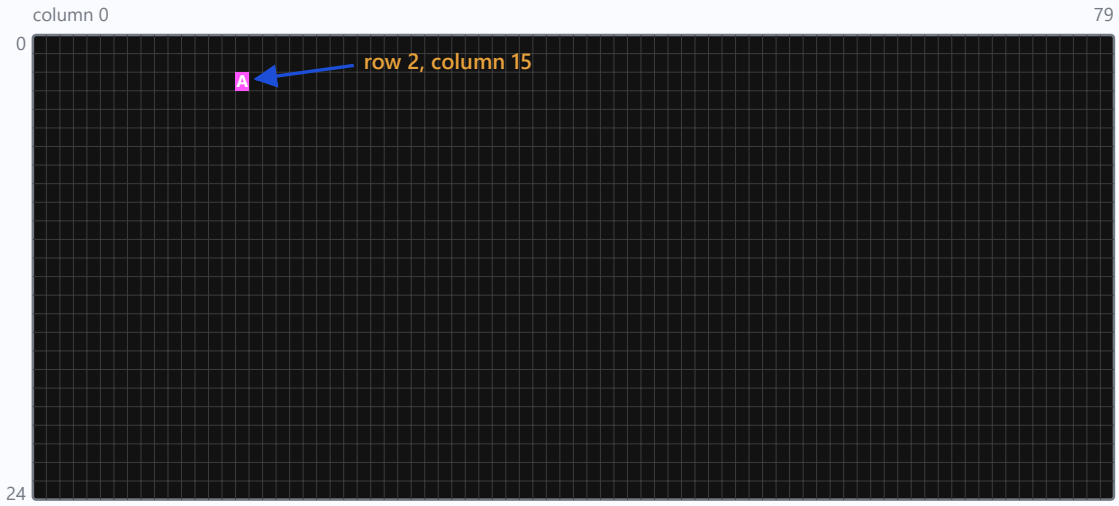


Figure 4.1.18: The 80 × 25 text screen. A bright white 'A' on a magenta background has been written into the cell at row 2, column 15.

The bits of the attribute byte encode the color. DFh = 1101 1111b is read as follows:

- Bit 7 = 1: blinking of the character on a real screen; bright background in emu8086.
- Bits 6–4 = 101: background color magenta (5).
- Bits 3–0 = 1111: foreground color bright white (15).

The foreground color selects one of 16 colors with four bits, and the background color one of 8 colors with three bits. The basic colors are 0 black, 1 blue, 2 green, 3 cyan, 4 red, 5 magenta, 6 brown, 7 light gray; 8–15 are their bright versions, and 14 is yellow. The 1Eh in the earlier example thus means yellow text (1110) on a blue background (001), and 07h gray text on a black background.

Example 4.1.3

What would the code look like that writes a yellow 'X' on a green background at row 10, column 40 of the screen? The color byte is 2Eh.

The relative address is (10 × 80 + 40) × 2 = 840 × 2 = 1680 = 690h. The ASCII code of the character 'X' is 58h. Since the character is in the low byte and the color in the high byte, the word to write is 2E58h. 2Eh = 0010 1110b: background 010 (green), foreground 1110 (yellow).

```
MOV AX, 0B800h
MOV DS, AX
MOV BX, 690h
MOV WORD PTR [BX], 2E58h
```

`WORD PTR` is necessary in the last instruction; otherwise the size of the constant stays ambiguous. The code writes 58h to B8690h and 2Eh to B8691h.

4.1.8 MOV examples

Each of the short MOV examples in the emu8086 documentation shows a single direction of transfer: memory to register, register to memory, register to register, constant to memory. The sections below trace them in order. All of them are COM programs; initially CS = DS = 0700h and IP = 0100h.

4.1.8.1 Memory to register

```
ORG 100h
MOV AX, [1000h]   ; A1 00 10
RET
```

Let the bytes `CD AB` be at DS:1000h beforehand. The instruction reads these two bytes with direct addressing. By the little-endian rule, the CDh at 1000h goes to AL and the ABh at 1001h to AH: AX = ABCDh. The instruction is three bytes; after it executes, IP = 0103h. The physical address is 07000h + 1000h = 08000h. Memory does not change; MOV only copies.

Step	Instruction	AX	IP (after)	[1000h] [1001h]
0	(initial)	0000h	0100h	CD AB
1	`MOV AX, [1000h]`	ABCDh	0103h	CD AB
2	`RET`	ABCDh	0000h	CD AB

Table 4.1.20: Trace of memory to register (DS:1000h = CD AB).

The appearance of this program in the emu8086 window consists of three parts. On the left are the registers; the high and low halves (H, L) of AX, BX, CX, and DX are shown in separate boxes. Initially CS = DS = 0700h, IP = 0100h, SP = FFFEh. In the middle is the memory dump. The physical address 07100h, that is, 0700:0100, holds the three bytes of the instruction (`A1 00 10`). On the right is the disassembled code: `MOV AX, [01000h]` followed by `RET` (`C3`). The *single step* button executes one instruction and the registers that changed are highlighted in color. In the real emu8086 the DS:1000h region is initially zero; so in the window AX = 0000h remains after the instruction runs. The value ABCDh is obtained on the assumption that CD AB was written to memory beforehand.

4.1.8.2 Register to memory

```
ORG 100h
MOV BX, 1234h     ; BB 34 12
MOV [2000h], BX   ; 89 1E 00 20
RET
```

Step	Instruction	BX	IP (after)	[2000h] [2001h]
0	(initial)	0000h	0100h	00 00
1	`MOV BX, 1234h`	1234h	0103h	00 00
2	`MOV [2000h], BX`	1234h	0107h	34 12
3	`RET`	1234h	0000h	34 12

Table 4.1.21: Trace of register to memory (DS:2000h initially 00 00).

The first instruction loads an immediate value into BX. The second writes BX to a direct address: BL (34h) to 2000h and BH (12h) to 2001h. This instruction is four bytes. Because a register other than AX is used, the short `A3` form cannot be chosen. The general opcode `89`, a ModR/M byte, and two address bytes are needed. The ModR/M byte `1E` = 00 011 110 reads as REG = BX, R/M = direct address. IP advances from 0103h to 0107h. BX does not change. The physical address is 07000h + 2000h = 09000h.

The RET in the last step pulls the COM program's return address (0000h) from the top of the stack. IP after RET is therefore 0000h; the program fetches its next instruction from 0700:0000, that is, from the `INT 20h` at the start of the PSP. The last step is like this in all COM examples.

4.1.8.3 XCHG: swapping

Swapping the places of two values is a basic operation of many algorithms, such as sorting. The 8086 has a separate instruction for it.

Definition 4.1.9

The instruction `XCHG destination, source` exchanges the contents of the two operands in a single instruction. One operand can be a register and the other a register or memory. Memory with memory, segment registers, and constants cannot be used. The two operands must be the same size. XCHG does not affect the flags.

```
MOV CX, 10   ; CX = 000Ah
MOV DX, 20   ; DX = 0014h
XCHG CX, DX  ; swap
```

When numbers carry no suffix, they are taken as decimal. After the first instruction CX = 10 = 000Ah, and after the second DX = 20 = 0014h. After XCHG, CX = 0014h and DX = 000Ah. The flags do not change. The machine code of the instruction is `87 CA` : `87` is the word swap, and `CA` = 11 001 010 means REG = CX, R/M = DX.

Step	Instruction	CX	DX
1	`MOV CX, 10`	000Ah	0000h
2	`MOV DX, 20`	000Ah	0014h
3	`XCHG CX, DX`	0014h	000Ah

Table 4.1.22: Swapping two registers.

4.1.8.4 XCHG rules

- Register ↔ register: `XCHG AX, BX`.
- Register ↔ memory: `XCHG AL, [SI]`. The writing order does not matter; `XCHG [SI], AL` is the same instruction.
- There is no memory ↔ memory.
- Segment registers and constants cannot be used. Swapping with a constant is meaningless anyway: nothing can be written into a constant.
- The sizes must be the same; the flags do not change.
- The form `XCHG AX, reg` has a special one-byte code: `90h` + register number. `XCHG AX, BX` = `93`. `XCHG AX, AX` changes nothing and its code is `90`. This byte is also the `NOP` (no operation) instruction.

Without XCHG, a temporary register would be needed for the swap:

```
MOV AX, CX    ; AX ← CX (backup)
MOV CX, DX    ; CX ← DX
MOV DX, AX    ; DX ← old CX
```

This way spends three instructions and one more register. The old value of AX is also lost. XCHG is ahead in time too. On the 8086 register-to-register MOV takes 2 periods and XCHG 4; three MOVs take 6 periods. The one-byte swap with AX takes 3 periods. A swap between a register and memory is expensive: 17 periods plus the EA time. The reason is that the processor first reads from memory and then writes to memory; two separate bus cycles are needed.

The use of XCHG with memory has a special importance in multiprocessor systems. Because the read and the write are done in a single instruction, the chance of another processor accessing the same byte in between is narrowed. When used with the `LOCK` prefix, the 8086 keeps the $\overline{\text{LOCK}}$ pin active for the duration of the instruction and does not hand the bus to another bus master. The locks that ensure a resource is used by only one processor at a time are built on this mechanism.

4.1.8.5 Code trace: swapping with XCHG

The program below shows the use of XCHG with memory, and the same swap done with and without XCHG, side by side.

```
ORG 100h
MOV SI, OFFSET num
MOV AL, 5Ah
XCHG AL, [SI]      ; AL ⇔ [SI]
MOV CX, 1111h
MOV DX, 2222h
MOV AX, CX         ; backup
MOV CX, DX
MOV DX, AX         ; swapped
XCHG CX, DX        ; one instruction
RET
num DB 0A5h
```

Step	Instruction	AX	CX	DX	SI	num
1	`MOV SI, OFFSET num`	0000h	0000h	0000h	0116h	A5h
2	`MOV AL, 5Ah`	005Ah	0000h	0000h	0116h	A5h
3	`XCHG AL, [SI]`	00A5h	0000h	0000h	0116h	5Ah
4	`MOV CX, 1111h`	00A5h	1111h	0000h	0116h	5Ah
5	`MOV DX, 2222h`	00A5h	1111h	2222h	0116h	5Ah
6	`MOV AX, CX`	1111h	1111h	2222h	0116h	5Ah
7	`MOV CX, DX`	1111h	2222h	2222h	0116h	5Ah
8	`MOV DX, AX`	1111h	2222h	1111h	0116h	5Ah
9	`XCHG CX, DX`	1111h	1111h	2222h	0116h	5Ah

Table 4.1.23: Trace of the swap with XCHG (num = 0116h, initially A5h).

In step 1, SI takes the relative location (0116h) of the variable `num`. In memory `num` = A5h. In step 2, AL = 5Ah. The `XCHG AL, [SI]` (`86 04`) in step 3 swaps a register with a memory byte: AL = A5h, `num` = 5Ah. No intermediate register was needed. This single instruction contains both the bus cycle that reads memory and the one that writes to it.

Steps 4 and 5 put recognizable values in CX and DX. Steps 6–8 do the same swap without XCHG. In step 6, a backup of CX is taken into AX; the previous value of AX (00A5h) is lost. In step 7, CX = 2222h. At this moment the old value of CX is held only in AX. In step 8, DX = 1111h and the swap is complete. Three instructions and one register have been spent. The single `XCHG CX, DX` in step 9 undoes the swap: CX = 1111h, DX = 2222h. AX is not touched in the process.

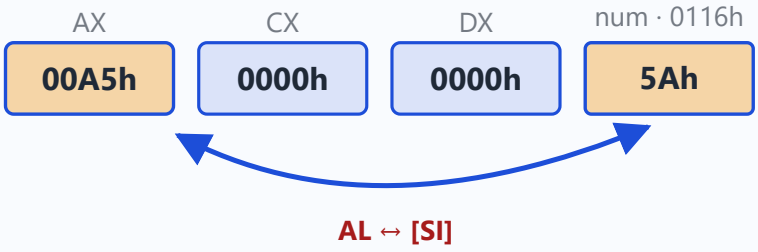


Figure 4.1.19: Step 3: `XCHG AL, [SI]` swaps a register and a memory byte in one instruction; AL = A5h, num = 5Ah.

4.1.8.6 Register to register

```
MOV SI, 3000h  ; SI = address
MOV DI, SI     ; DI = SI
MOV AL, [SI]   ; DS:3000h
MOV [DI+1], AL ; DS:3001h
```

This example shows what copying between registers that hold addresses means. Let DS:3000h hold 5Ah. The first instruction assigns a relative location (3000h) to SI. The second, `MOV DI, SI`, makes DI = 3000h. What is copied is the address, not the data in memory. Now two registers point to the same cell. The third instruction reads 5Ah with `[SI]`: AL = 5Ah. The fourth writes it to `[DI+1]` = 3001h. In memory, 3000h–3001h is now `5A 5A`.

Step	Instruction	Machine code	SI	DI	AL	[3000h] [3001h]
1	`MOV SI, 3000h`	`BE 00 30`	3000h	0000h	00h	5A ??
2	`MOV DI, SI`	`8B FE`	3000h	3000h	00h	5A ??
3	`MOV AL, [SI]`	`8A 04`	3000h	3000h	5Ah	5A ??
4	`MOV [DI+1], AL`	`88 45 01`	3000h	3000h	5Ah	5A 5A

Table 4.1.24: Trace of register to register (DS:3000h = 5Ah).

In C this is like making two pointers equal: `q = p;` followed by `q[1] = p[0];`. The pointers themselves are copied, not the data they point to.

4.1.8.7 Writing a constant to memory

```
ORG 100h
MOV BYTE PTR [4000h], 65  ; 'A'
MOV WORD PTR [4002h], 65
RET
```

Let DS:4000h–4003h initially be `00 99 00 99`. The first instruction writes 65 as a single byte. 65 = 41h is the ASCII code of the letter 'A'. The machine code is `C6 06 00 40 41` and takes five bytes: the opcode, ModR/M, two address bytes, and one data byte. IP becomes 0105h. 4001h does not change.

The second instruction writes the same constant with `WORD PTR`. 65 is now the word 0041h; 41h goes to 4002h and 00h to 4003h. The 99h at 4003h is erased. The instruction is six bytes, `C7 06 02 40 41 00`, and IP advances to 010Bh. The constant has no size of its own; the PTR specifier determines it. The number 65 in the source is the same in both instructions, but it enters the machine code once as one byte and once as two.

Step	Instruction	IP (after)	4000h 4001h 4002h 4003h
0	(initial)	0100h	00 99 00 99
1	`MOV BYTE PTR [4000h], 65`	0105h	**41** 99 00 99
2	`MOV WORD PTR [4002h], 65`	010Bh	41 99 **41 00**
3	`RET`	0000h	41 99 41 00

Table 4.1.25: Trace of writing a constant to memory (DS:4000h = 00 99 00 99).

4.1.8.8 Working with 8-bit halves

Each of AX, BX, CX, and DX has two 8-bit halves that can be used with separate register names: AH and AL, BH and BL, and so on. Writing to one half does not affect the other half. This makes it possible to hold two small values in a single register and to process the bytes of a word separately.

Copying the low byte. In the code below, AL is copied to BL.

```
MOV AX, 1234h   ; AH = 12h, AL = 34h
MOV BX, 0FFFFh
MOV BL, AL      ; BL only
```

The first instruction makes AX = 1234h: AH = 12h, AL = 34h. The second fills BX with FFFFh so that the effect can be seen. The third instruction (`8A D8`) changes only BL: BL = 34h. BH = FFh stays in place and BX = FF34h. Copying from a 16-bit register to an 8-bit register (`MOV BL, AX`) is invalid; the sizes must be the same.

Adding to the high byte. When arithmetic is done on the halves, the carry is seen not to pass from one half to the other.

```
MOV AX, 00FFh   ; AH = 00h, AL = FFh
ADD AH, 01h     ; AH only
ADD AL, 01h     ; AL only
MOV AX, 01FFh
ADD AX, 1       ; 16 bits
```

Step	Instruction	Machine code	AX	CF	ZF	SF	AF	PF
1	`MOV AX, 00FFh`	`B8 FF 00`	00FFh	0	0	0	0	0
2	`ADD AH, 01h`	`80 C4 01`	01FFh	0	0	0	0	0
3	`ADD AL, 01h`	`04 01`	0100h	1	1	0	1	1
4	`MOV AX, 01FFh`	`B8 FF 01`	01FFh	1	1	0	1	1
5	`ADD AX, 1`	`05 01 00`	0200h	0	0	0	1	1

Table 4.1.26: Adding to the high and low byte (flags: CF, ZF, SF, AF, PF).

In step 2, AH = 00h + 01h = 01h and AX = 01FFh. The result is not zero and there is no carry. Since 01h has an odd number of 1 bits, PF = 0. In step 3, AL = FFh + 01h is computed. The 8-bit result is 00h and a carry comes out as the ninth bit: CF = 1, ZF = 1. A carry also comes out of the low four bits, so AF = 1. There are zero 1s in 00h; zero is an even number and PF = 1. The carry does not pass to AH: AX = 0100h. This is an 8-bit addition and AH is not part of the operation.

In step 4, AX is set to 01FFh again. Because MOV does not change the flags, the flags of the previous step (CF = 1, ZF = 1) remain as they were. In the 16-bit addition of step 5, 01FFh + 1 = 0200h. This time the carry has passed from AL to AH, because the operation is 16 bits. Since no carry comes out of 16 bits, CF = 0. AF is 1 again; F + 1 in the low four bits produces a carry. PF looks only at the low byte of the result: PF = 1 for 00h. The instruction `ADD AX, 1` is encoded in the short AX-specific form as `05 01 00`.

Common mistake

Thinking the carry of an 8-bit addition passes to the other half of the register. `ADD AL, 1` with AL = FFh does not increment AH; it only makes CF = 1. If a 16-bit value is to be added to in 8-bit pieces, `ADC` (add with carry) is used on the high half: `ADD AL, 1` followed by `ADC AH, 0`.

4.1.8.9 Assignment with number bases

In assembly source code, constants can be written in different bases. Whichever base is used, the assembler converts the number to a binary value and writes it into the machine code. The program below uses three bases together.

```
mov al, 00000101b ; binary
mov bl, 0ah       ; hex
mov cl, 10o       ; octal
add al, bl        ; 5 + 10 = 15
sub al, cl        ; 15 - 8 = 7
```

Step	Instruction	Machine code	AL	BL	CL	CF	PF
1	`mov al, 00000101b`	`B0 05`	05h	00h	00h	0	0
2	`mov bl, 0ah`	`B3 0A`	05h	0Ah	00h	0	0
3	`mov cl, 10o`	`B1 08`	05h	0Ah	08h	0	0
4	`add al, bl`	`02 C3`	0Fh	0Ah	08h	0	1
5	`sub al, cl`	`2A C1`	07h	0Ah	08h	0	0

Table 4.1.27: Trace of assignment with number bases.

The binary number 00000101b is 5 and AL = 05h. The hexadecimal number 0Ah is 10; a 0 is written in front because it begins with a letter. The octal number 10o is 1 × 8 + 0 = 8 and CL = 08h. In the machine codes the constants are now independent of base: `05`, `0A`, `08`. In step 4, AL = 5 + 10 = 15 = 0Fh. 0000 1111b has four 1 bits; four is even, so PF = 1. In step 5, AL = 15 − 8 = 7 = 07h. Since no borrow is needed, CF = 0 stays. 0000 0111b has three 1s and PF = 0.

4.1.8.10 Writing numbers

Table 4.1.28 summarizes the writing rules of constants in assemblers compatible with emu8086 and MASM. The decimal value of all the examples is 25; only 0FFh and the character row differ.

Base	Suffix	Example	Decimal value
Decimal	none or `d`	`25`, `25d`	25
Binary	`b`	`00011001b`	25
Octal	`o` or `q`	`31o`	25
Hexadecimal	`h`	`19h`, `0FFh`	25, 255
Character	`' '`	`'A'`	65 (41h)

Table 4.1.28: Number bases and their suffixes.

If a hexadecimal number begins with one of the letters A–F, a 0 is put in front: `0Ah`, `0B800h`. This rule is so that the assembler can tell a number from a name. If `Ah` is written, it is confused with the register AH, and if `B800h` is written, with a label or variable name. The assembler assumes that every name begins with a letter and every number with a digit. emu8086 also accepts the C-style form `0x1000`. For octal numbers, because the letter `o` resembles zero, some programmers prefer `q`.

Example 4.1.4

How is the number 200 written in binary, octal, and hexadecimal? Do `MOV AL, 200` and `MOV AL, 0C8h` produce the same machine code?

Since 200 = 128 + 64 + 8, the binary is 11001000b. With groups of three bits, 11 001 000 = 3 1 0, that is, 310o; with groups of four bits, 1100 1000 = C 8, that is, 0C8h. A 0 is put in front because C is a letter. Both instructions produce `B0 C8`; the base only changes how the source code reads.

4.1.9 Common mistakes

Most errors in memory access do not give an assembly error. The program runs but reads from the wrong address or corrupts the wrong byte. These errors are found in emu8086 by checking the effective address and physical address in the memory window. Table 4.1.29 collects the most frequent ones.

Wrong	What happens?	Correct
`MOV SI, arr`	SI is loaded with the value of the first element, not the address of the array.	`MOV SI, OFFSET arr` or `LEA SI, arr`
`INC SI` (word array)	Reads from the middle of a word; the halves of two elements combine.	`ADD SI, 2`
variable with `MOV AL, [BP]`	Reads from the SS segment; in an EXE program the wrong byte arrives.	`[BX]`, `[SI]`, or `DS:[BP]`
`MOV [BX], 5`	Size is ambiguous; the assembler reports an error or guesses.	`MOV BYTE PTR [BX], 5`
`MOV AX, B800h`	B800h is taken for a label name.	`MOV AX, 0B800h`
Changing DS and leaving it	All the following variable accesses go to another segment.	`PUSH DS` … `POP DS`, or ES
`MOV AX, [0200h]` → expecting 3412h	Little-endian was forgotten: if memory has 34 12, AX = 1234h.	low address = low byte

Table 4.1.29: Common mistakes in memory access.

The first row depends on the assembler's syntax. In MASM and emu8086, a name defined with data is treated as a variable. The notation `MOV SI, arr` means the same as `MOV SI, [arr]` and reads the first word of the array. SI then points to an unrelated place. For the address, the `OFFSET` operator or the `LEA` (load effective address) instruction is used. In some assemblers such as NASM, on the other hand, a bare name always means an address and square brackets are required for the contents. This difference must not be overlooked when moving from one source to another.

Common mistake

Forgetting to shift the segment when computing a physical address. For DS = 2000h, BX = 0150h the physical address is 20000h + 0150h = 20150h, not 2000h + 0150h = 2150h. In the reverse mistake the offset is shifted: 2000h + 1500h = 3500h. The segment is always written with a 0 appended on its right, and the offset stays as it is.

Common mistake

Carrying an effective address overflow into 20 bits. With BP = FFFEh, the EA for `[BP+4]` is 0002h, not 1 0002h. The effective address always wraps in 16 bits; the overflowing bit is not added to the segment value. The segment:offset sum, on the other hand, wraps at 20 bits.

4.1.10 Exercises and solutions

The six exercises in the slides are solved here with their intermediate steps.

1. With DS = 3000h, BX = 0010h, SI = 0005h, what are the effective and physical addresses of the operand `[BX+SI+20h]`?

There is no BP in the address; the segment is DS. EA = 0010h + 0005h + 0020h = 0035h. The physical address is 3000h × 10h + 0035h = 30000h + 0035h = 30035h.

2. With SS = 4000h, BP = FFFEh, what are the effective and physical addresses of the operand `[BP+4]`?

EA = FFFEh + 0004h = 1 0002h; it wraps in 16 bits and EA = 0002h. Since BP is present, the segment is SS: 40000h + 0002h = 40002h.

3. Which of the notations `[BX+BP]`, `[DI+SI]`, `[BX+DI+300]`, `[DX]` are valid?

Only `[BX+DI+300]` is valid: one base, one index, and one d16. `[BX+BP]` contains two base registers and `[DI+SI]` two index registers. DX cannot be used in address calculation at all.

4. If [0500h] = 78h and [0501h] = 56h, what is AX after `MOV AX, [0500h]`? What is AL after `MOV AL, [0501h]`?

When reading a word, the low address is the low byte: AL = 78h, AH = 56h, AX = 5678h. The second instruction reads only the byte at 0501h: AL = 56h.

5. What does the code look like that writes a yellow 'X' on a green background at row 10, column 40 of the screen? The color byte is 2Eh.

The solution is given in detail in Example 4.1.3. The relative address is (10 × 80 + 40) × 2 = 1680 = 690h; the word to write is 2E58h.

```
MOV AX, 0B800h
MOV DS, AX
MOV BX, 690h
MOV WORD PTR [BX], 2E58h
```

6. How is the program that prints 'A' on the screen written with ES without changing DS?

B800h is loaded into ES instead of DS and the `ES:` prefix is added to the write instruction. DS stays in the program's segment (0700h).

```
MOV AX, 0B800h
MOV ES, AX
MOV CL, 'A'
MOV CH, 11011111b
MOV BX, 15Eh
MOV ES:[BX], CX      ; 26 89 0F
```

The machine code of the write instruction is its unprefixed form (`89 0F`) with `26` added in front. At the end of the program B815Eh = 41h, B815Fh = DFh, and DS = 0700h stays.

4.1.11 Summary

- **Addressing**: a memory operand has the form `[base + index + displacement]`. The base is BX or BP, the index SI or DI, and the displacement d8 or d16; each is optional. BX and BP, or SI and DI, cannot be used together. These rules give 24 combinations and are encoded in the MOD and R/M fields of the ModR/M byte.
- **Mode types**: the immediate and register modes do not go to memory. The direct, register indirect, based, indexed, and based-indexed modes compute an effective address. String instructions use DS:SI and ES:DI implicitly; I/O instructions go to a separate port space, and jumps to an address relative to IP.
- The **effective address** is computed in 16 bits and wraps: with BP = FFFEh, `[BP+4]` → 0002h. d8 is sign-extended. EA calculation takes 5–12 clock periods on the 8086; a segment prefix adds 2.
- **Segment choice**: SS if BP is in the address, otherwise DS. A segment prefix (`ES:` = 26h, `CS:` = 2Eh, `SS:` = 36h, `DS:` = 3Eh) changes this for a single instruction. CS:IP, SS:SP, and the string destination ES:DI cannot be changed.
- **Physical address** = segment × 10h + EA; 1234:7890 → 19BD0h. At most 4096 segment:offset pairs lead to one physical address. *n* address lines select 2^n locations; the 8086's 20 lines give 1 MB, and the 16-bit offset gives the 64 KB segment.

- **Size**: `byte ptr` / `word ptr` (`b.` / `w.` in emu8086) specify the access size; if the other operand is a register, the size is known from it. The 8086 is **little-endian**: the low byte at the low address. In a word array the byte offset is index × 2.
- **MOV** copies and does not affect the flags. There is no MOV from memory to memory, from a constant to a segment register, or from segment to segment; CS and IP cannot be destinations.
- **XCHG** swaps two operands in one instruction; segment registers, constants, and two memory operands cannot be used. It does not affect the flags.
- **Video memory** is in the B800h segment. Each cell is a character + attribute byte; relative address = (row × 80 + column) × 2.

4.1.12 Key terms

addressing mode immediate addressing register addressing direct addressing
register indirect addressing based / indexed addressing relative addressing base register
index register displacement sign extension effective address (EA) ModR/M byte segment offset
physical address paragraph (16 bytes) address bus default segment segment override prefix
program segment prefix (PSP) .COM program data type specifier byte / word
little-endian / big-endian row-major order exchange (XCHG) string I/O port
video memory (text mode) attribute byte clock cycle bus cycle

---

4.1.13 Test Yourself

1. With ES = 5000h and DI = FFF0h, from which physical address does the instruction `MOV AL, ES:[DI+20h]` read?

   Answer

   EA = FFF0h + 0020h = 1 0010h; it wraps in 16 bits and EA = 0010h. The prefix selects ES: 50000h + 0010h = 50010h. Without the wraparound, 60010h would have been found; this address is outside the ES segment.

2. How many bytes is the instruction `MOV AL, [BP+SI+5]` and which segment does it access? What is the ModR/M byte?

   Answer

   The segment is SS, because BP is in the address. Since the displacement 5 is in the range −128…127, d8 is chosen: MOD = 01. REG = 000 (AL), R/M = 010 (`[BP+SI]`). ModR/M = 01 000 010 = 42h. The instruction is `8A 42 05`, that is, 3 bytes.

3. Starting at DS:0300h, memory holds the bytes `78 56 34 12`. What are AX and BL after `MOV AX, [0301h]` and `MOV BL, [0303h]`? What happens on the data bus in the first instruction?

> **Answer**
>
> AX: the 56h at 0301h goes to AL and the 34h at 0302h to AH; AX = 3456h. BL = 12h. 0301h is an odd address. The 8086 reads a word starting at an odd address with two separate bus cycles; the result is the same, but the access takes 4 clock periods longer.

4. What is AX when the following program finishes?

```
org 100h
mov bx, offset t
mov si, 3
mov al, [bx+si]
mov ah, [bx+si-2]
ret
t db 10h,20h,30h,40h,50h
```

> **Answer**
>
> `[bx+si]` is byte number 3 of t (counting from 0): AL = 40h. `[bx+si-2]` is byte number 1: AH = 20h. AX = 2040h. Since the program is 12 bytes long, t is at 010Ch; the result is independent of this address.

5. Physical address 45678h is to be reached with DS:SI. What are the smallest and largest values DS can take? How many different pairs are there?

> **Answer**
>
> Largest DS: the first four digits, 4567h; SI = 45678h − 45670h = 0008h. Smallest DS: DS × 10h ≥ 45678h − FFFFh = 35679h, that is, DS ≥ 3567.9h → 3568h; SI = 45678h − 35680h = FFF8h. The number of pairs is 4567h − 3568h + 1 = 1000h = 4096.

6. What is the capacity of a memory chip with 18 address lines and 8 data lines? What if it had 18 address lines and 4 data lines?

> **Answer**
>
> 2^{18} = 262,144 locations; each location is 1 byte → 256 KB. With 4 data lines each location is 4 bits: 262,144 × 4 bits = 1,048,576 bits = 128 KB.

7. Which of the following are accepted by an 8086 assembler? `MOV ES, [BX]` · `MOV [SI], DS` · `MOV`

`AX, [DX]` · `XCHG [SI], [DI]` · `MOV BYTE PTR [DI], 300`

Answer

The first two are valid: a segment register can be loaded from memory (`8E 07`) and written to memory (`8C 1C`). `[DX]` is invalid; DX is not an address register. XCHG cannot take two memory operands. 300 does not fit in a byte (at most 255); the assembler reports an error.

8. What is the relative address in video memory of the last cell of the 80 × 25 text screen (row 24, column 79)? At which physical address is this cell's color byte?

Answer

(24 × 80 + 79) × 2 = 1999 × 2 = 3998 = 0F9Eh. The character is at B8000h + 0F9Eh = B8F9Eh and the color byte at the next address, B8F9Fh. Since the whole screen is 4000 bytes, the last byte is 0F9Fh.

9. How many clock periods do the instructions `MOV AX, [BX+SI+6]` and `MOV AX, [BX+DI+6]` take on the 8086? What changes if the EA comes out as an odd address?

Answer

MOV from memory to a register is 8 periods plus the EA time. For `[BX+SI+d]` the EA is 11 and for `[BX+DI+d]` 12 periods: 19 and 20 periods. If the EA is odd, the word is read in two bus cycles and 4 periods are added to the time: 23 and 24 periods.

4.1.14 Further reading

- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 3 (Addressing Modes): data, program, and stack addressing modes with examples from the 8086 to the Pentium; it is the source closest to the addressing part of these notes.
- Intel, *8086/8088 User's Manual: Programmer's and Hardware Reference*: ModR/M encoding, the segment prefix, and the instruction timing table (including the EA times) are found here at first hand.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), Chapter 4: data transfer, little-endian, OFFSET, PTR, and indirect addressing; real-mode examples.
- The emu8086 documentation, *8086 assembler tutorial for beginners*, Part 2: Memory Access and MOV instruction; it is the source of the MOV examples in the slides.

4.1.15 References

1. The emu8086 documentation, *8086 assembler tutorial for beginners*, part 2: memory access, MOV instruction.
2. Intel, *8086/8088 User's Manual*, addressing modes and instruction encoding.
3. B. B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, Chapter 3 (Addressing Modes).
4. Documentation of IBM PC text-mode video memory (B800h) and the attribute byte.
5. K. R. Irvine, *Assembly Language for x86 Processors*, 7th edition, Pearson.

PART II · CHAPTER 4.2

Variables

A program's data is, in the end, nothing but bytes in memory. The 8086 reaches these bytes with 20-bit physical addresses; but working with an address such as `5A73:235B` is both hard and error-prone for the programmer. Assembly language therefore lets us give names to memory locations: variables. These notes first describe how variables are defined with the DB, DW, and DD directives. They then show what the assembler does when it assigns them relative locations and in what order multi-byte values are written to memory. Next they extend the same idea to arrays, strings, and tables; treat the difference between a variable's value and its address through LEA and OFFSET; and introduce EQU constants, which occupy no memory at all, and the location counter. The last section traces, line by line, small programs that move, add, and compare data with variables and process 32-bit values. At each step it is shown how the instruction changes the registers, flags, and memory, and what happens on the 8086's buses.

LEARNING OUTCOMES

Define variables
Define byte, word, and double-word variables and arrays with DB, DW, DD, `?`, and DUP; apply the naming rules and value ranges.

Work out the memory layout
Find at which relative location, and in which byte order, every instruction and every variable of a source program is placed; explain the effect of ORG 100h, the PSP, and alignment.

Work with arrays
Compute the byte offset of an element in byte and word arrays, strings, and two-dimensional tables, and access the element by index.

Separate the address from the value
Say what LEA, OFFSET, and a variable name load into a register; use pointer variables.

Use constants
Use EQU constants and the location counter (`$`) to write values that occupy no memory and update themselves automatically.

Trace programs
Trace step by step, at the register, flag, and memory level, programs that move, add, and compare data between variables and process 32-bit values.

4.2.1 Variables

The 8086's memory is a long strip of one million bytes numbered from 00000h to FFFFFh. To reach a byte on this strip, the processor produces a 20-bit physical address from a segment register and an offset. Machine language knows only these numbers. A program that stores a counter, a sum, or a text in memory might have to write the data's address at every use. Such a program is very hard to read and to change. Moving a datum one byte over would mean correcting, one by one, every instruction in which that address appears.

Assembly language solves this problem with names. The programmer gives a memory location a meaningful name; the assembler takes over the job of computing the address and substituting it at every use. This section describes how named memory locations, that is, variables, are defined and how the assembler and the processor see them.

4.2.1.1 What is a variable?

Definition 4.2.1

A **variable** is a data area in memory bound to a name, whose value can change while the program runs. A variable has three properties: its **name**, its **address** in memory (its relative location within the segment), and its **type**, which determines the number of bytes it occupies.

For the programmer, using the name `var1` is much easier than using the address `5A73:235B`. The name exists only in the source code. When the assembler translates the source file into machine code, it replaces every name with the relative location it has reserved for that variable. No name remains in the running program; the processor sees only the 16-bit relative location embedded in the instruction. The concept of a variable therefore lives not in the hardware but in the assembler.

The assembler supports two basic types. The **BYTE** type is 8 bits and is defined with `DB` (*define byte*). The **WORD** type is 16 bits, a word of the 8086, and is defined with `DW` (*define word*). A third directive, `DD` (*define double word*), reserves a 32-bit, that is, 4-byte, double word. The general form of a definition is as follows:

```
name  DB  value
name  DW  value
name  DD  value
```

The following two lines define two variables, of one byte and two bytes:

```
counter DB 10h     ; BYTE
sum     DW 0FFFFh  ; WORD
```

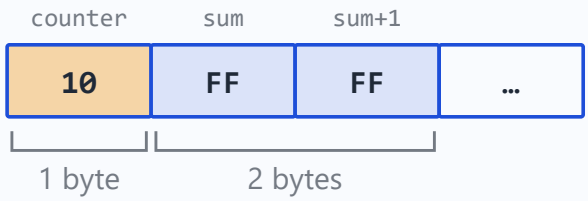


Figure 4.2.1: `counter` occupies a single byte and holds 10h. `sum` occupies two bytes: FFh and FFh at the addresses `sum` and `sum+1`.

Figure 4.2.1 shows the memory counterpart of these two definitions. The assembler reserves one byte for `counter` and writes 10h into it. Right after it, it reserves two bytes for `sum`. The name `sum` stands for the address of the first of these two bytes. The second byte is reached with `sum+1`. No gaps are left between variables: definitions are laid out consecutively in memory in the order they are written in the source.

Definition 4.2.2

A **directive** is a source line that gives information to the assembler and is not itself translated into a processor instruction. `ORG`, `END`, and `EQU` produce no bytes. `DB`, `DW`, and `DD`, on the other hand, produce not instructions but **data** bytes: they write the given initial values into memory, into the program.

This distinction is important. A line such as `MOV AL, counter` is a machine instruction that the processor will execute. `counter DB 10h` is nothing to be executed; it only puts the value 10h into the program's byte image. When the program is loaded from disk into memory, this byte is loaded too and becomes the variable's initial value. There is no need to assign an initial value to the variable separately.

4.2.1.2 Naming rules and the initial value

A variable name is a label and is written into the assembler's symbol table. It therefore has to obey some rules:

- A name **starts with a letter**; letters and digits may follow. MASM and emu8086 also accept the underscore and a few special characters (`_`, `@`, `$`, `?`).

- Register names (`AX` , `SI` ...) and instruction names (`MOV` , `ADD` ...) are reserved words; they cannot be variable names.
- The assembler does not distinguish upper and lower case: `VAR1` , `var1` , and `Var1` are the same name.
- The same name cannot be defined twice. This rule applies to constants and labels too.

A data definition does not have to have a name. The line `DB 0Ah` reserves a byte in memory and writes 0Ah into it; this byte has no name but it has an address. It can be reached by adding a number to the address of the preceding named variable. This is why, in multi-line tables, naming only the first line is enough.

The initial value can be written in any number form: decimal (`10`), hexadecimal (`0Ah`), binary (`00001010b`), octal (`12o`), or a character in quotes (`'A'`). The assembler converts all of them to the same binary pattern. For variables whose initial value does not matter, `?` is written. `?` means “the initial value of this area is undefined”; emu8086 puts 00 in these bytes, but it is wrong for a program to rely on that value.

Definition	Result	Explanation
`num1 DB 5`	valid	a name that starts with a letter, one byte, value 05h
`1num DB 5`	invalid	the name starts with a digit
`ax DW 0`	invalid	a register name cannot be a variable name
`DB 0Ah`	valid	nameless; reserves a byte, has an address
`result DW ?`	valid	a word with no initial value; 00 00 in emu8086
`letter DB 'A'`	valid	the character's ASCII code: 41h

Table 4.2.1: Valid and invalid definitions.

4.2.1.3 Value ranges

A byte has 8 bits, that is, $2^8 = 256$ different patterns. These patterns can be interpreted in two ways. In the **unsigned** interpretation the patterns are the numbers 0 to 255. In the **signed** interpretation the two's-complement representation is used: if the top bit is 1 the number is negative, and the range is −128 to 127. For words and double words the same logic applies to 16 and 32 bits.

Directive	Size	Unsigned	Signed	Example
`DB`	1 byte	0 … 255	−128 … 127	`x DB -1` → FF
`DW`	2 bytes	0 … 65.535	−32.768 … 32.767	`y DW -2` → FE FF
`DD`	4 bytes	0 … 4.294.967.295	-2^{31} … $2^{31} - 1$	`z DD 1` → 01 00 00 00

Table 4.2.2: Sizes and value ranges of the data-definition directives.

When accepting an initial value, the assembler looks at the union of the two ranges. For `DB`, every number from −128 to 255 fits in a byte. When `x DB -1` is written, the assembler computes the 8-bit two's complement of −1: the inverse of 1 is 1111 1110, plus 1 gives 1111 1111 = FFh. Had `x DB 255` been written, FFh would again sit in memory. The two definitions produce the same byte in memory. `DB 300`, on the other hand, does not fit in a byte, because 300 = 12Ch needs nine bits; the assembler reports an error on this line and does not silently truncate the value.

Definition 4.2.3

In memory only the **bit pattern** is stored; the information about whether a byte is signed or unsigned is not in memory. The meaning of the pattern is determined by the **instruction** that uses it: `JAE`, `JB`, and CF interpret it as unsigned; `JGE`, `JL`, `CBW`, and OF interpret it as signed.

This definition prevents a common confusion when working with variables. Writing `DB -1` does not tell the assembler "this variable is signed." The programmer only chooses the initial pattern. The same FFh byte can be processed as 255 in one instruction and as −1 in the next. The trace below shows this step by step.

4.2.1.4 Code trace: is FFh 255 or −1?

```
ORG 100h
MOV AL, x        ; FFh
ADD AL, 1        ; 255 + 1 ?
MOV AL, x
CBW              ; signed: -1
MOV BX, y        ; FFFEh
NEG BX           ; -(-2)
RET
x DB -1
y DW -2
```

The program starts at 0100h. The seven instructions take 16 bytes in total (`A0 10 01`, `04 01`, `A0 10 01`, `98`, `8B 1E 11 01`, `F7 DB`, `C3`); that is why `x` lands at 0110h and `y` at 0111h. In memory `x` = FFh, and `y` is the bytes FE FF in little-endian order, that is, FFFEh.

The line `MOV AL, x` is, in the assembler's eyes, `MOV AL, [0110h]`. The machine code is `A0 10 01`: the opcode A0 means "load into AL the byte at the 16-bit relative location that follows," and the bytes 10 01 are 0110h. It is useful to remember that the execution unit (EU) does not itself access memory. The EU takes the relative location from the queue and hands the read job to the bus interface unit (BIU). The BIU computes the physical address as DS × 10h + 0110h and performs a read bus cycle. If

emu8086's load segment is 0700h, the physical address is 07000h + 0110h = 07110h. Since this address is even, the byte comes over the low half of the data bus (D0–D7). The instruction takes 10 clock periods. As a result AL = FFh and AX = 00FFh. `MOV` does not change the flags.

`ADD AL, 1` processes the pattern with an unsigned view. 1111 1111 + 0000 0001 = 1 0000 0000; the ninth bit does not fit in a byte. AL = 00h is left and the bit that spills out is written to the carry flag: CF = 1. Since the result is zero, ZF = 1. With the signed view the same addition is −1 + 1 = 0 and is correct; since there is no signed overflow, OF = 0 remains. A single addition writes the results of both interpretations into the flags at once: CF reports unsigned overflow and OF signed overflow. The program decides which to look at.

`MOV AL, x` makes AL FFh again; the flags keep their previous values. `CBW` (*convert byte to word*) takes AL as a **signed** number and extends it to 16 bits: it copies the top bit of AL into the eight bits of AH. The top bit of AL = FFh is 1; AH = FFh. AX = FFFFh, which is −1 in 16 bits. This operation is called sign extension. Had AL been thought of as unsigned 255, the correct 16-bit equivalent would be 00FFh; `CBW` does not make this choice, it always extends as signed. `CBW` takes only 2 clock periods and does not change the flags.

`MOV BX, y` reads the two bytes together: the FEh at 0111h goes to the low byte (BL) and the FFh at 0112h to the high byte (BH). BX = FFFEh, which is 65,534 in the unsigned view and −2 in the signed view. Because `y` sits at an odd address, the 8086 reads this word with two separate bus cycles. `NEG BX` takes the two's complement of the number, that is, computes 0 − BX: 0000h − FFFEh = 0002h. Because the operand is nonzero, `NEG` sets CF = 1. The result is not zero, so ZF = 0.

Line	AX	BX	CF	ZF	OF	Interpretation
2 `MOV AL, x`	00FFh	0000h	0	0	0	AL: 255 or −1
3 `ADD AL, 1`	0000h	0000h	1	1	0	unsigned overflow: CF
4 `MOV AL, x`	00FFh	0000h	1	1	0	flags preserved
5 `CBW`	FFFFh	0000h	1	1	0	sign extension: −1
6 `MOV BX, y`	FFFFh	FFFEh	1	1	0	65,534 or −2
7 `NEG BX`	FFFFh	0002h	1	0	0	0 − (−2) = 2

Table 4.2.3: Trace of the value-ranges program (each row is the state after the instruction executes).

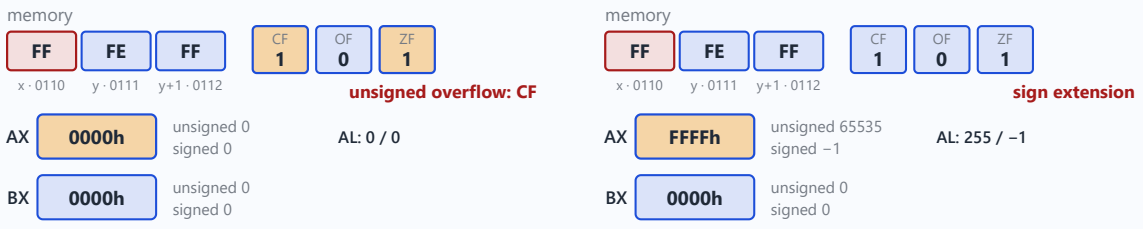


Figure 4.2.2: On the left, after `ADD AL, 1` : AL = 00h, and the unsigned overflow is reported by CF = 1. On the right, after `CBW` : AL = FFh is taken as signed −1 and extended to AX = FFFFh.

Table 4.2.3 and Figure 4.2.2 show two faces of the same program. The two variables in memory never changed; what changed is the interpretation of the instructions that read the patterns. `ADD` and CF treated the pattern as unsigned, `CBW` and `NEG` as a signed number.

> **Common mistake**
>
> It is wrong to assume that a variable defined with `x DB -1` will behave like a negative number from then on. The assembler converts `-1` only into the pattern FFh and records the type as BYTE; it does not keep the sign information. If this variable is compared with `JAE`, it behaves like 255.

4.2.1.5 The variable's type and size checking

The assembler also writes a **type** next to every variable name. A name defined with DB has type BYTE (1 byte), one defined with DW has type WORD (2 bytes), and one defined with DD has type DWORD (4 bytes). The type determines how many bytes of the variable an instruction reads or writes, and provides a safety check at assembly time.

The rule is simple: if the other operand in the instruction is a register, the register's size and the variable's type must match. `MOV AX, w` is valid, because both AX and w are 16 bits. `MOV AL, w` is invalid; the assembler reports a "1 byte / 2 bytes" mismatch and stops assembly. If the other operand is a constant, the variable's type determines how many bits of the constant are written: `MOV w, 5` writes two bytes, 05 00, to memory.

Memory operands that contain no variable name have no type. `[BX]` says only an address; it does not say whether there is a byte or a word at that address. So `MOV [BX], 5` is ambiguous and gives an error. The programmer writes the size explicitly with `BYTE PTR` or `WORD PTR`: `MOV BYTE PTR [BX], 5`. The same specifier also serves to change a variable's type **for that instruction only**: `MOV AL, BYTE PTR w` reads the low byte of the word variable. The variable's definition and type do not change.

Instruction	Result
`MOV AL, w`	error: operand sizes do not match (1 byte / 2 bytes)
`MOV AL, BYTE PTR w`	AL = 34h (low byte)
`MOV AL, BYTE PTR w+1`	AL = 12h (high byte)
`MOV w, 5`	w = 0005h; 05 00 is written to memory
`MOV AX, b`	error: operand sizes do not match (2 bytes / 1 byte)
`MOV [BX], 5`	error: size is ambiguous; BYTE PTR or WORD PTR is needed
`MOV b, w`	error: there is no memory-to-memory MOV
`INC b`	b = 06h; only one byte is incremented

Table 4.2.4: Type-checking examples (`w DW 1234h`, `b DB 5`).

Type checking is done only at assembly time. The processor knows no types while running. The size of an instruction is encoded in the machine code with a bit: in most 8086 opcodes the lowest bit (the **W bit**) means the operation is on a byte if 0 and on a word if 1. For example `MOV AL, [address]` is encoded with A0 and `MOV AX, [address]` with A1. The assembler chooses the right W bit by looking at the variable's type. Had an access of the wrong size got through assembly, the processor would have executed it without objection. The size error the assembler gives therefore works in the programmer's favor: it catches, right from the start, an error that produces a wrong value and is hard to notice.

The error in the line `MOV b, w`, on the other hand, is not a type error but a hardware rule. None of the 8086's data-movement instructions takes two memory operands. The instruction format has only one ModR/M field, and this field can define at most one memory operand. A memory-to-memory move is always done through a register with two instructions.

4.2.1.6 Two names for the same memory with LABEL

Sometimes the same memory area is to be processed both byte by byte and as a word. Writing `BYTE PTR` or `WORD PTR` at every access is possible but not readable. The `LABEL` directive instead gives a second name and type to the same address.

> **Definition 4.2.4**
>
> The directive `name LABEL type` reserves no memory. It gives a new name, of the stated type (`BYTE`, `WORD`, `DWORD`), to the address of the first byte that follows it.

```
ORG 100h
MOV AX, wval       ; WORD
MOV BL, bytes      ; low
MOV BH, bytes+1    ; high
MOV bytes, 0FFh    ; one byte
MOV CX, wval       ; word
RET
wval LABEL WORD
bytes DB 34h, 12h
```

The six instructions fill the range 0100h to 0114h; `RET` is at 0114h. Since `wval LABEL WORD` produces no bytes, both `wval` and `bytes` point to address 0115h. In memory 34h sits at 0115h and 12h at 0116h.

`MOV AX, wval` uses a name of type WORD, so it reads both bytes at once: AX = 1234h. The assembler encodes this line as `A1 15 01`. `MOV BL, bytes` uses the BYTE-type name of the same address and reads only 34h; there is no need to write PTR. `MOV BH, bytes+1` reads the second byte: BH = 12h. BX

= 1234h. The same result as AX is the natural outcome of the little-endian order: the byte at the low address goes to the low half.

In the line `MOV bytes, 0FFh` the destination's type is BYTE. The assembler produces the 5-byte instruction `C6 06 15 01 FF`, and the processor writes only to address 0115h. Memory is now FF 12. The last read, `MOV CX, wval`, takes the same two bytes as a word: CX = 12FFh. A change made through the byte name is seen through the word name, because the two names point to a single memory location.

Line	AX	BX	CX	[0115h] [0116h]
2 `MOV AX, wval`	1234h	0000h	0000h	34 12
3 `MOV BL, bytes`	1234h	0034h	0000h	34 12
4 `MOV BH, bytes+1`	1234h	1234h	0000h	34 12
5 `MOV bytes, 0FFh`	1234h	1234h	0000h	FF 12
6 `MOV CX, wval`	1234h	1234h	12FFh	FF 12

Table 4.2.5: Trace of the LABEL program.

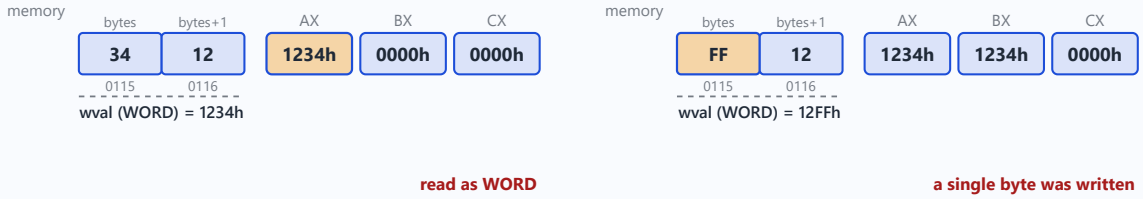


Figure 4.2.3: On the left `MOV AX, wval` : the two bytes are read as a WORD. On the right `MOV bytes, 0FFh` : because of the BYTE type only 0115h changes; wval is now 12FFh.

`LABEL` is used in programs that process a buffer both as bytes and as words. For example, a serial-port buffer is filled byte by byte but compared two bytes at a time. The program size grows only by the two bytes of the `DB` line; the `LABEL` line leaves no trace in memory.

4.2.2 Memory layout

Defining a variable means asking the assembler for space in memory. Knowing exactly where this space is, is indispensable when tracing a program and debugging. Only someone who knows the layout can read in a memory dump which byte belongs to which variable, and which number in an instruction's machine code is an address. This section follows how a small program turns into bytes from start to finish.

4.2.2.1 A sample program and its byte image

```
ORG 100h
MOV AL, var1   ; AL ← var1
MOV BX, var2   ; BX ← var2
RET            ; end the program
VAR1 DB 7      ; BYTE, 7
var2 DW 1234h  ; WORD, 1234h
```

The layout of the program is deliberately chosen: code first, then data. Since the variables sit after the `RET`, the processor never reaches them. Execution ends at `RET`; the bytes behind it are read only as data. In the source the variable's name is written once as `VAR1` and once as `var1`. Since the assembler is case-insensitive, the two are the same variable.

Figure 4.2.4: The program's memory image. Machine code between 0100h and 0107h, `VAR1` at 0108h, `var2` at 0109h–010Ah. The variable names have turned into relative locations inside the instructions.

The assembler translates the source file into the 11-byte sequence in Figure 4.2.4. The first instruction `MOV AL, var1` is three bytes: `A0 08 01`. A0 is the short opcode that loads a byte from a direct address into AL, and the two bytes that follow are the address itself: 0108h. The second instruction `MOV BX, var2` is four bytes: `8B 1E 09 01`. The opcode 8B means "load a word register from memory," the byte 1E (ModR/M) says the destination is BX and the address is given directly in the instruction, and 09 01 is 0109h. The third instruction is the single-byte `C3` (`RET`). Then comes the data: 07 at 0108h, 34 at 0109h, and 12 at 010Ah.

Two points stand out in this image. First, there is no name in the machine code; `var1` and `var2` have given way to the numbers 0108h and 0109h. Second, even the addresses are written little-endian: 0108h sits in the instruction in the order 08 01. The little-endian order applies not only to data but to every 16-bit number embedded in an instruction.

Definition 4.2.5

In **little-endian** order, the **low byte of a multi-byte value goes to the low address** and the high byte to the next address. The 8086 is little-endian: `DW 1234h` sits in memory as 34 12, and `DD 12345678h` as 78 56 34 12.

This order matches the 8086's hardware. The low half of the 16-bit data bus (D0–D7) carries the even-addressed byte and the high half (D8–D15) the odd-addressed byte. When a word starting at an even address is read, the low byte automatically arrives in the low half of the register and the high byte in the high half. No byte swapping is needed.

4.2.2.2 Code trace: var1 and var2

When tracing the program step by step, seeing IP advance by the length of the instruction after each instruction is the easiest way to verify the byte image.

Line	Machine code	IP	AL	BX	Explanation
1 `ORG 100h`	—	0100h	00h	0000h	initial state after loading
2 `MOV AL, var1`	A0 08 01	0103h	07h	0000h	[0108h] is read
3 `MOV BX, var2`	8B 1E 09 01	0107h	07h	1234h	[0109h] and [010Ah] are read
4 `RET`	C3	0000h	07h	1234h	0000h popped from the stack; INT 20h in the PSP

Table 4.2.6: Trace of the sample program. IP is the value after the instruction executes.

Figure 4.2.5: On the left, after `MOV AL, [0108h]` : AL = 07h, IP = 0103h. On the right, after `MOV BX, [0109h]` : 34h goes to the low byte and 12h to the high byte, BX = 1234h, IP = 0107h.

In the first step the processor executes the three bytes at 0100h. The BIU computes the physical address as DS × 10h + 0108h and fetches 07h with a single read bus cycle. AL = 07h, IP = 0103h. In the second step the address is 0109h, which is odd. The low byte of the word is at 0109h and the high byte at 010Ah. These two bytes do not fall on the same half of the data bus. The 8086 first reads 0109h from the high half and then 010Ah from the low half, and arranges the two bytes in the right order in BX. The result is BX = 1234h; IP advances four bytes to 0107h.

The last instruction is `RET` . When a .COM program starts, DOS puts the value 0000h on the stack: SP = FFFEh, [FFFEh] = 0000h. `RET` takes this word into IP and increases SP by 2: IP = 0000h, SP = 0000h. At relative location 0000h of the segment is the PSP header in front of the program, and its first two bytes are `CD 20` , that is, `INT 20h` . This interrupt terminates the program and hands control back to DOS. That is why IP in the RET row of the trace table is 0000h, not 0108h. The data byte at 0108h is never executed as an instruction.

The trace in the slides assumes the program is loaded into segment 0B56h: DS = CS = 0B56h, and the full address of `VAR1` is 0B56:0108. emu8086, on the other hand, loads .COM programs into segment 0700h; the full address of the same variable there is 0700:0108. The two notations do not contradict.

The segment value depends on where DOS loads the program in memory and may change on every run. The relative locations, however, are fixed at assembly time. Since the program uses only relative locations, it works the same wherever it is loaded.

4.2.2.3 What does the assembler do?

While translating source code into a byte sequence, the assembler does two jobs together. It turns instructions into machine code and replaces variable names with **relative locations**. To be able to do this, it keeps a counter while reading the source file from start to finish.

> **Definition 4.2.6**
>
> The **location counter** is the counter in which the assembler holds the relative location where the next byte will be placed. `ORG` sets it to a given value; every instruction and every data definition increases it by the number of bytes it produces. In the source it is read with the symbol `$`.

In the sample program `ORG 100h` makes the counter 0100h. Since `MOV AL, var1` produces three bytes, the counter becomes 0103h; `MOV BX, var2` produces four bytes and the counter is 0107h; after `RET` it is 0108h. When the assembler reaches the `VAR1` definition, it writes the counter's current value, 0108h, into the **symbol table** as this name's address. Since `DB 7` produces one byte, the counter becomes 0109h, and `var2` takes this address.

There is a problem here. The line `MOV AL, var1` comes **before** the `VAR1` definition. When the assembler reaches this line, it does not yet know the address of `VAR1`. The solution is to read the source twice. In the first pass only the number of bytes each line will take is computed and the symbol table is filled. In the second pass, since all the addresses are known, the machine code is produced. This is why variables can be defined after the place where they are used in the source.

A variable's **relative location** is its distance in bytes from the start of the segment. The relative location of `VAR1` is 0108h. The full (logical) address is written with the segment value: 0B56:0108. The physical address is 0B56h × 10h + 0108h = 0B560h + 0108h = 0B668h. `var2` is at 0109h, its full address is 0B56:0109, and since it is of type WORD it occupies 0109h and 010Ah.

When .COM programs are loaded, DOS gives all four segment registers the same value: CS = DS = ES = SS. Code and data therefore sit in the same segment, and there is no need to load DS separately to access a variable. The relative location in the instruction finds the right place in the segment DS points to.

4.2.2.4 The ORG 100h directive and the .COM program

The `ORG` (*origin*) directive tells the assembler the starting value of the location counter. `ORG 100h` declares that the first byte of the program will be loaded at relative location 0100h of the segment. The assembler takes this value into account when replacing variable names with addresses. If `ORG` is forgotten, the counter starts at 0000h and every variable address is computed 100h lower than its real place. The program assembles without an error but reads the wrong bytes. That is why emu8086 assumes 100h for .COM sources in which `ORG` is not written. `ORG` is a directive; it is not translated into machine code and produces no bytes.

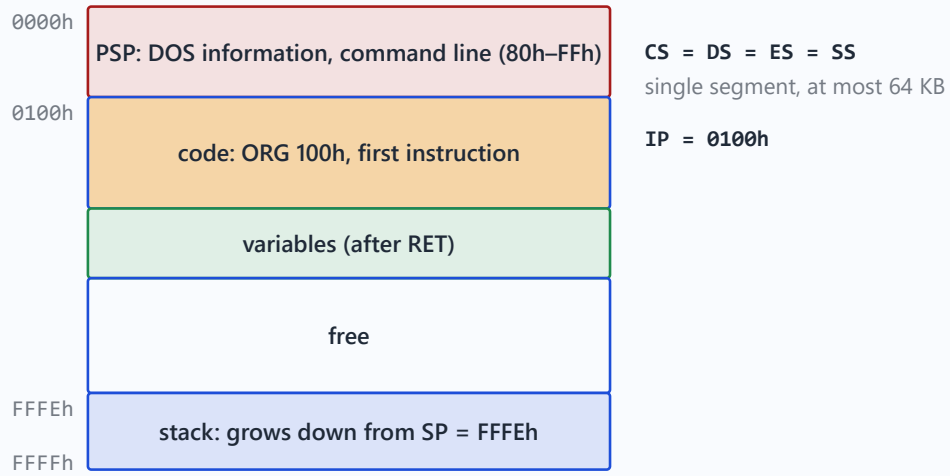


Figure 4.2.6: The layout of a .COM program in its single 64 KB segment. The PSP at 0000h–00FFh, code and variables from 0100h, and at the top the stack growing downward from FFFEh. The four segment registers carry the same value.

Why 100h? When loading a .COM program, DOS places a header called the **PSP** (program segment prefix) in the first 256 bytes of the segment (0000h–00FFh). The PSP holds information about the program: at 0000h the `INT 20h` that ends the program, at 80h the length of the command line, and between 81h and FFh the command-line parameters. The program starts right after this header, at 0100h. The stack is at the very top of the segment: it starts with SP = FFFEh and grows downward. Code, data, and stack share a single 64 KB segment (Figure 4.2.6).

The layout of .EXE programs is different. The code segment of an .EXE file starts at relative location 0000h and a separate data segment is used for the variables. Since each segment can grow up to 64 KB, large programs are written in the .EXE format. In this course most of the examples are written with emu8086's .COM template; a single segment makes it easier to follow the addresses of the variables.

4.2.2.5 Alignment and EVEN

The address at which a word variable starts does not affect the correctness of the program, but it affects its speed. The reason is the 8086's memory interface.

Figure 4.2.7: A word at an even address is read in a single bus cycle, a word at an odd address in two bus cycles. The 16-bit data bus carries two bytes together only if they start at an even address.

The 8086's memory is divided into two banks. Even-addressed bytes sit in the low bank and are connected to the data lines D0–D7. Odd-addressed bytes sit in the high bank and are connected to D8–D15. The processor signals which bank it selects with two signals: if the address line A0 is 0 the low bank is active, and if the $\overline{\text{BHE}}$ (*bus high enable*) signal is 0 the high bank is active. For a word starting at an even address, A0 = 0 and $\overline{\text{BHE}}$ = 0 are given at the same time; the two bytes arrive together in a single bus cycle.

In a word starting at an odd address, the two bytes are on different lines. The byte at 0201h is in the high bank, and the byte at 0202h is in the low bank of the next line. The 8086 notices this by itself and performs two bus cycles: first it reads 0201h from the high half, then 0202h from the low half. The program still gets the right value; only, because each cycle takes four clock periods, 4 periods are lost per word access.

Example 4.2.1

The instruction `MOV AX, w` takes 10 clock periods on the 8086; if `w` is at an odd address it takes 14. If this instruction is executed 1000 times in a loop, the difference is 1000 × 4 = 4000 periods. On a 5 MHz 8086 one period is 1 / 5 MHz = 200 ns. The total loss is 4000 × 200 ns = 800,000 ns = 0.8 ms.

To ensure aligned placement, the `EVEN` directive is used in MASM and TASM. If the location counter is odd, `EVEN` rounds it up to an even number by increasing it by one and puts a padding byte in between; if the counter is already even, it does nothing. In the code segment the padding is usually the byte `90h` (`NOP`). The definitions do not move; they only shift.

```
a  DB 5          ; 0200h
   EVEN          ; 0201h: pad
w  DW 1234h      ; 0202h (even)
```

`a` lands at 0200h and the counter becomes 0201h. `EVEN` rounds the counter up to 0202h and puts a padding byte at 0201h. `w` starts at an even address; memory holds 05, padding, 34, 12. Alignment is not needed for byte variables, because a single byte is read in a single bus cycle at every address. On the 8088, whose data bus is 8 bits, every word needs two bus cycles anyway; there alignment makes no difference.

4.2.2.6 Where do variables go?

The processor does not distinguish whether the bytes in memory are code or data. Whatever byte IP points to, it decodes as an instruction. Where in the program the variables are written is therefore important: data bytes that fall on the execution path are taken for instructions. There are three common layouts.

Place	Example	Points to watch
At the end of the code	`RET` · `x DB 5`	Execution ends at RET and never reaches the data. The most commonly used layout in .COM.
At the start of the code	`JMP start` · `x DB 5` · `start:`	JMP skips the data; if forgotten, the data bytes are executed as instructions.
Separate data segment (.EXE)	`.DATA` · `x DB 5`	`MOV AX, @data` · `MOV DS, AX` is mandatory at the start of the program.

Table 4.2.7: Where variables go in the program.

The first layout is used in most of the programs in these notes. In the second layout the data are at the start of the program and the first instruction is a `JMP`. This layout has one advantage: the addresses of the variables are not affected by changes in the code. However the code grows or shrinks, the data right after the `JMP` always stay at the same relative location. For example, after a 2-byte short `JMP` the first variable is always at 0102h. If the `JMP` is forgotten, the processor decodes the data bytes at 0100h as instructions. For example, the bytes 0A 00 of `DW 10` are executed as `OR AL, [BX+SI]`; the program does meaningless things without giving an error.

The third layout is specific to .EXE programs. Code sits in the segment CS points to and data in the segment DS points to. The relative locations of the variables are computed from the start of the data segment. When the program starts, DOS sets DS not to the program's data but to the PSP. That is why the first two instructions of an .EXE program load DS. `@data` is the segment value of the data segment. Since an immediate value cannot be loaded into a segment register, the value is first taken into AX.

Common mistake

Forgetting the lines `MOV AX, @data` · `MOV DS, AX` in an .EXE program. `MOV AL, x` uses the right relative location but reads from the wrong segment, the one where the PSP is. Neither the assembler nor the processor reports an error; the program just gets a wrong value. In .COM programs this problem does not arise, because CS = DS.

4.2.3 Arrays

Single variables are enough for small programs. But for a hundred measurements read from a sensor, the letters of a text, or a conversion table, defining a hundred separate names is both impossible and pointless. Data of this kind is stored as consecutive bytes of the same type and referred to by a single name: an array.

4.2.3.1 The array concept

Definition 4.2.7

An **array** is a set of elements of the same type placed in consecutive locations in memory. The name of the array stands for the address of the first element; the later elements follow right behind it. An element is reached by adding the element's **byte offset** to the address of the array.

An array can be thought of as a chain of variables. If several values separated by commas are written after a `DB` or `DW` directive, the assembler lays them all out in order with no gaps between them. A text in quotes also turns into a byte array automatically: for each character its ASCII code is written.

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

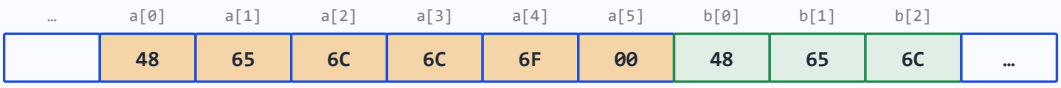


Figure 4.2.8: The arrays `a` and `b` sit one after the other in memory. The two definitions produce the same bytes: 48h 'H', 65h 'e', 6Ch 'l', 6Ch 'l', 6Fh 'o', and the string terminator 00h.

The two definitions in Figure 4.2.8 are written differently but produce the same six bytes in memory. `a` gives the numbers in hexadecimal; `b` writes the text in quotes and the assembler converts each letter to its ASCII code. The 0 at the end is a terminator and shows where the string ends. As far as memory is concerned, there is no difference between a “text” and a “number array”; both are byte arrays.

4.2.3.2 Code trace: accessing array elements

```
ORG 100h
MOV AL, a[3]   ; a + 3
MOV SI, 0
MOV BL, a[SI]  ; a + 0
INC SI
MOV BH, a[SI]  ; a + 1
MOV a[SI], 'E' ; a[1] ← 45h
RET
a DB 48h,65h,6Ch,6Ch,6Fh,0
```

The program begins with `ORG 100h`: the code is loaded from 0100h. The seven instructions take 21 bytes; the array `a` starts at 0115h. An array element is accessed in two ways: with a constant index or with a register.

`MOV AL, a[3]` is a constant-index access. The number inside the square brackets is not the element number but the **byte offset**: `a[3]` means "add 3 to the address of a." Since the offset is known at assembly time, the assembler does the addition itself: 0115h + 3 = 0118h. The instruction turns into `MOV AL, [0118h]` with a direct address and is encoded as `A0 18 01`. The byte at 0118h is 6Ch; AL = 6Ch, that is, 'l'. The processor does no addition while running.

`MOV SI, 0` clears the index register. `MOV BL, a[SI]` is an access with a register: the address is computed at run time as `a + SI`. This time the machine code is `8A 9C 15 01`. The ModR/M byte 9Ch says the address is "SI + a 16-bit displacement," and the displacement is the array's address, 0115h. The EU computes the effective address as 0115h + SI = 0115h and the BIU reads that byte: BL = 48h ('H').

`INC SI` increments the index; SI = 1 now points to the second element. The `MOV BH, a[SI]` instruction, in the same form, this time reads 0116h: BH = 65h ('e'). BX = 6548h. Here the basic idea of a loop is seen: the instruction stays the same, only the index register changes. A loop can scan the whole array with the same instruction by incrementing SI on each iteration.

`MOV a[SI], 'E'` writes into the array. Since the destination `a` is of type BYTE, there is no size ambiguity; the assembler produces the instruction `C6 84 15 01 45` and a single byte is written: [0116h] = 45h. The array is now "HEllo." `'E'` is a character constant in quotes and, to the assembler, no different from the number 45h.

Line	Effective address	AL	SI	BX	Array
2 `MOV AL, a[3]`	0118h	6Ch	0000h	0000h	H e l l o 0
3 `MOV SI, 0`	—	6Ch	0000h	0000h	H e l l o 0
4 `MOV BL, a[SI]`	0115h	6Ch	0000h	0048h	H e l l o 0
5 `INC SI`	—	6Ch	0001h	0048h	H e l l o 0
6 `MOV BH, a[SI]`	0116h	6Ch	0001h	6548h	H e l l o 0

Line	Effective address	AL	SI	BX	Array
7 `MOV a[SI], 'E'`	0116h	6Ch	0001h	6548h	H **E** l l o 0

Table 4.2.8: Trace of the array-access program (`a` = 0115h).

Figure 4.2.9: On the left the first step: `a[3]` = a + 3 bytes, AL = 6Ch ('l'). On the right the last write: `a[1]` ← 45h; the array becomes "HEllo."

The two access paths have different costs in hardware. `MOV AL, [0118h]` takes 10 clock periods. `MOV BL, a[SI]`, on the other hand, computes the effective address as "displacement + index"; this calculation adds 9 periods on the 8086, and the instruction takes 8 + 9 = 17 periods. A constant-index access is faster but not flexible; an index register lets each iteration access a different element.

4.2.3.3 Access forms and DUP

All forms of accessing an array element come down to the same effective address calculation. On the 8086 only **BX, BP, SI, and DI** can be used inside square brackets; these four registers can work as memory index registers. `MOV SI, 3` · `MOV AL, a[SI]` and `MOV AL, a[3]` read the same byte. The assembler treats the following notations as equivalent:

```
a[SI]  =  [a + SI]  =  [SI + a]  =  [SI]+a
```

Whichever is written, the machine code is the same: the address is the sum of the index register and a 16-bit displacement. The index is always in **bytes**. In a byte array the offset of the i-th element is i, but in a word array it is 2 × i. Unlike high-level languages such as C, the assembler does not multiply by the element size for the programmer.

For large arrays it is not necessary to write every value out. The `DUP` (*duplicate*) operator repeats a list of values the required number of times.

Definition 4.2.8

The expression `count DUP(value(s))` writes the list of values in the parentheses `count` times in a row. If `?` is written in the parentheses, an area with no initial value is reserved; `DUP` can be nested.

- `c DB 5 DUP(9)` is the same as `c DB 9, 9, 9, 9, 9`: 5 bytes.

- `d DB 5 DUP(1, 2)` is the same as `d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2`: 10 bytes. What is repeated is not a single value but the whole list.
- `buffer DB 100 DUP(?)` reserves a 100-byte area with no initial value. It is the usual way to make room for a line read from the keyboard or a calculation result.
- `table DW 10 DUP(0)` reserves 10 words, that is, 20 bytes, and zeroes them all.

Figure 4.2.10: Bytes produced with `DUP`: `5 DUP(9)` gives five 09s, `5 DUP(1, 2)` ten bytes: the pair 01 02 five times.

Arrays are also defined with `DW`: `w DW 1000h, 2000h` is a number array of two words. Values greater than 255 or less than −128 do not fit in a byte, so such numbers need `DW`. One place where `DW` cannot be used is long texts: `DW 'Hello'` is invalid. A word can hold at most two characters; there is no single meaningful way to split a five-character text into words. Texts are always defined with `DB`.

4.2.3.4 Word arrays

In a word array each element occupies two bytes. The element number and the byte offset then come apart, and the most frequent array mistake appears exactly at this point.

```
ORG 100h
MOV AX, w[0]   ; element 1
MOV BX, w[2]   ; element 2
MOV SI, 4
MOV CX, w[SI]  ; element 3
MOV DX, w[1]   ; careful!
RET
w DW 1111h, 2222h, 3333h
```

The code starts at 0100h and takes 19 bytes; `w` is at 0113h. In memory the bytes 11 11 22 22 33 33 sit from 0113h on. `MOV AX, w[0]` reads the first word: AX = 1111h. The second element is not at `w[1]` but at `w[2]`, because the first element occupies two bytes: BX = 2222h. The general rule is that the offset of the i-th element (counting from 0) is 2 × i. The index of the third element is 2 × 2 = 4: with `MOV SI, 4` · `MOV CX, w[SI]`, CX = 3333h.

The last line `MOV DX, w[1]` is wrong, but the assembler does not notice. Offset 1 is the middle of the first element. The instruction reads the word starting at 0114h: the low byte is the 11h at 0114h and the high byte the 22h at 0115h. DX = 2211h. This value is not any element of the array; it consists of one half each of two neighboring elements. In the same way `w[3]` combines the high byte of 2222h with the low byte of 3333h and gives 3322h. In a word array the index must always be even.

Line	Addresses read	AX	BX	CX	DX	SI
2 `MOV AX, w[0]`	0113h, 0114h	1111h	0000h	0000h	0000h	0000h
3 `MOV BX, w[2]`	0115h, 0116h	1111h	2222h	0000h	0000h	0000h
4 `MOV SI, 4`	—	1111h	2222h	0000h	0000h	0004h
5 `MOV CX, w[SI]`	0117h, 0118h	1111h	2222h	3333h	0000h	0004h
6 `MOV DX, w[1]`	0114h, 0115h	1111h	2222h	3333h	2211h	0004h

Table 4.2.9: Trace of the word-array program (`w` = 0113h).

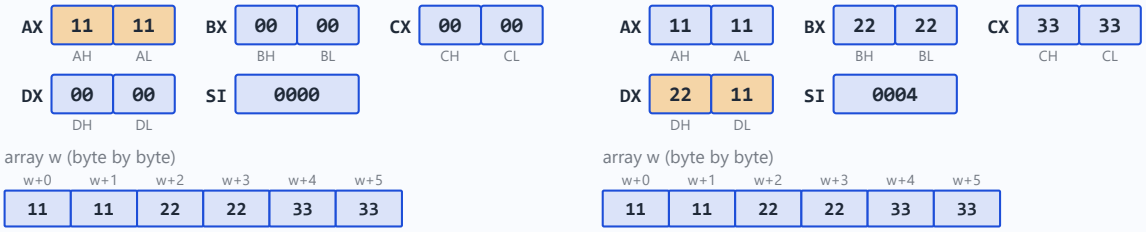


Figure 4.2.11: On the left `w[0]` : AX = 1111h. On the right `w[1]` : the high byte of the first element and the low byte of the second combine, DX = 2211h.

In Table 4.2.9 note that `w` starts at an odd address (0113h). In this layout each of `w[0]`, `w[2]`, and `w[4]` is a word at an odd address, and every read needs two bus cycles. Had an `EVEN` been written before the `DW` line, the array would start at 0114h and every element would be read in a single bus cycle. In that case the wrongly indexed `w[1]` would have fallen on an odd address.

Common mistake

Trying to access the i-th element of a word array with `w[i]`. The third element is not at `w[2]` but at `w[4]`. In loops the index register is incremented by 2, not 1, on each iteration: `ADD SI, 2`.

4.2.3.5 From definition to bytes

Being able to work out on paper which bytes a data definition produces in memory is necessary for reading memory dumps. The rule is always the same: values are written from left to right in order, each value occupies as much space as the type's size, and multi-byte values are in little-endian order. Table 4.2.10 gives the expansion of frequently met definitions.

Definition	Bytes	Size
`b DB 'Hello', 0`	48 65 6C 6C 6F 00	6 bytes
`a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h`	48 65 6C 6C 6F 00	6 bytes
`var2 DW 1234h`	34 12	2 bytes

Definition	Bytes	Size
`d DB 5 DUP(1, 2)`	01 02 01 02 01 02 01 02 01 02	10 bytes
`w DW 3 DUP(?), -1`	?? ?? ?? ?? ?? ?? FF FF	8 bytes
`mydouble DD 12345678h`	78 56 34 12	4 bytes
`x DB 300`	error: 300 does not fit in a byte	—

Table 4.2.10: Byte expansion of data definitions (?? = byte with no initial value; emu8086 writes 00).

The line `w DW 3 DUP(?), -1` has two parts. First three words with no initial value, that is, six bytes, are reserved; emu8086 writes 00 into them. Then −1 is added as a word. The 16-bit two's complement of −1 is FFFFh and is written in little-endian order as FF FF. The total size of the line is 8 bytes. Although the definition has a single name, it has four elements.

4.2.3.6 Strings and terminators

> **Definition 4.2.9**
>
> A **string** is a text made of consecutive character codes in memory. The place where the string ends is shown by a **terminator** byte (0 or '$') or by a length byte at the start.

Memory does not know where a string ends. A program that prints or copies a string has to find the end point by a rule. There are three common rules:

```
s1 DB 'Welcome', 0              ; zero-terminated
s2 DB 'Welcome$'                ; '$'-terminated (INT 21h / AH = 09h)
s3 DB 13, 10, 'Line', 13, 10, '$' ; CR LF: carriage return + line feed
s4 DB 7, 'Welcome'              ; length-prefixed: first byte = 7
s5 DB "Ali's", 0                ; single quote inside double quotes
```

Form	How is the end recognized?	Where is it used?
zero-terminated	when the byte read is 0	emu8086.inc `PRINT_STRING`, the C language
'$'-terminated	when the byte read is 24h	DOS string-printing service
length-prefixed	as many as the number in the first byte	DOS line-input buffer (AH = 0Ah)

Table 4.2.11: String forms.

The space a string occupies in memory is the sum of the number of characters and the terminator. `s1` is seven letters and a zero, that is, 8 bytes. The numbers 13 and 10 in `s3` are not characters but control codes: 13 (0Dh) is carriage return (CR) and 10 (0Ah) is line feed (LF). In DOS, to end a line and go to the start of the line below, the two are written together. `s4` is the length-prefixed form: the

first byte is 7, followed by seven characters. In this form no terminator is needed and the length of the string is learned with a single read. `s5` writes a text containing a single quote with double quotes; the assembler accepts both kinds of quotes.

If the terminator is forgotten, the code that prints the string does not stop. It takes the bytes behind the string for characters too and prints meaningless marks on the screen until it finds a 0 or '$'. A '$'-terminated string cannot print the '$' character itself, because it stops the moment it sees it. The codes of Turkish letters (ç, ğ, ı, ö, ş, ü), on the other hand, depend on the code page in use. These letters are not in the ASCII range 0–127; the same byte may appear as a different letter in a different code page.

4.2.3.7 Character constants and ASCII arithmetic

A character in quotes is only a number to the assembler. `MOV AL, 'A'` and `MOV AL, 41h` produce the same machine code (`B0 41`). The layout of the ASCII table makes it possible to do simple arithmetic on characters.

Characters	ASCII codes
'0' … '9'	30h … 39h
'A' … 'Z'	41h … 5Ah
'a' … 'z'	61h … 7Ah
space · '$'	20h · 24h
CR · LF	0Dh · 0Ah

Table 4.2.12: Frequently used ASCII codes.

The digit characters start at 30h and are in order. To obtain a number's value from a digit character it is therefore enough to subtract '0': '7' = 37h, 37h − 30h = 7. `SUB AL, '0'` does this job. In the reverse direction, to print a number between 0 and 9 on the screen, `ADD AL, '0'` is used. Every digit read from the keyboard is an ASCII code; it goes through this conversion before entering a calculation.

The difference between upper- and lower-case letters is 20h: 'a' = 61h, 'A' = 41h. 20h in binary is 0010 0000; that is, the two letters differ only in **bit 5**. In a lower-case letter bit 5 = 1 and in an upper-case letter it is 0. So to capitalize a letter it suffices to clear bit 5: `SUB AL, 20h` or `AND AL, 0DFh`. The mask 0DFh = 1101 1111 preserves all bits except bit 5. For example 'q' = 71h, 71h AND DFh = 51h = 'Q'. To lowercase a letter, bit 5 is set: `OR AL, 20h`; 'Q' = 51h, 51h OR 20h = 71h = 'q'.

Common mistake

Applying `AND AL, 0DFh` to all the bytes of a string without checking whether they are letters. Digits also have bit 5 = 1: '1' = 31h, 31h AND DFh = 11h and the digit turns into a control code. Before the conversion, the condition 'a' ≤ AL ≤ 'z' is tested.

4.2.3.8 Code trace: converting a string to upper case

The program below converts a zero-terminated string to upper case in place. It does not know the length; it stops when it sees the terminator. The letter check keeps the digits from being corrupted.

```
ORG 100h
LEA SI, s
again:
MOV AL, [SI]
CMP AL, 0          ; terminator?
JE done
CMP AL, 'a'
JB next
CMP AL, 'z'
JA next
SUB AL, 20h        ; 'a' - 'A'
MOV [SI], AL
next:
INC SI
JMP again
done:
RET
s DB 'Ab1c', 0
```

The code is 25 bytes and fills 0100h–0118h; the string starts at 0119h. `LEA SI, s` loads the string's address into SI: SI = 0119h. SI is from then on a pointer to the character being processed. The body of the loop follows the same sequence on every iteration:

1. `MOV AL, [SI]` reads the next byte.
2. `CMP AL, 0` and `JE done`: if the byte is 0, ZF = 1 and the loop ends.
3. `CMP AL, 'a'` and `JB next`: if AL < 61h (CF = 1), the byte is not a lower-case letter; it is passed over without writing.
4. `CMP AL, 'z'` and `JA next`: if AL > 7Ah, it is again passed over.
5. A byte that passes these two checks is a lower-case letter: `SUB AL, 20h` capitalizes it and `MOV [SI], AL` writes it back to the same address.
6. `INC SI` moves on to the next byte and `JMP again` returns to the start.

The comparisons are made with unsigned jumps (`JB`, `JA`), because ASCII codes are unsigned numbers. When 'A' = 41h is compared with 'a' = 61h, the operation 41h − 61h needs a borrow and CF = 1; `JB` is taken.

Round	SI	AL (read)	Decision	Written	String after
1	0119h	41h 'A'	41h < 61h: `JB` taken	—	A b 1 c 0
2	011Ah	62h 'b'	61h ≤ 62h ≤ 7Ah: a letter	42h 'B'	A B 1 c 0
3	011Bh	31h '1'	31h < 61h: `JB` taken	—	A B 1 c 0

Round	SI	AL (read)	Decision	Written	String after
4	011Ch	63h 'c'	letter	43h 'C'	A B 1 C 0
5	011Dh	00h	ZF = 1: `JE done`	—	A B 1 C 0

Table 4.2.13: Round-by-round trace of the upper-case loop (`s` = 0119h).

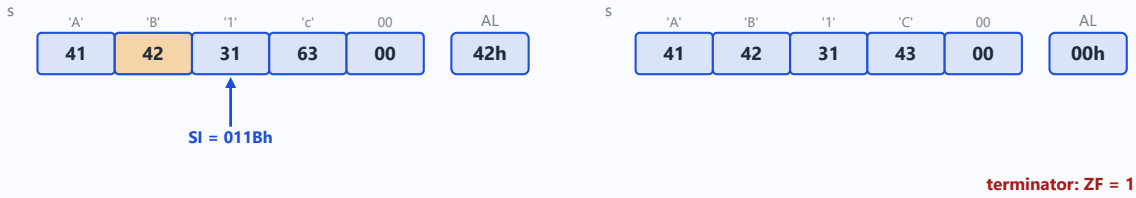


Figure 4.2.12: On the left the second round: 'b' (62h) − 20h = 42h ('B') is written to the same address, SI = 011Bh. On the right the last round: the byte read is 0, `CMP AL, 0` makes ZF = 1 and the loop ends.

At the end of the program SI = 011Dh, AL = 00h, and ZF = 1; the string in memory has become “AB1C.” The loop read five times but wrote only twice. The program used the string's length nowhere; the terminator became the stopping condition instead of a counter. The code works unchanged whether characters are added to or removed from the string. The digit '1' was preserved because it was caught by the letter check; without the check it would have been corrupted by 31h − 20h = 11h.

4.2.3.9 Two-dimensional arrays

A matrix or a table is a two-dimensional structure with rows and columns. Memory, however, is one-dimensional. When a two-dimensional array is placed in memory, the rows are laid out one after another.

Definition 4.2.10

In **row-major order** all the columns of row 0 are placed first, then all the columns of row 1. In a table with S columns, the byte offset of the element [row][column] from the start of the table is (row × S + column) × element size.

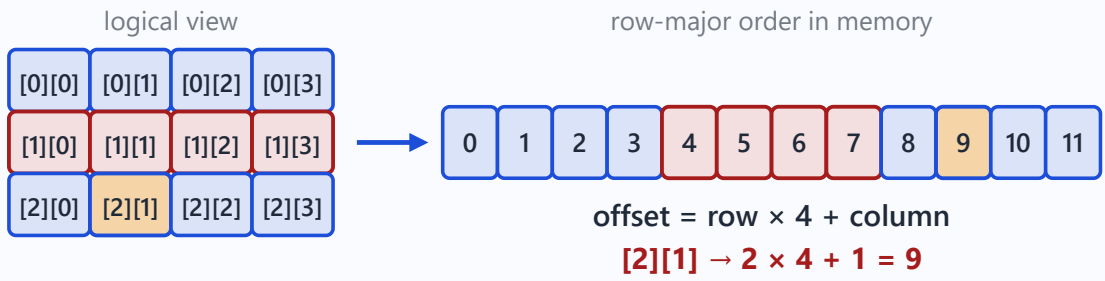


Figure 4.2.13: A byte table of three rows and four columns. The element [2][1] in the logical view is the 2 × 4 + 1 = 9th byte in memory.

```
mat DB 3 DUP(4 DUP(0)) ; 3 × 4
tab DW 3 DUP(4 DUP(0)) ; 24 bytes
```

A nested `DUP` reserves rows and columns together: "3 times (4 times 0)." `mat` is 12 bytes and `tab` 12 words, that is, 24 bytes. No trace of the nested structure is left in memory; there are only 12 or 24 consecutive bytes. What makes the table a table is the formula the program uses when computing the offset.

In the byte table in Figure 4.2.13 the offset of the element [2][1] is 2 × 4 + 1 = 9. In a word table the same calculation is multiplied by 2. For example, the element [1][2] of `tab` starts at byte (1 × 4 + 2) × 2 = 12. The number of rows does not appear in the formula. The number of rows does not affect the layout; it only determines the largest value the row index can take.

If the number of columns is a power of 2, shifting can be used instead of multiplication. For four columns, row × 4 is the same as shifting the row left twice: `SHL AL, 1` twice. On the 8086 an 8-bit `MUL` takes 70–77 clock periods while each `SHL` takes 2. This is why table sizes are so often chosen as powers of 2.

4.2.3.10 Code trace: reading a table element

```
NCOLS EQU 4
ORG 100h
MOV AL, row        ; AL = 2
MOV BL, NCOLS      ; BL = 4
MUL BL             ; AX = 2 × 4
ADD AL, col        ; + 1
MOV SI, AX         ; SI = 9
MOV DL, mat[SI]    ; mat[2][1]
RET
mat DB 11h, 12h, 13h, 14h
    DB 21h, 22h, 23h, 24h
    DB 31h, 32h, 33h, 34h
row DB 2
col DB 1
```

The elements of the table were chosen to carry the row and column numbers: 32h is column 2 (index 1) of row 3 (index 2). The code fills 0100h–0111h; `mat` is at 0112h, `row` at 011Eh, and `col` at 011Fh. Only the first row is named. The next two `DB` lines are nameless and sit right after the first row.

`MOV AL, row` takes the row index from memory: AL = 2. `MOV BL, NCOLS` uses a constant. `NCOLS EQU 4` occupies no memory; the assembler encodes this line as `MOV BL, 4` (`B3 04`). `MUL BL` multiplies AL and BL as unsigned numbers and writes the 16-bit result to AX: AX = 2 × 4 = 0008h. This is the starting offset of row 2 in the table. Since the upper half of the result (AH) is 0, CF = OF = 0.

`ADD AL, col` adds the column index: 8 + 1 = 9. Since AH is already 0, AX = 0009h. The offset must be moved to an address register, because on the 8086 AL or AX cannot be used in addressing; only BX, SI, DI, and BP can be inside square brackets. With `MOV SI, AX`, SI = 0009h. The last instruction `MOV DL, mat[SI]` computes the effective address as 0112h + 9 = 011Bh and reads the byte there: DL = 32h.

Line	AX	BL	SI	DL	Explanation
3 `MOV AL, row`	0002h	00h	0000h	00h	row index
4 `MOV BL, NCOLS`	0002h	04h	0000h	00h	constant embedded in the instruction
5 `MUL BL`	0008h	04h	0000h	00h	2 × 4: start of the row
6 `ADD AL, col`	0009h	04h	0000h	00h	+ column
7 `MOV SI, AX`	0009h	04h	0009h	00h	offset in the index register
8 `MOV DL, mat[SI]`	0009h	04h	0009h	32h	[0112h + 9] = [011Bh]

Table 4.2.14: Trace of the table-element program (`mat` = 0112h).

Figure 4.2.14: On the left, AX = 0008h after `MUL BL`: row 2 starts at byte 8 of the table. On the right, `MOV DL, mat[SI]`: byte 9 is read, DL = 32h.

Because the row and column are read from variables at run time, the same code works for every element of the table. In a word table the final offset would also be multiplied by 2; adding `SHL SI, 1` is enough for that.

The constant was deliberately named `NCOLS`. Had `COL EQU 4` been written, this name would have collided with `col DB 1`, because the assembler is case-insensitive. The assembler reports a "symbol defined twice" error and stops assembly.

4.2.4 Addresses

A variable is associated with two numbers: the **value** stored in it and its place in memory, that is, its **address**. In assembly, when a variable's name is written alone in an instruction, it stands for its value. `MOV AL, VAR1` brings the byte at VAR1's address into AL. Sometimes, however, the address itself is needed, not the value. Scanning an array with a pointer, telling a print service where a string is, or giving a procedure a large block of data without copying it are examples. This section describes two ways to put an address into a register: the `LEA` instruction and the `OFFSET` operator.

4.2.4.1 Getting a variable's address

Definition 4.2.11

`LEA` (*load effective address*) is a processor instruction: it computes the **effective address** (relative location) of a memory operand and loads it into the destination register. It does not access memory. `LEA BX, VAR1` loads the address of VAR1 into BX, and `LEA BX, a[SI]` loads the sum a + SI.

Definition 4.2.12

`OFFSET` is an **assembler operator**: it gives the relative location of a variable as a constant number at assembly time. `MOV BX, OFFSET VAR1` is, in the assembler's eyes, an immediate load such as `MOV BX, 010Dh`.

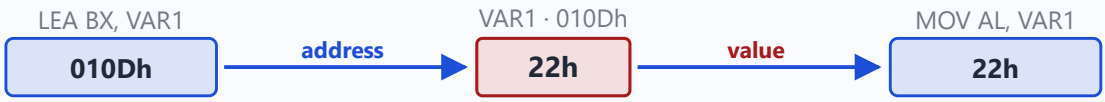


Figure 4.2.15: The address and the value of the same variable. `LEA BX, VAR1` loads the address (010Dh), `MOV AL, VAR1` the value (22h).

Instruction	Loaded into BX	Memory access
`MOV BX, VAR1`	the **value** of VAR1 (contents)	yes: two bytes are read
`LEA BX, VAR1`	the **address** of VAR1	no: computed in the EU only
`MOV BX, OFFSET VAR1`	the **address** of VAR1	no: the address is inside the instruction

Table 4.2.15: What is loaded into the register? (with `VAR1 DW` defined)

A variable's address is very handy for **passing a parameter to a procedure**. Copying the array itself to a subroutine that will process a thousand-byte array is both slow and wasteful of space. Instead only the 16-bit address is passed, for example in SI; the subroutine reaches the array through this address. DOS's string-printing service uses the same idea: the address of the string is expected in DX.

4.2.4.2 Code trace: updating a variable with LEA

```
ORG 100h
MOV AL, VAR1          ; value
LEA BX, VAR1          ; address
MOV BYTE PTR [BX], 44h
MOV AL, VAR1          ; new value
RET
VAR1 DB 22h
END
```

The program's byte image is: `A0 0D 01` (3), `BB 0D 01` (3), `C6 07 44` (3), `A0 0D 01` (3), and `C3` (1). The code is 13 bytes and `RET` is at 010Ch; `VAR1` lands at address 0100h + 13 = 010Dh. emu8086 translates `LEA BX, VAR1`, which contains no base or index register, into the same three-byte code as `MOV BX, OFFSET VAR1`: the opcode `BB` and the 16-bit address. MASM, the real 8086 assembler, encodes the same line as a four-byte LEA in the form `8D 1E 0D 01`.

`MOV AL, VAR1` reads the value of VAR1: AL = 22h. `LEA BX, VAR1`, on the other hand, loads the address of VAR1 into BX: BX = 010Dh. BX is now a pointer to VAR1. `MOV BYTE PTR [BX], 44h` writes through this pointer. Because `[BX]` is a typeless memory operand, `BYTE PTR` is mandatory. The instruction writes 44h to DS:010Dh. The `MOV AL, VAR1` on the last line reads the variable by name again and sees the new value: AL = 44h. The same memory location has been accessed both by name and through a pointer. `END` is the directive that marks the end of the source file and produces no bytes.

Line	AL	BX	[010Dh]	Explanation
2 `MOV AL, VAR1`	22h	0000h	22h	value is read
3 `LEA BX, VAR1`	22h	010Dh	22h	address is loaded, memory is not read
4 `MOV BYTE PTR [BX], 44h`	22h	010Dh	44h	write through the pointer
5 `MOV AL, VAR1`	44h	010Dh	44h	new value is read

Table 4.2.16: Trace of the LEA program (`VAR1` = 010Dh).

Figure 4.2.16: On the left, after `LEA BX, VAR1`, BX = 010Dh and memory holds 22h. On the right, after `MOV BYTE PTR [BX], 44h`, VAR1 = 44h.

In terms of hardware, the difference between `LEA` and `MOV` is in the bus cycle. In `MOV AL, VAR1` the EU gives the address to the BIU, the BIU performs a read bus cycle, and the byte that arrives is written to AL. In `LEA BX, VAR1` there is no read bus cycle: emu8086 puts the address into the instruction as a constant and the EU writes it directly to BX; this form takes 4 periods. In a `LEA` with a register (`LEA BX, a[SI]`) the EU computes the address at run time but does not send the result to the BIU, again writing it directly to the register; its time is 2 clock periods plus the effective address calculation.

4.2.4.3 Are LEA and OFFSET the same?

The same program can also be written with `OFFSET` instead of `LEA` :

```
ORG 100h
MOV AL, VAR1
MOV BX, OFFSET VAR1
MOV BYTE PTR [BX], 44h
MOV AL, VAR1
RET
VAR1 DB 22h
END
```

The result is the same: BX takes the variable's address, the variable becomes 44h, and AL finally reads 44h. `MOV BX, OFFSET VAR1` is an ordinary immediate load: `BB 0D 01`, three bytes. emu8086 translates the line `LEA BX, VAR1` into exactly this code too; the bytes of the two programs are identical and `VAR1` is at 010Dh in both. For a simple variable there is no difference between the two notations. In MASM, however, `LEA BX, VAR1` is encoded as a four-byte real LEA instruction (`8D 1E 0D 01`).

The real difference appears with indexed addresses. `OFFSET` works at assembly time; it can give only constant addresses known to the assembler. `OFFSET a[SI]` is meaningless, because the value of SI is not known at assembly time. `LEA BX, a[SI]`, on the other hand, computes the address **at run time** with SI's value at that moment. More complex expressions such as `LEA SI, [BX+DI+4]` are also possible; `LEA` works here like a small adder that does not change the flags.

	`LEA BX, VAR1`	`MOV BX, OFFSET VAR1`
When is it computed?	for an indexed address at run time, in the EU	at assembly time, in the assembler
Machine code (emu8086)	direct address: BB imm16 (3 bytes); indexed: 8D ModR/M (+ displacement)	BB imm16 (3 bytes)
Time (8086)	direct address: 4 periods; indexed: 2 + EA	4 periods
Indexed address	possible: `LEA BX, a[SI]`	not possible
Flags	unchanged	unchanged

Table 4.2.17: Comparison of LEA and OFFSET.

For a constant address, `MOV ... OFFSET` is both short and fast. When an indexed address has to be computed, `LEA` is used. Whichever is chosen, only BX, SI, DI, and BP can appear inside square brackets.

4.2.4.4 Code trace: LEA a[BX] and OFFSET a

```
ORG 100h
MOV BX, 2
LEA SI, a[BX]      ; SI = a + BX
MOV DI, OFFSET a   ; fixed address
ADD DI, BX         ; add by hand
MOV AL, [SI]
INC BX
LEA SI, a[BX]      ; new address
MOV AH, [SI]
RET
a DB 'ABCD'
```

The code is 22 bytes; the array `a` starts at 0116h and holds the bytes 'A' 'B' 'C' 'D' (41h–44h). `MOV BX, 2` prepares the index. `LEA SI, a[BX]` (`8D B7 16 01`) computes the address at run time: 0116h + 2 = 0118h. Memory is not read; SI only receives the address of 'C'.

`MOV DI, OFFSET a` (`BF 16 01`) loads the constant known at assembly time: DI = 0116h. This constant cannot include BX. A separate addition is needed to reach the same address: with `ADD DI, BX`, DI = 0118h = SI. The two ways gave the same address; LEA did it in a single instruction.

`MOV AL, [SI]` reads the byte at 0118h: AL = 43h ('C'). `INC BX` makes the index 3. The second `LEA SI, a[BX]` produces the new result with the same machine code: 0116h + 3 = 0119h. The instruction did not change; the address changed because BX changed. `MOV AH, [SI]` makes AH = 44h ('D'); AX = 4443h. DI stays at 0118h, because the value that came from OFFSET does not follow the change in BX.

Line	BX	SI	DI	AX
2 `MOV BX, 2`	0002h	0000h	0000h	0000h
3 `LEA SI, a[BX]`	0002h	0118h	0000h	0000h
4 `MOV DI, OFFSET a`	0002h	0118h	0116h	0000h
5 `ADD DI, BX`	0002h	0118h	0118h	0000h
6 `MOV AL, [SI]`	0002h	0118h	0118h	0043h
7 `INC BX`	0003h	0118h	0118h	0043h
8 `LEA SI, a[BX]`	0003h	0119h	0118h	0043h
9 `MOV AH, [SI]`	0003h	0119h	0118h	4443h

Table 4.2.18: Trace of the LEA a[BX] and OFFSET a program (`a` = 0116h).

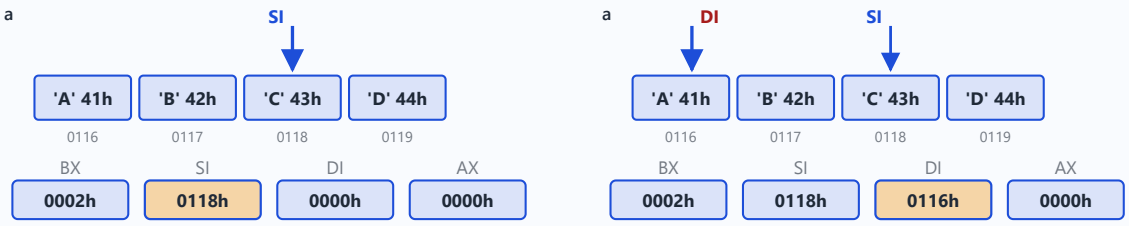


Figure 4.2.17: On the left `LEA SI, a[BX]` : SI = 0116h + 2 = 0118h. On the right `MOV DI, OFFSET a` : DI = 0116h, a fixed address unaware of BX.

4.2.4.5 Pointer variables

An address can be stored not only in a register but also in memory. If the value of a `DW` is a variable name, the assembler writes the **address** of that variable to memory.

Definition 4.2.13

A **pointer variable** is a variable that stores the address of another variable. `p DW arr` defines a 16-bit near pointer, and `far_p DD arr` a 32-bit far pointer made of an offset and a segment.

```
ORG 100h
MOV BX, p          ; 010Bh
LEA SI, p          ; 010Fh
MOV DI, OFFSET arr
RET
arr   DB 'KAYA'    ; 010Bh
p     DW arr       ; 010Fh, holds 010Bh
far_p DD arr       ; offset + segment
```

The code takes 11 bytes (`8B 1E 0F 01` , `BE 0F 01` , `BF 0B 01` , `C3` ; emu8086 translates the line `LEA SI, p` into three bytes like `MOV SI, OFFSET p`). `arr` starts at 010Bh and occupies four bytes; `p` is at 010Fh and holds 010Bh, that is, the bytes 0B 01. `far_p` is at 0111h and takes four bytes: first the offset (0B 01), then the segment value. In emu8086's load segment these bytes are 0B 01 00 07.

Instruction	What is loaded into the register?	Value
`MOV BX, p`	the address inside p (the address of the array)	010Bh
`LEA SI, p`	p's own address	010Fh
`MOV DI, OFFSET arr`	the address of the array, a constant at assembly time	010Bh

Table 4.2.19: Three different loads with a pointer variable.

The first and third rows in Table 4.2.19 give the same number but by different routes. `MOV DI, OFFSET arr` takes the number from inside the instruction and does not change. `MOV BX, p` , on the other hand, reads the number from memory. Because p sits in memory, the program can change it: `ADD p, 2` moves the pointer two bytes further into the array. To use a pointer, it must first be taken into an address register, because the 8086 cannot read memory directly with an address that is in memory: `MOV BX, p` · `MOV AL, [BX]` .

A far pointer is used to point at data in a different segment. The two words in it are loaded in one go with the `LDS` or `LES` instructions. `LDS SI, far_p` writes the low word to SI and the high word to DS; `LES DI, far_p` does the same job for DI and ES. Since `far_p` holds the bytes 0B 01 00 07, after `LDS SI, far_p` SI = 010Bh and DS = 0700h; then `MOV AL, [SI]` reads 'K' (4Bh).

4.2.4.6 Code trace: a pointer in memory

```
ORG 100h
MOV BX, p          ; contents of p
MOV AL, [BX]       ; arr[0]
ADD p, 2           ; advance
MOV BX, p
MOV AH, [BX]       ; arr[2]
RET
arr DB 'KAYA'
p    DW arr        ; address
```

The code is 18 bytes; `arr` is at 0112h and `p` at 0116h with the initial value 0112h. `MOV BX, p` reads the contents of p: BX = 0112h. `MOV AL, [BX]` takes the byte the pointer points to: AL = 4Bh ('K'). `ADD p, 2` is an addition that writes directly to memory. Because the constant is small, the assembler chooses the 5-byte form `83 06 16 01 02`; the opcode 83h sign-extends the 8-bit constant to 16 bits. The processor first reads p, adds 2, and writes the result back. This instruction needs one read and one write bus cycle. p is now 0114h. BX, on the other hand, keeps holding the old address; the register and the pointer in memory are independent of each other.

`MOV BX, p` reloads the pointer: BX = 0114h. `MOV AH, [BX]` makes AH = 59h ('Y'); AX = 594Bh.

Line	BX	AX	p
2 `MOV BX, p`	0112h	0000h	0112h
3 `MOV AL, [BX]`	0112h	004Bh	0112h
4 `ADD p, 2`	0112h	004Bh	0114h
5 `MOV BX, p`	0114h	004Bh	0114h
6 `MOV AH, [BX]`	0114h	594Bh	0114h

Table 4.2.20: Trace of the pointer-in-memory program (`arr` = 0112h, `p` = 0116h).

Figure 4.2.18: On the left `MOV AL, [BX]` : the 'K' that the pointer points to is read. On the right `ADD p, 2` : the pointer in memory points to 0114h; BX stays at the old address.

The address of the array was never written in the code; the pointer in memory decided which element would be read. On the 8086, which has few registers, this is the way to keep the locations of many arrays. At each use the pointer is first taken into BX, SI, or DI.

Common mistake

Confusing the value with the address. `MOV SI, arr` loads not the address of the array but its first word (if arr is DB, it gives a size error). For the address, write `LEA SI, arr` or `MOV SI, OFFSET arr` . For a pointer variable the reverse holds: `MOV SI, p` gives the address, and `LEA SI, p` the pointer's own location.

4.2.5 Constants

Numbers that do not change are common in programs: the number of characters in a row, the number of columns of a table, the number of a port. Writing these numbers by hand at every place they occur creates two problems. The code becomes unreadable, because it is not clear what `80` means. If the value changes, all its uses have to be found and fixed, and it is easy to miss one. Keeping the number in a variable solves these problems but takes space in memory and requires a memory read at every use. Constants solve both problems at no cost.

4.2.5.1 Defining a constant with EQU

Definition 4.2.14

A **constant** is a name defined in the form `name EQU expression` that exists only during assembly. At every use the assembler replaces the name with the value of the expression. A constant occupies no memory and cannot be changed after it is defined.

```
k      EQU 5
ROWLEN EQU 80
MOV AX, k         ; AX = 5
MOV BX, ROWLEN*2  ; BX = 160
MOV CL, k+1       ; CL = 6
```

Constants carry a name similar to variables, but live only in the assembler's symbol table. When the assembler sees the line `MOV AX, k`, it encodes it as `MOV AX, 5`: `B8 05 00`. A constant's name, like a variable name, is case-insensitive and cannot collide with another name. An expression can also be given: `ROWLEN*2` is computed as 160 at assembly time and 160 is embedded in the instruction. If the same value is used in different places in the program, the value is updated from a single place, the EQU line.

	`k EQU 5`	`k DB 5`
Memory	uses none	1 byte
`MOV AL, k`	AL ← 5 (immediate value, `B0 05`)	AL ← [k] (from memory, `A0` + address)
Can it be changed?	no: `MOV k, 7` is an error	yes: `MOV k, 7`
When does it exist?	at assembly time only	at run time, in memory

Table 4.2.21: Comparison of a constant and a variable.

The line `MOV k, 7` gives an error with an EQU constant. When the assembler replaces `k` with 5, the result is `MOV 5, 7`, and a number as a destination is meaningless. With a variable the same line is valid, because `k` points to a memory location.

4.2.5.2 Code trace: EQU is embedded in the instruction

```
k       EQU 5
ROWLEN EQU 80
ORG 100h
MOV AX, k         ; MOV AX, 5
MOV BX, ROWLEN*2 ; MOV BX, 160
MOV CL, k+1       ; MOV CL, 6
MOV DL, v         ; from memory
RET
v DB 5            ; takes 1 byte
```

The first two lines write k = 5 and ROWLEN = 80 into the symbol table; neither produces a byte. The first byte after `ORG 100h` is therefore directly the code of `MOV AX, k`. The program's byte image is `B8 05 00` · `BB A0 00` · `B1 06` · `8A 16 0D 01` · `C3` · `05`. The code takes 13 bytes and `v` lands at 010Dh.

Address	Machine code	Instruction	AX	BX	CL	DL
0100h	B8 05 00	`MOV AX, 5`	0005h	0000h	00h	00h
0103h	BB A0 00	`MOV BX, 160`	0005h	00A0h	00h	00h
0106h	B1 06	`MOV CL, 6`	0005h	00A0h	06h	00h
0108h	8A 16 0D 01	`MOV DL, [010Dh]`	0005h	00A0h	06h	05h

Table 4.2.22: Trace of the EQU program.

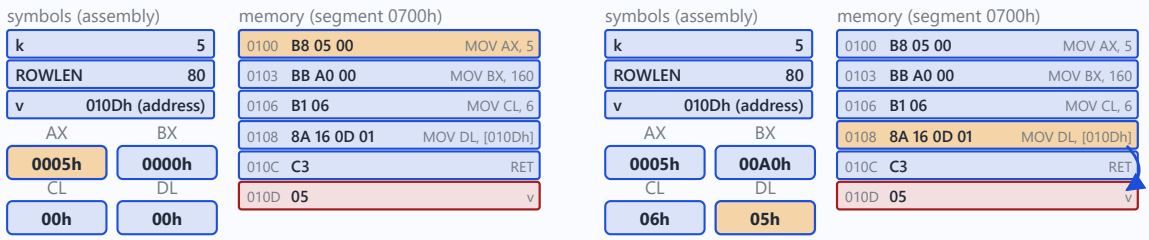


Figure 4.2.19: On the left `MOV AX, k` : the constant 5 is inside the instruction (B8 05 00). On the right `MOV DL, v` : the instruction carries the address of v (010Dh) and the value is read from memory.

In the first three instructions the value is in the instruction's own bytes. Since 160 = 00A0h, `BB A0 00` is written; `k+1` = 6 is also computed at assembly time and becomes the single-byte `B1 06`. The processor does no addition or multiplication while running. The immediate value enters the prefetch queue along with the instruction. The EU takes it from the queue and no separate data bus cycle is needed. Loading an immediate value into a register takes 4 clock periods on the 8086.

The fourth instruction is different. `v` is a variable; the instruction carries not the value but the address (`0D 01`). The EU supplies the effective address, the BIU performs a read bus cycle from DS:010Dh, and DL = 05h. This instruction takes 8 + 6 = 14 periods. `v` occupies one byte in memory;

`k` and `ROWLEN` occupy none.

In MASM and emu8086 the `=` directive also defines a constant: `count = 3`. The difference is that a name defined with `=` can be redefined later in the source. This suits names used as counters at assembly time. While the program runs, neither occupies memory and neither can be changed.

4.2.5.3 The location counter and size operators

The location counter is the counter in which the assembler keeps the relative location of the next byte, and it is read in the source with `$`. `$` gives its value on the line where it is used. This is the way to find the size of an array without counting by hand.

```
s DB 'Welcome', 0
n EQU $ - s        ; 8
w DW 10 DUP(0)
```

When the definition of `s` ends, the location counter points to the address one past the last byte of the string. `$ - s` is the difference of the two addresses, that is, the number of bytes of the string: 7 letters + 1 terminator = 8. This difference is assigned to an EQU constant and becomes an assembly-time constant. If the string is changed, `n` takes the new length automatically. It is essential that the EQU line come **right after** the array. If another definition comes in between, its bytes are counted too and `n` comes out wrong.

MASM also offers operators that read the same information from a variable's definition. `TYPE` gives the byte size of an element, `LENGTHOF` the number of elements, and `SIZEOF` the total number of bytes (in older MASM versions and in emu8086, `LENGTH` and `SIZE`).

Expression	Value	Meaning
`n`	8	number of bytes of s
`TYPE s`	1	element size (DB)
`LENGTHOF s`	8	number of elements
`TYPE w`	2	element size (DW)
`LENGTHOF w`	10	number of elements
`SIZEOF w`	20	LENGTHOF × TYPE

Table 4.2.23: Values of the location counter and the size operators.

All these expressions are assembly-time constants. `MOV CX, LENGTHOF w` embeds the number 10 in the instruction and is encoded as `B9 0A 00`; it takes no extra space in memory. In a word array, LENGTHOF is used for a loop counter and SIZEOF for the length in bytes. In a byte array the two are the same.

4.2.5.4 Code trace: the largest element using the array's size

```
ORG 100h
JMP start
arr DB 12, 47, 5, 31
n EQU $ - arr      ; 4
big DB ?
start:
MOV CX, n          ; CX = 4
MOV SI, 0
MOV AL, arr[SI]    ; first guess
check:
CMP AL, arr[SI]
JAE skip           ; AL >= element
MOV AL, arr[SI]    ; new guess
skip:
INC SI
LOOP check
MOV big, AL
RET
```

In this program the data are at the start. The two-byte `JMP start` (`EB 05`) at 0100h skips the data. `arr` occupies 0102h–0105h and `big` 0106h; the code continues from 0107h. The line `n EQU $ - arr` is right after the array: at that point `$` = 0106h and the difference is 0106h − 0102h = 4. `MOV CX, n` is therefore encoded as `B9 04 00` .

The algorithm rests on the idea of "the largest so far." AL starts with the first element and becomes the guess. On each iteration the guess is compared with the next element. If the guess is smaller, the element becomes the new guess. `CMP AL, arr[SI]` performs AL − element and only sets the flags. `JAE` (*jump if above or equal*) compares as unsigned: it jumps if CF = 0, that is, if AL ≥ element. `LOOP check` decrements CX by one and returns to `check` if CX is not zero. `LOOP` and `INC` do not change CF; the CF in the table is the result of the last `CMP` .

Round	Comparison	CF	JAE	AL	SI	CX
initial	—	0	—	0Ch (12)	0000h	0004h
1	12 − 12	0	taken	0Ch (12)	0001h	0003h
2	12 − 47	1	not taken	2Fh (47)	0002h	0002h
3	47 − 5	0	taken	2Fh (47)	0003h	0001h
4	47 − 31	0	taken	2Fh (47)	0004h	0000h

Table 4.2.24: Round-by-round trace of the largest-element loop (at the end of each round).

Figure 4.2.20: On the left the second round: 12 < 47, CF = 1, `JAE` is not taken and AL = 47 (2Fh). On the right, `MOV big, AL` after the loop: the largest element is written to memory.

After the fourth round CX = 0 and `LOOP` falls through. `MOV big, AL` writes the result to 0106h: big = 2Fh = 47. The loop count was written by hand nowhere. If an element were added to the array, `n` would become 5 and the code would run five rounds unchanged. The comparison is unsigned, because the values in the array are assumed to be in the range 0–255. Had the array contained negative numbers, the signed `JGE` would have been used instead of `JAE`; otherwise −1 (FFh) would have been picked as the largest number.

4.2.6 Example programs

All the rules about variables come together in small programs of a few lines. The programs in this section are based on emu8086's sample programs. Each does a basic job: moving data between variables, arithmetic, comparison, or multiword values. After each program, a trace table gives the registers, flags, and variables at every step.

4.2.6.1 Moving data between variables

```
ORG 100h
jmp start
value1 DW 10    ; a 2-byte variable
value2 DW 0     ; a second one
start:
MOV AX, value1  ; AX ← value1
MOV value2, AX  ; value2 ← AX
RET
end start
```

The program uses the layout that puts the data first. The `jmp start` at 0100h is a two-byte short jump (`EB 04`): the number 04 is the displacement to be added to the address of the instruction after the jump (0102h). The target is 0102h + 4 = 0106h. The four bytes in between are the variables: `value1` at 0102h (0A 00) and `value2` at 0104h (00 00). The `jmp` makes the processor jump over the data. Had there been no jump, the bytes 0A 00 would have been executed as the instruction `OR AL, [BX+SI]`.

Since the 8086 has no memory-to-memory `MOV`, the move is done in two steps. `MOV AX, value1` (`A1 02 01`) takes the value into AX: AX = 000Ah. `MOV value2, AX` (`A3 04 01`) writes AX to memory: 0104h now holds 0A 00. Both instructions are the short forms specific to AX and take 10 clock periods. This is also why AX was chosen as the intermediate register. The `end start` on the last line declares the end of the source and says that the program's entry point is the label `start`.

Line	IP	AX	value1 (0102h)	value2 (0104h)
2 `jmp start`	0106h	0000h	000Ah (0A 00)	0000h (00 00)
6 `MOV AX, value1`	0109h	000Ah	000Ah	0000h
7 `MOV value2, AX`	010Ch	000Ah	000Ah	000Ah (0A 00)

Table 4.2.25: Trace of the data-movement program.

Figure 4.2.21: On the left `MOV AX, value1`: AX = 000Ah. On the right `MOV value2, AX`: the bytes of value2 become 0A 00; now value1 = value2 = 10.

The program begins with `ORG 100h` and ends with `RET`. Since `ORG` produces no bytes, the addresses do not change: `jmp start` is still at 0100h. After `MOV value2, AX`, the `RET` at 010Ch hands control back to DOS. Without this line the processor would have continued executing the bytes after 010Ch as instructions.

4.2.6.2 Adding variables

```
ORG 100h
jmp start
value1 DW 105h  ; initially 105h
value2 DW 103h  ; initially 103h
result DW ?     ; undefined
start:
MOV AX, value1  ; AX ← value1
ADD AX, value2  ; AX ← AX + value2
MOV result, AX  ; result ← AX
RET
end start
```

The three variables are at 0102h, 0104h, and 0106h; this time `jmp start` skips six bytes (`EB 06`). `result` is defined with `?`. emu8086 puts 00 00 in these bytes but its value is not to be relied on; the program writes it before reading it.

`MOV AX, value1` makes AX = 0105h. `ADD AX, value2` (`03 06 04 01`), unlike `MOV`, can use the memory operand directly: it adds AX and the word at [0104h]. The BIU reads the word, the ALU adds, and the result is written to AX. The instruction takes 9 + 6 = 15 clock periods. The sum can be done digit by digit: 5 + 3 = 8, 0 + 0 = 0, 1 + 1 = 2; there is no carry in any digit and AX = 0208h. Decimal check: 261 + 259 = 520 = 208h.

ADD sets all six status flags according to the result. Since no carry comes out of the top bit, CF = 0. The result is nonzero, ZF = 0. The top bit is 0, SF = 0. Since the sum of two positive numbers stays positive, there is no signed overflow, OF = 0. In the low nibble 5 + 3 = 8 and no carry passes from bit 3 to bit 4, AF = 0. The parity flag looks only at the low byte of the result: 08h = 0000 1000 has an odd number (one) of 1s, so PF = 0.

Line	AX	CF	ZF	SF	OF	PF	AF	result (0106h)
initial	0000h	0	0	0	0	0	0	00 00
7 `MOV AX, value1`	0105h	0	0	0	0	0	0	00 00
8 `ADD AX, value2`	0208h	0	0	0	0	0	0	00 00
9 `MOV result, AX`	0208h	0	0	0	0	0	0	08 02

Table 4.2.26: Trace of the addition program.

Figure 4.2.22: On the left `ADD AX, value2` : 0105h + 0103h = 0208h, all flags 0. On the right `MOV result, AX` : little-endian 08 02 in memory.

Had the sum not fit in a word, CF = 1. For example, with value1 = 0F000h and value2 = 2000h, 0F000h + 2000h = 11000h; only 1000h fits in AX and the 1 that spills out goes to CF. A program that checks for overflow looks at CF with `JNC` right after the `ADD` :

```
MOV AX, v1
ADD AX, v2
MOV v3, AX
JNC done
MOV overflow, 1
done:
```

Since `MOV v3, AX` does not change the flags, `JNC` still sees the CF of the addition. For 105h + 103h, CF = 0 and `overflow` stays 0; for 0F000h + 2000h, v3 = 1000h and `overflow` = 1.

4.2.6.3 Comparing two variables

```
ORG 100h
MOV AX, value1     ; AX ← value1
CMP AX, value2     ; AX − value2
JGE greater_or_equal
MOV AX, value2     ; if less, value2
JMP store_max
greater_or_equal:
MOV AX, value1
store_max:
MOV max_value, AX  ; store result
RET
value1 DW 10
value2 DW 5
max_value DW ?
```

The program writes the larger of two numbers to `max_value`. The code is in 0100h–0114h; `value1` is at 0115h, `value2` at 0117h, and `max_value` at 0119h. In this layout the data come after the `RET`.

`CMP` (*compare*) subtracts but **does not store** the result; it only sets the flags. `CMP AX, value2` computes 000Ah − 0005h = 0005h: CF = 0, ZF = 0, SF = 0, OF = 0. AX does not change. `JGE` (*jump if greater or equal*) looks at the result of the signed comparison and jumps if SF = OF. Here SF = OF = 0, so it jumps and goes to `greater_or_equal`. The machine code of `JGE` is `7D 05`: the target is found by adding 5 to the address of the next instruction, 0109h, giving 010Eh.

The `MOV AX, value1` at the jump target makes AX 000Ah again. Since AX already holds value1, this line is redundant but harmless. `MOV max_value, AX` writes the result to 0119h: 0A 00. `RET` hands control back to DOS.

Line	AX	CF	ZF	SF	OF	max_value	Explanation
2 `MOV AX, value1`	000Ah	0	0	0	0	0000h	
3 `CMP AX, value2`	000Ah	0	0	0	0	0000h	10 − 5 = 5, result discarded
4 `JGE ...`	000Ah	0	0	0	0	0000h	SF = OF: taken
8 `MOV AX, value1`	000Ah	0	0	0	0	0000h	redundant line
10 `MOV max_value, AX`	000Ah	0	0	0	0	000Ah	0A 00 in memory

Table 4.2.27: Trace of the comparison program (value1 = 10, value2 = 5).

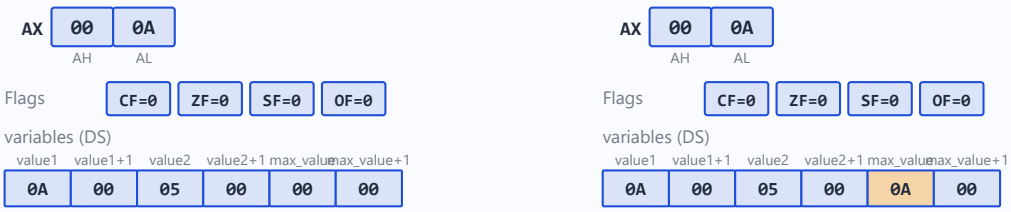


Figure 4.2.23: On the left `JGE` : it is taken because SF = OF. On the right `MOV max_value, AX` : the larger value, 10 (000Ah), is written to memory.

4.2.6.4 Comparison: the other path

To see the program's other branch, the values of the variables are changed:

```
value1 DW 3
value2 DW 7
max_value DW ?
```

The rest of the code is the same; the addresses do not change either. `MOV AX, value1` makes AX = 0003h. `CMP AX, value2` computes 0003h − 0007h. In the unsigned view 7 cannot be subtracted from 3 and a borrow is needed: CF = 1. The 16-bit result is FFFCh, that is, −4; since the top bit is 1, SF = 1. There is no signed overflow, because 3 − 7 = −4 is within the range: OF = 0. In the low nibble 3 − 7 also needs a borrow, AF = 1. The low byte of the result, FCh = 1111 1100, has six 1s, PF = 1.

This time SF = 1, OF = 0, and since SF ≠ OF, `JGE` is not taken. The processor moves on to the next instruction: `MOV AX, value2` makes AX = 0007h. `JMP store_max` jumps unconditionally and skips the `greater_or_equal` branch. `MOV max_value, AX` writes 0007h. Whichever of the two branches the program goes through, it joins again at `store_max` .

Line	AX	CF	ZF	SF	OF	max_value	Explanation
2 `MOV AX, value1`	0003h	0	0	0	0	0000h	
3 `CMP AX, value2`	0003h	1	0	1	0	0000h	3 − 7 = FFFCh, discarded
4 `JGE ...`	0003h	1	0	1	0	0000h	SF ≠ OF: not taken
5 `MOV AX, value2`	0007h	1	0	1	0	0000h	the larger one
6 `JMP store_max`	0007h	1	0	1	0	0000h	unconditional jump
10 `MOV max_value, AX`	0007h	1	0	1	0	0007h	07 00 in memory

Table 4.2.28: Trace of the comparison program on the other path (value1 = 3, value2 = 7).

Figure 4.2.24: On the left `CMP AX, value2`: CF = 1 and SF = 1 for 3 − 7. On the right, because `JGE` is not taken, `MOV AX, value2` executes: AX = 0007h.

4.2.6.5 Signed and unsigned comparison

The jump instruction in the comparison program was deliberately chosen as `JGE`. The 8086 has two families of jumps. `JGE`, `JG`, `JL`, and `JLE` make a signed comparison and look at SF, OF, and ZF. `JAE`, `JA`, `JB`, and `JBE` make an unsigned comparison and look at CF and ZF. The same `CMP` sets the flags both families need; the programmer chooses the interpretation with the jump instruction.

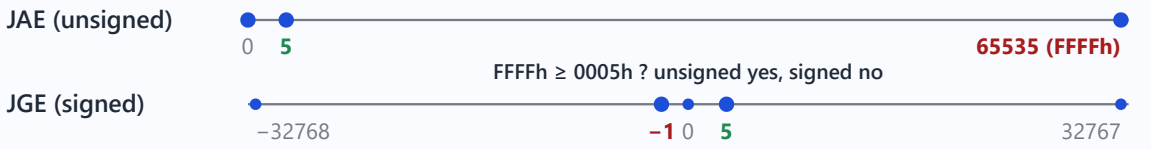


Figure 4.2.25: FFFFh and 0005h on two number lines. In the unsigned view FFFFh = 65,535 is the largest number; in the signed view it is −1 and less than 5.

Example 4.2.2

Let value1 = −1 (FFFFh in memory) and value2 = 5. `CMP AX, value2` computes FFFFh − 0005h = FFFAh: CF = 0 (as unsigned, 65,535 ≥ 5), SF = 1, OF = 0. Because SF ≠ OF, `JGE` is not taken and max_value = 0005h; this is correct in the signed view, since −1 < 5. Had `JAE` been written, it would have been taken because CF = 0 and max_value = FFFFh; this is correct in the unsigned view, since 65,535 > 5.

Figure 4.2.25 shows the place of the same pattern on two number lines. Which jump is right depends on what the variable represents. A value that can be negative, such as a temperature or a position, calls for a signed jump; a counter or an address calls for an unsigned one.

The `RET` at the end of the program is mandatory. The data definitions come right after the code. Without the `RET`, the processor would have executed the bytes `0A 00 05 00 …` at 0115h as instructions. The solution is either to put a `RET` at the end of the code, or to move the data to the start and jump over it with `jmp start`. An improvement of the program is also possible: the `MOV AX, value1` under `greater_or_equal` is redundant. `JGE` can jump directly to `store_max` and the program becomes one instruction shorter.

4.2.6.6 A double-word variable

The 8086 is a 16-bit processor; it has no 32-bit register. Still, it may be necessary to work with numbers larger than 65,535. Such values are stored in memory as 4 bytes with `DD`. In the processor they are held in two 16-bit registers, by convention in the pair **DX:AX**: DX takes the high word and AX the low word.

```
org 100h
jmp start
mydouble dd 12345678h  ; double word
mywords  dw 5678h      ; 2 separate words
         dw 1234h
mybytes  db 78h        ; 4 separate bytes
         db 56h
         db 34h
         db 12h
data dd 00010010000110100010101100111000b
start:  ; load data into DX:AX
mov ax, word ptr data
mov dx, word ptr [data+2]
ret
```

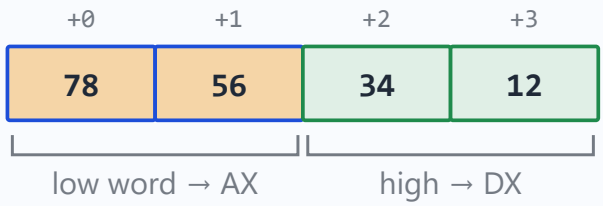


Figure 4.2.26: 12345678h sits in memory as 78 56 34 12. The low word (+0, +1) goes to AX and the high word (+2, +3) to DX.

The four definitions in the program have exactly the same counterpart in memory. `mydouble` gives the value as a single double word; the assembler writes it in little-endian order as 78 56 34 12. `mywords` produces the same bytes with two words: first the low word 5678h (78 56), then the high word 1234h (34 12). `mybytes` writes the four bytes one by one. `data` gives the value in binary: groups of four bits 0001 0010 0011 0100 0101 0110 0111 1000, that is, 12345678h. For the assembler it does not matter in which base or in which pieces the number was written; what remains in memory is four bytes arranged in little-endian order.

4.2.6.7 Code trace: loading 32 bits into DX:AX

```
mov ax, word ptr data      ; data+0
mov dx, word ptr [data+2]  ; data+2
ret
data dd 12345678h
```

`data` is of type DD; when used with a word register, the sizes do not match. `word ptr` tells the assembler that only two bytes will be read in this access. The code is 8 bytes; `data` is at 0108h. `mov ax, word ptr data` (`A1 08 01`) reads the bytes 78 56 at 0108h and 0109h: AX = 5678h. `mov dx, word ptr [data+2]` (`8B 16 0A 01`) reads the bytes 34 12 at 010Ah and 010Bh: DX = 1234h. The pair DX:AX now holds 1234:5678h, that is, the value 12345678h.

Line	Addresses read	DX	AX
1 `mov ax, word ptr data`	0108h, 0109h (78 56)	0000h	5678h
2 `mov dx, word ptr [data+2]`	010Ah, 010Bh (34 12)	1234h	5678h

Table 4.2.29: Trace of the double-word load (`data` = 0108h).

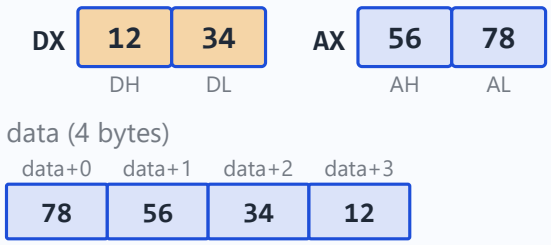


Figure 4.2.27: After the two loads DX = 1234h, AX = 5678h; DX:AX = 12345678h.

Arithmetic on a 32-bit value is also done in two parts. To add 1 to the value `data dd 0FFFFFFFFh`, 1 is first added to the low word, and then the carry that spills out of the low word joins the high word:

```
ADD AX, 1     ; low word
ADC DX, 0     ; high word + carry
```

Initially AX = FFFFh and DX = FFFFh. `ADD AX, 1` makes AX = 0000h and CF = 1. `ADC` (*add with carry*) adds CF as well as the second operand: DX = FFFFh + 0 + 1 = 0000h. The result is DX:AX = 00000000h, and this last addition also produces CF = 1, because the 32-bit result overflowed too. If the same operation is done for 0001FFFFh, DX:AX = 00020000h. Had `ADD DX, 0` been written instead of `ADC` on the second line, the carry would have been lost and the result would wrongly come out as 00010000h.

4.2.6.8 Common mistakes with variables

Most mistakes made when working with variables are of the kind the assembler cannot catch: the program assembles, runs, and gives a wrong result. The list below collects the most frequent ones.

- **Confusing the value with the address.** `MOV SI, arr` loads the first word; for the address, `LEA SI, arr` or `MOV SI, OFFSET arr` is needed.
- **Moving from memory to memory.** There is no `MOV v1, v2`; the value goes through a register. The assembler catches this.
- **The index in a word array.** The 3rd element is at `w[4]`; the index is in bytes. `w[1]` combines the halves of two elements.
- **Relying on the `?` value.** `total DW ?` may not be zero; `MOV total, 0` is written before the summation starts.
- **Forgetting the suffix.** `DW 10` is decimal 10 (000Ah); for hexadecimal write `10h`. Hexadecimal numbers that begin with a letter get a leading 0: `0FFh`, `0ABCDh`.
- **Name collision.** `COL EQU 4` and `col DB 1` are the same name; the assembler reports a “defined twice” error.
- **Putting data in the path of execution.** If there is no `RET` or `JMP`, the data bytes are taken for instructions.
- **Forgetting the terminator.** Without a 0 or '$', printing continues with the bytes behind the string.

4.2.6.9 Summary of data definition

Notation	Meaning	Memory
`x DB 7`	one byte	07
`y DW 1234h`	one word	34 12
`z DD 12345678h`	double word	78 56 34 12
`s DB 'Hi', 0`	string + terminator	48 69 00
`t DB 3 DUP(0)`	repetition	00 00 00
`u DW ?`	no initial value	?? ?? (emu8086: 00 00)
`k EQU 10`	constant	occupies no memory

Table 4.2.30: Data-definition notations and their counterparts in memory.

To get an address, write `LEA BX, x` or `MOV BX, OFFSET x`; to read the value, `MOV AL, x`. If the size of an access is to differ from the variable's type, `byte ptr` or `word ptr` is used.

4.2.7 Summary

- A variable is a data area in memory bound to a name. The assembler replaces the name with a relative location; the running program has no name, only the address embedded in the instruction.
- `DB`, `DW`, and `DD` reserve 1, 2, and 4 bytes respectively and give the variable the type BYTE, WORD, or DWORD. `?` reserves an area with no initial value; its value is not to be relied on.
- Only the bit pattern is stored in memory. Whether the pattern is signed or unsigned is determined by the instruction that uses it: CF, `JAE`, `JB` interpret it as unsigned; OF, `JGE`, `JL`, `CBW` as signed.
- The assembler does type checking: the size of the register and the variable must match. For typeless operands and for a deliberate size change, `BYTE PTR` / `WORD PTR` is used. `LABEL` gives a second name and type to the same address.
- The 8086 is little-endian: the low byte of multi-byte values is at the low address. This rule applies to the addresses inside instructions too.
- With the location counter the assembler computes the relative location of every byte and turns names into addresses in two passes. `ORG 100h` declares that a .COM program starts at 0100h, after the PSP; in .COM, CS = DS = ES = SS.
- A word starting at an odd address is read in two bus cycles and costs 4 clock periods. `EVEN` aligns words to even addresses.
- Data is written either after the `RET` or behind a `JMP`; the processor takes data bytes in its path for instructions. In an .EXE program DS is loaded first.
- An array is consecutive elements of the same type. The index is in bytes: in a word array the i-th element is at offset `2 × i`. `DUP` repeats value lists and can be nested.
- Strings are defined with `DB` and their end is marked by 0, '$', or a length byte. Upper and lower case differ only in bit 5; subtracting '0' from a digit character gives the number.
- Two-dimensional arrays are stored in row-major order: offset = (row × number of columns + column) × element size.
- `LEA` computes the effective address at run time and does not access memory; `OFFSET` gives a constant address at assembly time. An indexed address can be obtained only with `LEA`. Pointer variables store an address in memory.
- `EQU` constants exist only at assembly time, are embedded in the instruction as immediate values, and occupy no memory. `$ - arr` computes the size of an array automatically.
- There is no memory-to-memory move; the value goes through a register. `CMP` sets the flags without storing the result. 32-bit values are held in the pair DX:AX and added with the `ADD` / `ADC` pair.

4.2.8 Key terms

variable data definition (DB / DW / DD) initial value directive byte word double word signed / unsigned number two's complement sign extension byte ptr / word ptr little-endian offset location counter ($) .COM program program segment prefix (PSP) aligned / unaligned access bus cycle array string ASCII row-major order effective address (EA) OFFSET operator pointer variable constant (EQU) immediate value carry flag (CF) overflow flag (OF)

4.2.9 Test Yourself

1. The definitions `p DB 'Az', 0` · `q DW 300` · `r DD 0Ah` start at 0300h. What are the address and value of each byte? How many bytes do the definitions occupy in total?

   Answer

   0300h: 41h ('A'), 0301h: 7Ah ('z'), 0302h: 00h. Since 300 = 012Ch, 0303h: 2Ch, 0304h: 01h. `r` is at 0305h–0308h: 0A 00 00 00. Total 3 + 2 + 4 = 9 bytes.

2. How many bytes does `tab DW 5 DUP(4 DUP(?))` occupy? In row-major order, what is the byte offset of the element [3][2] from `tab`?

   Answer

   5 × 4 = 20 words = 40 bytes. The offset is (3 × 4 + 2) × 2 = 28 = 1Ch. To read it, `MOV SI, 28` · `MOV AX, tab[SI]` can be written.

3. Right after the line `s DB 'Test', 13, 10, '$'`, `n EQU $ - s` is written. What is `n`? What would happen if `t DB 0` came in between?

   Answer

   4 letters + CR + LF + '$' = 7. Had `t DB 0` come in between, `$` would have advanced by one byte and `n` = 8; the constant would no longer give the size of the string.

4. With BX = 0100h, DI = 0020h, and CF = 1, `LEA SI, [BX+DI+4]` is executed. What does SI become?

Does CF change? Is anything read from memory?

Answer

SI = 0100h + 0020h + 4 = 0124h. `LEA` does not change the flags; CF = 1 stays. Only the address is computed; no read bus cycle is made.

5. With AL = 80h and BL = 7Fh, `CMP AL, BL` is executed. What are CF, SF, and OF? Is `JA` taken, and is `JG` taken afterwards?

Answer

80h − 7Fh = 01h. As unsigned, 128 ≥ 127, no borrow: CF = 0, ZF = 0, `JA` is taken. As signed, −128 − 127 = −255 is out of range: OF = 1, and since the result is 01h, SF = 0. Because SF ≠ OF, the result is −128 < 127 and `JG` is not taken.

6. `w DW 0102h, 0304h` is defined. What are AX and BL after `MOV AX, w[2]` and `MOV BL, BYTE PTR w+1`? Why does `MOV BL, w+1` not assemble?

Answer

`w[2]` is the second element: AX = 0304h. Since memory is 02 01 04 03, the byte at `w+1` is 01h: BL = 01h. Because `w` is of type WORD, `MOV BL, w+1` gives a size mismatch; `BYTE PTR` is needed.

7. `MOV DX, OFFSET msg` and `LEA DX, msg` give the same result. How many bytes is each and roughly how many clock periods does each take on the 8086? Which can be used for an indexed address?

Answer

`MOV DX, OFFSET msg` is an immediate load: 3 bytes (BA + 16 bits), 4 periods. emu8086 translates the line `LEA DX, msg` into the same code; both are 3 bytes and 4 periods (the real LEA in MASM, however, is 4 bytes, 8D 16 + 16 bits, and 2 + 6 = 8 periods). An indexed address (`LEA DX, msg[SI]`) can be obtained only with LEA.

8. With DX:AX = 0001FFFEh, `ADD AX, 3` · `ADC DX, 0` are executed to add 3. What are AX, DX, and CF after each instruction?

Answer

`ADD AX, 3` : FFFEh + 3 = 10001h; AX = 0001h, CF = 1. `ADC DX, 0` : 0001h + 0 + 1 = 0002h, CF = 0. The result is DX:AX = 00020001h = 0001FFFEh + 3.

9. With the location counter at 0105h, `a DB 1, 2` · `EVEN` · `w DW 5` is written. At which address does `w` land? Why would `MOV AX, w` be slower without `EVEN`?

   Answer

   `a` occupies 0105h–0106h and the counter becomes 0107h. `EVEN` puts a padding byte at 0107h and `w` lands at 0108h. Without `EVEN`, `w` would be at 0107h, starting at an odd address. The word would be read in two bus cycles and the access would take 4 periods longer (14 instead of 10).

4.2.10 Further reading

- The emu8086 documentation, *8086 assembler tutorial for beginners*, chapter 3 (variables): DB, DW, arrays, DUP, LEA, OFFSET, and EQU are explained with the examples that are the starting point of the programs in these notes.
- Kip R. Irvine, *Assembly Language for x86 Processors*, the data-definition chapter: data directives, little-endian order, `$`, and the TYPE, LENGTHOF, and SIZEOF operators are given with detailed examples.
- Barry B. Brey, *The Intel Microprocessors*, the chapters on addressing modes and data-movement instructions: LEA, LDS/LES, and the machine code and hardware-level operation of direct addressing are covered.
- Microsoft, *MASM 6.1 Programmer's Guide*: the precise definitions and rules of the DB/DW/DD, DUP, EQU, `=`, OFFSET, LABEL, and EVEN directives.

4.2.11 References

1. The emu8086 documentation, *8086 assembler tutorial for beginners*, chapter 3: variables (DB, DW, DUP, LEA, OFFSET, EQU).
2. Barry B. Brey, *The Intel Microprocessors*, 8th edition, Pearson.
3. Kip R. Irvine, *Assembly Language for x86 Processors*, Pearson (data-definition directives).
4. Microsoft, *MASM 6.1 Programmer's Guide*: DB/DW/DD, DUP, EQU, OFFSET.
5. Intel, *The 8086 Family User's Manual*, 1979 (instruction encoding and clock-period counts).

PART II · CHAPTER 5

Interrupts and BIOS/DOS Services

An interrupt is an event that makes the processor set aside its current work and run a special service routine. The chapter explains the types of interrupts, the interrupt vector table, how INT and IRET save and restore FLAGS, CS and IP on the stack, processor exceptions, and masking hardware interrupts with the 8259A. The BIOS and DOS services part covers screen output with INT 10h, keyboard input with INT 16h, character and string handling with INT 21h, and reading the date and time, including INT 1Ah. Terminating a program correctly and an application that combines these services close the chapter.

NOTES IN THIS CHAPTER

5.1 Interrupts

5.2 Interrupt Functions

PART II · CHAPTER 5.1

Interrupts

A processor that only ran its own program would know nothing of the outside world. A key is pressed, a timer produces a pulse, a disk drive has data ready; the processor must respond to these events in time. An interrupt lets the processor drop what it is doing at an instruction boundary, run a routine that belongs to the event, and then resume where it left off. These notes begin with what an interrupt is and how it differs from polling. They then cover the 8086 interrupt vector table and the effect of INT and IRET on the stack, followed by exceptions generated by the processor itself, hardware interrupts on the INTR and NMI pins, the 8259A interrupt controller, and interrupt latency. The last part writes an interrupt service routine from scratch, installs its vector, and restores the old vector when the program ends. Every piece of code is traced step by step together with the registers, flags, vector table and stack.

LEARNING OUTCOMES

Explain the interrupt concept
Explain what an interrupt is, how it differs from polling, and where software, hardware and internal interrupts come from.

Compute a vector address
Use the interrupt vector table to find the vector address for type n (n × 4), then the ISR's CS:IP from the bytes stored there and its physical address.

Trace INT and IRET on the stack
Show step by step the order in which FLAGS, CS and IP are pushed during INT n and IRET, how SP changes, and why IF and TF are cleared.

Recognize exceptions
Explain the two causes of a divide error (INT 0), the condition under which INTO (INT 4) fires, and the single-step (INT 1) and breakpoint (INT 3) mechanisms.

Follow hardware interrupts
Follow INTR acknowledgment, interrupt priorities, the 8259A's IRR, IMR and ISR registers, and the masking of a single IRQ with the IMR; compute interrupt latency in clock cycles.

Write an interrupt service routine
Write your own ISR, install its vector safely, save and restore the old vector, and follow the rules for writing ISRs.

5.1.1 The interrupt concept

5.1.1.1 What is an interrupt?

While a program runs, the processor fetches, decodes and executes instructions in order. After each instruction IP points to the next one, and apart from jumps and calls this order is never disturbed. Yet events that have nothing to do with the program happen all around the computer. A user presses a key, the timer chip produces pulses at fixed intervals, a character arrives on the serial port. The processor must not miss these events and must respond to them quickly.

Definition 5.1.1

An **interrupt** is an event that makes the processor temporarily set aside its current work and run a special routine. This routine is called an **interrupt service routine** (ISR). When the ISR finishes, the processor resumes the program with the instruction after the one that was interrupted.

The last sentence of the definition is the important one. The interrupted program does not notice that it was interrupted. Its registers, flags and memory continue with the values they would have had without the interrupt. Guaranteeing this is the job of both the processor and the programmer who writes the ISR. The processor saves the return address and the flags on its own. The ISR saves the registers it uses and restores them on exit.

An everyday analogy captures the steps of an interrupt. You are reading a book when the phone rings. You first mark your page, then answer the phone and talk. When the call ends, you return to the marked page and carry on reading. Marking the page is saving the return address; the phone call is the ISR; returning to the marked page is the IRET instruction. If you forget to mark the page, you will not know where to continue. The processor would be in the same position if it entered an ISR without pushing the return address on the stack.

Figure 5.1.1: The main program is interrupted by a request or an INT instruction, the ISR runs, and IRET returns to where it left off. The return address stays on the stack for the whole time.

Figure 5.1.1 shows this flow on a time axis. The main program stops at some point, control passes to the ISR, and the last instruction of the ISR, IRET, hands control back to the main program. An interrupt can be started in two ways: the program executes an INT instruction, or a request arrives from outside the processor. In both cases what the processor does afterwards is the same.

Interrupts have a second, more everyday use. On an 8086-based PC there are ready-made routines for writing a character to the screen, reading a key from the keyboard, accessing the disk, or ending a program. These are the routines of the BIOS, which sits in ROM on the motherboard, and of the operating system, DOS. A program calls them with an INT instruction, without knowing their addresses. To write a character to the screen, for example, it does not need dozens of instructions that drive the video hardware; `INT 10h` does the job. For this reason software interrupts are also seen as ready-made services that make programming easier. The details of the interrupt functions that work with hardware such as the screen, the keyboard, the disk drive and memory management are the subject of the notes of Chapter 5.2, Interrupt Functions. These notes cover the interrupt mechanism itself: how the processor accepts an interrupt and how it returns.

5.1.1.2 Polling or interrupts?

A processor can learn the state of a device in two ways. The first is to keep asking the device, "Is the data ready?" This method is called **polling**. In a loop the processor reads the device's status port, checks the ready bit, and reads again if it is not set. The second is for the device to announce itself. When its data is ready, the device drives a pin of the processor; the processor finishes the current instruction and switches to the ISR. This is the interrupt method.

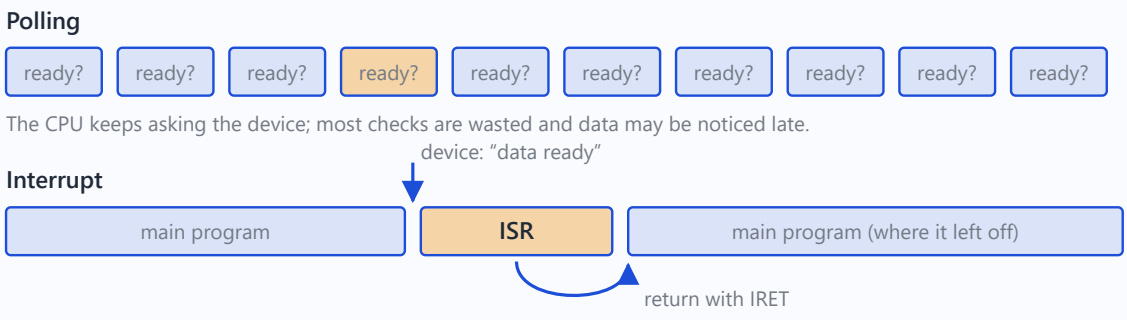


Figure 5.1.2: Top, polling: the processor keeps asking the device "ready?" and most queries are wasted. Bottom, interrupt: the processor runs the main program, the device signals when data is ready, the ISR runs,

and IRET returns.

Figure 5.1.2 compares the two methods on the same time axis. The cost of polling can be seen in a single number. The shortest polling loop that asks for the keyboard's state has three instructions: an `IN` that reads the port, a `TEST` that checks the ready bit, and a conditional jump that goes back to the start if the device is not ready.

```
; polling loop (port number and bit are examples)
poll:  IN   AL, 64h      ; read the status port (10 clocks)
       TEST AL, 01h      ; data-ready bit (4 clocks)
       JZ   poll         ; not ready: ask again (16 clocks)
```

According to Intel's 8086 manual, an `IN` with a fixed port takes 10 clock cycles, a `TEST` of a register with an immediate takes 4, and a taken `JZ` takes 16. One pass of the loop is 30 clock cycles; on a 5 MHz 8086 a clock cycle is 200 ns, so the loop turns once every 6 µs. The processor asks the device about 166,000 times per second. A fast typist presses ten keys per second. Of those 166,000 queries only ten get a "yes" and the rest are wasted. Worse, the processor can do nothing else in the meantime.

Polling less often saves processor time, but then the event is noticed late. If the program asks every 10 ms, the response to a key press can be delayed by up to 10 ms. The longer the interval, the greater the risk of losing data. For example, characters arriving on a serial port at 9600 bit/s are about 1 ms apart. If the device has a one-character buffer, a character that is not read within 1 ms is overwritten by the next.

With interrupts this dilemma disappears. The processor runs the main program as long as there is no event; the moment an event occurs, at the latest when the current instruction finishes, it switches to the ISR. The price is the machinery. A routine must be written for each event, its address placed in a table, the stack used for the return address, and the hardware needs lines that carry the request to the processor. Table 5.1.1 compares the two methods.

Criterion	Polling	Interrupt
Hardware	simple; no extra line needed	a request line and, if needed, an interrupt controller
Processor time	wasted; most queries return "no"	spent only when an event occurs
Response time	depends on the polling interval; an event may be noticed late	short; at the latest when the running instruction finishes
Software structure	a single loop	vector table, stack and ISR
Suited to	single-task, simple systems; very fast, continuous data	rare events at unpredictable times; multitasking systems

Table 5.1.1: Comparison of polling and interrupts.

Polling is not always bad. If a small microcontroller circuit is only waiting for one button, polling is the simplest solution. If a very fast device produces data every few microseconds, a tight polling loop can be faster than bearing the overhead of the interrupt machinery for every item. A general-purpose computer, on the other hand, has independent events at unpredictable times: keyboard, timer, disk and serial port. Polling all of them in one loop both keeps the processor busy and slows the response. This is why the PC architecture is built on interrupts.

Common mistake

Generalizing "with an interrupt the processor never waits for the device" to "an interrupt is always faster than polling." The interrupt mechanism itself takes time: on the 8086, accepting a hardware interrupt takes 61 clock cycles and IRET takes 24. For an event that arrives thousands of times per second, this overhead can exceed the cost of polling. The advantage of interrupts is that for rare events the processor does not wait in vain.

5.1.1.3 Interrupt kinds and type numbers

The 8086 distinguishes up to 256 different interrupts. Each interrupt has a **type number** between 0 and 255 (00h–FFh). When the processor accepts an interrupt, the first thing it does is determine this number; it then reads from a table which routine to go to for that number. Interrupts fall into three groups, according to where the type number comes from and what starts the interrupt.

Definition 5.1.2

A **software interrupt** is one called deliberately by an `INT n` instruction in the program; the type number n is inside the instruction. A **hardware interrupt** is one started when an external device drives the processor's INTR or NMI pin. An **internal interrupt**, or **exception**, is one the processor generates itself while executing an instruction; division by zero is the best-known example.

Software interrupts. BIOS and DOS services are used this way: `INT 10h` is the screen, `INT 16h` the keyboard, `INT 21h` the DOS services. Two more special instructions belong to this group. `INT 3` is the breakpoint instruction used by debuggers and is encoded in a single byte. `INTO` calls INT 4 when the overflow flag OF = 1 and does nothing when OF = 0. Because they serve debugging and error trapping, these two instructions are often mentioned together with exceptions.

Hardware interrupts. The 8086 has two input pins for interrupt requests. **INTR** (interrupt request) is the maskable input: while IF = 0 the processor ignores this pin. Requests from the keyboard, timer, disk and serial port arrive on it. **NMI** (non-maskable interrupt) is the non-maskable input; it is accepted whatever IF is. It is reserved for urgent events that a program cannot switch off and ignore, such as a memory parity error or loss of power.

Internal interrupts. The processor turns some conditions into interrupts by itself. When DIV or IDIV divides by zero, or the quotient does not fit in the destination register, INT 0 is generated. While the trap flag TF = 1, INT 1 is generated after every instruction; this lets debuggers run a program one step at a time.

Table 5.1.2 gives the role of the first five type numbers. The 8086 itself uses these five types; their numbers are fixed and a programmer must not use them for anything else.

Type	Source	Description
00h	internal	Divide error: DIV/IDIV by zero, or the quotient does not fit
01h	internal	Single step (TF = 1); used by debuggers
02h	hardware	NMI pin (non-maskable)
03h	software	Breakpoint; the one-byte INT 3 (CCh)
04h	software	INTO: overflow interrupt if OF = 1
05h–FFh	—	Intel reserved 05h–1Fh for later processors; on the PC the BIOS, DOS and user programs use them

Table 5.1.2: Reserved interrupt types of the 8086.

Intel stated explicitly in its documentation that it was reserving the range 05h–1Fh for its future processors. The designers of the IBM PC did not heed this warning and placed BIOS services in the range 05h and 08h–1Fh. For example, the BIOS routine that prints the screen on the printer uses type 05h. With the 80186 and 80286, type 05h became the bound-range-exceeded interrupt of the BOUND instruction. The same number thus acquired two different meanings, and PC software wrestled with this collision for years. The story is a good illustration of why reserved numbers must stay reserved.

5.1.1.4 Comparing the three sources

At first glance the three kinds of interrupt look very different: one is an instruction, one is a pin, one is an error condition. They differ in only two respects: where the type number comes from and when the interrupt occurs. Table 5.1.3 puts these differences side by side.

	Software (INT n)	Hardware (INTR / NMI)	Internal (exception)
Triggered by	an instruction in the program	an external device, a pin	the processor itself
Timing	synchronous: always at the same place	asynchronous: after any instruction	synchronous: at the faulting instruction
Type number	inside the instruction	INTR: from the 8259A, via INTA; NMI: always 2	fixed: 0, 1
When IF = 0	still runs	INTR waits, NMI runs	still runs

	Software (INT n)	Hardware (INTR / NMI)	Internal (exception)
Typical use	BIOS/DOS service, INT 3, INTO	keyboard, timer, disk	divide error, single step

Table 5.1.3: Comparison of software, hardware and internal interrupts.

The words **synchronous** and **asynchronous** are used here with respect to the program flow. A software interrupt occurs at the same point of the program, in the same way, on every run. An exception is the same: the same DIV instruction with the same data gives the same error every time. A hardware interrupt, on the other hand, is an event whose instruction in the program cannot be known in advance. The user presses a key once at the start of a loop and once in the middle. For this reason the ISR of a hardware interrupt must be written as if it could run at any point of the program. This is where the obligation of ISRs to save registers comes from.

There is an important common point under the table. Once the type number is determined, all three kinds follow the same steps: FLAGS, CS and IP are pushed, IF and TF are cleared, and CS:IP is loaded from the vector table. In other words there is a single interrupt mechanism inside the processor; the sources only determine with which number and at what moment that mechanism is entered.

Common mistake

Assuming that the `CLI` instruction turns off all interrupts. CLI only clears IF and affects only the INTR input. Software interrupts such as `INT 21h`, exceptions such as division by zero, and NMI still run while IF = 0. When CLI is used to "protect a critical piece of code from interrupts", remember that NMI can still arrive.

5.1.2 The interrupt vector table

5.1.2.1 Structure of the table

After the processor learns the type number of an interrupt, it must find where the ISR for that interrupt is. On the 8086 this information is kept in a table at the very beginning of memory. The location of the table is fixed; on every interrupt the hardware looks at the same address.

Definition 5.1.3

An **interrupt vector** is the 4-byte far address of the service routine of an interrupt type: 2 bytes of IP (offset) and 2 bytes of CS (segment). The **interrupt vector table** (IVT) is the 1 KB table that holds the vectors of the 256 interrupt types in order and occupies physical addresses 00000h–003FFh.

The size of the table is easy to compute: 256 types × 4 bytes = 1024 bytes = 1 KB. The vector of type 0 starts at 00000h, that of type 1 at 00004h, that of type 2 at 00008h, and so on. The vector of the last type, FFh, is in bytes 003FCh–003FFh. The address at which the vector of type n starts is therefore a simple multiplication:

> vector address = n × 4 (within segment 0000h)

The layout of the four bytes of a vector is also fixed. Figure 5.1.3 shows the table. The word at the lower address (n × 4 and n × 4 + 1) holds the IP value. The word at the higher address (n × 4 + 2 and n × 4 + 3) is the CS value. Both words are stored in little-endian order, like all words on the 8086: low byte first, then high byte.

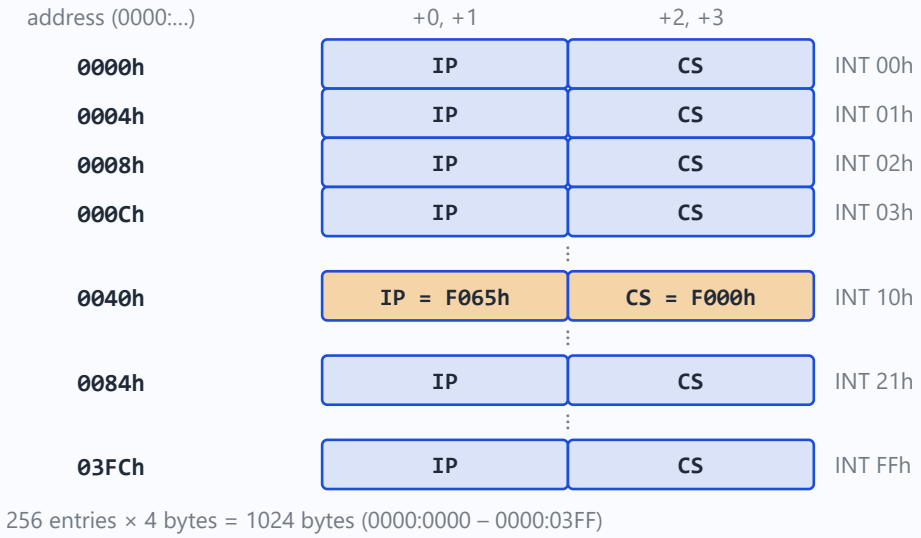


Figure 5.1.3: The interrupt vector table. Each row is the 4-byte vector of one interrupt type: IP first, then CS. The value F000:F065 in the INT 10h row is the address of the video routine in the original IBM PC BIOS and is given as an example.

The answer to why the table is at the very beginning of memory lies in the design of the 8086. The hardware does not have to look at any register to find the table; the address is produced from the type number alone. The operation n × 4 is a two-bit shift in the circuit. The 8 bits of the type number are placed on address lines A2–A9; A0, A1 and A10–A19 are zero. Reading a vector therefore needs no extra adder or register. The price is that the first 1 KB of memory must always be RAM. Programs have to be able to change the table; if it were in ROM, no program could install its own ISR.

The table is an ordinary region of memory with no special protection. Any program that sets a segment register to 0000h can read and write any byte of the table. This flexibility lets programs install their own ISRs; it also means that a faulty program can corrupt the table and crash the whole system. Later processors with protected mode solve this problem by moving the table into a protected structure. In the real mode of the 8086, the safety of the table is left entirely to the programmer.

Common mistake

Thinking of the four bytes of a vector in the order "CS first, then IP". The notation CS:IP writes the segment first and so invites this error. In memory the order is reversed: at address n × 4 is the low byte of IP, and at n × 4 + 2 is the low byte of CS.

5.1.2.2 The vector address step by step

It helps to follow the rule in action on an example. Suppose the program is executing `INT 10h`. The processor finds the vector and switches to the ISR in four steps.

Example 5.1.1

Let us find the vector address for INT 10h, the ISR's CS:IP, and its physical address. Suppose the four bytes of the table starting at address 0000:0040h are, in order, `65 F0 00 F0`.

1. **Type number.** The second byte of the instruction is the type number: n = 10h = 16.
2. **Vector address.** n × 4 = 10h × 4 = 40h (decimal: 16 × 4 = 64). The vector is in bytes 0000:0040h–0000:0043h. In binary, shifting 0001 0000b left by two bits gives 0100 0000b = 40h.
3. **Reading the bytes.** At 0040h there is 65h and at 0041h F0h; the low word is IP = F065h. At 0042h there is 00h and at 0043h F0h; the high word is CS = F000h.
4. **Physical address.** CS × 16 + IP = F0000h + F065h = FF065h. The ISR starts inside the BIOS ROM in the topmost 64 KB of memory.

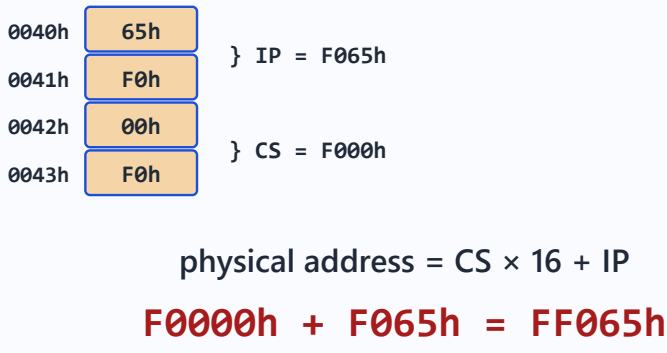


Figure 5.1.4: Top, bytes 0040h–0043h: low word IP = F065h, high word CS = F000h. Bottom, the physical address calculation: F0000h + F065h = FF065h.

The bytes read in the example are shown in Figure 5.1.4. The processor fetches these two words with two separate memory reads. The 8086 reads a word at an even address in a single bus cycle. Because both n × 4 and n × 4 + 2 are even, each word arrives in one cycle. The 4-byte alignment of the vector table is also convenient in this respect.

The physical address calculation in the fourth step is the segmentation rule the 8086 uses for all memory accesses: the 16-bit segment value is shifted left by four bits (multiplied by 16) and the offset is added. The segment CS = F000h starts at F0000h. IP = F065h is a position inside this segment. The result is the 20-bit address FF065h.

There is an important detail about the order here. Before the processor reads the vector and loads CS:IP, it writes the return information to the stack. So while the first instruction of the ISR is fetched from FF065h, the old CS, IP and FLAGS are already on the stack. If this order were broken, that is, if CS:IP were loaded first, the return address would be lost. The exact order of the steps is in Table 5.1.5.

5.1.2.3 From table to routine and back

Two-way calculations are done on the vector table. In the first direction one goes from a type number to the address of the vector; this is the multiplication n × 4. In the second direction one finds, from an address in the table, which type's vector it is; this is a division of the address by 4. The second direction is needed when looking at a byte sequence in a memory dump and working out what it is.

Example 5.1.2

In a memory dump the bytes `2E E8 00 F0` are seen starting at address 0000:0058h. a) Which interrupt's vector is it? b) What are the ISR's CS:IP and physical address? c) At which addresses is the vector of type 1Ch in the same table?

1. a) 58h / 4 = 16h. Because the address is a multiple of 4, it is the start of a vector: type 16h, that is, the BIOS keyboard service INT 16h.
2. b) Low word: 2Eh, E8h → IP = E82Eh. High word: 00h, F0h → CS = F000h. Physical address F0000h + E82Eh = FE82Eh.
3. c) 1Ch × 4 = 70h. IP is in bytes 00070h–00071h and CS in bytes 00072h–00073h.

If the address were not exactly divisible by 4, for example 005Ah, it would not be the start of a vector but the CS half of the vector of type 16h (5Ah = 4 × 16h + 2).

The calculation is the same for all types: IP is at n × 4 and n × 4 + 1, CS at n × 4 + 2 and n × 4 + 3. For type 00h the address comes out as 0000h, for type FFh as 03FCh; all addresses in between are multiples of 4.

Common mistake

Mistaking the type number for the address directly (INT 21h → 00021h) or multiplying by 2 instead of 4 (INT 21h → 00042h). The correct value is 21h × 4 = 84h. To avoid slips in hexadecimal multiplication, one way is to convert to decimal first: 21h = 33, 33 × 4 = 132 = 84h.

5.1.2.4 How the PC divides up the type numbers

The first five of the 256 type numbers belong to the 8086. The rest are left to the designers of the system that uses the processor. On the IBM PC and DOS the numbers are divided up roughly as follows (Table 5.1.4).

Type	Vector address	Role
00h–04h	000h–013h	the 8086's own types (divide error, single step, NMI, breakpoint, overflow)
05h	014h	BIOS: print the screen
08h–0Fh	020h–03Fh	hardware interrupts IRQ0–IRQ7 (08h timer, 09h keyboard)
10h	040h	BIOS: video / screen services
13h	04Ch	BIOS: disk services
16h	058h	BIOS: keyboard services
1Ah	068h	BIOS: clock / date
20h	080h	DOS: terminate program
21h	084h	DOS: general services (selected with AH)
33h	0CCh	mouse driver
60h–67h	180h–19Fh	reserved for user programs
70h–77h	1C0h–1DFh	IRQ8–IRQ15 of the second 8259A on the PC/AT

Table 5.1.4: Frequently used interrupt types on the IBM PC and DOS.

The practical conclusion from this table is that a program that installs its own ISR should choose an unused type. The range 60h–67h is set aside for this purpose. The examples in these notes that install their own ISR therefore use types 60h, 61h and 90h. A program that changes the vector of a type in use, for example INT 21h, redirects all DOS services to its own routine. This is sometimes done on purpose. For example, a program changes the vector of type 0 so that its own routine runs instead of DOS's divide error routine. In that case it is mandatory to restore the old vector when the program ends.

5.1.3 INT and IRET

5.1.3.1 The steps of INT n

The `INT n` instruction is two bytes: the opcode CDh and the type number n. For example `INT 21h` sits in memory as `CD 21`. While executing it the processor does a series of things the programmer cannot see. They resemble a procedure call but include more. Table 5.1.5 gives the steps in order.

Step	Operation	Effect	Bus cycle
1	FLAGS is pushed on the stack	SP ← SP − 2, [SS:SP] ← FLAGS	memory write
2	IF and TF are cleared	IF = 0, TF = 0	—
3	CS is pushed on the stack	SP ← SP − 2, [SS:SP] ← CS	memory write
4	IP is pushed on the stack (the address of the instruction after INT)	SP ← SP − 2, [SS:SP] ← IP	memory write
5	vector is read	IP ← [0000:n×4], CS ← [0000:n×4+2]	two memory reads
6	the ISR runs and ends with IRET	IP, CS, FLAGS are popped from the stack in that order	three memory reads

Table 5.1.5: The steps of the INT n instruction and the effect of each on the stack, registers and bus.

Every row of the table has a reason. **Step 1** saves the flags. The result of the interrupted program's last comparison, for example ZF, may change during the ISR. If the program is interrupted between a `CMP` and a `JE` and the flags are not saved, JE jumps the wrong way. Saving the flags automatically is therefore essential. The CALL instruction does not save the flags, because the program knows when a procedure will be called.

Step 2 clears IF and TF. IF = 0 prevents the ISR from being interrupted by another maskable interrupt at its very first instruction. TF = 0 prevents the ISR from being stopped step by step by the single-step interrupt. It matters that this clearing is done *after* FLAGS has been pushed. The copy on the stack carries the old IF and TF values. When IRET restores that copy, IF and TF return to their values from before the interrupt.

Steps 3 and 4 save the return address. Because the ISR may be in another segment, IP alone is not enough; CS is saved too. The IP written to the stack is not the address of the INT instruction itself but the address of the instruction *after* INT. The processor has already advanced IP by two bytes when fetching INT.

Step 5 reads the vector and loads it into CS:IP. From this moment the first instruction the processor fetches is the first instruction of the ISR. The BIU's prefetch queue is also flushed; the bytes in the queue belonged to the instructions after INT and are now invalid.

A total of 6 bytes, that is 3 words, is written to the stack. The whole of INT n takes 51 clock cycles on the 8086. A significant part of these cycles is spent on five memory accesses: three writes to the stack and two reads from the vector table. Flushing the queue and refetching the first bytes of the ISR also take time.

Definition 5.1.4

IRET (interrupt return) is the return instruction from an interrupt service routine. It pops IP, CS and FLAGS from the stack in that order; SP increases by 2 on each pop. It is a single byte (CFh) and takes 24 clock cycles on the 8086.

IRET reads back in reverse order what INT wrote to the stack. This is natural because the stack works last in, first out: the IP pushed last is popped first, and the FLAGS pushed first are popped last. When FLAGS comes back, IF and TF take their old values too. A program that had IF = 1 before the interrupt therefore continues with IF = 1 after IRET.

Table 5.1.6 gives the encoding and timing of the interrupt instructions that appear in these notes.

Instruction	Machine code	Length	Clock cycles
`INT n`	CD n	2 bytes	51
`INT 3`	CC	1 byte	52
`INTO`	CE	1 byte	53 (OF = 1) / 4 (OF = 0)
`IRET`	CF	1 byte	24
`CLI` / `STI`	FA / FB	1 byte	2

Table 5.1.6: Machine code, length and clock cycles of the 8086 interrupt instructions (8086 manual).

5.1.3.2 The stack step by step: INT 10h

Let us follow the effect of the steps on the stack with concrete numbers. Suppose the `INT 10h` instruction in a .COM program is at address 0700:0104. When emu8086 loads a .COM program, CS = DS = SS = 0700h and SP = FFFEh. Initially only IF = 1; because bit 1 and bits 12–15 of the 8086 FLAGS register always read as 1, FLAGS = F202h. Suppose the vector of INT 10h is F000:F065, as in Example 5.1.1.

Step	Operation	SP	[FFF8h]	[FFFAh]	[FFFCh]	CS:IP	FLAGS
0	start	FFFEh	—	—	—	0700:0104	F202h
1	FLAGS pushed	**FFFCh**	—	—	**F202h**	0700:0104	F202h
2	IF = TF = 0	FFFCh	—	—	F202h	0700:0104	**F002h**
3	CS pushed	**FFFAh**	—	**0700h**	F202h	0700:0104	F002h
4	IP pushed	**FFF8h**	**0106h**	0700h	F202h	0700:0104	F002h

Step	Operation	SP	[FFF8h]	[FFFAh]	[FFFCh]	CS:IP	FLAGS
5	CS:IP ← vector	FFF8h	0106h	0700h	F202h	**F000:F065**	F002h
6	IRET: IP, CS popped	**FFFCh**	0106h	0700h	F202h	**0700:0106**	F002h
7	IRET: FLAGS popped	**FFFEh**	0106h	0700h	F202h	0700:0106	**F202h**

Table 5.1.7: The stack and registers during INT 10h at 0700:0104 and the IRET at the end of the ISR (SS = 0700h).

The rows of Table 5.1.7 read as follows. In **step 1** SP is first decreased by 2 and becomes FFFCh, and the current value of FLAGS, F202h, is written to this address. In **step 2** IF and TF in the register are cleared; with bit 9 (IF) of F202h cleared, FLAGS = F002h. The copy on the stack stays F202h. In **step 3** SP becomes FFFAh and CS = 0700h is written. In **step 4** SP becomes FFF8h and the return address is written. Because INT 10h is two bytes, the return address is 0104h + 2 = 0106h. In **step 5** CS:IP is loaded from the vector and the ISR starts running. In this example the ISR writes a character to the screen.

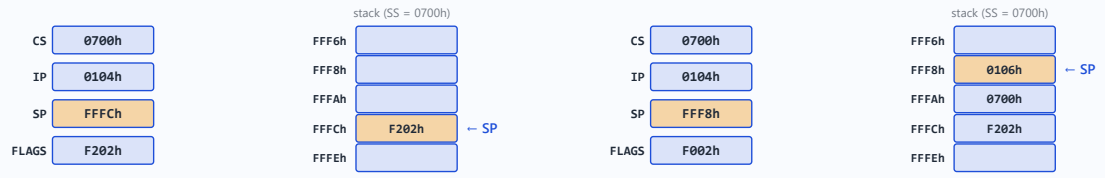


Figure 5.1.5: Left, after FLAGS is pushed (SP = FFFCh); right, after the return address 0106h is pushed (SP = FFF8h). Toward the top of the stack the three words are laid out as FLAGS, CS, IP.

When the ISR finishes and IRET runs, the stack empties in the opposite direction. In **step 6** the 0106h at [FFF8h] is popped into IP, the 0700h at [FFFAh] into CS, and SP = FFFCh. In **step 7** the F202h at [FFFCh] is popped into FLAGS and SP returns to FFFEh. IF is 1 again. The program continues from 0700:0106, that is, from the instruction after INT 10h.

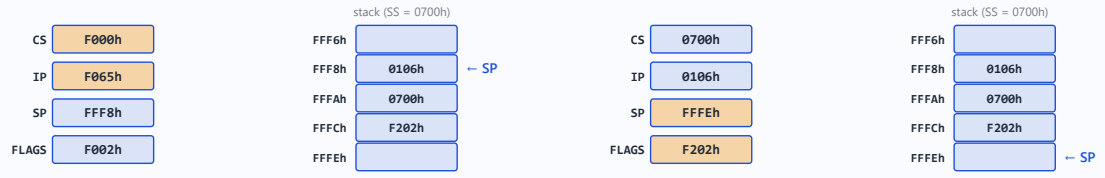


Figure 5.1.6: Left, while the ISR runs: CS:IP = F000:F065, FLAGS = F002h. Right, after IRET: CS:IP = 0700:0106, SP = FFFEh, FLAGS = F202h. The three words on the stack are not erased; only SP moves back.

A detail in the last row deserves attention: after IRET the three words on the stack remain in memory (Figure 5.1.6). POP and IRET do not erase memory; they only increase SP. The region above SP is now considered "free", and the next PUSH or interrupt overwrites these values. This is why old values are visible when you look at the stack window of a debugger.

Example 5.1.3

When the same program is run in emu8086 with its own ISR placed in the INT 10h vector, the words written to the stack can be read. The code is:

```
org 100h
mov ah, 0Eh      ; 0100h, 2 bytes
mov al, 'A'      ; 0102h, 2 bytes
int 10h          ; 0104h, 2 bytes (CD 10)
ret              ; 0106h: return address
```

Right after INT 10h executes, SP = FFF8h and the stack holds the words [FFF8h] = 0106h, [FFFAh] = 0700h, [FFFCh] = F202h. After IRET, SP = FFFEh. The values are exactly the same as in Table 5.1.7.

Common mistake

Taking the IP pushed on the stack to be the address of the INT instruction itself (0104h in this example). The value pushed is the address of the next instruction (0106h). Otherwise IRET would take the processor back to the INT and the program would enter an infinite loop.

5.1.3.3 INT/IRET versus CALL/RET

The similarity between INT and a procedure call is obvious: both write the return address to the stack and jump elsewhere. The differences show why ISRs need a separate return instruction. Table 5.1.8 compares the two pairs.

	CALL / RET	INT n / IRET
Target address	in the instruction (a label)	from the vector table (n × 4)
Pushed on the stack	IP (near) or CS + IP (far)	FLAGS + CS + IP (6 bytes)
Flags	unchanged	IF = 0, TF = 0
Return	RET: pops IP (and CS)	IRET: pops IP, CS, FLAGS
Instruction length	3 bytes (near CALL)	2 bytes (INT 3: 1 byte)
Invoked by	only the program	the program, the processor (exception) or hardware

Table 5.1.8: Comparison of CALL/RET and INT n/IRET.

The most important row is the last. Only the program calls a procedure; at that point the program knows the state of the flags and, if necessary, saves them itself. An ISR, on the other hand, can be invoked by a hardware event without the program's knowledge. That is why the job of saving the flags is given to the processor.

The target address row also gives an important flexibility. The assembler writes the target of a CALL into the instruction, and this value does not change. The target of INT is read from the table at run time. Changing the vector redirects every INT call to another routine without changing any code. This is how DOS, drivers and resident programs can add their own functions to the BIOS services. The technique of **interrupt chaining** rests on this flexibility. The new routine does its own work and then jumps to the old vector, so that the old routine runs too.

In the instruction length row, INT being 2 bytes is also an advantage. A far CALL is 5 bytes; calling a BIOS routine with a far CALL means both longer code and a fixed address that depends on the BIOS version. INT 10h, by contrast, is the same two bytes on every BIOS.

5.1.3.4 An ISR that ends with RET: code trace

To see the practical consequence of the difference in Table 5.1.8, we trace a deliberately faulty program. The program installs its own ISR on type 61h and calls it with INT 61h, but ends the ISR with a near RET instead of IRET.

```
org 100h
mov ax, 0
mov es, ax
mov word ptr es:[61h*4], offset isr
mov word ptr es:[61h*4+2], cs
int 61h
hlt             ; SP = ?
isr:
ret             ; BUG!
```

The code runs line by line as follows. `mov ax, 0` and `mov es, ax` set ES to 0000h. Because an immediate value cannot be loaded directly into a segment register, 0 is put into AX first. From then on every address written with `es:[…]` is inside the vector table. The third line writes the offset of the `isr` label to the IP half of the vector of type 61h, at address 61h × 4 = 184h. The assembler computes the expression `61h*4` itself and puts 0184h in the instruction. The fourth line writes the value of CS to address 186h. CS and the ISR are in the same segment, so the vector points to 0700:0114.

Counting the bytes of the program makes the IP values in the trace table easier to understand. The lengths and addresses of the instructions are: `mov ax, 0` 3 bytes (0100h–0102h), `mov es, ax` 2 bytes (0103h–0104h), the word write with an ES prefix 7 bytes (0105h–010Bh), the CS write 5 bytes (010Ch–0110h), `int 61h` 2 bytes (0111h–0112h), `hlt` 1 byte (0113h) and `ret` 1 byte (0114h). So the `isr` label is 0114h and the return address of INT 61h is 0113h.

Line	Instruction	CS:IP (next)	SP	IF	Stack / memory
2	`mov ax, 0`	0700:0103	FFFEh	1	[FFFEh] = 0000h
3	`mov es, ax`	0700:0105	FFFEh	1	ES = 0000h

Line	Instruction	CS:IP (next)	SP	IF	Stack / memory
4	`mov … [61h*4], offset isr`	0700:010C	FFFEh	1	[0000:0184h] = 0114h
5	`mov … [61h*4+2], cs`	0700:0111	FFFEh	1	[0000:0186h] = 0700h
6	`int 61h`	**0700:0114**	**FFF8h**	**0**	[FFF8h] = 0113h, [FFFAh] = 0700h, [FFFCh] = F202h
9	`ret` (wrong)	**0700:0113**	**FFFAh**	0	[FFFAh] = 0700h, [FFFCh] = F202h left on the stack
7	`hlt`	0700:0114	FFFAh	0	processor halted, IF = 0

Table 5.1.9: Trace of the program whose ISR ends with RET (state after each line executes; SS = 0700h).

Two lines of the trace are critical (Table 5.1.9). In **line 6**, `int 61h` writes 6 bytes to the stack: FLAGS (F202h), CS (0700h) and the return IP (0113h). SP = FFF8h and IF = 0, and CS:IP is loaded from the vector as 0700:0114. In **line 9**, the near `ret` in the ISR pops only the word at the top of the stack: IP = 0113h. CS does not change; because the ISR and the program are in the same segment, this time we happen to return to the right segment. SP, however, increases by only 2 and becomes FFFAh. The expected value is FFFEh; the stack is off by 4 bytes. Because FLAGS was not popped from the stack, IF remains 0.

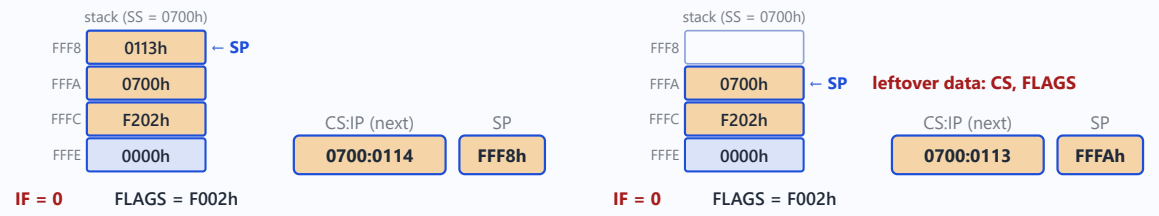


Figure 5.1.7: Left, after `int 61h` : three words on the stack, SP = FFF8h, IF = 0. Right, after the faulty `ret` : only IP has been popped; CS and FLAGS remain on the stack as leftover data, SP = FFFAh, IF is still 0.

When the program reaches `hlt`, there are two problems at once (Figure 5.1.7). First, interrupts are off. HLT stops the processor until an interrupt arrives. Because IF = 0, no request from INTR can wake the processor; only NMI or RESET can. In this small program the processor waits forever. Second, the stack is out of step. If there were a `ret` instead of HLT, it would take the 0700h at the top of the stack for the return address and jump to 0700:0700. There is no meaningful code to execute there.

Trying to fix the problem with RETF is not enough either. The far return instruction pops IP and CS and increases SP by 4. The 2 bytes of FLAGS still remain on the stack and IF does not return to its old state. The rule is therefore absolute: ISRs always end with IRET.

Common mistake

Writing an ISR inside PROC/ENDP and putting `RET` at the end out of habit. The assembler does not object, because RET is a valid instruction. The error shows up only at run time, often long after the ISR, as a wrong return address or as interrupts that never come again. The symptom being far from the cause makes this bug hard to find.

5.1.4 Exceptions

Exceptions are the way the processor reports to the program the special situations it meets while executing an instruction. The 8086 has two true exceptions: divide error (INT 0) and single step (INT 1). The breakpoint (INT 3) and overflow interrupt (INT 4 via INTO) are really software interrupts called by an instruction. But because both are used for debugging and error trapping, they are treated together with exceptions. Intel's 8086 manual groups these four interrupts, together with INT n, under the heading "internal interrupts".

5.1.4.1 Divide error (INT 0)

The DIV and IDIV instructions write the quotient to a fixed-size register. In an 8-bit division the dividend is in AX, the quotient is written to AL and the remainder to AH. In a 16-bit division the dividend is in DX:AX, the quotient goes to AX and the remainder to DX. If the quotient does not fit in the destination register, there is no way to produce a correct result. In this case the processor does not write the result and raises the type 0 interrupt.

Definition 5.1.5

A **divide error** is the type 0 interrupt that the processor raises when, in a DIV or IDIV instruction, the divisor is zero or the quotient does not fit in the destination register. The destination is AL in an 8-bit division and AX in a 16-bit division.

The second condition in the definition is often overlooked. A divide error can occur even without division by zero.

Example 5.1.4

`DIV BL` runs with AX = 1000 (03E8h) and BL = 2. What happens? How can the same division be done without an error?

1. 8-bit division: AX / BL = 1000 / 2 = 500. The quotient is to be written to AL.
2. AL can hold at most 255. Since 500 > 255, the quotient does not fit. Although the divisor is not zero, the processor raises INT 0.

3. The solution is a 16-bit division: DX = 0, AX = 1000, BX = 2 and `DIV BX`. The dividend DX:AX = 0000:03E8h = 1000. The quotient 500 = 01F4h is written to AX and the remainder 0 to DX. AX can hold at most 65,535, and 500 fits easily.

A smaller borderline example: AX = 01FEh (510) / 2 = 255 = FFh fits; AX = 0200h (512) / 2 = 256 does not fit and raises INT 0. These values were verified by running them in emu8086.

The rule can be put briefly: in an 8-bit DIV, if AH ≥ BL, that is, if the high half of the dividend is not smaller than the divisor, the quotient exceeds 255 and the error occurs. In the example AH = 03h and BL = 02h, and since 03h ≥ 02h the error is unavoidable. For a 16-bit DIV the condition is DX ≥ divisor. This check can be made before the division with a CMP.

In the signed division IDIV the limits are the signed range. On the 8086 the quotient of an 8-bit IDIV must lie between −127 and +127; for a 16-bit IDIV the range is −32,767 to +32,767. This limit leaves out the smallest value possible in two's complement (−128 or −32,768). For example, with AX = −128 (FF80h) and BL = 1, `IDIV BL` finds the quotient −128 and on the 8086 still raises INT 0. Later processors accept this smallest value.

Return address. The IP that the 8086 pushes on a divide error is the address of the instruction *after* DIV. So IRET resumes the program by skipping the division. Execution continues as if the division had never been done; the destination registers keep their old values. On the 80286 and later processors this behavior was changed, and the address of the faulting instruction itself is pushed. The error routine can then remove the cause and retry the instruction. An error routine written for the 8086 does not have this option; it can only report to the program in some way that the result is invalid.

What does DOS do? When a program starts, the vector of INT 0 points to DOS's own routine. This routine prints "Divide overflow" on the screen and terminates the program. A program can change this behavior by installing its own routine. The simplest form of this is seen in the trace below.

Common mistake

Thinking of a divide error only as "division by zero" and checking only whether the divisor is zero. When the dividend is large and the divisor small, the quotient does not fit in the destination and the same interrupt occurs. Checking before the division both that the divisor is nonzero and that the quotient will fit (AH < divisor for 8 bits, DX < divisor for 16 bits) prevents INT 0 in the first place.

5.1.4.2 Our own INT 0 handler: code trace

The following program redirects the vector of type 0 to its own routine and deliberately divides by zero. The routine leaves the program an "invalid result" marker by writing FFFFh to AX.

```
org 100h
mov ax, 0
mov es, ax           ; ES = 0: IVT
mov word ptr es:[0*4], offset div_err
mov word ptr es:[0*4+2], cs
mov ax, 100
mov bl, 0
div bl               ; INT 0!
mov dl, al           ; returns here
ret
div_err:
mov ax, 0FFFFh       ; error flag
iret
```

The first two lines set ES to 0000h. `es:[0*4]` is the IP half of the vector of type 0, that is, the very start of the table. The third line writes the offset of the `div_err` label there, and the fourth line writes CS to 0000:0002h. Before the program, the vector pointed to the emulator's default routine; in this trace the old value is taken to be F000:1000h. In this short example the old vector is not saved. In a real program it is saved and restored before exit.

Let us count the offsets: `mov ax, 0` is 3 bytes, `mov es, ax` 2, the two vector writes 7 and 5, `mov ax, 100` 3, `mov bl, 0` 2, `div bl` 2, `mov dl, al` 2 and `ret` 1 byte. So `div bl` is at 0116h, `mov dl, al` at 0118h and the `div_err` label at 011Bh.

Line	Instruction	AX	BL	DL	CS:IP (next)	SP	IF	Vector 0 / stack
2	`mov ax, 0`	0000h	00h	00h	0700:0103	FFFEh	1	F000:1000
3	`mov es, ax`	0000h	00h	00h	0700:0105	FFFEh	1	F000:1000
4	`mov … [0*4], offset div_err`	0000h	00h	00h	0700:010C	FFFEh	1	**F000:011B**
5	`mov … [0*4+2], cs`	0000h	00h	00h	0700:0111	FFFEh	1	**0700:011B**
6	`mov ax, 100`	**0064h**	00h	00h	0700:0114	FFFEh	1	0700:011B
7	`mov bl, 0`	0064h	00h	00h	0700:0116	FFFEh	1	0700:011B
8	`div bl` → INT 0	0064h	00h	00h	**0700:011B**	**FFF8h**	**0**	[FFF8h] = 0118h, [FFFAh] = 0700h, [FFFCh] = F202h
12	`mov ax, 0FFFFh`	**FFFFh**	00h	00h	0700:011E	FFF8h	0	stack unchanged
13	`iret`	FFFFh	00h	00h	**0700:0118**	**FFFEh**	**1**	stack emptied
9	`mov dl, al`	FFFFh	00h	**FFh**	0700:011A	FFFEh	1	—

Line	Instruction	AX	BL	DL	CS:IP (next)	SP	IF	Vector 0 / stack
10	`ret`	FFFFh	00h	FFh	0700:0000	0000h	1	return to DOS

Table 5.1.10: Trace of the program with our own INT 0 handler (state after each line executes).

The rows of Table 5.1.10 read in three phases. **Lines 4–5** change the vector in two pieces. After line 4, half of the vector is new and half is old: F000:011B. This intermediate state points to no routine. In this program the intermediate state is harmless, because only the program's own DIV can trigger type 0. When the vectors of hardware interrupts are changed, however, this intermediate state is a serious danger.

In **line 8**, `div bl` divides by zero. The processor does not write the quotient and starts the type 0 interrupt. FLAGS (F202h), CS (0700h) and the return IP are pushed on the stack. The IP pushed is not the address of DIV, 0116h, but the address of the next instruction, 0118h. SP = FFF8h and IF = 0, CS:IP ← 0700:011B. AX has not changed; the dividend stays 0064h.

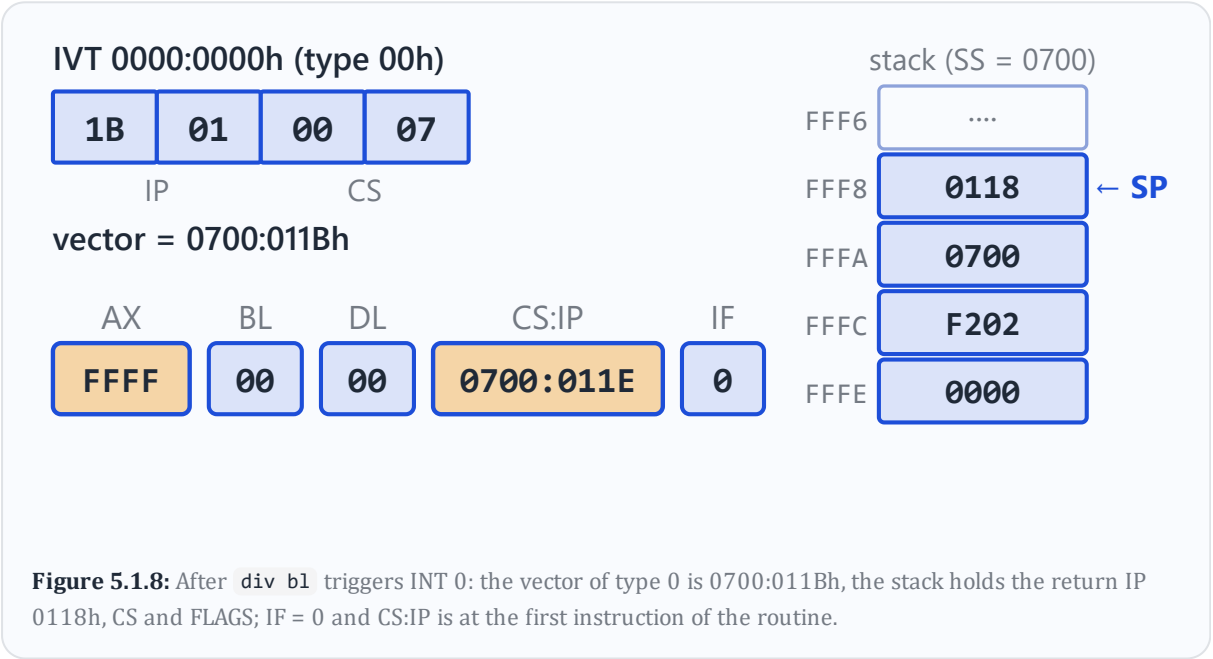


Figure 5.1.8: After `div bl` triggers INT 0: the vector of type 0 is 0700:011Bh, the stack holds the return IP 0118h, CS and FLAGS; IF = 0 and CS:IP is at the first instruction of the routine.

Lines 12–13 are the routine. `mov ax, 0FFFFh` puts an invalid marker in the result register. `iret` pops IP = 0118h, CS = 0700h and FLAGS = F202h; SP = FFFEh and IF = 1. The DIV does not run again. In **line 9** the main program moves AL into DL with `mov dl, al` and learns of the error from DL = FFh. The `ret` on the last line takes the 0000h at the top of the stack in a .COM program and returns to DOS through the INT 20h at the start of the PSP.

The marking method in this program has a weakness. FFh can also be the result of a real division: 255 / 1 = 255. In real programs the error information is returned instead in a separate variable or in a flag, for example CF. To change CF, the ISR must modify the copy of FLAGS on the stack, because IRET restores that copy.

5.1.4.3 INTO and the overflow interrupt (INT 4)

In signed addition and subtraction, OF = 1 if the result does not fit in the signed range. The processor does not raise an interrupt by itself in this case; testing OF is the program's job. `INTO` (interrupt on overflow) makes this test in a single byte.

Definition 5.1.6

INTO is a one-byte (CEh) instruction that calls the type 4 interrupt if OF = 1 and does nothing if OF = 0. It is written immediately after a signed addition or subtraction.

Example 5.1.5

`ADD AL, BL` followed by `INTO` runs with AL = 64h and BL = 32h. Then the same operation is repeated with BL = F6h. In each case, what are AL, OF and CF, and is INT 4 called?

1. 64h + 32h = 96h. In the unsigned reading, 100 + 50 = 150 ≤ 255; there is no carry, CF = 0. In the signed reading, the sum of two positive numbers is +150, which exceeds +127. The top bit of 96h is 1; the result looks like −106. OF = 1 and INTO calls INT 4. AL stays 96h; INTO does not correct the result.
2. 64h + F6h = 15Ah; AL = 5Ah and CF = 1. In the signed reading +100 + (−10) = +90 is correct; the sum of two numbers of different signs cannot overflow: OF = 0. INTO does nothing. CF = 1 does not affect INTO.

Running the two cases in emu8086 gives the same result: in the first the INT 4 routine runs, in the second it does not.

The cost of INTO is small: with OF = 0 it takes only 4 clock cycles. A program that does signed arithmetic can therefore put an INTO after every important operation and catch overflows in a central routine. In practice, though, DOS does not install a meaningful routine for INT 4. The default vector often points to an empty routine consisting only of IRET. A program that is going to use INTO must install its own INT 4 routine.

Common mistake

Assuming that INTO looks at CF. INTO looks only at OF; an unsigned overflow (CF = 1) does not trigger INTO. A program that works with unsigned numbers catches overflow with `JC`, not with INTO.

5.1.4.4 Single step (INT 1)

The most effective way to find a bug in a program is to execute it instruction by instruction and look at the registers after each one. The 8086 supports this in hardware. While the trap flag TF, bit 8 of the FLAGS register, is 1, the processor raises the type 1 interrupt after every instruction. The INT 1 routine, which is part of the debugger, prints the registers on the screen and waits for the user to press a key. It then returns to the program with IRET, the program executes one more instruction, and the cycle repeats.

There is no instruction that changes TF directly. CLC/STC, CLD/STD and CLI/STI have no counterpart for TF. TF is changed through the stack: FLAGS is pushed, popped into a register, bit 8 is set, and the value is loaded back into FLAGS.

```
PUSHF
POP  AX
OR   AH, 01h      ; bit 8 = TF
PUSH AX
POPF              ; TF = 1
```

If FLAGS = F202h at the start, then after `PUSHF` and `POP AX` AX = F202h. Bit 0 of AH is bit 8 of FLAGS; `OR AH, 01h` makes AX F302h. `POPF` loads this value into FLAGS and TF = 1. To clear TF, the same sequence is written with `AND AH, 0FEh`.

There is a subtle but important point here. The INT 1 routine is also a sequence of instructions. If every instruction of this routine also raised INT 1 while TF = 1, the routine would interrupt itself forever. What prevents this is the second of the INT n steps: on entering an interrupt, the processor pushes FLAGS on the stack and clears TF. The INT 1 routine runs with TF = 0, so its own instructions are not single-stepped. When the routine returns with IRET, the copy of FLAGS on the stack restores TF = 1 and tracing continues.

The single-step interrupt has the lowest priority of all interrupts (Table 5.1.12). If another request is also waiting at the same instruction boundary, that one is serviced first.

5.1.4.5 Breakpoint (INT 3)

Single stepping is too slow for stopping at every instruction of a long program. Often you want the program to run at full speed up to a certain point and stop there. This point is called a **breakpoint**. On the 8086 breakpoints are set with INT 3.

INT 3 has a special one-byte encoding: CCh. Written in the general `INT n` form it would be the two-byte `CD 03`. A debugger sets and removes a breakpoint in four steps:

1. It saves the first byte of the instruction to stop at and writes CCh in its place.

2. The program runs at full speed. When that address is reached, CCh, that is INT 3, executes. FLAGS, CS and the address after INT 3 are pushed on the stack; the debugger's INT 3 routine shows the registers.
3. The debugger writes the saved byte back and decreases the return IP on the stack by 1. The return address then becomes the address where CCh was, that is, where the original instruction starts.
4. With IRET the original instruction is executed from the beginning, undamaged.

The reason for the one-byte encoding lies in the first step. The shortest instructions of the 8086 are a single byte: `INC AX`, `PUSH BX`, `CLC` and so on. If the two-byte `CD 03` were written in place of such an instruction, the second byte would corrupt the first byte of the next instruction. If the program reached that next instruction by a jump, a corrupted instruction would now be executed. CCh, on the other hand, fits in place of the first byte of any instruction and affects only that instruction. Decreasing the return IP by 1 also depends on INT 3 being a single byte.

The "single step" button of emu8086 does the same job inside the emulator. On a real 8086 system, DOS's DEBUG program and similar tools work by installing their own routines on the INT 1 and INT 3 vectors.

> **Common mistake**
>
> Taking the CCh bytes for the program's own instructions when the code of a program running under a debugger is dumped from memory. At addresses where a breakpoint is set, CCh temporarily sits in memory. If the program reads its own code or computes a checksum, it may find a different result under the debugger.

5.1.5 Hardware interrupts

5.1.5.1 The INTR and NMI pins

External devices send interrupt requests to the 8086 over two pins. The behavior of the two pins differs from the way they are triggered to the way the type number is determined.

INTR (pin 18) is the maskable interrupt request input. It is level-triggered: the processor samples the pin during the last clock cycle of each instruction and sees the request if the pin is 1 at that moment. If IF = 0 the request is ignored; if the pin stays at 1 it is seen as soon as IF = 1. The type number is not fixed inside the processor. When the processor accepts the request, it produces two pulses on the INTĀ output (pin 24, interrupt acknowledge) and reads the type number from the data bus. This arrangement makes it possible to connect many devices behind a single INTR pin; each device can be given a different type number.

NMI (pin 17) is the non-maskable interrupt input. It is rising-edge triggered: the 0-to-1 transition of the pin is recorded in a latch inside the processor, so the request is not lost. IF is not consulted; it cannot be turned off with CLI. The type number is always 2, so no INTA cycle is needed and the vector is read directly from 00008h. On the IBM PC, NMI is used to report memory parity errors. Circuits that sense loss of power also use NMI to give the processor a few milliseconds to save important data before the supply fails completely.

	INTR	NMI
Pin	18	17
Triggering	level (sampled at the end of an instruction)	rising edge (latched internally)
When IF = 0	held off	accepted
Type number	from the data bus via INTA (08h–0Fh on the PC)	fixed at 2
Use	keyboard, timer, disk, serial port	parity error, power loss

Table 5.1.11: Comparison of INTR and NMI.

Level triggering has a consequence. The device must hold INTR at 1 until the processor accepts the request. If the request comes as a very short pulse and the processor is running with IF = 0 at that moment, the request is lost. On the PC, the 8259A between the devices and the processor solves this problem. The 8259A stores incoming requests in its own register and holds INTR until the request is serviced.

> **Common mistake**
>
> Thinking that the vector of NMI is also read with INTA. The INTA cycle is only for INTR. The type of NMI is fixed (2) and the processor does not ask outside for it. In the same way, the types of INT 0, INT 1, INT 3 and INT 4 are fixed and have nothing to do with external hardware.

5.1.5.2 The INTR acknowledge sequence

Let us follow step by step the chain of events from the moment a key is pressed to the execution of the first instruction of the keyboard ISR. On the PC the keyboard controller is connected to input IRQ1 of the 8259A.

1. The keyboard controller activates the IRQ1 line. The 8259A records the request and, if the mask allows, drives the INTR pin to 1.
2. The 8086 finishes the instruction it is executing. Interrupts are not accepted in the middle of an instruction; the processor samples INTR only at instruction boundaries.
3. If IF = 0 the request is held off and INTR stays at 1. If IF = 1 the request is accepted.

4. The processor produces two INTĀ pulses. The first tells the 8259A to "get ready". On the second pulse the 8259A puts the type number on data bus lines D0–D7; on the PC this is 09h for the keyboard.
5. From here on it is the same as INT n: FLAGS, CS, IP are pushed on the stack; IF = TF = 0; CS:IP ← [n × 4], that is, [0000:0024h].
6. When the ISR has read the key and finished its work, it sends the 8259A an EOI (end of interrupt) and returns with IRET.

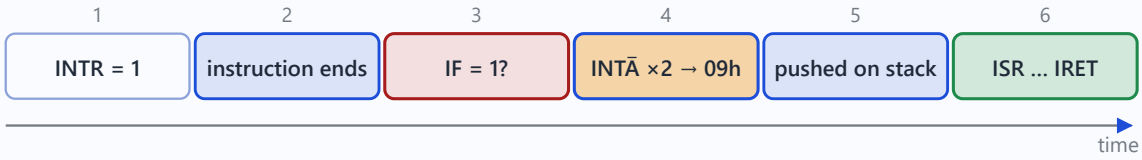


Figure 5.1.9: Timeline from an INTR request to the ISR: the request arrives, the running instruction finishes, IF is tested, the type number is read with two INTA pulses, the return information is pushed on the stack, and the ISR ends with IRET.

The fourth step in Figure 5.1.9 is the only step that separates a hardware interrupt from a software interrupt. The INTA cycle resembles a read bus cycle: the processor puts no meaningful address on the address bus, activates INTĀ instead of RD̄, and takes the byte on the data bus. Between the two pulses, in maximum mode, the LOCK̄ signal is also held active so that no other bus master can take the bus in the meantime. Once the type number is received, the processor behaves as if there were an `INT 09h` instruction in the program.

The rule in the second step, that an interrupt is accepted only at an instruction boundary, is the foundation of ISR design. An instruction is never left half done. For example, while `ADD [BX], AX` reads and writes memory, no interrupt comes in between; the ISR always sees memory either as it was before the instruction or as it is after it. A task done by two separate instructions, however, can be interrupted between the two. This must be kept in mind for data shared by the ISR and the main program.

5.1.5.3 Priority and the PC interrupt map

If several interrupt requests are waiting at the same instruction boundary, the processor decides which to look at first by a fixed order. Table 5.1.12 gives the priority order of the 8086.

Rank	Source	Rationale
1 (highest)	divide error, INT n, INTO	arises from the instruction being executed; it is part of that instruction
2	NMI	urgent hardware event; non-maskable
3	INTR	ordinary hardware request; can be held off by IF
4 (lowest)	single step (TF)	debugging; waiting costs nothing

Table 5.1.12: Interrupt priority order of the 8086.

The logic of the order is clear. An interrupt produced by an instruction itself is part of that instruction's execution and is handled first. Among external requests, NMI comes before INTR because it is an urgent event that a program cannot turn off. Single step is for a person to look at the registers; waiting a few microseconds costs nothing.

This priority is among the processor's *own* inputs. The priority of the eight devices behind INTR relative to one another is decided by the 8259A. In the PC's standard setting IRQ0 has the highest priority and IRQ7 the lowest. Table 5.1.13 gives the hardware interrupts of the IBM PC. At boot the BIOS programs the 8259A so that IRQ0 gets type 08h; the type of IRQ n is therefore 08h + n.

IRQ	Device	Type	Vector address
0	timer (18.2 Hz)	08h	00020h
1	keyboard	09h	00024h
3 / 4	serial ports (COM2 / COM1)	0Bh / 0Ch	0002Ch / 00030h
6	floppy disk	0Eh	00038h
7	printer	0Fh	0003Ch

Table 5.1.13: IBM PC hardware interrupts (type = 08h + IRQ number).

The timer's 18.2 Hz frequency comes from an interesting calculation. The PC's 8253 timer chip is fed by a 1,193,182 Hz clock. When the counter is run with the largest divisor, 65,536, the output frequency is 1,193,182 / 65,536 ≈ 18.2 Hz. That means an INT 08h about every 55 ms. The BIOS uses this interrupt to count the time of day.

The keyboard example also shows the division of labor between a hardware interrupt and a software interrupt. When a key is pressed, the hardware interrupt INT 09h runs first. This ISR reads the key's code from the keyboard controller and puts it into the BIOS keyboard buffer. When a program wants to read a key, it calls the software interrupt INT 16h; INT 16h reads from this buffer. This way the user can press keys even while the program is busy. The keys wait in the buffer and are taken in order when the program is ready. The details of the buffer and the INT 16h services are the subject of the notes of Chapter 5.2, Interrupt Functions.

5.1.5.4 The 8259A interrupt controller

The 8086 has a single INTR pin, while the PC has at least six interrupt sources. The chip that gathers these sources onto one pin, sets a priority order among them and gives the processor the right type number is the 8259A.

Definition 5.1.7

The **programmable interrupt controller** (8259A) is a chip that puts eight interrupt request inputs (IRQ0–IRQ7) into a priority order, masks them, and places the corresponding type number on the data bus during the processor's INTA cycle. On the PC it is at ports 20h and 21h.

Inside the 8259A are three registers, each eight bits wide with each bit corresponding to one IRQ (Figure 5.1.10).

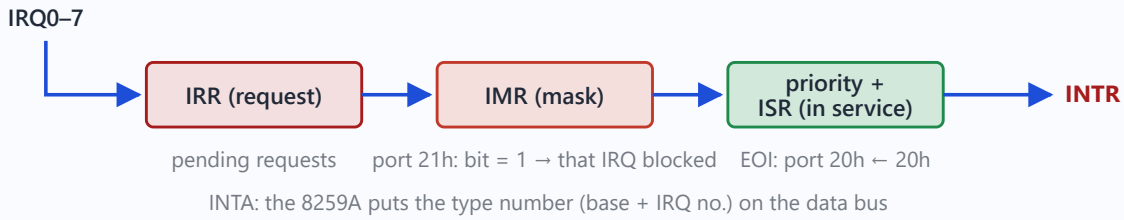


Figure 5.1.10: Inside the 8259A. Requests IRQ0–7 are written to the IRR, those masked in the IMR are filtered out, the priority resolver selects the highest-priority request and drives INTR. During INTA the type number (base + IRQ no.) is put on the data bus and the request's bit moves to the ISR register.

- **IRR** (interrupt request register): holds requests that have arrived but have not yet been serviced. When an IRQ line becomes active, the corresponding bit becomes 1.
- **IMR** (interrupt mask register): holds one mask bit for each IRQ. If the bit is 1, the request on that line is not passed on to the processor. While IF shuts off the whole INTR input, the IMR shuts off lines one by one. On the PC it is read and written at port 21h.
- **ISR** (in-service register): marks the IRQ currently being serviced. During INTA the bit of the selected request moves from the IRR to the ISR. Until the EOI arrives this bit stays 1, and the same or lower-priority IRQs are held off.

The third register's name, ISR, is the same as the abbreviation of interrupt service routine. To avoid confusion, these notes say "ISR register" when speaking of the register.

EOI. The bit in the ISR register does not clear by itself. When the service routine has finished its work, it must tell the 8259A so. This notification is called **end of interrupt** (EOI). On the PC it is done with two instructions:

```
MOV AL, 20h     ; non-specific EOI command
OUT 20h, AL     ; to the 8259A's command port
```

This command clears the highest-priority 1 bit in the ISR register. That the port number and the command value are both 20h is a coincidence. If the EOI is forgotten, the 8259A thinks that IRQ is still in service. The same and lower-priority IRQs never reach the processor again. For example, if the EOI is forgotten in the timer ISR, then because IRQ0 has the highest priority, the keyboard, disk and serial port interrupts all stop. The system appears frozen.

Cascading. Eight inputs are sometimes not enough. 8259As can be cascaded: if a slave 8259A is connected to each input of a master 8259A, the count rises to 8 × 8 = 64 IRQs. The IBM PC/AT connects a second 8259A to input IRQ2 of the master chip. The inputs of the second chip are called IRQ8–IRQ15 and map to types 70h–77h. On the AT, IRQ8 is the real-time clock and IRQ14 is the interrupt of the hard disk.

> **Common mistake**
>
> Reading a 1 bit in the IMR as "interrupt allowed". The IMR is a mask register: 1 blocks, 0 allows. For IF the situation is the opposite: IF = 1 allows. To avoid mixing up the two rules, it helps to read a 1 in the IMR as "masked" and a 1 in IF as "open".

5.1.5.5 Masking with the IMR: code trace

Sometimes a program needs to block the interrupt of just one device for a short time. For example, a piece of code that changes the keyboard buffer directly does not want the keyboard ISR to run in the meantime, but wants the timer to keep counting. CLI is too crude here, because it shuts off the whole INTR input. The right tool is the IMR.

```
org 100h
in  al, 21h        ; read the IMR
or  al, 00000010b  ; mask IRQ1
out 21h, al
mov cx, 3          ; critical work
in  al, 21h
and al, 11111101b  ; unmask IRQ1
out 21h, al
ret
```

The program applies the **read–modify–write** pattern twice. Only one bit of the IMR is to be changed; the other seven bits are the settings of other programs, the BIOS or the operating system. So the IMR is read first, the wanted bit is set with OR or cleared with AND, and the result is written back. Writing `MOV AL, 02h / OUT 21h, AL` directly would mask IRQ1, but it would unmask all the other IRQs.

In the trace the initial value of the IMR is taken, as an example, to be B8h: 1011 1000b. In this value IRQ3, 4, 5 and 7 are masked; IRQ0 (timer), IRQ1 (keyboard), IRQ2 and IRQ6 (floppy disk) are enabled.

Line	Instruction	AL (binary)	AL	CX	IMR	IRQ1
2	`in al, 21h`	1011 1000	**B8h**	0000h	B8h	enabled
3	`or al, 00000010b`	1011 10**1**0	**BAh**	0000h	B8h	enabled
4	`out 21h, al`	1011 1010	BAh	0000h	**BAh**	**masked**
5	`mov cx, 3`	1011 1010	BAh	**0003h**	BAh	masked
6	`in al, 21h`	1011 1010	BAh	0003h	BAh	masked
7	`and al, 11111101b`	1011 10**0**0	**B8h**	0003h	BAh	masked
8	`out 21h, al`	1011 1000	B8h	0003h	**B8h**	**enabled**
9	`ret`	1011 1000	B8h	0003h	B8h	enabled

Table 5.1.14: Trace of the program that masks IRQ1 with the IMR (IMR initially B8h).

Table 5.1.14 reads as follows. In **line 2** the IMR is read: AL = B8h. In **line 3** `or al, 02h` sets only bit 1: 1011 1000 OR 0000 0010 = 1011 1010 = BAh. The other bits are preserved. The IMR has not changed yet; only AL has. In **line 4** `out` writes the value to the 8259A and IMR = BAh. From this moment a request on IRQ1 does not reach INTR; IF and the other IRQs are not affected.

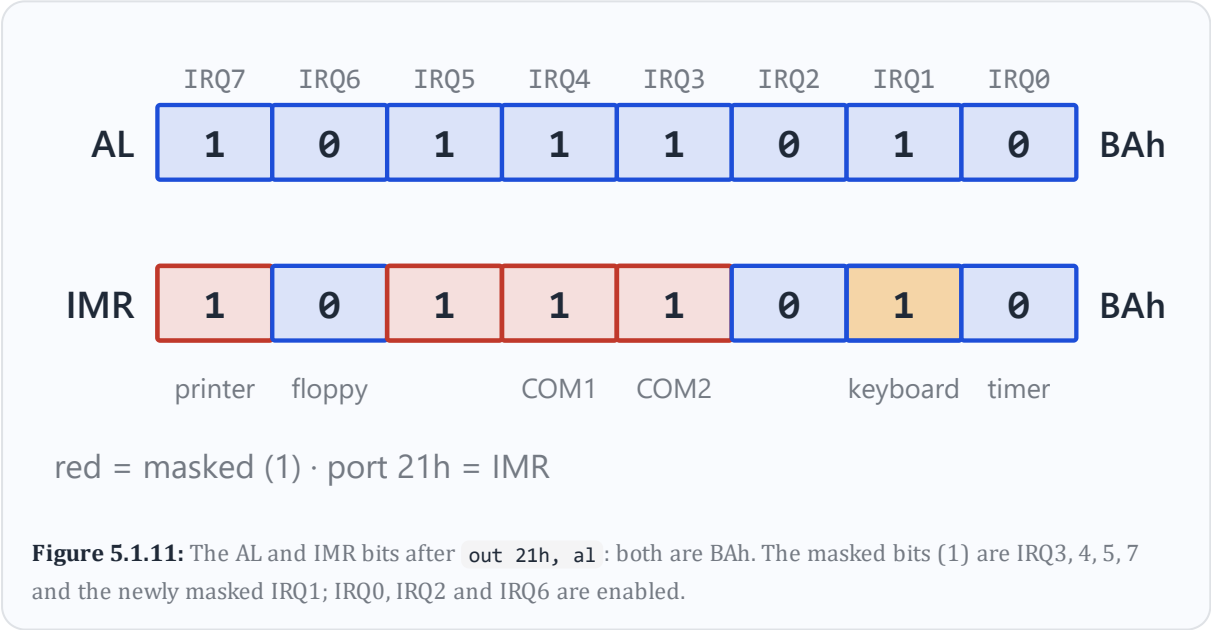


Figure 5.1.11: The AL and IMR bits after `out 21h, al`: both are BAh. The masked bits (1) are IRQ3, 4, 5, 7 and the newly masked IRQ1; IRQ0, IRQ2 and IRQ6 are enabled.

Line 5 stands in for the work that must not be interrupted. If the keyboard is pressed in the meantime, the request is not lost: the 8259A stores it in the IRR. **Lines 6–8** remove the mask with the same pattern. With `and al, 0FDh`, the mask 1111 1101b clears only bit 1: 1011 1010 AND 1111 1101 = 1011 1000 = B8h. With `out` the IMR becomes B8h again. If a keyboard request is waiting in the IRR, it now reaches INTR and is serviced. In **line 9** the program returns; the mask is back to its original state and nobody else's bits have been touched.

The difference between the IMR and CLI is plain in this trace. CLI holds off all maskable interrupts; the timer's pulses wait too, and the clock may fall behind. The IMR holds off only the selected device. The price is that access to the IMR takes two IN/OUT instructions and, because it writes to an I/O port, is slower than CLI.

> **Common mistake**
>
> Writing to the IMR without reading it first: `MOV AL, 02h` and `OUT 21h, AL`. These two lines mask IRQ1 but unmask IRQ3, 4, 5 and 7. If a device with no ISR installed is connected to those lines, the first request that arrives sends the processor to a meaningless vector.

5.1.5.6 When an interrupt is held off

The 8086 accepts an interrupt only at instruction boundaries. In some cases this boundary is deliberately moved forward by one instruction. Table 5.1.15 collects these cases.

Situation	Behavior	Reason
in the middle of an instruction	the request waits until the running instruction finishes	so that no instruction is left half done
prefixes (LOCK, REP, segment prefix)	no interrupt is accepted between a prefix and its instruction	the prefix is part of the instruction
after `MOV SS, …` or `POP SS`	no interrupt arrives until the next instruction has also finished	so that the following `MOV SP, …` completes safely
after `STI`	one more instruction executes, then the interrupt is accepted	so that no interrupt slips between `STI` / `IRET` or `STI` / `HLT`
string instruction with REP	can be interrupted between iterations; after IRET it continues for the remaining CX	so that a long copy does not increase the delay
`HLT` and `WAIT`	INTR or NMI wakes the processor	the wait ends with an interrupt

Table 5.1.15: Situations in which INTR is not accepted immediately.

After SS is loaded. Moving the stack takes two instructions: `MOV SS, AX` and `MOV SP, 0FFFEh`. If an interrupt arrived between the two, the processor would push FLAGS, CS and IP at the address given by the new SS and the old SP. This address is most likely on top of some other data. For this reason the 8086 postpones interrupts by one instruction after every instruction that writes a segment register. For this to work safely, `MOV SP` must be written immediately after `MOV SS`.

After STI. STI enables interrupts not immediately but after the instruction following it has finished. This small delay makes two common patterns safe. The first is the `STI` / `IRET` sequence at the end of an ISR: a new interrupt cannot arrive before IRET finishes and the stack is emptied. The second is the `STI` / `HLT` sequence. The program wants to say "enable interrupts and wait until an interrupt arrives". If an interrupt slipped in between STI and HLT, the ISR would run and return, and HLT would wait needlessly until the next interrupt.

String instructions with REP. A copy of 60,000 bytes with `REP MOVSB` takes hundreds of thousands of clock cycles. Interrupts cannot be allowed to wait for all that time. The 8086 therefore can interrupt a REP instruction between iterations. The return address pushed on the stack is the address of the REP prefix; SI, DI and CX hold the progress so far. After IRET the instruction restarts and runs for the remaining CX iterations. This scheme has a flaw specific to the 8086: on return from an interrupt the processor remembers only one prefix. If an instruction with two prefixes such as `REP ES: MOVSB` is interrupted, the segment prefix is lost on return and the copy continues from the wrong segment. For this reason such a line is put between CLI and STI, or written so that no segment prefix is needed.

HLT and WAIT. HLT halts the processor. INTR (with IF = 1) or NMI wakes the processor; the ISR runs and IRET returns to the instruction after HLT. This is the most economical way of writing a main program as "sleep until something happens". WAIT waits for the 8087 coprocessor to finish its work; an interrupt that arrives during this wait is also accepted.

Common mistake

Loading SP first and then SS when moving the stack, or putting another instruction between the two. The postponement protects only the *one* instruction after the one that writes a segment register. The correct order is `MOV SS, AX` followed immediately by `MOV SP, …`; the safest approach is to put these two lines between CLI and STI as well.

5.1.5.7 Interrupt latency

When a device requests an interrupt, the ISR is expected to start running at once. In reality this time is not zero and for some devices it is bounded.

Definition 5.1.8

Interrupt latency is the time between the arrival of an interrupt request and the start of the first instruction of the ISR.

Latency has two parts. The first is waiting for the instruction that is running when the request arrives to finish. The second is the acceptance process itself: the INTA cycles, three stack writes, two vector reads and refilling the queue. Table 5.1.16 gives the relevant times.

Operation	Clock cycles	Time at 5 MHz
INT n	51	10,2 µs
INT 3	52	10,4 µs
INTO (OF = 1 / OF = 0)	53 / 4	10,6 / 0,8 µs
INTR acceptance (including two INTA)	61	12,2 µs

Operation	Clock cycles	Time at 5 MHz
IRET	24	4,8 µs
16-bit IDIV (register)	165–184	33–36,8 µs

Table 5.1.16: Clock cycle counts of interrupt-related operations on the 8086 (8086 manual).

Example 5.1.6

On a 5 MHz 8086 an INTR arrives just after a 16-bit IDIV has started. What is the worst-case interrupt latency in clock cycles and in µs? What would it be if the processor ran at 8 MHz?

1. In the worst case the running instruction must finish: IDIV takes at most 184 cycles.
2. Then the acceptance process: 61 cycles. The total is 184 + 61 = 245 cycles.
3. At 5 MHz a clock cycle is 1 / 5 MHz = 200 ns. 245 × 200 ns = 49 µs.
4. At 8 MHz a cycle is 125 ns: 245 × 125 ns ≈ 30.6 µs.

If the program had turned interrupts off with CLI at that moment, the time spent with them off adds directly to this latency. A 100 µs CLI–STI interval raises the worst-case latency to about 149 µs at 5 MHz.

What these numbers mean depends on the device. For the timer interrupt, which arrives 18.2 times per second, 49 µs is negligible; there are 55 ms between two interrupts. A serial port is different. Characters arriving at 9600 bit/s are about 1.04 ms apart, and at 115,200 bit/s about 87 µs apart. With a serial port controller that has a one-character buffer, the ISR must read the previous character before the next one arrives. If the latency plus the ISR's running time exceeds this interval, a character is lost.

This calculation gives directly the two basic rules of ISR writing. First, the CLI–STI interval must be kept short, because this time adds to the latency of all devices. Second, the ISR itself must be kept short. While an ISR runs, IF = 0 and the same or lower-priority IRQs wait until the EOI. A long ISR has the same effect on all the other devices as a long CLI.

The share of the processor an ISR takes can also be computed. Suppose the timer ISR takes 200 clock cycles. Running 18.2 times per second, it spends 200 × 18.2 = 3640 cycles. That is only 0.07% of the 5,000,000 cycles per second of a 5 MHz processor. The advantage of interrupts over polling is clear in this figure.

5.1.6 Writing our own interrupt service routine

5.1.6.1 Installing a vector: INT 90h

Installing an interrupt service routine takes three jobs: writing the routine, choosing an unused type, and writing the routine's address into that type's vector. The following program does these three jobs in the simplest form and calls its own interrupt once.

```
org 100h
mov ax, 0
mov es, ax                          ; ES = 0: IVT segment
cli                                 ; interrupts off
mov word ptr es:[90h*4], offset isr ; IP
mov word ptr es:[90h*4+2], cs       ; CS
sti                                 ; interrupts on
int 90h                             ; our own interrupt
ret
isr:
mov ah, 0Eh
mov al, '*'
int 10h
iret                                ; not RET!
```

Every decision in the program has a reason.

- **Choice of type.** 90h is a type that neither the BIOS nor DOS uses on the PC. Its vector is at address 90h × 4 = 240h; IP is at 0000:0240h–0241h and CS at 0000:0242h–0243h.
- **ES = 0.** The vector table is in segment 0000h. Since DS points to the program's own data, the table is accessed through ES, with the `es:` prefix. Because an immediate value cannot be loaded into a segment register, 0 is put into AX first.
- **CLI / STI.** The vector is written as two words with two separate MOVs. After the first MOV, the IP half of the vector is new and the CS half is old. If an interrupt of this type arrived at that moment, the processor would jump to a meaningless address. CLI holds off maskable interrupts during this short interval. Since only the program itself calls type 90h, there is no danger in this example; but for hardware types the same pattern is mandatory, and applying it every time a vector is written is a good habit.
- **The end of the ISR.** The ISR writes an asterisk to the screen with the BIOS INT 10h teletype service and returns with IRET. Another software interrupt can be called from inside an ISR; with nested interrupts, 6 more bytes are added to the stack for each level.

When the program runs, `*` is printed on the screen and the program returns to DOS. This short example does not save and restore the old vector. After the program ends, the vector of type 90h keeps pointing to memory into which another program will now be loaded. If another program calls

INT 90h, the random bytes there are executed as instructions. The correct form, which saves and restores the old vector, is traced separately below.

Under DOS, instead of writing vectors directly in memory, two DOS services can be used: `INT 21h` with AH = 25h sets a vector and AH = 35h reads one. These services manage interrupts themselves while a vector is written and make the operating system aware of the vectors. That makes this the safer way. The register layout of the services is given in the notes of Chapter 5.2, Interrupt Functions; here the direct-write method is used to show how the vector table works.

5.1.6.2 Our own ISR on INT 60h: code trace

A slightly different form of the same arrangement passes a parameter to the ISR in a register. The main program puts the character to be written in AL; the ISR only prepares AH and calls INT 10h. Type 60h, which is reserved for user programs, is used.

```
org 100h
mov ax, 0
mov es, ax           ; ES = 0: IVT
cli
mov word ptr es:[60h*4], offset isr
mov word ptr es:[60h*4+2], cs
sti
mov al, '#'
int 60h              ; our own interrupt
ret
isr:
mov ah, 0Eh
int 10h
iret
```

The offsets are as follows: `mov ax, 0` 0100h, `mov es, ax` 0103h, `cli` 0105h, the two vector writes 0106h and 010Dh, `sti` 0112h, `mov al, '#'` 0113h, `int 60h` 0115h, `ret` 0117h and `isr` 0118h. The vector is at address 60h × 4 = 180h.

Line	Instruction	AX	ES	CS:IP (next)	SP	FLAGS	IVT 0180h–0183h / stack	Screen
2	`mov ax, 0`	0000h	0700h	0700:0103	FFFEh	F202h	?? ?? ?? ??	
3	`mov es, ax`	0000h	**0000h**	0700:0105	FFFEh	F202h	?? ?? ?? ??	
4	`cli`	0000h	0000h	0700:0106	FFFEh	**F002h**	?? ?? ?? ??	
5	`mov … [60h*4], offset isr`	0000h	0000h	0700:010D	FFFEh	F002h	**18 01 ?? ??**	
6	`mov … [60h*4+2], cs`	0000h	0000h	0700:0112	FFFEh	F002h	**18 01 00 07**	

Line	Instruction	AX	ES	CS:IP (next)	SP	FLAGS	IVT 0180h–0183h / stack	Screen
7	`sti`	0000h	0000h	0700:0113	FFFEh	**F202h**	18 01 00 07	
8	`mov al, '#'`	**0023h**	0000h	0700:0115	FFFEh	F202h	18 01 00 07	
9	`int 60h`	0023h	0000h	**0700:0118**	**FFF8h**	**F002h**	[FFF8h] = 0117h, [FFFAh] = 0700h, [FFFCh] = F202h	
12	`mov ah, 0Eh`	**0E23h**	0000h	0700:011A	FFF8h	F002h	stack unchanged	
13	`int 10h`	0E23h	0000h	0700:011C	FFF8h	F002h	stack unchanged	**#**
14	`iret`	0E23h	0000h	**0700:0117**	**FFFEh**	**F202h**	stack emptied	#
10	`ret`	0E23h	0000h	**0700:0000**	**0000h**	F202h	return to DOS	#

Table 5.1.17: Trace of the INT 60h program (state after each line executes; SS = 0700h).

Table 5.1.17 divides into four phases. **Preparation (lines 2–3).** ES is initially 0700h, like the other segment registers of a .COM program. After `mov es, ax`, ES = 0000h and `es:[180h]` now points to byte 0000:0180h in the vector table. The value of these four bytes of the table before the program is unknown; it is shown as "??" in the trace.

Writing the vector (lines 4–7). `cli` clears IF and FLAGS drops from F202h to F002h. Line 5 writes offset isr = 0118h to the IP half; in memory, in little-endian order, `18 01` is seen. Line 6 writes 0700h to the CS half: `00 07`. The vector is now 0700:0118. With `sti`, IF = 1 and FLAGS = F202h. Because bit 1 and bits 12–15 of FLAGS on the 8086 are always 1, IF alone makes a difference of 0200h.

The interrupt (lines 8–9). `mov al, '#'` puts 23h in AL. This value is the parameter passed to the ISR in a register. `int 60h` pushes FLAGS (F202h), CS (0700h) and the return IP 0117h on the stack. INT 60h starts at 0115h and is two bytes; the return address is 0117h. IF = 0 and CS:IP is loaded as 0700:0118 from the vector at 0000:0180h.

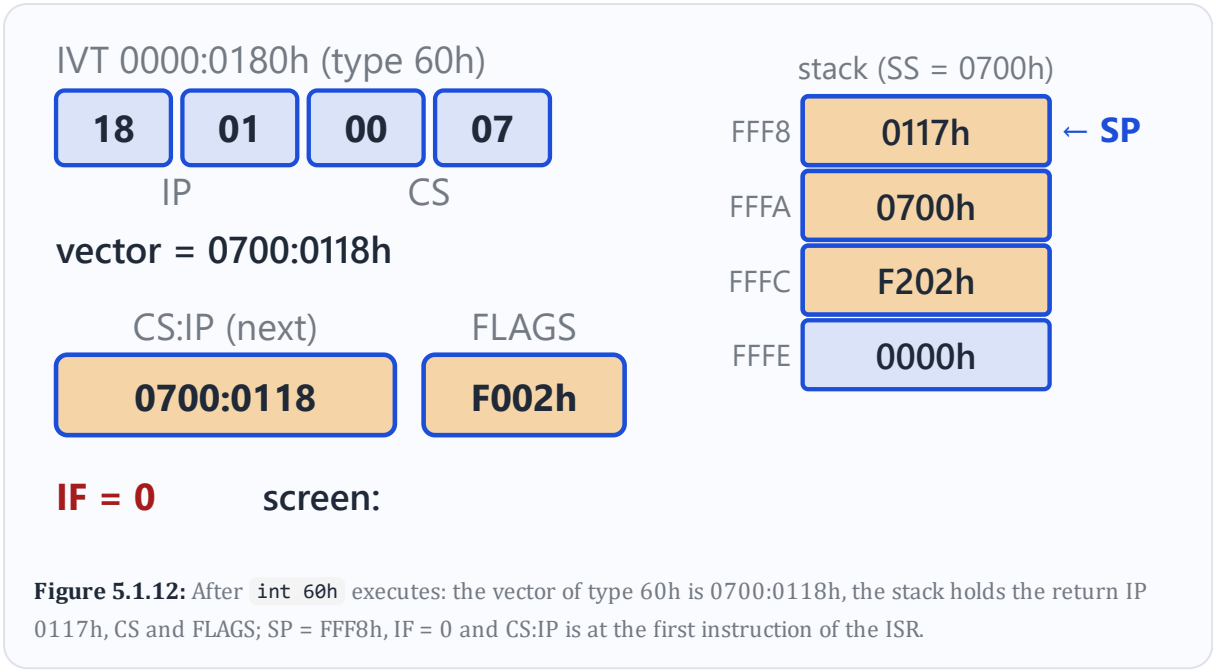


Figure 5.1.12: After `int 60h` executes: the vector of type 60h is 0700:0118h, the stack holds the return IP 0117h, CS and FLAGS; SP = FFF8h, IF = 0 and CS:IP is at the first instruction of the ISR.

The ISR and the return (lines 12–14 and 10). In the ISR, `mov ah, 0Eh` makes AX 0E23h. AH = 0Eh selects the BIOS teletype service; the character in AL will be written. `int 10h` writes `#` on the screen. This is a second interrupt called from inside an interrupt. The BIOS releases its own stack use, and when it returns SP is again FFF8h. `iret` pops IP = 0117h, CS = 0700h and FLAGS = F202h; SP = FFFEh and IF = 1. The `ret` in the main program returns to DOS.

Note that the ISR changes AX. The main program continues after the interrupt with AX = 0E23h. In this program AL is passed to the ISR as a parameter on purpose, so the change in AX causes no problem. But if the ISR were attached to a hardware interrupt, the main program would suddenly find AH equal to 0Eh. An ISR of this kind must save every register it uses.

5.1.6.3 Save, replace and restore a vector: code trace

What the previous two programs lack is that they change the table and leave it that way. A correct program first saves the old vector and restores it when done. The following program does this for type 3 and also shows how the ISR protects the main program's registers.

```
org 100h
xor ax, ax
mov es, ax          ; ES = 0: IVT
mov ax, offset isr3
xchg ax, es:[3*4]   ; new IP ⇔ old
mov old_vec, ax
mov ax, cs
xchg ax, es:[3*4+2] ; new CS ⇔ old
mov old_vec+2, ax
mov ax, 1234h
int 3               ; our ISR
mov bx, old_vec
mov es:[3*4], bx    ; restore
mov bx, old_vec+2
mov es:[3*4+2], bx
ret
isr3: push ax
mov ax, 0E21h       ; AH=0Eh, AL='!'
int 10h
pop ax
iret
old_vec dw ?, ?
```

The new element in the program is the `XCHG` instruction. `xchg ax, es:[3*4]` swaps AX with the word in memory in a single instruction: while the new IP is written to memory, the old IP comes into AX. If the read and the write were done with two separate MOVs, a gap of two instructions would remain between them. With XCHG it is impossible for the new value to be written without the old one being read, or for the old value to be lost before it is read. The old vector is saved in a two-word variable named `old_vec`. `xor ax, ax` zeroes AX; it is one byte shorter than MOV but changes the flags: ZF = 1 and PF = 1. That is why FLAGS is F246h in this program.

The type 3 vector before the program is F000:1003h: emu8086 sets the default vector for each type n to F000:(1000h + n) (F000:1000h for type 0). On a real PC this value varies with the BIOS and any debugger that is loaded; the correctness of the program does not depend on it. The offsets: `int 3` is at 011Ch and is one byte, `isr3` is at 0130h, `old_vec` at 0138h.

Line	Instruction	AX	BX	Type 3 vector	CS:IP (next)	SP	IF	Stack / output
2–3	`xor ax, ax` / `mov es, ax`	0000h	0000h	F000:1003	0700:0104	FFFEh	1	—
4	`mov ax, offset isr3`	**0130h**	0000h	F000:1003	0700:0107	FFFEh	1	—
5	`xchg ax, es:[3*4]`	**1003h**	0000h	**F000:0130**	0700:010C	FFFEh	1	—
6	`mov old_vec, ax`	1003h	0000h	F000:0130	0700:010F	FFFEh	1	old_vec = 1003h

Line	Instruction	AX	BX	Type 3 vector	CS:IP (next)	SP	IF	Stack / output
7	`mov ax, cs`	**0700h**	0000h	F000:0130	0700:0111	FFEEh	1	—
8	`xchg ax, es:[3*4+2]`	**F000h**	0000h	**0700:0130**	0700:0116	FFEEh	1	—
9	`mov old_vec+2, ax`	F000h	0000h	0700:0130	0700:0119	FFEEh	1	old_vec+2 = F000h
10	`mov ax, 1234h`	**1234h**	0000h	0700:0130	0700:011C	FFEEh	1	—
11	`int 3`	1234h	0000h	0700:0130	**0700:0130**	**FFF8h**	**0**	[FFF8h] = 011Dh, [FFFAh] = 0700h, [FFFCh] = F246h
17	`push ax`	1234h	0000h	0700:0130	0700:0131	**FFF6h**	0	[FFF6h] = 1234h
18	`mov ax, 0E21h`	**0E21h**	0000h	0700:0130	0700:0134	FFF6h	0	—
19	`int 10h`	0E21h	0000h	0700:0130	0700:0136	FFF6h	0	**output: !**
20	`pop ax`	**1234h**	0000h	0700:0130	0700:0137	**FFF8h**	0	—
21	`iret`	1234h	0000h	0700:0130	**0700:011D**	**FFEEh**	**1**	stack emptied
12–13	`mov bx, old_vec` / `mov es:[3*4], bx`	1234h	**1003h**	**0700:1003**	0700:0126	FFEEh	1	—
14–15	`mov bx, old_vec+2` / `mov es:[3*4+2], bx`	1234h	**F000h**	**F000:1003**	0700:012F	FFEEh	1	—
16	`ret`	1234h	F000h	F000:1003	**0700:0000**	**0000h**	1	return to DOS

Table 5.1.18: Trace of the program that saves, replaces and restores the vector (SS = 0700h).

The first phase of Table 5.1.18 is changing the vector. In **line 5** XCHG writes the new IP (0130h) to the table and brings the old IP (1003h) into AX. The vector is then F000:0130: half new, half old. In **line 8** the second XCHG writes the new CS (0700h) and brings the old CS (F000h) into AX. The vector is now 0700:0130, and the old vector F000:1003 is saved in the variable `old_vec`. Because only the program calls type 3, CLI was not needed in this example. When the vector of a hardware type is changed, these lines are put between CLI and STI.

The second phase is the interrupt. In **line 10** AX = 1234h; this is the main program's value that must be protected. In **line 11** `int 3` is a single byte (CCh). FLAGS (F246h), CS (0700h) and the return IP 011Dh are pushed on the stack. The return address is 011Ch + 1 = 011Dh; the one-byte nature of INT 3 is visible here as well. CS:IP ← 0700:0130.

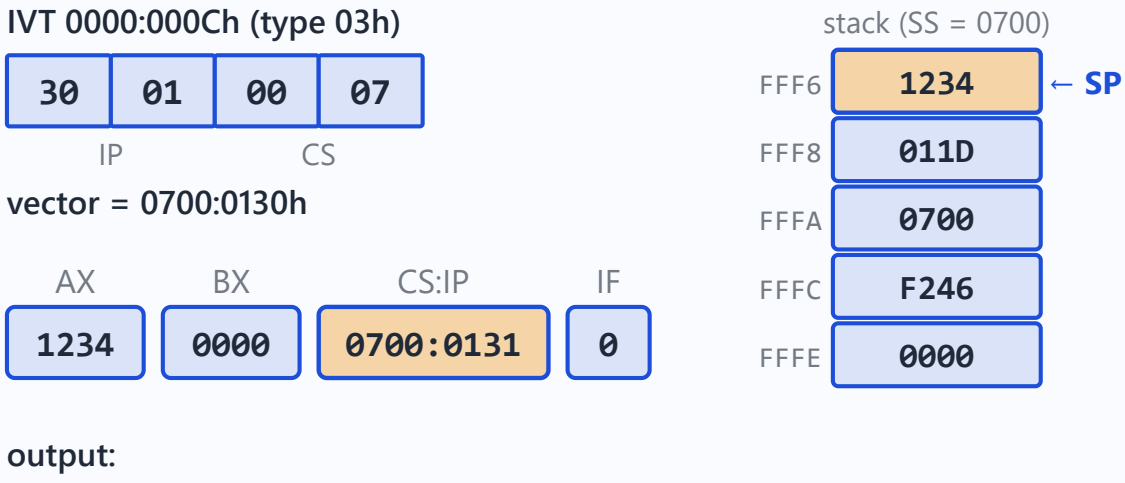


Figure 5.1.13: After the ISR's first instruction `push ax`: there are four words on the stack. At the bottom the FLAGS, CS and IP pushed by the interrupt, on top the AX = 1234h saved by the ISR; SP = FFF6h.

The third phase is the ISR. In **line 17** the ISR first pushes the AX it will use: SP = FFF6h and [FFF6h] = 1234h (Figure 5.1.13). **Lines 18–19** write `!` on the screen with AX = 0E21h. In **line 20** `pop ax` restores 1234h; SP = FFF8h and the top of the stack is again the IP pushed by the interrupt. At this point the number of PUSHes and POPs must be equal. If they were not, IRET would take the wrong words for the return address. In **line 21** `iret` pops IP = 011Dh, CS = 0700h and FLAGS = F246h; SP = FFFEh and IF = 1.

The last phase is the restoration. **Lines 12–15** write the two words in the `old_vec` variable back to the table. Here too the vector passes for a moment through a meaningless intermediate value such as 0700:1003. In **line 16**, when the program returns, AX = 1234h is preserved and the vector table is as it was before the program. This is what it means for a program to exit without leaving a trace in the vector table.

Common mistake

Forgetting to restore the vector on every exit path of the program. If the program restores only at its normal end, but returns early to DOS in an error case, the vector points to code that is no longer loaded. The next INT 3 or hardware interrupt crashes the system. The restoration code must be placed at a single point through which all the program's exits pass.

5.1.6.4 Rules for writing an ISR

The previous traces brought out the rules of ISR writing one by one. Table 5.1.19 collects these rules and the symptoms that appear when they are not followed.

Rule	If ignored
Every register used is protected with PUSH/POP (the processor protects FLAGS).	The main program continues with a register whose value has changed on its own; the bug appears at random moments.
A hardware ISR sends the 8259A an EOI: `MOV AL, 20h` · `OUT 20h, AL`.	The 8259A does not clear the bit in the ISR register; the same and lower-priority IRQs never arrive again.
The ISR is kept short; long work is left to the main program through a flag variable.	Interrupt latency grows and data from fast devices is lost.
INT 21h is not called from a hardware ISR.	DOS is not reentrant; the data of the interrupted DOS call gets corrupted.
The vector is written with CLI or with AH = 25h, and the old one is restored on exit.	After the program ends, the interrupt jumps to memory where something else now lives.
The ISR ends with IRET.	The stack is off by 4 bytes and IF stays off.

Table 5.1.19: Rules for writing an ISR and the problems that appear when they are ignored.

Protecting registers. A hardware ISR can run after any instruction of the program. It cannot be known which register the program is using at that moment. For this reason the ISR pushes every register it uses on entry and pops them in reverse order on exit. There is no need to save the flags; INT and IRET do that themselves. Segment registers must not be forgotten either. If the ISR changes DS to reach its own data, it must save DS too.

A short ISR and a flag variable. Work that takes long is not done inside the ISR. The ISR does only the urgent part: it reads the data from the device, puts it in a buffer and sets a variable to 1. The main program looks at this variable in its loop and does the long work with interrupts enabled. This pattern looks like polling, but what is polled is a variable in memory rather than a slow device, and the ISR has removed the risk of losing data.

Definition 5.1.9

A **reentrant** routine is one that works correctly even when it is called again before an earlier call has finished, for example because of an interrupt. To achieve this, the routine keeps its persistent state not in fixed memory cells but on the stack or in registers.

Not calling DOS. While running, DOS keeps its internal variables and its own stack in fixed memory areas. If a hardware interrupt arrives while the program is in the middle of an INT 21h call and the ISR calls INT 21h again, the second call overwrites the variables and the stack of the first. When the first call is resumed, the data are corrupted. Because DOS is not reentrant, hardware ISRs do not use DOS services. BIOS services such as INT 10h are often safer in this respect; the example ISRs above use INT 10h for this reason and are called by a software interrupt.

STI inside an ISR. An ISR starts with IF = 0. A long ISR can allow higher-priority hardware interrupts at the start of its work with STI; the 8259A already holds off the same and lower-priority IRQs until the EOI. In this case the ISR itself must also be written to be reentrant, because the interrupt of

another device can run another routine in the middle of the ISR and touch the variables the ISR shares.

The skeleton of a typical hardware ISR that applies these rules is as follows. The code is written to show the order of the rules; the lines in the middle change with the device.

```
isr_irq:
    push ax              ; save the registers used
    push ds
    mov  ax, seg data    ; the ISR's own data segment
    mov  ds, ax
    in   al, 60h          ; get data from the device (example port)
    mov  last_code, al    ; put in the buffer / variable
    mov  ready, 1         ; flag for the main program
    mov  al, 20h          ; EOI
    out  20h, al
    pop  ds               ; restore in reverse order
    pop  ax
    iret                  ; not RET
```

Common mistake

Putting an EOI in an ISR called by a software interrupt. EOI is only for hardware interrupts coming from the 8259A. If the ISR of a software interrupt such as INT 60h sends an EOI, the 8259A clears the bit of some other IRQ that is in service at that moment, and the interrupt scheme of that device is disrupted.

5.1.7 Exercises and solutions

The six exercises in the slides are solved here with their intermediate steps.

1. At which physical addresses is the INT 13h vector located?

13h × 4 = 4Ch (decimal: 19 × 4 = 76). The vector is in bytes 0004Ch–0004Fh. IP is at 0004Ch–0004Dh and CS at 0004Eh–0004Fh.

2. If SP = FFFEh before INT 21h runs, what is SP inside the ISR? What is on the stack?

INT pushes three words, 6 bytes: FFFEh − 6 = FFF8h. At the top of the stack, at [FFF8h], is the return IP; at [FFFAh], CS; at [FFFCh], FLAGS. The copy of FLAGS on the stack carries IF = 1; the FLAGS in the register has IF = TF = 0.

3. With IMR = B8h, write the instructions that also mask IRQ0 (the timer) alone. What does the IMR become?

The read–modify–write pattern is used: `IN AL, 21h` · `OR AL, 01h` · `OUT 21h, AL`. 1011 1000 OR 0000 0001 = 1011 1001 = B9h. IRQ3, 4, 5 and 7 stay masked, and IRQ0 is masked too. As long as the timer stays masked the time of day does not advance; for this reason such a mask is kept very short.

4. The vector of type 61h is to be made 0700:0120h. Which bytes are written to which physical addresses?

61h × 4 = 184h. IP is written first, then CS; both are written in little-endian order. 00184h ← 20h, 00185h ← 01h, 00186h ← 00h, 00187h ← 07h. The physical address of this ISR is 07000h + 0120h = 07120h.

5. On a 5 MHz 8086, an INTR arrives just after MUL BX (at most 133 cycles) has started. What is the maximum interrupt latency?

The MUL must finish first: 133 cycles. Then acceptance: 61 cycles. The total is 194 cycles. 194 × 200 ns = 38.8 µs ≈ 39 µs.

6. What problems does an ISR cause by ending with RET instead of IRET?

A near RET pops only IP. CS and FLAGS stay on the stack and SP is off by 4 bytes. Because FLAGS is not restored, IF stays 0 and maskable interrupts are never accepted again. If the ISR is in another segment, CS is also left wrong and the program continues in the wrong segment. The first RET or POP afterwards takes the CS left on the stack for data or a return address. All of these cases are seen in the trace of the ISR that ends with RET.

5.1.8 Summary

- An **interrupt** makes the processor drop its current work at an instruction boundary, run an interrupt service routine (ISR), and return to where it left off with IRET. The interrupted program does not notice that it was interrupted.
- In **polling** the processor keeps asking the device and wastes most of its time; with an **interrupt** the device alerts the processor when it is ready. The price of interrupts is the vector table, stack and ISR machinery.
- Interrupts come from three sources: **software** (INT n, INT 3, INTO), **hardware** (INTR, NMI) and **internal** (divide error INT 0, single step INT 1). Once the type number is determined, all three follow the same steps.
- The **interrupt vector table** is at 00000h–003FFh: 256 types × 4 bytes. The vector of type n is at address n × 4; the low word is IP, the high word is CS, and both are stored little-endian.
- **INT n**: FLAGS is pushed, IF = TF = 0, CS and IP are pushed (SP − 6), CS:IP ← vector. The IP pushed is the address of the instruction after INT. **IRET** pops IP, CS and FLAGS; IF and TF return to their old values.
- Ending an ISR with RET shifts the stack by 4 bytes and leaves IF off; RETF does not pop FLAGS either. ISRs always end with IRET.

- A **divide error** occurs both when the divisor is zero and when the quotient does not fit in the destination. The 8086 pushes the address of the instruction after DIV. **INTO** calls INT 4 only when OF = 1. While **TF** = 1, INT 1 is raised after every instruction; TF is changed through the stack. **INT 3** is a single byte (CCh) and is used for breakpoints.
- **INTR** is level-triggered, accepted only when IF = 1 and at an instruction boundary; the type number is read from the 8259A on the second INTA pulse. **NMI** is rising-edge triggered, non-maskable, and always type 2.
- **Priority**: divide error / INT n / INTO, then NMI, then INTR, and last single step. On the PC the type of IRQ n is 08h + n; timer 08h, keyboard 09h.
- In the **8259A**, the IRR holds pending requests, the IMR holds the masks (port 21h, bit 1 = blocked), and the ISR register holds the IRQ in service. A hardware ISR sends an EOI at its end: `MOV AL, 20h` · `OUT 20h, AL`. The IMR is changed with the read–modify–write pattern.
- After a load of SS and after STI, an interrupt is postponed by one instruction; string instructions with REP can be interrupted between iterations.
- **Interrupt latency** = the remaining time of the running instruction + the acceptance time (61 cycles for INTR). Time spent with interrupts off by CLI adds to latency; the CLI–STI interval and the ISR are kept short.
- To install your own ISR, an unused type is chosen and the vector is written between CLI/STI (or with INT 21h, AH = 25h). The old vector is saved and restored on exit. The ISR protects the registers it uses and ends with IRET.

5.1.9 Key terms

interrupt interrupt service routine (ISR) polling hardware / software interrupt exception interrupt vector interrupt vector table (IVT) interrupt flag (IF) trap flag (TF) single-step breakpoint (INT 3) maskable / non-maskable interrupt interrupt acknowledge (INTA) programmable interrupt controller (8259A) interrupt mask register (IMR) end of interrupt (EOI) interrupt latency reentrant stack offset physical address

5.1.10 Test Yourself

1. In a memory dump the bytes `A5 FE 00 F0` are seen starting at address 0000:0020h. Which type's vector is it, and which device does it belong to on the PC? What is the physical address of the ISR?

   Answer

   20h / 4 = 08h: type 08h, that is, the timer interrupt attached to IRQ0. IP = FEA5h, CS = F000h. Physical address F0000h + FEA5h = FFEA5h.

2. Before the `INT 21h` at 0700:0200 runs, SP = FFF0h and FLAGS = F246h. At the first instruction of the ISR, what are SP, the three words on the stack, and FLAGS?

   Answer

   SP = FFF0h − 6 = FFEAh. [FFEAh] = 0202h (INT 21h is two bytes), [FFECh] = 0700h, [FFEEh] = F246h. In the FLAGS register IF (bit 9) is cleared: F246h − 0200h = F046h. TF is already 0.

3. With BL = 2, what does `DIV BL` give for AX = 01FEh and for AX = 0200h?

   Answer

   510 / 2 = 255 = FFh fits in AL: AL = FFh, AH = 00h. 512 / 2 = 256 does not fit; INT 0 is raised even though the divisor is not zero. A quick check: in the second case AH = 02h ≥ BL = 02h.

4. With IMR = 00h, what instruction sequence masks the timer and keyboard interrupts together? What is the new IMR? What happens if a key is pressed in the meantime?

   Answer

   `IN AL, 21h` · `OR AL, 03h` · `OUT 21h, AL`; the IMR becomes 03h. The request of the key that was pressed waits in the 8259A's IRR; it is serviced when the mask is lifted.

5. On an 8 MHz 8086, an INTR arrives just after an IDIV of at most 184 cycles has started. What is the worst-case interrupt latency in µs?

   Answer

   184 + 61 = 245 cycles. At 8 MHz a cycle is 125 ns: 245 × 125 ns ≈ 30.6 µs.

6. To move the stack, a program writes `MOV SP, 0FFFEh` first and then `MOV SS, AX`. Why is this order not safe?

   Answer

   The postponement protects only the instruction after the one that writes a segment register. An interrupt can arrive between the two instructions, that is, after MOV SP. At that moment SP is new and SS is old, and FLAGS, CS and IP are pushed to a wrong place. The correct order is `MOV SS` first, followed immediately by `MOV SP`.

7. A hardware ISR uses BX and DX but pushes and pops only BX. What symptom is seen in the main

program?

Answer

Because an interrupt can arrive after any instruction of the program, DX changes at moments the main program does not know about. The program usually works correctly, but gives a wrong result in the rare cases when the interrupt arrives exactly while DX is in use. The bug is hard to reproduce and looks random.

8. A debugger has written CCh in place of a one-byte instruction at address 0150h. Which IP does INT 3 push on the stack? Why does the debugger change this value?

Answer

Because INT 3 is a single byte, the IP pushed is 0151h. The debugger writes the saved original byte back to 0150h and decreases the IP on the stack by 1, making it 0150h. After IRET the original instruction is executed from the beginning, undamaged.

9. A memory parity error occurs while a program has interrupts turned off with CLI. Is the interrupt accepted? From which address is the vector read?

Answer

On the PC a parity error is reported through NMI. NMI is non-maskable; it is accepted even when IF = 0. Its type is fixed at 2 and the vector is read from bytes 2 × 4 = 00008h–0000Bh. No INTA cycle is performed.

5.1.11 Further reading

- Intel, *The 8086 Family User's Manual* (1979), the Interrupts section: interrupt types, priority order, the timing of INTR and NMI, and instruction clock cycle counts are found in this source. The values in Tables 5.1.6 and 5.1.16 are taken from it.
- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 12: interrupts in real mode, programming and cascading the 8259A, the PC interrupt scheme.
- A. S. Tanenbaum, *Modern Operating Systems* (3rd ed.): the place of interrupts in the operating system, device drivers, and the comparison of interrupts and polling.
- R. Brown, *Ralf Brown's Interrupt List*: a number-by-number reference list of all BIOS, DOS and driver interrupts on the PC.

5.1.12 References

1. A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.
2. Intel, *The 8086 Family User's Manual*, 1979 — Interrupts section.
3. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapters 6 and 12.
4. emu8086 help documents: the "Interrupts" and "Supported interrupts" pages.
5. R. Brown, *Ralf Brown's Interrupt List* (BIOS/DOS interrupt reference).

PART II · CHAPTER 5.2

Interrupt Functions

An 8086 program does not have to drive the hardware from scratch every time it wants to print text on the screen, read a key from the keyboard, or find out the time. The BIOS in the computer's ROM, and DOS running on top of it, offer these jobs as ready-made routines. A program reaches these routines through a software interrupt: it writes the service number into register AH and the parameters into the other registers, then executes an `INT n` instruction. These notes cover the services of the four most used interrupts: INT 10h manages the screen, INT 16h the keyboard, INT 21h the DOS character, string, date and time services, and INT 1Ah the clock counter. Writing directly to screen memory without the BIOS, the BIOS data area, and the proper ways to terminate a program are also covered. Each service is shown at register level by the step-by-step trace of a short program.

LEARNING OUTCOMES

Call services
Call BIOS and DOS services with the service number in AH and register parameters; know in advance which registers will change.

Write to the screen
Print characters, colored characters and strings with INT 10h and INT 21h; position the cursor; set the attribute byte; write directly to B800h and A000h memory.

Read from the keyboard
Read keys with INT 16h and INT 21h, separate the scan code from the ASCII code, poll for a key without waiting, and take buffered line input.

Read the clock
Read the date and time from DOS as binary and from the RTC as packed BCD and print them; measure time with the tick counter in the BIOS data area.

Terminate a program
End a program properly with RET, INT 20h or INT 21h/4Ch; explain the difference between the COM and EXE templates.

Trace programs
Trace a program that uses services step by step and determine the state of the registers, flags, memory and screen at each step.

5.2.1 BIOS and DOS services

5.2.1.1 Service layers

Writing a single character to the screen means several separate jobs at the hardware level. The code of the character is written to the right cell of the video card's memory. The color byte of the cell is set. The new position of the cursor is reported to the controller on the video card through an I/O port. If the end of the line has been reached, output moves to the next line, and at the end of the screen all the lines are scrolled up by one. It is both laborious and dangerous for every program to do these jobs itself. When the video card changes, all the programs would have to be rewritten.

IBM PC compatible computers solve this problem with layers. Figure 5.2.1 shows four layers. At the bottom is the hardware: the video card, the keyboard controller, the disk and the clock. Hardware is reached directly through I/O ports (`IN` , `OUT`) and memory addresses. One level up is the BIOS. The BIOS routines know the details of the hardware and hide them behind a fixed interface. DOS, above it, offers more abstract services such as file, string, date and program management. At the top is the application program, which only calls these services.

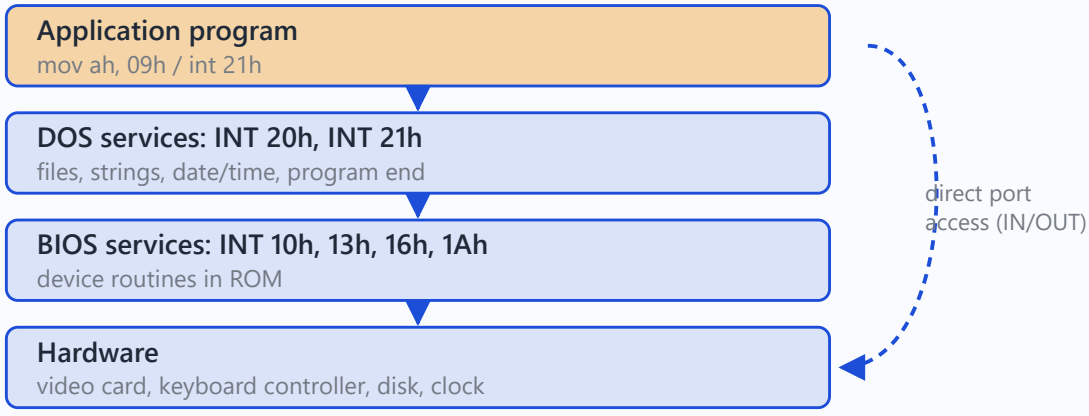


Figure 5.2.1: The service layers. The application calls DOS with `INT 21h` and the BIOS with interrupts such as `INT 10h` , `INT 16h` and `INT 1Ah` . DOS has the BIOS do most of its work; the BIOS reaches the hardware with IN/OUT instructions and memory access.

Definition 5.2.1

The **BIOS** (Basic Input/Output System) is a collection of routines that live in the ROM on the motherboard and drive the hardware directly. It works without an operating system. When the computer is switched on, it tests the hardware, writes the addresses of its own routines into the interrupt vector table, and loads the operating system from the disk. **DOS** (Disk Operating System) is the operating system that runs on top of the BIOS and offers file, memory and program management.

The tool a program uses to reach a BIOS or DOS service is the **software interrupt**. The `INT n` instruction makes the processor read entry number n of the interrupt vector table in memory and branch to the address found there. The return address and flags are pushed on the stack; when the service routine finishes, it returns to the program with `IRET`. The subject here is not this mechanism itself but the services it offers. What matters is this: the program does not have to know the address of the service in memory. It knows only its number.

The benefit of using numbers instead of addresses is great. The BIOSes of different manufacturers place their routines at different addresses. Which version of DOS is loaded also changes the addresses. The vector table is filled at boot with the real addresses of that machine. So a program that says `INT 10h` reaches its own screen routine on every PC compatible machine. On the 8086, an `INT n` instruction takes 51 clock cycles; a direct far `CALL` takes 28. The cost is nearly twice the time, and the return is portability.

The same job can be done at different layers. There are three ways to write a character to the screen: DOS's `INT 21h/02h` service, the BIOS `INT 10h/0Eh` service, and writing directly to screen memory (segment B800h). The upper layer is more portable; its output can also be redirected to a file. The lower layer is faster and more flexible; it gives full control over color and position. A program chooses the layer according to the need and can mix layers in the same program.

5.2.1.2 The service-call pattern

An interrupt number corresponds not to a single service but to a family of services. `INT 21h` does hundreds of jobs such as writing a character, writing a string, opening a file, reading the time and ending the program. Which job is to be done is decided by the number written to register AH before the interrupt. This number is called the **service number** or **function number**. Because AH is 8 bits wide, an interrupt can have at most 256 services.

Definition 5.2.2

The **service-call pattern** has four steps: (1) AH ← the service number; (2) the other parameters are written to registers such as AL, BX, CX, DX, DS:DX or ES:BP; (3) `INT n` is executed; (4) the result is read from registers (AL, AX, CX, DX) or flags (ZF, CF). Which register carries what is defined separately for each service.

The shortest example is making DOS print a letter on the screen:

```
mov ah, 02h    ; service: print char
mov dl, 'A'    ; parameter
int 21h        ; call DOS
```

The first line loads 02h into AH: DOS's "write character" service. The second line puts the ASCII code of the character to be written, 41h, into DL. This service expects the character in DL. The third line calls the interrupt. The processor reads the vector at address 0000:0084h (21h × 4 = 84h) and branches to the DOS routine. The routine writes the character where the cursor is, advances the cursor by one, and returns with `IRET`. The program continues from the instruction after `int 21h`.

What the processor does at the hardware level in this pattern is clear. `MOV AH, 02h` is a two-byte instruction (`B4 02`); the BIU takes it from the prefetch queue, the EU writes the register, and no memory bus cycle is needed. `INT 21h`, on the other hand, pushes three words on the stack (FLAGS, CS, IP) and reads two words from the vector table: five memory bus cycles. The service routine itself is hundreds of instructions. So even printing a single character takes thousands of clock cycles. This time is too short for the user to notice, but it becomes visible in a loop that fills the screen character by character.

Table 5.2.1 lists the frequently used interrupts and their providers.

Interrupt	Provider	Role
`INT 10h`	BIOS	video: mode, cursor, character, string, pixel
`INT 13h`	BIOS	disk
`INT 16h`	BIOS	keyboard
`INT 1Ah`	BIOS	clock counter, real-time clock
`INT 20h`	DOS	terminate the program (COM)
`INT 21h`	DOS	hundreds of services: character and string I/O, files, date and time, exit

Table 5.2.1: Frequently used BIOS and DOS interrupts.

Services may change the registers they use. This is the most error-prone side of the pattern. For example `INT 21h/02h` puts the character in DL into AL on return; `INT 21h/09h` sets AL to 24h. A value kept in AL is lost after these calls. Values that will be needed later are kept in a register the

service does not touch, or on the stack. Some services also use AL for a sub-selection. The VESA video card services are an example: in the service called with `AX = 4F00h`, AH = 4Fh selects the family and AL = 00h selects the job within that family.

> **Common mistake**
>
> Leaving a character that was read in AL and putting another `INT 21h` call in between. For example, a key is first read with `AH = 01h`, then `AH = 02h`, `DL = 0Dh` is called for a carriage return. This call makes AL 0Dh and the key that was read is lost. The value read must be copied to another register such as BL right after the call.

5.2.1.3 ASCII and control characters

Every character written to the screen and read from the keyboard is a number. On the PC these numbers come from the **ASCII** (American Standard Code for Information Interchange) encoding. ASCII assigns characters to the codes 0–127. The codes 20h–7Eh are printable characters: space, digits, letters and punctuation. The codes 00h–1Fh are **control characters**. These carry meaning not as a shape on the screen but as an action.

Code	Name	Effect
07h	BEL	bell
08h	BS	cursor back one (character not erased)
09h	TAB	tab (in DOS services)
0Ah	LF	cursor down one line
0Dh	CR	cursor to the start of the line; the code of the Enter key
1Bh	ESC	the code of the Esc key

Table 5.2.2: Frequently encountered control characters and their effect in INT 10h/0Eh and INT 21h/02h.

The codes of the printable characters are not randomly scattered; a few regularities make programs shorter. Table 5.2.3 summarizes them. The codes of the digits start at 30h: '0' = 30h, '9' = 39h. The numeric value of a digit is therefore found by subtracting 30h from its code. Conversely, a number from 0 to 9 becomes a printable digit when 30h is added. Uppercase letters are 41h–5Ah and lowercase letters 61h–7Ah. The difference between the two groups is exactly 20h, a single bit: bit 5. To convert a lowercase letter to uppercase, bit 5 is cleared (`SUB AL, 20h` or `AND AL, 0DFh`); to convert uppercase to lowercase, it is set (`OR AL, 20h`).

Character	ASCII	Relation
space	20h	the first printable character
'0' … '9'	30h–39h	digit = code − 30h
'A' … 'Z'	41h–5Ah	bit 5 = 0

Character	ASCII	Relation
'a' ... 'z'	61h–7Ah	bit 5 = 1 (+20h)
'$'	24h	end of string in the INT 21h/09h service

Table 5.2.3: Regularities in the ASCII table.

Starting a new line takes two separate characters: CR (0Dh) and LF (0Ah). This distinction is a legacy of the typewriter. CR returns the print head to the start of the line; LF pushes the paper up one line. Figure 5.2.2 compares the three cases. If only CR is sent, the cursor returns to the start of the line and the next text lands on top of the same line. If only LF is sent, the cursor goes down one line but its column does not change; the text drifts to the right like a staircase. When both are sent together, the cursor comes to the start of the next line.

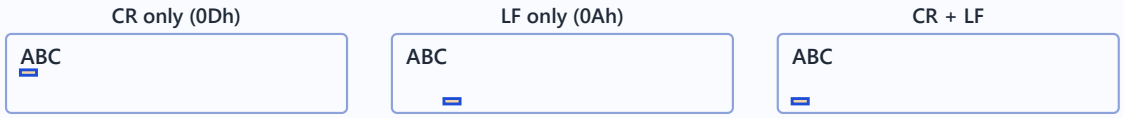


Figure 5.2.2: The effect of CR and LF. CR alone returns to the start of the line; LF alone goes down a line but keeps the column; CR LF together open a new line.

Letters outside the English alphabet (such as the Turkish ç, ğ, ı, ö, ş, ü) are not in 7-bit ASCII. DOS shows them with codes between 128 and 255, according to the active **code page**; for Turkish this page is CP857. The same byte may appear as a different letter in another code page. For this reason the example programs in the slides and in these notes use no such letters in strings: plain ASCII text only.

Example 5.2.1

AL = 37h ('7') and BL = 32h ('2') are two digit characters. To print their sum as a single digit, 30h is first subtracted from both: 07h and 02h. The sum is 09h. To print it, 30h is added: 39h = '9'. If the characters were added directly, 37h + 32h = 69h = 'i' would result. ASCII codes are names of characters, not numbers; they are converted to numbers before arithmetic.

5.2.2 INT 10h: video services

5.2.2.1 The service table and video modes

All the BIOS services related to the screen are gathered under `INT 10h`. Selecting the screen mode, moving the cursor, writing colored characters, printing a string in a single call, clearing a region of the screen and drawing pixels in graphics mode are all done with these services. Table 5.2.4 gives the most frequently used ones with their input and output registers.

AH	Service	Input	Output / notes
00h	set video mode	AL = mode (03h: 80×25 color text)	the screen is cleared
01h	cursor shape	CH = start, CL = end scan line	CH = 20h → cursor hidden
02h	set cursor position	BH = page, DH = row, DL = column	–
03h	get cursor position	BH = page	DH, DL = position; CH, CL = shape
06h	scroll window up	AL = number of lines (0 = clear), BH = attribute, CH,CL – DH,DL = corners	clears the screen or a window
09h	write character and attribute	AL = character, BH = page, BL = attribute, CX = repeat count	cursor does not advance
0Ah	write character only	AL, BH, CX	the cell's attribute is kept
0Ch	write pixel	AL = color, BH = page, CX = x, DX = y	in graphics modes
0Eh	teletype output	AL = character, BH = page	cursor advances; CR, LF, BS, BEL are interpreted
13h	write string	AL = mode, BH, BL, CX = length, DH,DL = position, ES:BP = string	on AT and later BIOSes

Table 5.2.4: INT 10h video services.

The video card can work in different **video modes**. In text modes the screen is made of cells, each holding one character. In graphics modes the screen is made of pixels and the color of each pixel is written individually. The mode is selected with the `AH = 00h` service by giving the number in AL. Table 5.2.5 shows the frequently used modes and the segment at which the video memory starts.

AL	Mode	Video memory
01h	40×25 text, 16 colors	B800h
03h	80×25 text, 16 colors	B800h
07h	80×25 monochrome	B000h
04h	320×200 graphics, 4 colors	B800h
13h	320×200 graphics, 256 colors	A000h

Table 5.2.5: Frequently used video modes.

Selecting a mode also clears the screen. So the two lines often seen at the start of programs are the shortest way to get a clean 80×25 screen:

```
mov ah, 00h
mov al, 03h    ; or mov ax, 0003h
int 10h        ; clean 80×25 screen
```

AH and AL can be loaded together with a single `MOV AX, 0003h` instruction. Since AH is the high byte of AX it becomes 00h, and since AL is the low byte it becomes 03h. This three-byte instruction is both shorter and faster than two separate two-byte `MOV`s.

In mode 03h the screen is 25 rows and 80 columns (Figure 5.2.3). Rows are numbered from 0 to 24 and columns from 0 to 79. The top-left corner is (0, 0) and the bottom-right corner is (24, 79). In the BIOS services the row number is given in DH and the column number in DL. The emu8086 screen window is in this mode by default; it supports graphics modes only to a limited extent.

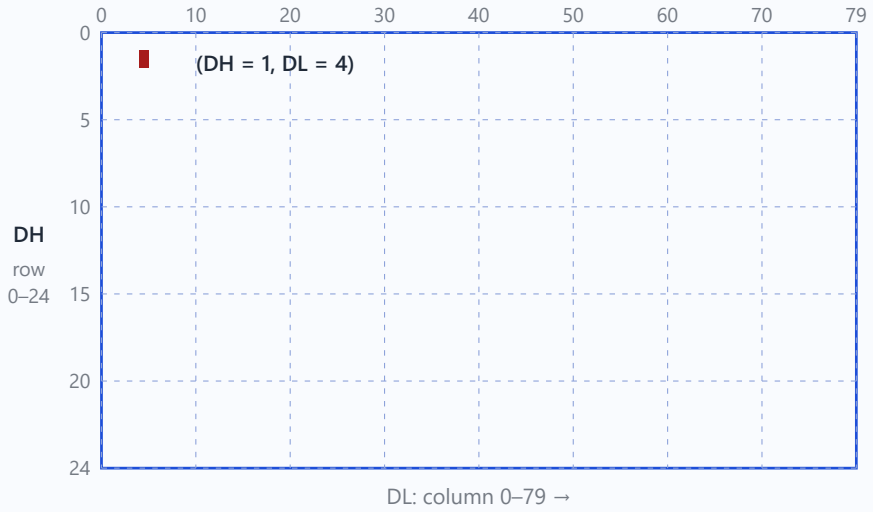


Figure 5.2.3: The 80×25 text screen. The row is given in DH (0–24) and the column in DL (0–79); the marked cell is the position (DH = 1, DL = 4).

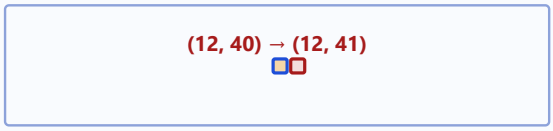
5.2.2.2 The cursor

The **cursor** is the position on the screen where the next character will be written. The BIOS keeps a cursor position for each display page. `AH = 02h` moves the cursor to row DH and column DL; `AH = 03h` reads the cursor position into DH and DL. Both services expect a **display page** number in BH. In 80×25 mode the video card's memory is large enough to hold several screens; VGA has 8 pages (0–7). Only one of the pages is visible. Page 0 is normally used.

Row and column can be loaded together with a single `MOV DX` instruction. `mov dx, 0C28h` puts 0Ch = 12 in DH and 28h = 40 in DL: the middle of the screen. The code below first moves the cursor to the middle, then reads the position and moves it one to the right.

```
; move the cursor to the middle
mov ah, 02h
mov bh, 0
mov dx, 0C28h  ; row 12, column 40
int 10h
; move the cursor one right
mov ah, 03h
mov bh, 0
int 10h        ; DH, DL = position
inc dl
mov ah, 02h
int 10h
```

The first four lines move the cursor to (12, 40). The next three lines read the position; on return DH = 12, DL = 40, and the shape of the cursor is written to CH, CL. `INC DL` makes the column 41. The last two lines write the new position. BH is not reloaded because it did not change after the 03h call. Figure 5.2.4 shows these steps and the cell's place in screen memory.



DH = 0Ch = 12 (row), DL = 28h = 40 (column)
read with 03h → inc dl → write with 02h: (12, 41)
offset (12 × 80 + 40) × 2 = 07D0h

Figure 5.2.4: `mov dx, 0C28h` places the cursor at (12, 40); when the position read with 03h and incremented with `inc dl` is written with 02h, the cursor moves to (12, 41).

The cursor position is not merely a BIOS variable; it is also held in the video card's hardware. In color text mode, two registers of the display controller (the CRTC) determine in which cell the cursor blinks. The BIOS writes to these registers with `OUT` instructions through I/O ports 3D4h (index) and 3D5h (data). The `AH = 02h` call spares the program this port detail. The same position is also stored in the BIOS data area; characters written with `INT 21h` also start from this cursor position.

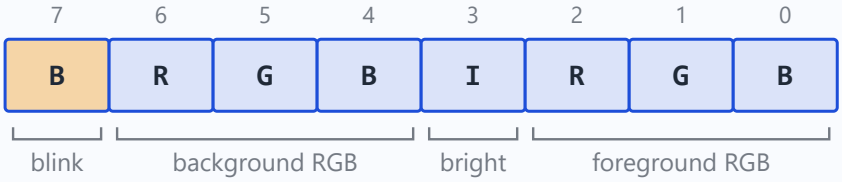
Every cell also has a place in screen memory. In mode 03h, 2 bytes are used per cell and the rows follow one another. The offset of the (row, column) cell is computed as follows:

$$\text{offset} = (\text{row} \times 80 + \text{column}) \times 2$$

For (12, 40) we get (960 + 40) × 2 = 2000 = 07D0h. For (1, 4) it is (80 + 4) × 2 = 168 = 00A8h. At the even address is the character's code, and at the odd address after it is its color.

5.2.2.3 The attribute byte

In text mode the color of each cell is determined by one byte. This byte is called the **attribute byte**. Figure 5.2.5 shows the bits of the byte. The low four bits (bits 3–0) are the foreground, that is the color of the text. Bits 2–0 turn on the red, green and blue components; bit 3 makes the color brighter. Bits 6–4 are the color of the background: again red, green, blue. Bit 7 turns on blinking of the character.



Example: 1Eh = 0001 1110b → blue background, yellow (bright brown) text

Figure 5.2.5: The attribute byte: bit 7 blink, bits 6–4 background (RGB), bit 3 intensity, bits 2–0 foreground (RGB). 1Eh = 0001 1110b is yellow text on a blue background.

Definition 5.2.3

The **attribute byte** is the byte that determines the foreground and background color and the blinking of a character cell in text mode. Written in hexadecimal, the high digit gives the background and the low digit the foreground: attribute = background × 16 + foreground. If bit 7 = 1 the character blinks (depending on the BIOS setting, the background can instead be shown bright).

The sixteen colors come from all the combinations of four bits. Table 5.2.6 gives the numbers and names. The bright colors (8–F) are formed by adding bit 3 to their dark counterparts: 1 is blue, 9 light blue; 4 is red, C light red. Two exceptions stand out: 6 is brown, not yellow, and its bright version E is yellow. 7 is light gray and F is white.

No.	Color	No.	Color
0	black	8	dark gray
1	blue	9	light blue
2	green	A	light green
3	cyan	B	light cyan
4	red	C	light red
5	magenta	D	light magenta
6	brown	E	yellow
7	light gray	F	white

Table 5.2.6: The sixteen colors of text mode.

Example 5.2.2

The default attribute is 07h: background 0 (black), foreground 7 (light gray). For white text on a blue background, background 1 and foreground F are chosen: 1 × 16 + 15 = 31 = 1Fh. Yellow text on a red background is 4Eh. 8Fh = 1000 1111b, because bit 7 is set, is blinking white text on a black background. Since only three bits are given to the background, a light green (A) background cannot be chosen directly: if A0h is written, the top 1 bit falls on bit 7 and blinking is turned on over a green background.

The attribute explorer in the slides produces the attribute byte and the `INT 10h/09h` call that uses it when a foreground and a background are selected. The same calculation is also easy by hand: the two color numbers are written side by side as two hexadecimal digits.

5.2.2.4 Writing characters: 09h and 0Eh

INT 10h offers two main services for writing characters, and the two behave very differently. `AH = 09h` takes the character from AL and its color from BL, and writes the same character CX times side by side. It does not advance the cursor. `AH = 0Eh` is **teletype** output: it writes the character where the cursor is and advances the cursor by one. At the end of a line it moves to the next line, and on the last line it scrolls the screen up. It also interprets the characters CR, LF, BS and BEL as commands. Table 5.2.7 puts the differences side by side.

Feature	AH = 09h	AH = 0Eh (teletype)
Inputs	AL = character, BH = page, BL = attribute, CX = repeat count	AL = character, BH = page
Color	yes (BL)	none in text mode; the cell's color is kept
Cursor	does not advance	advances; wraps to the next line at the end of a line
CR, LF, BS, BEL	printed as symbols (♪, ◘ ...)	interpreted as commands
Screen scrolling	none	the screen scrolls up on the last line
Typical use	colored boxes, frames, the same character CX times	plain text flow

Table 5.2.7: Comparison of AH = 09h and AH = 0Eh.

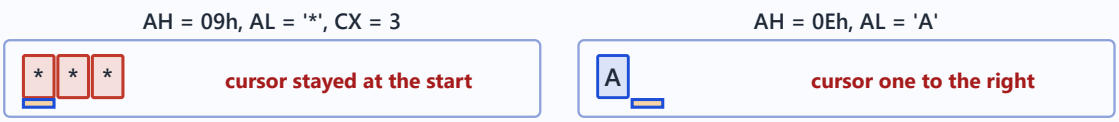


Figure 5.2.6: 09h writes three stars but the cursor stays on the first cell; 0Eh writes 'A' and moves the cursor one to the right.

That 09h does not advance the cursor looks odd at first, but it suits the purpose of the service. To draw the top edge of a frame, the same character is written CX times side by side; the program decides where the cursor goes. There are two common ways to write colored plain text. In the first, each character is written in color with 09h and CX = 1, and then the cursor is moved one to the right with 02h. In the second, the whole string is written in a single call with 13h.

> **Common mistake**
>
> Writing characters one after another with 09h without moving the cursor. Each new character is written over the previous one and only the last is visible on the screen. After 09h the cursor is either advanced by hand with `AH = 02h` or the colored cell is written over with 0Eh.

5.2.2.5 Code trace: mode, cursor, 09h and 0Eh

The following program uses four services one after another. First it clears the screen and moves the cursor to (1, 4), writes three yellow stars on a red background there, and then writes 'A' and 'B' with teletype. The purpose of the program is to show the difference between 09h and 0Eh in cursor and color handling in one place.

```
org 100h
mov ax, 0003h    ; AH=00h, AL=03h
int 10h          ; clears screen
mov ah, 02h      ; set cursor
mov bh, 0        ; page 0
mov dx, 0104h    ; DH=1, DL=4
int 10h
mov ah, 09h      ; char + attribute
mov al, '*'
mov bl, 4Eh      ; yellow on red
mov cx, 3        ; 3 times
int 10h          ; cursor stays!
mov ah, 0Eh      ; teletype
mov al, 'A'
int 10h          ; cursor moves
mov al, 'B'
int 10h
ret
```

The program is a .COM program: it begins with `org 100h`, and at the start the general registers are zero. The first two lines set mode 03h. The screen is cleared and the cursor comes to (0, 0). The next four lines move the cursor. The `mov dx, 0104h` instruction writes 01h to DH and 04h to DL. Because of the little-endian layout the bytes of the instruction are `BA 04 01`. After `int 10h` the cursor is at (1, 4) but there is nothing on the screen yet.

Then the 09h call is prepared. AL = 2Ah ('*'), BL = 4Eh, CX = 3 are loaded. BH is still 0; it does not need to be written again. After the call, the cells (1, 4), (1, 5) and (1, 6) hold yellow stars on a red background. The cursor, however, has not budged: it is still at (1, 4).

In the last part AH = 0Eh is loaded and 'A' is put in AL. `INT 10h` writes 'A' at (1, 4), where the cursor is; that is, over the first star. Then it moves the cursor to (1, 5). Because the 0Eh service does not change AH, only AL is changed for the second character. 'B' is written over the second star at (1, 5) and the cursor moves to (1, 6). `AB*` appears on the screen. Because 0Eh takes no attribute in text mode, 'A' and 'B' keep the color 4Eh left by the stars: yellow on red.

Step	Instruction	AX	BX	CX	DX	Cursor	Explanation
1	`mov ax, 0003h`	0003h	0000h	0000h	0000h	(0, 0)	AH = 00h (set mode), AL = 03h (80×25 text)
2	`int 10h`	0003h	0000h	0000h	0000h	(0, 0)	mode set, screen cleared, cursor at (0, 0)
3	`mov ah, 02h`	0203h	0000h	0000h	0000h	(0, 0)	AH = 02h: set cursor position
4	`mov bh, 0`	0203h	0000h	0000h	0000h	(0, 0)	BH = page 0
5	`mov dx, 0104h`	0203h	0000h	0000h	0104h	(0, 0)	DH = 1 (row), DL = 4 (column)
6	`int 10h`	0203h	0000h	0000h	0104h	(1, 4)	cursor moved to (1, 4); screen empty
7	`mov ah, 09h`	0903h	0000h	0000h	0104h	(1, 4)	AH = 09h: character + attribute
8	`mov al, '*'`	092Ah	0000h	0000h	0104h	(1, 4)	AL = '*' = 2Ah
9	`mov bl, 4Eh`	092Ah	004Eh	0000h	0104h	(1, 4)	BL = 4Eh: red background, yellow text
10	`mov cx, 3`	092Ah	004Eh	0003h	0104h	(1, 4)	CX = 3 repeats
11	`int 10h`	092Ah	004Eh	0003h	0104h	(1, 4)	`***` in cells (1, 4)–(1, 6); cursor in place
12	`mov ah, 0Eh`	0E2Ah	004Eh	0003h	0104h	(1, 4)	AH = 0Eh: teletype
13	`mov al, 'A'`	0E41h	004Eh	0003h	0104h	(1, 4)	AL = 'A'
14	`int 10h`	0E41h	004Eh	0003h	0104h	(1, 5)	'A' over the first star; cursor advanced
15	`mov al, 'B'`	0E42h	004Eh	0003h	0104h	(1, 5)	AL = 'B'
16	`int 10h`	0E42h	004Eh	0003h	0104h	(1, 6)	'B' over the second star: `AB*` on the screen
17	`ret`	0E42h	004Eh	0003h	0104h	(1, 6)	0000h popped from the stack → IP
18	`INT 20h (PSP:0000h)`	0E42h	004Eh	0003h	0104h	(1, 6)	the INT 20h in the PSP ends the program

Table 5.2.8: Trace of the mode, cursor, 09h and 0Eh program (each row is the state after the instruction executes).

The trace also shows how the BIOS services treat the registers. No call changed AX, BX, CX or DX; they changed only the screen and the cursor. Unlike the INT 21h services, these INT 10h services preserve AL. Even so, this assurance should not be relied on without looking at the documentation of each service. Read services such as `AH = 03h` deliberately change DX and CX on return.

5.2.2.6 Writing strings: 13h

`AH = 13h` writes a string, in color, in a single call. It is found in the BIOS of the AT and later computers. It has many parameters: AL gives the write mode, BH the page, BL the attribute, CX the length of the string, DH and DL the starting position. The address of the string is in **ES:BP**. This differs from DOS's string service (DS:DX) and is often confused with it. No marker such as '$' is looked for at the end of the string; the length is taken from CX.

AL	String format	Cursor
00h	characters only; all with the attribute in BL	stays in place
01h	characters only; all with the attribute in BL	moves to the end of the string
02h	character, attribute pairs	stays in place
03h	character, attribute pairs	moves to the end of the string

Table 5.2.9: The AH = 13h write modes (AL).

```
org 100h
mov ax, 1301h    ; AH=13h, AL=01h
mov bx, 001Eh    ; BH=0, BL=1Eh
mov cx, 7        ; string length
mov dx, 0205h    ; row 2, col 5
lea bp, msg      ; ES:BP → string
int 10h
ret
msg db 'Welcome'
```

The program loads four register pairs with one instruction each. AX = 1301h means AH = 13h and AL = 01h: all characters in the color of BL, the cursor advances. BX = 001Eh is BH = 0 for the page and BL = 1Eh for the color; yellow text on a blue background. CX = 7 is the length of the string `'Welcome'`. DX = 0205h puts the start at row 2, column 5. `LEA BP, msg` loads the offset of the string into BP.

The offset of the string is found from the byte counts of the preceding instructions. The four three-byte `MOV`s occupy 0100h–010Bh. emu8086 assembles the directly addressed `LEA BP, msg` instruction to the same three-byte code as `MOV BP, OFFSET msg` (`BD 12 01`); the instruction is at 010Ch–010Eh. The two-byte `INT 10h` is at 010Fh–0110h and the one-byte `RET` at 0111h. So `msg` starts at 0112h. In a .COM program all the segment registers point to the same value, so ES = DS and ES:BP points to the string correctly.

Step	Instruction	AX	BX	CX	DX	BP	Cursor	Explanation
1	`mov ax, 1301h`	1301h	0000h	0000h	0000h	0000h	(0, 0)	AH = 13h, AL = 01h (attribute in BL, cursor advances)
2	`mov bx, 001Eh`	1301h	001Eh	0000h	0000h	0000h	(0, 0)	BH = 0 (page), BL = 1Eh (blue background, yellow text)
3	`mov cx, 7`	1301h	001Eh	0007h	0000h	0000h	(0, 0)	string length 7
4	`mov dx, 0205h`	1301h	001Eh	0007h	0205h	0000h	(0, 0)	DH = 2, DL = 5: starting position
5	`lea bp, msg`	1301h	001Eh	0007h	0205h	0112h	(0, 0)	BP = the offset of `msg` ; ES = DS
6	`int 10h`	1301h	001Eh	0007h	0205h	0112h	(2, 12)	`Welcome` starting at (2, 5); cursor at (2, 12)
7	`ret`	1301h	001Eh	0007h	0205h	0112h	(2, 12)	program ends

Step	Instruction	AX	BX	CX	DX	BP	Cursor	Explanation
8	`INT 20h` (`PSP:0000h`)	1301h	001Eh	0007h	0205h	0112h	(2, 12)	

Table 5.2.10: Trace of the program that writes a colored string with AH = 13h.

The call writes `Welcome` into seven cells starting at (2, 5) and, because AL = 01h, moves the cursor to (2, 12): 5 + 7 = 12. With AL = 00h the string would still be written but the cursor would stay at (2, 5). If AL = 02h or 03h had been chosen, the string would have to carry its own attribute byte after each character; BL would then not be used, and CX would still be the number of characters.

> **Common mistake**
>
> Putting the string address in DS:DX for 13h and using DX twice. In this service DX is the position; if `mov dx, offset msg` is written, the position is spoiled, and BP points to some random place. Another mistake is to leave CX out and put '$' at the end of the string. 13h does not recognize '$'; it prints as many characters as CX says.

5.2.2.7 Window scrolling and clearing the screen: 06h

`AH = 06h` scrolls a rectangular region of the screen, that is a **window**, up by AL lines. The lines vacated at the bottom of the window are painted with the space character in the BH attribute. If AL = 0 is given, instead of scrolling, the whole window is cleared. This feature is the standard way of clearing the screen without changing mode and in the color you want. `AH = 07h` does the same job downward.

Register	Meaning
AL	number of lines to scroll; 0 = clear the window
BH	attribute of the vacated lines
CH, CL	top-left corner (row, column)
DH, DL	bottom-right corner (row, column)

Table 5.2.11: The registers of the AH = 06h service.

The following program first clears the whole screen to a blue background, then opens a red box in the middle.

```
org 100h
mov ax, 0600h    ; AH=06h, AL=0
mov bh, 1Fh      ; blue/white
mov cx, 0000h    ; (0,0)
mov dx, 184Fh    ; (24,79)
int 10h
mov ax, 0600h
mov bh, 4Fh      ; red/white
mov cx, 0714h    ; (7,20)
mov dx, 103Bh    ; (16,59)
int 10h
ret
```

In the first call the corners are CX = 0000h and DX = 184Fh. Since 18h = 24 and 4Fh = 79, the window is the whole screen. BH = 1Fh gives the cleared cells a blue background and white text color. After the call, every cell of the screen holds the space character and the attribute 1Fh. Characters written afterwards, even with 0Eh, take this color. Selecting mode 03h again also clears the screen but returns the colors to the default 07h.

In the second call AX is made 0600h again. This is necessary because on some BIOSes INT 10h is not guaranteed to preserve AX. CX = 0714h makes the top-left corner (7, 20) and DX = 103Bh makes the bottom-right corner (16, 59). The box is 16 − 7 + 1 = 10 rows and 59 − 20 + 1 = 40 columns. Only this region is cleared to red; the outside stays blue. Neither call moves the cursor; after clearing, the cursor is moved to (0, 0) or into the box with `AH = 02h`.

Step	Instruction	AX	BX	CX	DX	Explanation
1	`mov ax, 0600h`	0600h	0000h	0000h	0000h	AH = 06h (scroll up), AL = 0 (clear)
2	`mov bh, 1Fh`	0600h	1F00h	0000h	0000h	BH = 1Fh: blue background, white text
3	`mov cx, 0000h`	0600h	1F00h	0000h	0000h	top-left corner (0, 0)
4	`mov dx, 184Fh`	0600h	1F00h	0000h	184Fh	bottom-right corner (24, 79)
5	`int 10h`	0600h	1F00h	0000h	184Fh	the whole screen is cleared to a blue background
6	`mov ax, 0600h`	0600h	1F00h	0000h	184Fh	AH, AL again for the second call
7	`mov bh, 4Fh`	0600h	4F00h	0000h	184Fh	BH = 4Fh: red background, white text
8	`mov cx, 0714h`	0600h	4F00h	0714h	184Fh	top-left (7, 20)
9	`mov dx, 103Bh`	0600h	4F00h	0714h	103Bh	bottom-right (16, 59)
10	`int 10h`	0600h	4F00h	0714h	103Bh	a 10-row × 40-column box is cleared to red
11	`ret`	0600h	4F00h	0714h	103Bh	program ends

Table 5.2.12: Trace of the program that clears the screen and a window with AH = 06h.

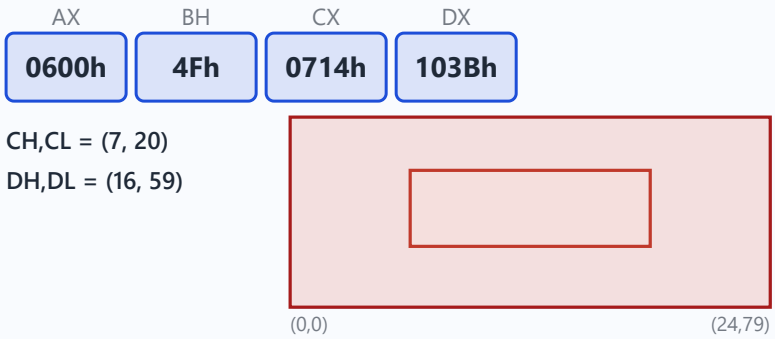


Figure 5.2.7: Diagram of the screen after the second call: the outer rectangle is the 80 × 25 screen cleared with 1Fh (blue background), the inner rectangle is the (7, 20)–(16, 59) window cleared with 4Fh (red background). The colors are not the real screen colors but the diagram's highlight colors.

Example 5.2.3

To clear the lower half of the screen (rows 13–24) to a black background, AX = 0600h and BH = 07h are loaded. The corners are CX = 0D00h (row 13, column 0) and DX = 184Fh (row 24, column 79). To scroll a region up by a single line, AL = 1 is given. With AX = 0601h, CX = 1400h, DX = 184Fh, the window of rows 20–24 scrolls up one line. Row 24 is vacated in the BH color.

5.2.3 Direct access to video memory

5.2.3.1 B800h text memory

INT 10h services always end up doing the same job: writing bytes to the video card's memory. This memory appears in the processor's address space like ordinary RAM. In the 8086's 1 MB address space the range A0000h–BFFFFh is reserved for video cards. In color text modes the video memory starts at physical address B8000h; a program reaches it with segment B800h. The video card's controller reads this memory from beginning to end dozens of times per second and draws each cell on the screen. A value written to memory appears on the screen on the next scan.

Definition 5.2.4

Video memory is the memory, mapped into the processor's address space, that holds the content the video card shows on the screen. In mode 03h it is 4000 bytes (80 × 25 × 2) starting at B800:0000h. Each cell occupies two bytes: at the even address the ASCII code of the character, at the odd address after it the attribute byte.

Writing directly to memory is much faster than calling the BIOS. An `INT 10h` call executes hundreds of instructions, while a direct write is a single `MOV` instruction. There is a price, though. A direct write does not move the cursor and does not update the cursor information the BIOS keeps. The output also does not go through DOS redirection. The program computes the cell address itself and must know which mode the video card is in. emu8086 supports this memory; the cells written appear in the screen window.

Figure 5.2.8: A screen cell is two bytes. When AX = 4E41h is written with `mov es:[di], ax`, the low byte 41h ('A') goes to the even address and the high byte 4Eh (the attribute) to the odd address.

Writing to a cell as a word places the character and its color in one instruction. The 8086 is little-endian: the low byte of a word is written to the lower address and the high byte to the higher address. That is why the character is put in the AL half of AX and the attribute in the AH half. Figure 5.2.8 shows this layout. Because the address of the word is even, the 8086 does this write in a single bus cycle: all 16 bits of the data bus are used, and the BHE and A0 lines are active together. Writing a word to an odd address would take two separate bus cycles.

5.2.3.2 Code trace: writing to B800h

The following program first puts a yellow 'A' on a red background in cell (1, 4). Then, with `REP STOSW`, it fills the first three cells of the screen with the period character. In the form in the slides, the second part uses CX = 2000 to fill the whole screen; here CX = 3 so that it stays traceable.

```
org 100h
mov ax, 0B800h
mov es, ax
mov di, (1*80+4)*2  ; 00A8h
mov ax, 4E41h
mov es:[di], ax
xor di, di
mov ax, 072Eh
mov cx, 3           ; first 3 cells
cld
rep stosw
ret
```

The first two lines make ES B800h. An immediate value cannot be written directly into a segment register; there is no 8086 instruction `MOV ES, 0B800h`. So the value is first put into AX. The 0 at the start of the number is there so that the assembler does not take `B800h` for a name. The expression

`(1*80+4)*2` in the third line is evaluated at assembly time; only the result, 00A8h, goes into the machine code. No multiplication is done while the program runs.

`MOV ES:[DI], AX` names the ES segment explicitly. In a memory access with DI the default segment is DS; here DS points to the program's segment, so the segment prefix (the `26h` byte) is mandatory. When the instruction runs, the physical address is B800h × 16 + 00A8h = B80A8h. 41h is written to this address and 4Eh to address B80A9h. The character appears on the screen at once, and the cursor stays where it was. According to the 8086 manual this instruction takes 9 + EA cycles; for [DI] the EA adds 5 cycles and the segment prefix adds 2: about 16 clock cycles. Writing the same character with INT 10h takes hundreds of times as long.

In the second part `XOR DI, DI` zeroes DI; this is one byte shorter than `MOV DI, 0`. AX = 072Eh means AL = 2Eh ('.') and AH = 07h (light gray). `CLD` clears the direction flag (DF), so the string instruction moves forward, increasing DI. In each iteration `REP STOSW` writes ES:[DI] ← AX, increases DI by 2 and decreases CX by 1; it stops when CX reaches zero. After three iterations DI = 0006h and CX = 0.

Step	Instruction	AX	CX	DI	ES	DF	Explanation
1	`mov ax, 0B800h`	B800h	0000h	0000h	0700h	0	the segment value goes into AX first
2	`mov es, ax`	B800h	0000h	0000h	B800h	0	ES = B800h: the segment of the text screen
3	`mov di, (1*80+4)*2`	B800h	0000h	00A8h	B800h	0	(1 × 80 + 4) × 2 = 168 = 00A8h
4	`mov ax, 4E41h`	4E41h	0000h	00A8h	B800h	0	AL = 41h ('A'), AH = 4Eh (attribute)
5	`mov es:[di], ax`	4E41h	0000h	00A8h	B800h	0	B80A8h ← 41h, B80A9h ← 4Eh; 'A' on the screen at once
6	`xor di, di`	4E41h	0000h	0000h	B800h	0	DI = 0: the top-left cell
7	`mov ax, 072Eh`	072Eh	0000h	0000h	B800h	0	AL = 2Eh ('.'), AH = 07h
8	`mov cx, 3`	072Eh	0003h	0000h	B800h	0	three cells will be filled
9	`cld`	072Eh	0003h	0000h	B800h	0	DF = 0: DI moves forward
10	`rep stosw`	072Eh	0000h	0006h	B800h	0	three times ES:[DI] ← AX, DI += 2
11	`ret`	072Eh	0000h	0006h	B800h	0	program ends

Table 5.2.13: Trace of the program that writes directly to screen memory.

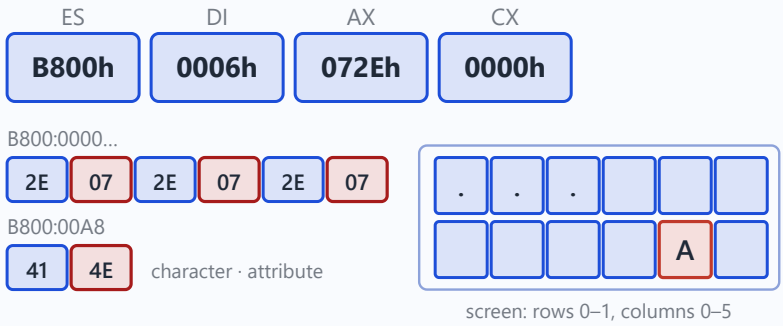


Figure 5.2.9: Video memory when the program ends: three times `2E 07` starting at B800:0000h, and `41 4E` at B800:00A8h. Three periods appear in the first row of the screen and an 'A' in the fifth column (column 4) of the second row.

To fill the whole screen, CX = 2000 is given: 80 × 25 cells. On the 8086 `REP STOSW` takes about 9 + 10 × CX clock cycles. For 2000 cells this is 20,009 cycles; on a 5 MHz processor about 4 ms. If the same job were done with 2000 separate `INT 10h/0Eh` calls, it would take dozens of times longer because of scrolling and cursor updates.

Common mistake

Putting the character and the attribute the wrong way round: if `mov ax, 414Eh` is written, the low byte 4Eh ('N') is the character and 41h is the color, blue text on a red background. In little-endian order the character is in AL and the attribute in AH. A second common mistake is writing `MOV [DI], AX` without setting ES. Without a segment prefix the write goes to the DS segment, that is, over the program's own data.

5.2.3.3 Mode 13h and the pixel address

Mode 13h is VGA's graphics mode with 320 × 200 resolution and 256 colors. It is selected with `mov ax, 0013h` and `int 10h`. The greatest convenience of this mode for the programmer is the structure of the memory: each pixel is a single byte, and the pixels follow one another row by row. The video memory is 64,000 bytes (320 × 200) from A000:0000h to A000:F9FFh. The value of the byte is the color number of the pixel; the video card converts this number to a real color in a **palette** table. The first 16 colors of the default palette are the same as the text colors: 0Eh is yellow, 0Fh is white.

Each row is 320 = 140h bytes. The offset of pixel (x, y) is therefore computed as follows:

$$\text{offset} = y \times 320 + x$$

For pixel (100, 50), 50 × 320 + 100 = 16,100 = 3EE4h. The last pixel (319, 199) is at 199 × 320 + 319 = 63,999 = F9FFh, the last byte of the memory. Figure 5.2.10 shows the offsets of the first rows and of the last row.

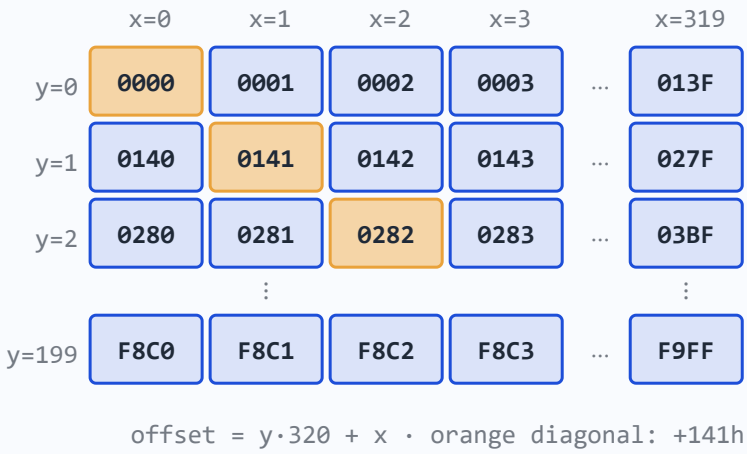


Figure 5.2.10: Pixel offsets in mode 13h. Each row is 140h bytes; the same column one row down is +140h, and one row down and one to the right on the diagonal is +141h further.

The BIOS offers a pixel service in this mode too. `AH = 0Ch` writes the color in AL to the pixel at column CX (x, 0–319) and row DX (y, 0–199); BH is the page and is 0 in mode 13h. `AH = 0Dh` reads the pixel's color into AL with the same CX, DX. Working with coordinates is easy, but one interrupt call per pixel is slow.

Example 5.2.4

The offset of pixel (160, 100) in the middle of the screen is 100 × 320 + 160 = 32,160 = 7DA0h; its physical address is A0000h + 7DA0h = A7DA0h. The top-right corner (319, 0) is at 013Fh, and the bottom-left corner (0, 199) is at 199 × 320 = 63,680 = F8C0h.

5.2.3.4 Drawing a diagonal

A diagonal starting at the top-left corner is made of the pixels (0, 0), (1, 1), (2, 2) … First, the version that uses coordinates with the BIOS:

```
mov ax, 0013h
int 10h          ; 320×200, 256 colors
xor cx, cx       ; x = 0
xor dx, dx       ; y = 0
mov bh, 0        ; page
dot: mov ax, 0C0Eh  ; AH=0Ch, AL=yellow
int 10h
inc cx           ; one right
inc dx           ; one down
cmp dx, 200
jb dot           ; 200 pixels
```

The loop reloads AX on every iteration. The reason is that some BIOSes change AX in the 0Ch service; the safe way is to set AH and AL before every call. CX and DX increase together. When DX reaches 200 the `JB` does not jump and the loop ends having drawn 200 pixels. Since the screen is 320 pixels wide, the diagonal reaches the bottom edge, not the right edge.

The same line can also be drawn by writing directly to A000h memory. This time we work with offsets instead of coordinates:

```
mov ax, 0013h
int 10h
mov ax, 0A000h
mov es, ax
xor di, di       ; (0,0) → offset 0
mov al, 0Eh      ; yellow
mov cx, 200
draw: mov es:[di], al
add di, 141h     ; 320 + 1
loop draw
mov ah, 0
int 16h          ; wait for a key
mov ax, 0003h
int 10h          ; back to text mode
```

Why is 141h added? +140h (320) goes to the same column of the next row down, +1 to the next column right. Together they make 141h = 321 and give the next pixel on the diagonal. The last pixel (199, 199) is at 199 × 321 = 63,879 = F987h. For the opposite diagonal descending from the top-right corner, 13Fh (320 − 1) is added at each step. At the end the program waits for a key and then returns to text mode; this way the user can see the line and the DOS prompt does not stay on a graphics screen.

The speed difference between the two versions is large. In the direct version the loop body is three instructions: a memory write, an addition and LOOP. In the BIOS version an interrupt is called for every pixel, and the interrupt routine itself computes the address from the coordinates. emu8086's graphics support is limited: drawing with the 0Ch service is slow, and a direct write to A000h may not be visible in every version. In a PC emulator such as DOSBox both versions are fully visible.

To keep the trace short, the following program draws only four pixels and does not change the mode. It shows clearly that DI increases by 141h in each iteration of the loop.

```
org 100h
mov ax, 0A000h
mov es, ax
xor di, di       ; (0,0) → offset 0
mov al, 0Eh      ; yellow
mov cx, 4        ; first 4 pixels
draw: mov es:[di], al
add di, 141h     ; 320 + 1
loop draw
ret
```

Step	Instruction	AL	CX	DI	ES	Explanation
1	`mov ax, 0A000h`	00h	0000h	0000h	0700h	the segment value goes through AX
2	`mov es, ax`	00h	0000h	0000h	A000h	ES = A000h: the mode 13h video memory
3	`xor di, di`	00h	0000h	0000h	A000h	DI = 0: pixel (0, 0)
4	`mov al, 0Eh`	0Eh	0000h	0000h	A000h	color 0Eh (yellow)
5	`mov cx, 4`	0Eh	0004h	0000h	A000h	four pixels
6	`mov es:[di], al`	0Eh	0004h	0000h	A000h	A0000h ← 0Eh: pixel (0, 0)
7	`add di, 141h`	0Eh	0004h	0141h	A000h	one row down, one column right
8	`loop draw`	0Eh	0003h	0141h	A000h	CX = 3 ≠ 0: back to `draw`
9	`mov es:[di], al`	0Eh	0003h	0141h	A000h	A0141h ← 0Eh: pixel (1, 1)
12	`mov es:[di], al`	0Eh	0002h	0282h	A000h	A0282h ← 0Eh: pixel (2, 2)
15	`mov es:[di], al`	0Eh	0001h	03C3h	A000h	A03C3h ← 0Eh: pixel (3, 3)
16	`add di, 141h`	0Eh	0001h	0504h	A000h	DI = 0504h = the place of (4, 4)
17	`loop draw`	0Eh	0000h	0504h	A000h	CX = 0: the loop ends
18	`ret`	0Eh	0000h	0504h	A000h	program ends

Table 5.2.14: Trace of the program that draws a four-pixel diagonal in A000h memory.

The DI values in the table, 0000h, 0141h, 0282h, 03C3h and 0504h, are multiples of 321: 0, 321, 642, 963 and 1284. When each is written as y × 320 + x, it turns out that x = y; for example 963 = 3 × 320 + 3. When the loop ends, DI points to the place of the fifth pixel, (4, 4), which has not been drawn. The physical addresses are A0000h, A0141h, A0282h and A03C3h.

Figure 5.2.11: The top-left 6 × 6 pixel region of the screen when the loop ends: (0, 0)–(3, 3) yellow, DI =

0504h (1284), that is y = 4, x = 4.

The pixel address explorer in the slides computes the (x, y) coordinates, offset and physical address of a point clicked on the screen. The diagonal and reverse-diagonal buttons animate the loop that adds 141h or 13Fh to DI.

5.2.4 INT 16h: keyboard services

5.2.4.1 Scan code and ASCII

When a key is pressed, the controller inside the keyboard sends a number that indicates the key's place on the keyboard. On the PC this number is read from I/O port 60h and, through the interrupt controller, gives rise to the IRQ 1 hardware interrupt. The BIOS keyboard interrupt routine (INT 09h) takes this number and finds its ASCII equivalent by looking at the state of Shift and Caps Lock. It then writes the two together as a word into the **keyboard buffer** in memory. The program reads keys from this buffer with `INT 16h`. This way keys pressed while the program is busy are not lost; they wait in line in the buffer.

Definition 5.2.5

The **scan code** is the code that indicates the physical position of the key pressed on the keyboard. The same key gives the same scan code whether or not Shift is held: 1Eh for 'a' and 'A'. The **ASCII code**, on the other hand, indicates the key's current meaning, that is, the character produced: 61h for 'a', 41h for 'A'.

AH	Service	Output
00h	wait for a key and read it	AH = scan code, AL = ASCII; the key is removed from the buffer
01h	is a key available? (does not wait)	ZF = 1: no key · ZF = 0: AX = key; the key is not removed from the buffer
02h	Shift states	AL bits: 0 right Shift, 1 left Shift, 2 Ctrl, 3 Alt, 4 Scroll Lock, 5 Num Lock, 6 Caps Lock, 7 Insert

Table 5.2.15: INT 16h keyboard services.

`AH = 00h` waits until a key is pressed if there is no key in the buffer. When a key arrives, it puts the scan code in AH and the ASCII code in AL and removes the key from the buffer. The key is not echoed on the screen; if something is to be seen on the screen, the program writes it itself. Table 5.2.16 lists the values returned by some keys.

Key	AH (scan)	AL (ASCII)	Key	AH (scan)	AL
a / A	1Eh	61h / 41h	↑ / ↓	48h / 50h	00h
1	02h	31h	← / →	4Bh / 4Dh	00h
Enter	1Ch	0Dh	F1 … F10	3Bh … 44h	00h
Esc	01h	1Bh	Backspace	0Eh	08h
Space	39h	20h	Ctrl+C	2Eh	03h

Table 5.2.16: INT 16h/00h return values of some keys (IBM PC/AT, US layout).

Keys such as the arrow keys, function keys, Home and End have no ASCII equivalent. These are called **extended keys**, and INT 16h returns AL = 00h for them. The program looks at AL first; if AL is zero, it recognizes the key from the scan code in AH. The F1 key, for example, gives AH = 3Bh, AL = 00h. Letters pressed with Ctrl fall into the first 32 codes of ASCII: Ctrl+A = 01h, Ctrl+C = 03h. The code is the low five bits of the letter's code.

The job of the `MOV AH, 0` · `INT 16h` pair often seen at the end of programs is simple. The program waits for a key, so that the output screen can be read before it closes. The keyboard explorer in the slides shows which AH and AL values INT 16h/00h would return for a key that is pressed.

5.2.4.2 Code trace: read a key, print it in uppercase

The following program reads keys until Enter is pressed. It converts lowercase letters to uppercase and writes them, and writes other characters as they are. The program uses together the fact that INT 16h does not echo and the bit 5 rule of the ASCII layout.

```
org 100h
read:
mov ah, 0        ; wait for key
int 16h          ; AH=scan, AL=ASCII
cmp al, 0Dh      ; Enter?
je done
cmp al, 'a'
jb echo
cmp al, 'z'
ja echo
sub al, 20h      ; lower → upper
echo:
mov ah, 0Eh
int 10h          ; print it
jmp read
done:
ret
```

The loop starts at the `read` label. `INT 16h` waits for a key. `CMP AL, 0Dh` checks for Enter; if equal, `JE` ends the program. Then two comparisons test whether AL is in the range 'a'–'z'. `JB` and `JA` are unsigned comparison jumps: JB jumps when CF = 1; JA jumps when CF = 0 and ZF = 0. If AL is below 'a' or above 'z', it is not a letter, so control goes straight to `echo`. If it is inside the range, `SUB AL, 20h` clears bit 5. The `echo` part writes the character with teletype and the loop returns to the start.

The user presses Shift+o, k and Enter in turn. Table 5.2.17 gives the important steps; similar steps in the same iteration are skipped.

Step	Instruction	AX	CF	ZF	Screen (output)	Explanation
1	`mov ah, 0`	0000h	0	0	–	AH = 00h: wait for a key and read it
2	`int 16h`	184Fh	0	0	–	Shift+o: AH = 18h (scan), AL = 4Fh ('O')
3	`cmp al, 0Dh`	184Fh	0	0	–	4Fh − 0Dh: the result is not zero
4	`je done`	184Fh	0	0	–	not Enter, no jump
5	`cmp al, 'a'`	184Fh	1	0	–	4Fh < 61h: borrow, CF = 1
6	`jb echo`	184Fh	1	0	–	CF = 1: jump to `echo`
7	`mov ah, 0Eh`	0E4Fh	1	0	–	AH = 0Eh; 'O' is still in AL
8	`int 10h`	0E4Fh	1	0	`O`	O on the screen
9	`jmp read`	0E4Fh	1	0	`O`	back to the start of the loop
11	`int 16h`	256Bh	1	0	`O`	k: AH = 25h, AL = 6Bh
14	`cmp al, 'a'`	256Bh	0	0	`O`	6Bh ≥ 61h: CF = 0
15	`jb echo`	256Bh	0	0	`O`	no jump: may be a lowercase letter
16	`cmp al, 'z'`	256Bh	1	0	`O`	6Bh − 7Ah: borrow, CF = 1, ZF = 0
17	`ja echo`	256Bh	1	0	`O`	JA needs CF = 0 and ZF = 0; it does not jump
18	`sub al, 20h`	254Bh	0	0	`O`	6Bh − 20h = 4Bh = 'K'
20	`int 10h`	0E4Bh	0	0	`OK`	K on the screen
23	`int 16h`	1C0Dh	0	0	`OK`	Enter: AH = 1Ch, AL = 0Dh
24	`cmp al, 0Dh`	1C0Dh	0	1	`OK`	0Dh − 0Dh = 0: ZF = 1
25	`je done`	1C0Dh	0	1	`OK`	jump to `done`
26	`ret`	1C0Dh	0	1	`OK`	program ends

Table 5.2.17: Trace of the program that reads keys and prints them in uppercase (keys pressed: Shift+o, k, Enter).

On the first key AL = 4Fh ('O') arrives. `CMP AL, 'a'` computes 4Fh − 61h; the result is negative, a borrow is needed and CF = 1. `JB` jumps: the character is not a lowercase letter and is written as it is. On the second key AL = 6Bh ('k') arrives. 6Bh − 61h needs no borrow, CF = 0 and `JB` does not jump. 6Bh − 7Ah, however, needs a borrow: CF = 1, ZF = 0. For JA to jump both CF and ZF would have to be zero; it does not jump. So it is settled that AL lies in the range 61h ≤ AL ≤ 7Ah. `SUB AL, 20h` turns 6Bh into 4Bh, that is 'K'. The 0Dh that arrives with Enter sets ZF = 1 in the first comparison and the program ends.

One more detail can be seen in the AX column of the table. Before the teletype call AH = 0Eh is set; at the start of the loop `MOV AH, 0` clears AH again, but AL keeps the last character written. When INT 16h returns, both AH and AL change with the new key. In the end `OK` appears on the screen. Because

Enter itself is not echoed, the cursor stays on the line.

5.2.4.3 Polling without waiting

`AH = 00h` stops the program until a key arrives. A game or an animation, however, must keep working while it waits for a key. For this `AH = 01h` is used: the service returns at once and reports with ZF whether there is a key. If ZF = 1 there is no key in the buffer. If ZF = 0 there is a key and it is in AX, but it is not removed from the buffer. To actually take the key, the program then calls `AH = 00h`. Asking about the state of a device again and again in a loop is called **polling**.

```
org 100h
xor dx, dx       ; work counter
poll:
inc dx           ; background work
mov ah, 01h      ; key ready?
int 16h          ; ZF=1 → none
jz poll
mov ah, 00h      ; take the key
int 16h
ret
```

DX stands for the work done while waiting; it increases by one on each iteration. In a real program an animation frame would be drawn here or a calculation advanced. In the trace there is no key at the first two polls, and at the third the user has pressed the x key.

Step	Instruction	AX	DX	ZF	Explanation
1	`xor dx, dx`	0000h	0000h	1	the work counter is zeroed
2	`inc dx`	0000h	0001h	0	background work: 1
3	`mov ah, 01h`	0100h	0001h	0	AH = 01h: is a key available?
4	`int 16h`	0100h	0001h	1	buffer empty: ZF = 1
5	`jz poll`	0100h	0001h	1	ZF = 1: back to `poll`
6	`inc dx`	0100h	0002h	0	background work: 2
8	`int 16h`	0100h	0002h	1	still no key
10	`inc dx`	0100h	0003h	0	background work: 3
12	`int 16h`	2D78h	0003h	0	x is there: ZF = 0, AX = 2D78h; the key stays in the buffer
13	`jz poll`	2D78h	0003h	0	ZF = 0: leave the loop
14	`mov ah, 00h`	0078h	0003h	0	AH = 00h: take the key
15	`int 16h`	2D78h	0003h	0	the same key is read and removed from the buffer
16	`ret`	2D78h	0003h	0	program ends

Table 5.2.18: Trace of the loop that polls for a key with AH = 01h (x is pressed at the third poll).

In the first two iterations INT 16h returns ZF = 1 and `JZ` continues the loop. The program does not freeze: DX increases on each iteration. In the third iteration a key has arrived. INT 16h makes ZF = 0 and puts 2D78h in AX: scan code 2Dh, ASCII 78h ('x'). The key stays in the buffer. `JZ` does not jump and the loop is left. `AH = 00h` reads the same key and this time removes it from the buffer; AX is again 2D78h. At the end of the program DX = 3: the background work was done three times while waiting for the key.

> **Common mistake**
>
> Seeing the key with 01h and not taking it with 00h. Because 01h does not remove the key from the buffer, the same key still looks "available" on the next iteration of the loop; the program reacts to a single key endlessly. After responding to the key, the buffer is emptied with `AH = 00h`.

5.2.5 INT 21h: DOS character and string services

5.2.5.1 Frequently used services

`INT 21h` is DOS's main service interrupt. Of its hundreds of services, character and string I/O, date, time and program termination are the most used. Table 5.2.19 summarizes them.

AH	Service	Input	Output
01h	read a character and echo it	–	AL = character
02h	write a character	DL = character	AL = DL
07h / 08h	read a character, without echo	–	AL (08h checks for Ctrl+C)
09h	write a string	DS:DX → string ending with '$'	AL = 24h
0Ah	read a buffered line	DS:DX → buffer	[buffer+1] = count, buffer+2 = text
2Ah	get the date	–	CX = year, DH = month, DL = day, AL = day of the week (0 = Sunday)
2Ch	get the time	–	CH = hour, CL = minute, DH = second, DL = hundredths
2Bh / 2Dh	set the date / time	same registers as 2Ah / 2Ch	AL = 00h success
4Ch	terminate the program	AL = exit code	(does not return)

Table 5.2.19: Frequently used INT 21h services.

An important difference between DOS services and BIOS services is that they obey **I/O redirection**. If the program is run as `program > output.txt`, everything written with 02h and 09h goes to the file instead of the screen. The BIOS's INT 10h output always goes to the screen; DOS cannot see it. Similarly, 01h and 0Ah can take their input from a file.

5.2.5.2 INT 10h/0Eh or INT 21h/02h?

The two services that write a single character to the screen look very similar. Both write the character where the cursor is and advance the cursor; both interpret control characters such as CR and LF. Their differences come from the layer they work at. Table 5.2.20 compares them.

Feature	BIOS · INT 10h, AH = 0Eh	DOS · INT 21h, AH = 02h
Character	in AL	in DL
Software needed	only the video BIOS; also works in boot code	DOS must be loaded
Redirection	no; always to the screen	yes; to a file, to a printer
Color	in graphics mode with BL	none
AL on return	preserved	AL = DL
Ctrl+Break check	none	yes

Table 5.2.20: Comparison of INT 10h/0Eh and INT 21h/02h.

The choice depends on the environment the program will run in. A boot loader runs before DOS, so it can use only the BIOS. A tool whose output is to be saved to a file must choose the DOS service. The two can be mixed in the same program; both use the same cursor position kept by the BIOS.

5.2.5.3 Code trace: read and write a character

The following program reads a character from the keyboard, moves to a new line, and writes the letter after the character that was read. Although short, it shows clearly how AL gets clobbered between services.

```
org 100h
mov ah, 01h      ; read + echo
int 21h          ; AL = char
mov bl, al       ; save it
mov ah, 02h      ; write char
mov dl, 0Dh      ; CR: line start
int 21h
mov dl, 0Ah      ; LF: next line
int 21h
inc bl           ; next letter
mov dl, bl
int 21h
ret
```

Step	Instruction	AX	BX	DX	Screen (output)	Explanation
1	`mov ah, 01h`	0100h	0000h	0000h	–	AH = 01h: read and echo
2	`int 21h`	016Bh	0000h	0000h	`k`	k: AL = 6Bh; k on the screen
3	`mov bl, al`	016Bh	006Bh	0000h	`k`	the character read is saved in BL
4	`mov ah, 02h`	026Bh	006Bh	0000h	`k`	AH = 02h: write DL
5	`mov dl, 0Dh`	026Bh	006Bh	000Dh	`k`	DL = 0Dh (CR)
6	`int 21h`	020Dh	006Bh	000Dh	`k⏎`	cursor to the start of the line; AL = 0Dh
7	`mov dl, 0Ah`	020Dh	006Bh	000Ah	`k⏎`	DL = 0Ah (LF)
8	`int 21h`	020Ah	006Bh	000Ah	`k↲`	cursor to the next line; AL = 0Ah
9	`inc bl`	020Ah	006Ch	000Ah	`k↲`	BL = 6Ch ('l')
10	`mov dl, bl`	020Ah	006Ch	006Ch	`k↲`	DL = 'l'
11	`int 21h`	026Ch	006Ch	006Ch	`k↲l`	l on the screen; AL = 6Ch
12	`ret`	026Ch	006Ch	006Ch	`k↲l`	program ends

Table 5.2.21: Trace of the program that reads a character and writes the next one on a new line (key pressed: k). ⏎ means CR, ↲ means CR LF.

`AH = 01h` waits for a key and echoes it on the screen; when the user presses k, k appears on the screen and AL = 6Bh. The value read is copied to BL at once, because the following `INT 21h` calls will change AL. Indeed, after CR is written AL = 0Dh, and after LF is written AL = 0Ah. The 02h service puts DL into AL on return. AH stays 02h, so it is enough to load it once for the three calls. CR takes the cursor to the start of the line and LF to the next line. `INC BL` turns 6Bh into 6Ch, 'k' into 'l', and the last call writes this letter on the new line. The final state of the screen is two lines: the echoed k above, l below.

5.2.5.4 Code trace: printing a string

`AH = 09h` writes the string pointed to by DS:DX until it sees the '$' (24h) character. '$' is the terminator; it is not written itself. The 0Dh and 0Ah inside the string end the line, that is, control characters are interpreted here too.

Definition 5.2.6

A **'$'-terminated string** is the format expected by DOS's 09h service: the characters sit one after another in memory and the string ends at the first 24h byte. The length is not given separately. The service sets AL to 24h on return.

```
org 100h
mov dx, offset msg1  ; DS:DX → string
mov ah, 09h
int 21h
mov dx, offset msg2
int 21h              ; AH still 09h
ret
msg1 db 'Welcome, 8086!', 0Dh, 0Ah, '$'
msg2 db 'Good-bye!$'
```

The code part is 13 bytes (0100h–010Ch). So `msg1` starts at 010Dh. `msg1` is 14 characters plus CR, LF and '$', 17 bytes; `msg2` is at 010Dh + 17 = 011Eh. The `OFFSET` operator gives the offset of the variable, not its contents.

Step	Instruction	AX	DX	Screen (output)	Explanation
1	`mov dx, offset msg1`	0000h	010Dh	–	DS:DX → `msg1` (010Dh)
2	`mov ah, 09h`	0900h	010Dh	–	AH = 09h
3	`int 21h`	0924h	010Dh	`Welcome, 8086!↵`	written up to the '$'; AL = 24h
4	`mov dx, offset msg2`	0924h	011Eh	`Welcome, 8086!↵`	DX → `msg2` (011Eh)
5	`int 21h`	0924h	011Eh	`Welcome, 8086!↵Good-bye!`	AH is still 09h: the second string
6	`ret`	0924h	011Eh	`Welcome, 8086!↵Good-bye!`	program ends

Table 5.2.22: Trace of the program that writes two strings with AH = 09h.

The first call writes `Welcome, 8086!` and moves to a new line with CR LF. AL = 24h. Before the second call only DX is changed; AH is still 09h. Even so, loading AH before every call is a good habit against the risk that a line added to the program later spoils AH. The output is two lines. The '$' character itself cannot be printed with this service; to print it, the 02h service with DL = 24h is used.

Common mistake

Forgetting to put '$' at the end of the string. 09h writes whatever it finds in memory until it finds the first 24h byte: the program's code, other variables, random bytes. The screen fills with garbage, and the bell may even ring because of control characters. The reverse also happens: the string `'Price: 5$ TL$'` is written only as `Price: 5` , because the first '$' ends the string.

5.2.5.5 Buffered line input

Getting a name or a number from the user takes more than reading keys one by one. When the user types something wrong they erase it with Backspace, and when they press Enter the line is finished. `AH = 0Ah` has DOS do this editing. The program only prepares a buffer and puts its address in DS:DX. DOS reads keys until Enter is pressed, echoes them, and allows correction with Backspace and Esc. The program does not run in the meantime; the call returns when the line is finished.

Definition 5.2.7

The **0Ah buffer** has three parts. In byte 0 the program writes the capacity: the maximum number of characters to be accepted, Enter included. In byte 1 DOS writes the number of characters read (Enter is not counted). The text begins at byte 2 and the code of Enter, 0Dh, is appended to its end. The capacity cannot be 0.

Figure 5.2.12 shows the state after Ali is typed and Enter is pressed in a buffer defined as `buffer db 10, ?, 10 dup(?)`. Byte 0 = 0Ah (10 characters), byte 1 = 03h, then 'A', 'l', 'i' and 0Dh. The text does not end with '$'; to print it with 09h, the program puts '$' in place of 0Dh. When the buffer is full, DOS does not accept further keys and the bell rings; only Enter and Backspace keep working.

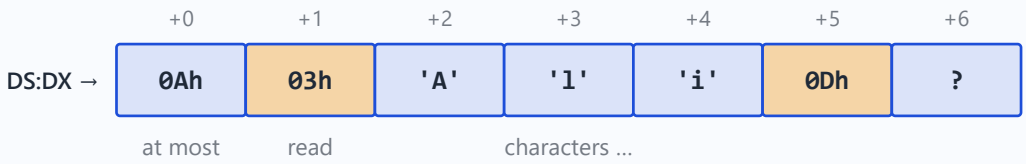


Figure 5.2.12: The 0Ah buffer after Enter: +0 capacity (0Ah), +1 number read (03h), the text from +2 and 0Dh at the end.

The following program reads the user's name and greets them.

```
org 100h
mov dx, offset buffer
mov ah, 0Ah          ; buffered input
int 21h
mov bl, [buffer+1]   ; count read
mov bh, 0
mov byte ptr buffer[bx+2], '$'
mov dx, offset greet
mov ah, 09h
int 21h
lea dx, [buffer+2]
int 21h
ret
buffer db 10, ?, 10 dup(?)
greet  db 0Dh, 0Ah, 'Hello $'
```

Step	Instruction	AX	BX	DX	Screen (output)	Explanation
1	`mov dx, offset buffer`	0000h	0000h	011Fh	–	DS:DX → buffer (011Fh)
2	`mov ah, 0Ah`	0A00h	0000h	011Fh	–	AH = 0Ah: read a line
3	`int 21h`	0A00h	0000h	011Fh	`Ali⏎`	buffer = 0Ah, 03h, 'A', 'l', 'i', 0Dh
4	`mov bl, [buffer+1]`	0A00h	0003h	011Fh	`Ali⏎`	BL = [0120h] = 3

Step	Instruction	AX	BX	DX	Screen (output)	Explanation
5	`mov bh, 0`	0A00h	0003h	011Fh	`Ali⏎`	BX = 0003h
6	`mov byte ptr buffer[bx+2], '$'`	0A00h	0003h	011Fh	`Ali⏎`	[011Fh + 3 + 2] = [0124h] ← '$'
7	`mov dx, offset greet`	0A00h	0003h	012Bh	`Ali⏎`	DX → `greet` (012Bh)
8	`mov ah, 09h`	0900h	0003h	012Bh	`Ali⏎`	AH = 09h
9	`int 21h`	0924h	0003h	012Bh	`Ali⏎↵Hello`	CR LF and `Hello` are written
10	`lea dx, [buffer+2]`	0924h	0003h	0121h	`Ali⏎↵Hello`	DX = 0121h: the start of the text
11	`int 21h`	0924h	0003h	0121h	`Ali⏎↵Hello Ali`	`Ali` is written
12	`ret`	0924h	0003h	0121h	`Ali⏎↵Hello Ali`	program ends

Table 5.2.23: Trace of the program that reads a name and greets (typed: Ali, Enter).

The program's code is 31 bytes (0100h–011Eh). emu8086 assembles the directly addressed `LEA DX, [buffer+2]` instruction to the same three-byte code as `MOV DX, OFFSET buffer+2` (`BA 21 01`); `buffer` therefore starts at 011Fh and `greet` at 012Bh. When the call returns, the buffer holds the bytes `0A 03 41 6C 69 0D` from 011Fh. `MOV BL, [buffer+1]` takes the number read, 3. BH is zeroed because the number will be used as BX in the address calculation. The address `buffer[bx+2]` is 011Fh + 3 + 2 = 0124h; exactly the byte where 0Dh sits. '$' is written there and the text can now be printed with 09h. The `BYTE PTR` specifier is here for clarity. Because `buffer` is defined with DB, the assembler already knows the write is a byte; in an address that has no variable name, such as `[BX+2]`, the specifier is mandatory.

Then `greet` is written: because it starts with CR LF, the output moves to a new line and `Hello` is printed. Since Enter itself echoed only CR, the cursor was at the start of the line; the LF takes it down to the next line. `LEA DX, [buffer+2]` loads the start of the text, 0121h, into DX. Since AH is still 09h, the last call writes `Ali`. `Hello Ali` appears on the screen.

Example 5.2.5

In a buffer defined as `buffer db 6, ?, 6 dup(?)`, the user types Micro12 and presses Enter. Because the capacity is 6 including Enter, at most 5 characters are accepted. 'M', 'i', 'c', 'r', 'o' are accepted; the bell rings at the keys '1' and '2'. When Enter arrives, the buffer becomes `06 05 4D 69 63 72 6F 0D`. The number read is 5; '$' is written at address buffer + 5 + 2 = buffer + 7 and only `Micro` is printed on the screen.

Common mistake

Not setting the first byte of the buffer, or reading the text from buffer+0. If the capacity is left 0, the service behaves unpredictably. The text always starts at buffer+2; buffer+0 and buffer+1 are numbers, and if they are written to the screen meaningless characters appear.

5.2.6 Date and time

5.2.6.1 AH = 2Ah and AH = 2Ch

DOS provides the system's date and time through two services. `AH = 2Ah` returns the date and `AH = 2Ch` the time. Table 5.2.24 shows the return registers. All the values are **binary** numbers. If the hour is 14, CH = 0Eh; if the minute is 35, CL = 23h. The year fits in the 16-bit CX: 2026 = 07EAh.

Register	AH = 2Ah (date)	AH = 2Ch (time)
CX / CH	CX = year (e.g. 07EAh = 2026)	CH = hour (0–23)
CL	(part of CX)	CL = minute (0–59)
DH	month (1–12)	second (0–59)
DL	day (1–31)	hundredths (0–99)
AL	day of the week (0 = Sunday … 6 = Saturday)	–

Table 5.2.24: Return values of INT 21h/2Ah and 2Ch.

To write binary values on the screen, each number is split into its decimal digits. For a number between 0 and 99 the shortest way is the `AAM` instruction. AAM divides AL by 10; it writes the quotient to AH and the remainder to AL. For 14, AH = 1 and AL = 4. Then 30h is added to the two digits and ASCII digits are obtained. For numbers greater than 99, such as the year, repeated division by 10 is used. The BIOS `INT 1Ah/02h` service, on the other hand, gives the time not in binary but as BCD, and AAM is not used there.

Example 5.2.6

`AAM` runs with AL = 23h (35): 35 ÷ 10 = 3, remainder 5. AH = 03h, AL = 05h, that is AX = 0305h. `ADD AX, 3030h` converts both bytes to ASCII at once: AX = 3335h, AH = '3', AL = '5'. When AH and then AL are written, 35 appears on the screen. For AL = 07h, AX = 0007h → 3037h, and 07 appears on the screen: AAM adds the leading zero to single-digit values by itself.

5.2.6.2 Code trace: printing the day name

The day of the week arrives in AL as a number from 0 to 6. The natural way to turn it into a name is a **lookup table**: the names of the days are laid out one after another in memory with equal lengths, and AL is converted to a position in the table. In the program below each name is 4 bytes including '$'; so the position is AL × 4.

```
org 100h
mov ah, 2Ah
int 21h          ; AL = 0…6
mov ah, 0
mov bl, 4
mul bl           ; AX = AL × 4
mov dx, offset days
add dx, ax
mov ah, 09h
int 21h
ret
days db 'Sun$Mon$Tue$Wed$'
     db 'Thu$Fri$Sat$'
```

The date comes from the system clock; the trace is for Wednesday, January 7, 2026. On return from `INT 21h`, CX = 07EAh (2026), DH = 01h (January), DL = 07h and AL = 3 (Wednesday). DX is 0107h at this moment. `MOV AH, 0` makes AX 0003h. `MUL BL` multiplies AL by BL and writes the 16-bit result to AX: 3 × 4 = 12 = 000Ch. This is the distance of the name `Wed$` from the start of the table.

`MOV DX, OFFSET days` erases the date in DX and loads the offset of the table, 0114h. If the date were needed later, it would have to be saved before this line. `ADD DX, AX` gives 0114h + 000Ch = 0120h: the first byte of `Wed$`. With `AH = 09h`, `Wed` is printed on the screen.

Step	Instruction	AX	BX	CX	DX	Screen (output)	Explanation
1	`mov ah, 2Ah`	2A00h	0000h	0000h	0000h	–	AH = 2Ah: get the date
2	`int 21h`	2A03h	0000h	07EAh	0107h	–	CX = 07EAh (2026), DH = 01h (month), DL = 07h (day), AL = 3 (Wednesday)
3	`mov ah, 0`	0003h	0000h	07EAh	0107h	–	AX = 0003h
4	`mov bl, 4`	0003h	0004h	07EAh	0107h	–	each name is 4 bytes
5	`mul bl`	000Ch	0004h	07EAh	0107h	–	AX = 3 × 4 = 12 = 000Ch
6	`mov dx, offset days`	000Ch	0004h	07EAh	0114h	–	DX = `days` = 0114h; the date is erased
7	`add dx, ax`	000Ch	0004h	07EAh	0120h	–	DX = 0120h: `Wed$`
8	`mov ah, 09h`	090Ch	0004h	07EAh	0120h	–	AH = 09h
9	`int 21h`	0924h	0004h	07EAh	0120h	`Wed`	Wed on the screen
10	`ret`	0924h	0004h	07EAh	0120h	`Wed`	program ends

Table 5.2.25: Trace of the program that prints the day name from a table (date Wednesday, January 7, 2026).

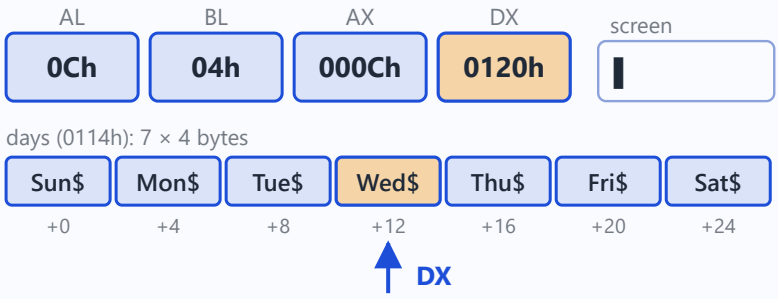


Figure 5.2.13: After `ADD DX, AX`: DX = 0120h points to byte +12 of the `days` table, that is, to `Wed$`.

The code part is 20 bytes, so `days` starts at 0114h. The lengths of the instructions: `MOV AH, imm` 2, `INT` 2, `MOV BL, imm` 2, `MUL BL` 2, `MOV DX, imm` 3, `ADD DX, AX` 2, `RET` 1 byte. Two left shifts (`SHL AX, 1` twice) could have been used instead of the multiplication, because 4 is a power of two. On the 8086 MUL takes 70–77 clock cycles, while two SHLs take only 4.

Common mistake

Skipping the `MOV AH, 0` line. In this program it does no harm, because `MUL BL` rewrites all of AX. But if the multiplication were done with two `SHL AX, 1`, the 2Ah left in AH would shift too: AX = 2A03h → A80Ch. DX = 0114h + A80Ch = A920h and 09h would print some unrelated place in memory. When computing a table position, the high half of AX must always be zeroed.

5.2.6.3 Code trace: printing the time

To write the time as HH:MM, two numbers are printed in the same way. The repeated part is put into a procedure: `print2` writes the number from 0 to 99 in AL as two digits.

```
org 100h
mov ah, 2Ch      ; get system time
int 21h          ; CH=hour CL=min
mov al, ch
call print2
mov dl, ':'
mov ah, 02h
int 21h
mov al, cl
call print2
ret
print2:          ; AL (0-99) → 2 digits
aam              ; AH=tens AL=ones
add ax, 3030h    ; ASCII
mov bx, ax
mov dl, bh
mov ah, 02h
int 21h
mov dl, bl
int 21h
ret
```

The trace is for the time 14:35:07.42. On return from `INT 21h`, CX = 0E23h (CH = 14, CL = 35) and DX = 072Ah (DH = 7 seconds, DL = 42 hundredths). `CALL print2` pushes the return address, the address of the instruction after `call`, 0109h, on the stack; SP drops from FFFEh to FFFCh. Inside the procedure `AAM` splits 14 into 1 and 4, and `ADD AX, 3030h` converts to ASCII.

The `MOV BX, AX` line is the key point of the procedure. The two digits are in AH and AL, but the first `INT 21h/02h` call will put DL into AL and erase the ones digit. So both are copied to BX and taken in turn from BH and BL into DL. The minute in CL, on the other hand, is not spoiled by any call: the procedure does not touch CX, and the 02h service does not change CX. The minute is safely taken with `MOV AL, CL`.

Step	Instruction	AX	BX	CX	DX	SP	Screen (output)	Explanation
1	`mov ah, 2Ch`	2C00h	0000h	0000h	0000h	FFFEh	–	AH = 2Ch: get the time
2	`int 21h`	2C00h	0000h	0E23h	072Ah	FFFEh	–	CH = 0Eh (14), CL = 23h (35), DH = 07h, DL = 2Ah (42)
3	`mov al, ch`	2C0Eh	0000h	0E23h	072Ah	FFFEh	–	AL = hour
4	`call print2`	2C0Eh	0000h	0E23h	072Ah	FFFCh	–	return address 0109h on the stack
5	`aam`	0104h	0000h	0E23h	072Ah	FFFCh	–	14 = 1 × 10 + 4: AH = 1, AL = 4
6	`add ax, 3030h`	3134h	0000h	0E23h	072Ah	FFFCh	–	two digits to ASCII: '1', '4'
7	`mov bx, ax`	3134h	3134h	0E23h	072Ah	FFFCh	–	saved in BX; 02h clobbers AL

Step	Instruction	AX	BX	CX	DX	SP	Screen (output)	Explanation
8	`mov dl, bh`	3134h	3134h	0E23h	0731h	FFFCh	–	DL = '1'
9	`mov ah, 02h`	0234h	3134h	0E23h	0731h	FFFCh	–	AH = 02h
10	`int 21h`	0231h	3134h	0E23h	0731h	FFFCh	`1`	1 on the screen; AL = 31h
11	`mov dl, bl`	0231h	3134h	0E23h	0734h	FFFCh	`1`	DL = '4'
12	`int 21h`	0234h	3134h	0E23h	0734h	FFFCh	`14`	4 on the screen
13	`ret`	0234h	3134h	0E23h	0734h	FFFEh	`14`	return to 0109h
14	`mov dl, ':'`	0234h	3134h	0E23h	073Ah	FFFEh	`14`	DL = ':'
16	`int 21h`	023Ah	3134h	0E23h	073Ah	FFFEh	`14:`	: on the screen
17	`mov al, cl`	0223h	3134h	0E23h	073Ah	FFFEh	`14:`	AL = minute = 23h; CL not clobbered
18	`call print2`	0223h	3134h	0E23h	073Ah	FFFCh	`14:`	return address 0114h
19	`aam`	0305h	3134h	0E23h	073Ah	FFFCh	`14:`	35 = 3 × 10 + 5
20	`add ax, 3030h`	3335h	3134h	0E23h	073Ah	FFFCh	`14:`	'3', '5'
24	`int 21h`	0233h	3335h	0E23h	0733h	FFFCh	`14:3`	3 on the screen
26	`int 21h`	0235h	3335h	0E23h	0735h	FFFCh	`14:35`	5 on the screen
27	`ret`	0235h	3335h	0E23h	0735h	FFFEh	`14:35`	return to 0114h
28	`ret`	0235h	3335h	0E23h	0735h	0000h	`14:35`	0000h from the stack; SP = 0000h
29	`INT 20h (PSP:0000h)`	0235h	3335h	0E23h	0735h	0000h	`14:35`	INT 20h ends the program

Table 5.2.26: Trace of the program that writes the time as HH:MM (time 14:35:07.42).

The last two lines show how the program ends. The `RET` returning from the procedure (step 27) pops 0114h from the stack and brings SP back to FFFEh. The main program's `RET` (step 28) loads the 0000h at the very top of the stack into IP, and SP = 0000h. 0000h is the start of the PSP, and the `INT 20h` that DOS placed there ends the program. `14:35` stays on the screen.

5.2.7 INT 1Ah and the BIOS data area

5.2.7.1 The tick counter and the real-time clock

On the PC two separate pieces of hardware keep time. The first is the 8253/8254 programmable timer. Channel 0 divides a 1,193,182 Hz clock by 65,536 and produces the IRQ 0 interrupt about 18.2 times per second. The BIOS routine attached to this interrupt (INT 08h) increments a counter in memory each time. This counter is called the **tick counter**; it holds the number of ticks since midnight. Because the counter is in memory, it is lost when the computer is switched off.

The second, found in the AT and later computers, is the **real-time clock** (RTC). The RTC is a battery-powered CMOS chip and keeps running while the computer is off. It keeps the hour, minute, second and date in its own registers as BCD; they are read through I/O ports 70h and 71h. At boot, DOS gets its time from the RTC and then advances it with the tick counter. `INT 1Ah` offers a BIOS interface to both clocks.

AH	Function	Return
00h	read the tick counter	CX:DX = number of ticks since midnight, AL = day-rollover flag
02h	read the RTC time	CH = hour, CL = minute, DH = second (BCD), DL = daylight saving; CF = 1: error
04h	read the RTC date	CH = century, CL = year, DH = month, DL = day (BCD); CF = 1: error

Table 5.2.27: INT 1Ah services.

Definition 5.2.8

Packed BCD (packed binary-coded decimal) is a representation that puts two decimal digits in one byte: the high four bits (the high nibble) hold the tens digit and the low four bits the ones digit. The minute 35 is stored in packed BCD as 35h = 0011 0101b; in binary it is 23h = 0010 0011b.

The benefit of BCD is that the hexadecimal notation directly shows the decimal value: 35h reads as "thirty-five". No division is needed to write it on the screen; the two nibbles are separated and 30h is added to each. The high nibble is separated by shifting four bits to the right, the low nibble by masking with 0Fh. The 8086 has no constant shift count greater than 1; the shift count is given in CL.

Step	Operation	Result
1	35h = 0011 0101b	high nibble 3, low nibble 5
2	`shr dl, cl` (CL = 4)	0000 0011b = 03h
3	`and dl, 0Fh`	0000 0101b = 05h
4	+30h	'3' = 33h, '5' = 35h

Table 5.2.28: Converting packed BCD 35h to ASCII.

`INT 21h/2Ch` gives the same minute in binary: CL = 23h. There AAM or division by 10 is needed. If the two methods are mixed up, a wrong number is written to the screen, but the error shows only for values of 10 and above, because between 0 and 9 BCD and binary are the same.

Example 5.2.7

The time is 14:23. The RTC gives the minute as CL = 23h (BCD). If this value is sent to the `print2` procedure that uses AAM, AAM counts it as binary 35 and 35 is written on the screen. DOS, on the other hand, gives the minute as CL = 17h (binary 23). If this value is sent to a BCD routine, the nibbles are separated as 1 and 7 and 17 is written on the screen. The correct approach is to print the RTC value with SHR/AND and the DOS value with AAM.

Among the interrupts that emu8086 supports, only 00h (tick counter) is available for INT 1Ah. The 02h service works in a real DOS environment or in DOSBox; in emu8086 the time is read with `INT 21h/2Ch`.

5.2.7.2 Code trace: printing the RTC time

The following program writes the RTC time as HH:MM:ss. The two procedures share an entry: `colon` first writes ':' and then, without any jump, falls through into `putbcd`. For the hour `putbcd` is called directly, for the minute and second `colon` is called.

```
org 100h
mov ah, 02h
int 1Ah          ; CH=hr CL=min DH=sec
jc  done         ; CF=1: clock error
mov bl, cl       ; save the minute
mov cl, 4        ; SHR count
mov bh, ch
call putbcd      ; "14"
mov bh, bl
call colon       ; ":35"
mov bh, dh
call colon       ; ":07"
done: ret
colon: mov dl, ':'
mov ah, 02h
int 21h
putbcd:          ; BH = packed BCD
mov dl, bh
shr dl, cl       ; high digit
add dl, 30h      ; ASCII digit
mov ah, 02h
int 21h
mov dl, bh
and dl, 0Fh      ; low digit
add dl, 30h
int 21h
ret
```

The program first looks at CF. If the RTC is not running or is being updated at that moment, the service returns CF = 1 and the program ends without printing. In the trace the time is 14:35:07 and CF = 0. On return CH = 14h, CL = 35h, DH = 07h; DL = 00h says that daylight saving is off.

The use of registers has been carefully planned. Since the shift count 4 will be written to CL, the minute is first copied to BL. The procedure expects the value in BH; the hour (CH), minute (BL) and second (DH) are put into BH in turn. DL is needed for writing characters; but since the second sits in DH, DL being clobbered is no problem. INT 21h/02h does not touch BX, CX or DH.

Step	Instruction	AX	BX	CX	DX	SP	CF	Screen (output)	Explanation
1	`mov ah, 02h`	0200h	0000h	0000h	0000h	FFFEh	0	–	AH = 02h: read the RTC time
2	`int 1Ah`	0200h	0000h	1435h	0700h	FFFEh	0	–	CH = 14h, CL = 35h, DH = 07h (packed BCD); DL = 00h
3	`jc done`	0200h	0000h	1435h	0700h	FFFEh	0	–	CF = 0: no jump
4	`mov bl, cl`	0200h	0035h	1435h	0700h	FFFEh	0	–	the minute is saved in BL
5	`mov cl, 4`	0200h	0035h	1404h	0700h	FFFEh	0	–	CL = 4: the shift count

Step	Instruction	AX	BX	CX	DX	SP	CF	Screen (output)	Explanation
6	`mov bh, ch`	0200h	1435h	1404h	0700h	FFFEh	0	–	BH = hour = 14h
7	`call putbcd`	0200h	1435h	1404h	0700h	FFFCh	0	–	straight into `putbcd`
8	`mov dl, bh`	0200h	1435h	1404h	0714h	FFFCh	0	–	DL = 14h = 0001 0100b
9	`shr dl, cl`	0200h	1435h	1404h	0701h	FFFCh	0	–	4 bits right: 01h
10	`add dl, 30h`	0200h	1435h	1404h	0731h	FFFCh	0	–	DL = 31h = '1'
12	`int 21h`	0231h	1435h	1404h	0731h	FFFCh	0	`1`	1 on the screen
13	`mov dl, bh`	0231h	1435h	1404h	0714h	FFFCh	0	`1`	DL = 14h again
14	`and dl, 0Fh`	0231h	1435h	1404h	0704h	FFFCh	0	`1`	top 4 bits cleared: 04h
15	`add dl, 30h`	0231h	1435h	1404h	0734h	FFFCh	0	`1`	DL = 34h = '4'
16	`int 21h`	0234h	1435h	1404h	0734h	FFFCh	0	`14`	4 on the screen
17	`ret`	0234h	1435h	1404h	0734h	FFFEh	0	`14`	return
18	`mov bh, bl`	0234h	3535h	1404h	0734h	FFFEh	0	`14`	BH = minute = 35h
19	`call colon`	0234h	3535h	1404h	0734h	FFFCh	0	`14`	entry at `colon`
22	`int 21h`	023Ah	3535h	1404h	073Ah	FFFCh	0	`14:`	: on the screen
23	`mov dl, bh`	023Ah	3535h	1404h	0735h	FFFCh	0	`14:`	falls through into `putbcd`
27	`int 21h`	0233h	3535h	1404h	0733h	FFFCh	0	`14:3`	3 on the screen
31	`int 21h`	0235h	3535h	1404h	0735h	FFFCh	0	`14:35`	5 on the screen
32	`ret`	0235h	3535h	1404h	0735h	FFFEh	0	`14:35`	return
33	`mov bh, dh`	0235h	0735h	1404h	0735h	FFFEh	0	`14:35`	BH = second = 07h
37	`int 21h`	023Ah	0735h	1404h	073Ah	FFFCh	0	`14:35:`	: on the screen
42	`int 21h`	0230h	0735h	1404h	0730h	FFFCh	0	`14:35:0`	0 on the screen
46	`int 21h`	0237h	0735h	1404h	0737h	FFFCh	0	`14:35:07`	7 on the screen
47	`ret`	0237h	0735h	1404h	0737h	FFFEh	0	`14:35:07`	return
48	`ret`	0237h	0735h	1404h	0737h	0000h	0	`14:35:07`	program ends: SP = 0000h

Table 5.2.29: Trace of the program that writes the RTC time as HH:MM:ss (time 14:35:07).

For the hour `putbcd` writes twice. DL = 14h = 0001 0100b; `SHR DL, CL` shifts four bits right and DL = 01h. With 30h added, '1' is written. Then DL is loaded with 14h again, `AND DL, 0Fh` clears the top four bits and DL = 04h; '4' goes to the screen. Since AH is still 02h after the first call, it is not reloaded for the second `INT 21h`. For the minute, the `colon` entry first writes ':'; then the instructions fall through into the `putbcd` label. A label is not an instruction, only an address. In the final state the screen shows `14:35:07`.

Common mistake

Writing `MOV CL, 4` without saving the minute. The 35h in CL is lost and 04h is printed in place of the minute. Another mistake is to write `SHR DL, 4` for the 8086. This form exists on the 80186 and later processors; on the 8086 the shift count is either 1 or CL.

5.2.7.3 The BIOS data area

The BIOS keeps the state of the hardware at a fixed place in memory: in segment 0040h, physically in the area starting at 00400h. This area is called the **BIOS data area**. INT 16h reads the keyboard buffer, INT 1Ah the tick counter, and INT 10h the video mode and cursor positions from here. A program can read the same bytes directly. Table 5.2.30 lists frequently used locations.

Address	Size	Contents
0040:0010	word	equipment list (the value returned by INT 11h)
0040:0013	word	base memory, KB (INT 12h)
0040:0017	byte	keyboard flags: bit 0 right Shift, 1 left Shift, 2 Ctrl, 3 Alt, 4 Scroll Lock, 5 Num Lock, 6 Caps Lock, 7 Insert
0040:001A	word	buffer head: the offset of the next key to be read
0040:001C	word	buffer tail: the offset of the next free place (head = tail → buffer empty)
0040:001E–003D	32 bytes	keyboard buffer: 16 words (ASCII + scan code), circular; at most 15 keys
0040:0049	byte	current video mode (03h, 13h ...)
0040:004A	word	number of columns on the screen (80 in mode 03h)
0040:0050	8 words	cursor positions of the 8 pages (low byte column, high byte row)
0040:006C	double word	tick counter: number of ≈ 18.2 Hz interrupts since midnight
0040:0070	byte	midnight-passed flag (INT 1Ah/00h returns it in AL and clears it)

Table 5.2.30: Selected locations in the BIOS data area.

The physical address is found by multiplying the segment value by 16 and adding the offset: 0040:006C = 00400h + 006Ch = 0046Ch. The same byte can also be reached with the address 0000:046Ch; the two segment:offset pairs point to the same physical address.

The keyboard buffer is a **circular** queue. INT 09h writes the new key at the "tail" position and advances the tail by two bytes; INT 16h reads from the "head" position and advances the head. When one of the positions reaches the end of the area (003Eh), it wraps back to 001Eh. If head and tail are equal, the buffer is empty. That is why the 16-word area holds at most 15 keys: if a 16th key were written, the tail would equal the head and a full buffer would be taken for an empty one.

Example 5.2.8

0024h is read at 0040:001Ah and 0028h at 0040:001Ch. Since each key is one word, the number of waiting keys is (0028h − 0024h) ÷ 2 = 2. The ASCII code of the next key is at 0040:0024h and its scan code at 0040:0025h. The physical address of 0040:001Ah is 0041Ah.

5.2.7.4 Waiting with the tick counter

The tick counter is a ready-made clock for short waits. The program reads the starting value and then waits in a loop until the counter has advanced by the desired amount. The code below also tests Caps Lock in the keyboard flags and checks whether there is a key waiting in the buffer.

```
; wait about 1 second
mov ax, 0040h
mov es, ax
mov bx, es:[006Ch]   ; starting tick
wait1:
mov ax, es:[006Ch]   ; current tick
sub ax, bx           ; ticks elapsed
cmp ax, 18           ; 18 ticks ≈ 0.99 s
jb wait1
; Caps Lock on? Any key waiting?
test byte ptr es:[0017h], 40h
jnz caps_on          ; bit 6 = 1
mov ax, es:[001Ah]   ; buffer head
cmp ax, es:[001Ch]   ; buffer tail
je buf_empty
```

The duration of a tick is computed from the timer's division ratio. Table 5.2.31 gives the values.

Quantity	Calculation	Value
Tick rate	1,193,182 Hz ÷ 65,536	18.2065 Hz
One tick	65.536 ÷ 1.193.182 s	≈ 54.9 ms
18 ticks	18 × 54.925 ms	≈ 0,989 s
5 seconds	5 × 18,2065	≈ 91 ticks
One day	the value at which the counter resets	1800B0h = 1,573,040 ticks

Table 5.2.31: Time values of the tick counter.

Because the wait is measured in ticks, the resolution is about 55 ms; shorter times cannot be measured this way. The code reads only the low word of the counter. Since a difference is taken, the result is correct even if the low word wraps from FFFFh to 0000h: a 16-bit subtraction gives the correct difference modulo 2^{16} (mod 65,536). Only the reset at midnight spoils the calculation, because the counter jumps from 1800B0h to 0.

The Caps Lock check is done with `TEST`. The mask 40h = 0100 0000b selects bit 6; if the bit is 1 the result is not zero, ZF = 0 and `JNZ` jumps. TEST only reads the flags; it does not change the byte in the BIOS data area. The buffer check compares head and tail; if they are equal there is no waiting key. emu8086 does not update the whole BIOS data area as a real PC does; the tick counter and keyboard buffer examples work fully in DOSBox.

Common mistake

Adding the wait time to the start value and looking for the counter to equal the resulting end value: `cmp ax, target` / `jne wait1`. The counter can pass the target at a moment when the loop does not read it in that very iteration, and the loop then runs until the next wraparound. It is safe to take a difference and ask "has at least this much passed?" with `JB`.

5.2.8 Program termination

5.2.8.1 Three ways and the exit code

When a program has finished its work, it must give control back to DOS. The processor knows nothing of an "end of the program"; it always executes the byte IP points to as an instruction. If the termination instruction is forgotten, the processor runs the bytes after the program, often the program's data, as instructions. The result is unpredictable: garbage may be printed on the screen, the system may lock up. DOS offers three ways to end a program (Table 5.2.32).

Method	How it works	Where
`ret`	In a COM program the top of the stack holds 0000h; RET returns to PSP:0000h, where `int 20h` (CD 20) sits.	COM only, if the stack is balanced
`int 20h`	DOS terminates the program; CS must point to the PSP segment.	COM
`mov ax, 4C00h` `int 21h`	Terminates with the exit code in AL and closes open files.	COM and EXE · recommended

Table 5.2.32: Three ways to terminate a program.

Definition 5.2.9

The **PSP** (Program Segment Prefix) is the 256-byte area DOS places in front of every program. The command line, the address of the environment variables and return information are kept here. Its first two bytes are `CD 20`, that is the `INT 20h` instruction. Because COM programs are loaded right after the PSP, the code begins at `ORG 100h`.

Why RET ends a COM program can be understood from the stack. When DOS starts a COM program, it sets SP to FFFEh and puts the word 0000h there. If the program has left the stack balanced, that is, every PUSH has been followed by a POP and every CALL by a RET, the last RET loads this 0000h into IP. CS is still the program's segment and CS:0000h is the start of the PSP. There `INT 20h` runs and the program ends. If an extra word is left on the stack, RET takes that word for an address and jumps to some random place.

The recommended way is `INT 21h/4Ch`. It works in both COM and EXE programs, does not depend on the value of CS, and gives DOS an **exit code**. The exit code is in AL: 0 means success, nonzero values mean an error. A batch file (`.BAT`) can test this code with `IF ERRORLEVEL 1` and take different paths.

5.2.8.2 COM and EXE templates

emu8086 offers two templates for a new program. In the COM template code, data and stack are in a single 64 KB segment. In the EXE template each sits in a separate segment.

```
org 100h
    mov dx, offset msg
    mov ah, 09h
    int 21h
    ret             ; or int 20h
msg db 'Welcome$'
```

In a COM program CS = DS = ES = SS; the program does not set the segment registers. Code, data and stack together can be at most 64 KB. The file is machine code only, with no header at the start. DOS copies the file right behind the PSP, at offset 100h, and starts execution from there.

```
.model small
.stack 100h
.data
msg db 'Welcome$'
.code
start:
mov ax, @data
mov ds, ax       ; set DS!
mov dx, offset msg
mov ah, 09h
int 21h
mov ax, 4C00h
int 21h
end start
```

An EXE file has a header at its start. The header holds which segments the program consists of and the places that must be adjusted according to the load address (relocation information). The entry point is given with `end start` and the program can be larger than 64 KB. At start DOS sets CS:IP to the entry point and SS:SP to the stack segment. DS and ES point to the PSP. Before accessing the data segment, the program must set DS itself: `mov ax, @data` and `mov ds, ax`. `@data` is the segment value of the data segment; DOS fills in this value at load time.

Step	Instruction	AX	DS	DX	Screen (output)	Explanation
1	`mov ax, @data`	0720h	0700h	0000h	–	@data = data segment 0720h; DS still points to the PSP
2	`mov ds, ax`	0720h	0720h	0000h	–	DS = 0720h: the data segment
3	`mov dx, offset msg`	0720h	0720h	0000h	–	`msg` is at the start of the data segment
4	`mov ah, 09h`	0920h	0720h	0000h	–	AH = 09h
5	`int 21h`	0924h	0720h	0000h	`Welcome`	from 0720:0000 up to the '$'
6	`mov ax, 4C00h`	4C00h	0720h	0000h	`Welcome`	AH = 4Ch, AL = 00h
7	`int 21h`	4C00h	0720h	0000h	`Welcome`	return to DOS with exit code 0

Table 5.2.33: Trace of the EXE template (segment values according to the emu8086 layout).

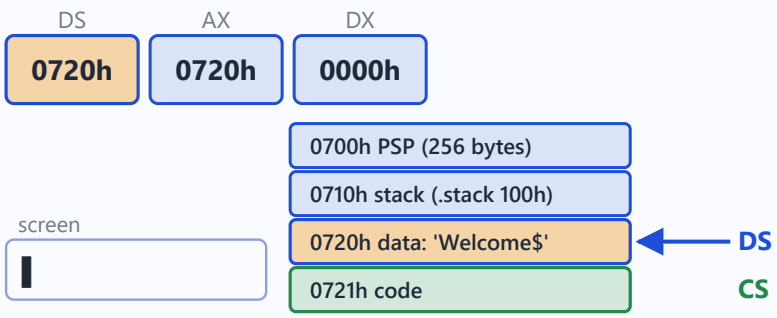


Figure 5.2.14: The segments after `MOV DS, AX`: PSP 0700h, stack 0710h, data 0720h (DS), code 0721h (CS). The segment values are emu8086's layout; on real DOS the numbers may differ, but the order and the need to set DS are the same.

At the start DS = ES = 0700h, that is, the PSP. The stack segment is 0710h: since the 256 bytes of the PSP are 16 paragraphs, 0700h + 10h. `.stack 100h` reserves a 256-byte stack, which is 10h paragraphs; the data segment therefore starts at 0720h. The 8 bytes of data fit in one paragraph and the code starts at 0721h. `MOV AX, @data` loads 0720h into AX; `MOV DS, AX` turns DS into the data segment. Because an immediate value cannot be written into a segment register, two steps are needed. `msg` is at the start of the data segment: `OFFSET msg` = 0000h. 09h writes `Welcome` starting at DS:DX = 0720:0000h. The last two lines return to DOS with exit code 0.

RET cannot be used in an EXE program. DOS does not put 0000h on an EXE's stack; SS:SP points to the program's own stack segment. RET jumps to whatever it finds there. If the `MOV DS, AX` line is forgotten, DX holds the right offset but the segment is wrong: DS:DX points to the start of the PSP and the bytes of the PSP, that is garbage, are printed on the screen.

5.2.9 Application: printing the result in binary

5.2.9.1 The program

The program in this part combines many services with a small calculation. First 5 + 10 − 1 = 14 is computed and the result is written to BL. Then the eight bits of BL, starting from the most significant bit (MSB), are printed on the screen one by one as the character '0' or '1'. At the end the binary suffix 'b' is written and a key press is awaited. The expected output is `00001110b`.

```
org 100h
mov al, 5        ; 00000101b
mov bl, 10       ; 0Ah = 00001010b
; 5 + 10 = 15 = 0Fh = 00001111b
add bl, al
; 15 - 1 = 14 = 0Eh = 00001110b
sub bl, 1
mov cx, 8        ; print result in binary
print:
mov ah, 2        ; print function
mov dl, '0'
test bl, 10000000b  ; test top bit
jz zero
mov dl, '1'
zero:
int 21h
shl bl, 1
loop print
; print the binary suffix b
mov dl, 'b'
int 21h
; wait for a key press
mov ah, 0
int 16h
ret
```

The number does not need to be "made binary" in memory or in a register; it is binary already. Printing means converting each bit to an ASCII character: 30h for a 0 bit, 31h for a 1 bit. On each iteration the loop looks at bit 7, writes the character, and shifts the number one bit left. The next bit thus takes the place of bit 7. After eight iterations all the bits have been written from left to right.

5.2.9.2 Reading bits with TEST and SHL

`TEST BL, 10000000b` ANDs BL with the mask bit by bit but writes the result nowhere; it only sets the flags. The mask has a single 1 bit, at bit 7. If bit 7 of BL is 0 the AND result is 0 and ZF = 1 is set. If bit 7 = 1 the result is 80h and ZF = 0. `JZ` jumps when ZF = 1. TEST also always clears CF and OF.

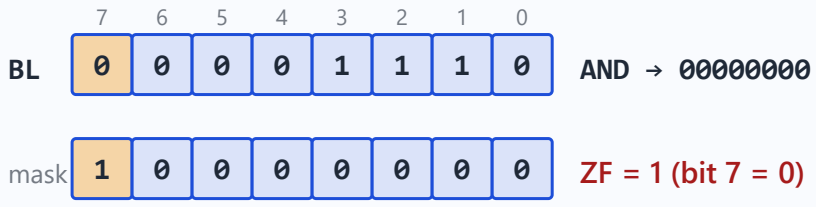


Figure 5.2.15: `TEST BL, 10000000b` : BL = 00001110b is ANDed with the mask and the result is 00000000b; ZF = 1 (bit 7 = 0). BL does not change.

`SHL BL, 1` shifts all the bits one place left. The bit that leaves from bit 7 goes to CF and a 0 enters at bit 0. The next bit is now in the place of bit 7. Because TEST does not store its result, it does not spoil BL; the only instruction that changes it, deliberately, is SHL.

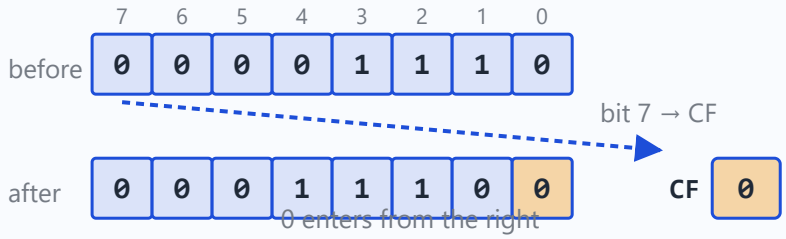


Figure 5.2.16: `SHL BL, 1` : 00001110b → 00011100b; the bit that leaves, bit 7 (0), goes to CF and a 0 enters from the right.

The same job can also be done by looking at CF after SHL: first `SHL BL, 1`, then `JC` tests whether the bit that left was 1. In this order the shift must be done before printing, because the value of the bit is now in CF, not in BL. There is also a reason for redoing `MOV AH, 2` on every iteration: `INT 21h/02h` does not spoil AH, but if another service is added to the loop later, AH will already be protected.

5.2.9.3 Code trace

How the flags change in the calculation part is seen in Table 5.2.34. The `MOV` instructions do not touch the flags. `ADD BL, AL` produces 0Ah + 05h = 0Fh and there is no carry (CF = 0). The result has four 1 bits; because the parity is even, PF = 1. `SUB BL, 1` produces 0Eh = 00001110b; three 1 bits make PF 0, and there is no borrow (CF = 0).

Step	Instruction	AH	DL	BL	CX	ZF	CF	PF	Screen (output)	Explanation
1	`mov al, 5`	00h	00h	00h	0000h	0	0	0	–	AL = 05h; MOV does not change the flags
2	`mov bl, 10`	00h	00h	0Ah	0000h	0	0	0	–	BL = 0Ah
3	`add bl, al`	00h	00h	0Fh	0000h	0	0	1	–	0Ah + 05h = 0Fh; four 1 bits: PF = 1
4	`sub bl, 1`	00h	00h	0Eh	0000h	0	0	0	–	0Fh − 1 = 0Eh; three 1 bits: PF = 0
5	`mov cx, 8`	00h	00h	0Eh	0008h	0	0	0	–	eight iterations for eight bits

Table 5.2.34: Trace of the calculation part (flags after each instruction).

The eight iterations of the loop follow the same pattern. Table 5.2.35 gives two iterations step by step: the first, where bit 7 is 0, and the fifth, where it is 1. In the first iteration TEST makes ZF = 1, `JZ` jumps and the '0' in DL is written. In the fifth iteration BL = E0h; TEST makes ZF = 0, JZ does not jump, `MOV DL, '1'` runs and '1' is written. In both iterations SHL throws the bit that leaves into CF, and LOOP decreases CX by one.

Step	Instruction	AH	DL	BL	CX	ZF	CF	PF	Screen (output)	Explanation
6	`mov ah, 2`	02h	00h	0Eh	0008h	0	0	0	–	iteration 1: AH = 2

Step	Instruction	AH	DL	BL	CX	ZF	CF	PF	Screen (output)	Explanation
7	`mov dl, '0'`	02h	30h	0Eh	0008h	0	0	0	–	default '0'
8	`test bl, 10000000b`	02h	30h	0Eh	0008h	1	0	1	–	0Eh AND 80h = 0: ZF = 1
9	`jz zero`	02h	30h	0Eh	0008h	1	0	1	–	ZF = 1: jump to `zero`
10	`int 21h`	02h	30h	0Eh	0008h	1	0	1	`0`	0 on the screen
11	`shl bl, 1`	02h	30h	1Ch	0008h	0	0	0	`0`	BL = 1Ch; the bit that left is 0
12	`loop print`	02h	30h	1Ch	0007h	0	0	0	`0`	CX = 7 ≠ 0: to `print`
34	`mov ah, 2`	02h	30h	E0h	0004h	0	0	0	`0000`	iteration 5: AH = 2
35	`mov dl, '0'`	02h	30h	E0h	0004h	0	0	0	`0000`	default '0'
36	`test bl, 10000000b`	02h	30h	E0h	0004h	0	0	0	`0000`	E0h AND 80h = 80h: ZF = 0
37	`jz zero`	02h	30h	E0h	0004h	0	0	0	`0000`	no jump
38	`mov dl, '1'`	02h	31h	E0h	0004h	0	0	0	`0000`	DL = '1'
39	`int 21h`	02h	31h	E0h	0004h	0	0	0	`00001`	1 on the screen
40	`shl bl, 1`	02h	31h	C0h	0004h	0	1	1	`00001`	BL = C0h; the bit that left, CF = 1
41	`loop print`	02h	31h	C0h	0003h	0	1	1	`00001`	CX = 3 ≠ 0: to `print`

Table 5.2.35: Iterations 1 and 5 of the loop step by step (in iteration 5, bit 7 = 1).

Table 5.2.36 is a summary of the eight iterations. BL shifts one place left on each iteration: 0Eh, 1Ch, 38h, 70h, E0h, C0h, 80h, 00h. The bit 7 column gives the order of the characters written: 0, 0, 0, 0, 1, 1, 1, 0. This is the binary notation of 0Eh. After the eighth iteration CX = 0 and LOOP does not jump.

Iteration	BL (before TEST)	Bit 7	Written	BL (after SHL)	CF	CX	Screen
1	0Eh = 00001110	0	`0`	1Ch	0	0007h	`0`
2	1Ch = 00011100	0	`0`	38h	0	0006h	`00`
3	38h = 00111000	0	`0`	70h	0	0005h	`000`
4	70h = 01110000	0	`0`	E0h	0	0004h	`0000`
5	E0h = 11100000	1	`1`	C0h	1	0003h	`00001`
6	C0h = 11000000	1	`1`	80h	1	0002h	`000011`
7	80h = 10000000	1	`1`	00h	1	0001h	`0000111`
8	00h = 00000000	0	`0`	00h	0	0000h	`00001110`

Table 5.2.36: Summary of the eight iterations (each row is the state after that iteration's LOOP instruction).

After the loop DL = 'b' is set, and because AH is still 2, `INT 21h` writes 'b' directly. `MOV AH, 0` and `INT 16h` wait for a key; in the trace, when Enter is pressed, AH = 1Ch (scan code), AL = 0Dh are returned. The last `RET` ends the program. The final state of the screen is as follows:

```
00001110b_
```

Example 5.2.9

Let the same loop run with BL = B5h. B5h = 1011 0101b. In the iterations bit 7 is, in turn, 1, 0, 1, 1, 0, 1, 0, 1 and `10110101b` is written on the screen. The bits that SHL throws into CF in the iterations follow the same order, because the bit that leaves is, on each iteration, the bit TEST looked at.

The lessons from the program are brief. Because LOOP decreases CX, CX is not used for anything else inside the loop. INT 21h/02h changes AL (AL = DL); if there is a value kept in AL it must be protected first. TEST sets the flags but does not change the operand; so the same register can be tested one after another with many masks.

5.2.9.4 A mini menu

The last program is the skeleton of a user interface: it shows a menu, reads a key, and acts according to the key. The '1' key says hello, 'q' ends the program, and any other key shows the menu again.

```
org 100h
show:
mov dx, offset menu
mov ah, 09h
int 21h
mov ah, 01h      ; read the choice
int 21h
cmp al, 'q'
je quit
cmp al, '1'
jne show
mov dx, offset greet
mov ah, 09h
int 21h
jmp show
quit:
mov ax, 4C00h    ; exit code 0
int 21h
menu  db 0Dh, 0Ah, '1:Hello q:Leave $'
greet db 0Dh, 0Ah, 'Hello!$'
```

The user presses 1 and then q. Table 5.2.37 gives the trace.

Step	Instruction	AX	DX	ZF	Screen (output)	Explanation
1	`mov dx, offset menu`	0000h	0121h	0	–	DX → the menu text
2	`mov ah, 09h`	0900h	0121h	0	–	AH = 09h

Step	Instruction	AX	DX	ZF	Screen (output)	Explanation
3	`int 21h`	0924h	0121h	0	`↵1:Hello q:Leave`	CR LF and the menu are written
4	`mov ah, 01h`	0124h	0121h	0	`↵1:Hello q:Leave`	AH = 01h: read and echo
5	`int 21h`	0131h	0121h	0	`↵1:Hello q:Leave 1`	1: AL = 31h
6	`cmp al, 'q'`	0131h	0121h	0	`↵1:Hello q:Leave 1`	31h − 71h ≠ 0: ZF = 0
7	`je quit`	0131h	0121h	0	`↵1:Hello q:Leave 1`	no jump
8	`cmp al, '1'`	0131h	0121h	1	`↵1:Hello q:Leave 1`	31h − 31h = 0: ZF = 1
9	`jne show`	0131h	0121h	1	`↵1:Hello q:Leave 1`	equal: no jump
10	`mov dx, offset greet`	0131h	0134h	1	`↵1:Hello q:Leave 1`	DX → `greet`
12	`int 21h`	0924h	0134h	1	`↵1:Hello q:Leave 1↵Hello!`	Hello! on a new line
13	`jmp show`	0924h	0134h	1	`↵1:Hello q:Leave 1↵Hello!`	back to the menu
16	`int 21h`	0924h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave`	the menu is written again
18	`int 21h`	0171h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave q`	q: AL = 71h
19	`cmp al, 'q'`	0171h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave q`	equal: ZF = 1
20	`je quit`	0171h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave q`	jump to `quit`
21	`mov ax, 4C00h`	4C00h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave q`	AH = 4Ch, AL = 00h
22	`int 21h`	4C00h	0121h	1	`↵1:Hello q:Leave 1↵Hello!↵1:Hello q:Leave q`	return to DOS with exit code 0

Table 5.2.37: Trace of the mini menu program (keys pressed: 1, q).

Because the menu text begins with CR LF, it is written on a new line at each display. `AH = 01h` reads the key and echoes it; the 1 the user pressed appears right next to the menu. The first comparison is between 31h and 71h ('q'); they are not equal, ZF = 0 and `JE` does not jump. The second comparison is made with '1'; they are equal, ZF = 1 and `JNE` does not jump. `greet` is written and `JMP` returns to the menu. In the second iteration q is read, `CMP AL, 'q'` makes ZF = 1 and `JE` jumps to `quit`. `MOV AX, 4C00h` sets AH = 4Ch and AL = 00h in a single instruction; the program ends with exit code 0. The final state of the screen is this:

```
1:Hello q:Leave 1
Hello!
1:Hello q:Leave q
```

The data sits after the last instruction. Because `INT 21h/4Ch` does not return, the processor never reaches the bytes of `menu`. If the termination lines were deleted, the processor would try to decode the bytes 0Dh, 0Ah, '1' ... as instructions.

5.2.9.5 Common mistakes

Most of the errors in programs that call services fall into a few patterns. Table 5.2.38 collects them with their causes and fixes.

Mistake	Result	Fix
'$' forgotten	INT 21h/09h writes memory until it finds '$'; garbage fills the screen	'$' at the end of the string
DS not set (EXE)	DX is right but the segment is wrong; the PSP is printed	`mov ax, @data` · `mov ds, ax`
AL clobbered	02h and 09h change AL; the character read is lost	save the value in another register
cursor after 09h	INT 10h/09h does not advance the cursor; the next text lands on top	move the cursor with 02h
DS:DX with 13h	the BIOS string service expects the address in ES:BP	`lea bp, msg` ; DX is the position
0Ah buffer	capacity not written, or the text read from buffer+0	byte 0 = capacity; text at buffer+2
LF only	the cursor goes down a line but not to the start of the line	CR + LF (0Dh, 0Ah)
No termination	the data is executed as instructions	RET (COM) or INT 21h/4Ch

Table 5.2.38: Frequent mistakes with interrupt services.

Common mistake

To rewrite a key that was read on a new line, this order is used: read with `AH = 01h`, write CR and LF with `AH = 02h`, then write the key with `MOV DL, AL`. When the user presses k, only the echoed k appears on the screen and the cursor goes down two lines. After the last 02h call AL = 0Ah; `MOV DL, AL` takes the LF, not the key. The key that was read is saved at once with `MOV BL, AL`, and then `MOV DL, BL` is written.

5.2.10 Exercises and solutions

The exercises in the slides are given below with their solutions. For the exercises that ask for a program, the code is followed by a short explanation of the solution. The solutions are not the only way; programs that do the same job with other services are also correct.

Exercise 1. How is a program written with INT 10h/13h that writes "8086" in white on a blue background exactly in the middle of the screen at (12, 38)?

```
org 100h
mov ax, 1301h      ; AH=13h, AL=01h
mov bx, 001Fh      ; BH=0, BL=1Fh
mov cx, 4          ; 4 characters
mov dx, 0C26h      ; row 12, column 38
lea bp, text       ; ES:BP → string
int 10h
ret
text db '8086'
```

The four characters are written to columns 38–41; since the middle of the 80-column screen is 40, the text is centered. DX = 0C26h means DH = 0Ch = 12 and DL = 26h = 38. BL = 1Fh is white text on a blue background. In a COM program ES = DS, so ES is not set separately. The place of the first cell in video memory is (12 × 80 + 38) × 2 = 07CCh.

Exercise 2. How is a program written that reads two digits (01h) from the user and prints their sum as a single digit (sum ≤ 9)?

```
org 100h
mov ah, 01h
int 21h            ; first digit
sub al, 30h
mov bl, al
mov ah, 01h
int 21h            ; second digit
sub al, 30h
add bl, al         ; BL = sum
mov ah, 02h
mov dl, 0Dh
int 21h
mov dl, 0Ah
int 21h
mov dl, bl
add dl, 30h        ; ASCII digit
int 21h
ret
```

The numeric value is found by subtracting 30h from each digit. The first number is saved in BL, because the second read changes AL. The sum forms in BL and is written after CR LF with 30h added. When the user presses 3 and 5, `35` appears on the screen and `8` on the line below.

Exercise 3. How is a program written that prints, in hexadecimal, the scan codes of the keys pressed until Esc is pressed?

```
org 100h
read:
mov ah, 0
int 16h            ; AH = scan, AL = ASCII
cmp al, 1Bh        ; Esc?
je done
lea bx, hex        ; XLAT table
mov dh, ah         ; save the scan code
mov al, dh
mov cl, 4
shr al, cl         ; high digit
xlat
mov dl, al
mov ah, 02h
int 21h
mov al, dh
and al, 0Fh        ; low digit
xlat
mov dl, al
int 21h
mov dl, ' '
int 21h
jmp read
done: ret
hex db '0123456789ABCDEF'
```

The scan code arrives in AH and is saved in DH, because AL and AH will change with the XLAT and 02h calls. `XLAT` performs AL ← [BX + AL]: it converts a number from 0 to 15 into the character in the table. The high digit is separated by shifting four bits right, the low digit by masking with 0Fh. When the a and 1 keys are pressed, `1E 02` appears on the screen; Esc ends the program.

Exercise 4. How is a program written that prints the text read with 0Ah in reverse?

```
org 100h
mov dx, offset buffer
mov ah, 0Ah
int 21h            ; read a line
mov ah, 02h
mov dl, 0Ah        ; next line (Enter echoed a CR)
int 21h
mov cl, [buffer+1] ; count read
mov ch, 0
jcxz done
mov si, offset buffer+1
add si, cx         ; last character
std                ; backwards
rev:
lodsb
mov dl, al
int 21h
loop rev
cld
done: ret
buffer db 20, ?, 20 dup(?)
```

The text starts at buffer+2 and its length is in [buffer+1]. The last character is at address buffer + 2 + count − 1 = buffer + 1 + count; SI is set there. `STD` sets the direction flag and `LODSB` does AL ← [SI] on each iteration and decreases SI by one. Because CX = count, LOOP writes each character once. For an empty line (count 0) `JCXZ` never enters the loop; otherwise LOOP would take CX from 0 to FFFFh and run 65,536 iterations. At the end of the program `CLD` returns the direction flag to its default. When Micro is typed, `orciM` appears on the next line.

Exercise 5. How is the date printed in the format DD.MM.YYYY? What method is used for the year?

```
org 100h
mov ah, 2Ah
int 21h            ; CX=year DH=month DL=day
mov bx, cx         ; save the year
mov al, dl
call print2        ; DD
mov dl, '.'
int 21h
mov al, dh
call print2        ; MM
mov dl, '.'
int 21h
mov ax, bx         ; year
mov cx, 0          ; digit counter
mov si, 10
div10:
xor dx, dx
div si             ; DX:AX / 10
push dx            ; remainder = digit
inc cx
test ax, ax
jnz div10
mov ah, 02h
emit:
pop dx
add dl, 30h
int 21h
loop emit
ret
print2:            ; AL (0-99) → 2 digits, DH preserved
push dx
aam
add ax, 3030h
push ax
mov dl, ah
mov ah, 02h
int 21h
pop ax
mov dl, al
mov ah, 02h
int 21h
pop dx
ret
```

Since the day and month are smaller than 99, AAM is enough; the `print2` procedure preserves the month in DH on the stack. The year is a 16-bit number and cannot be split with AAM. The year is divided by 10 repeatedly: `DIV SI` divides DX:AX by 10, writing the quotient to AX and the remainder to DX. The remainders come out in the order ones, tens, hundreds, thousands; the order of writing is

the reverse. So the remainders are pushed on the stack and then popped and written. DX is zeroed before every division, because DIV takes the dividend as DX:AX. For January 7, 2026, `07.01.2026` is written on the screen.

Exercise 6. How is the '$' character printed with `mov ah, 09h` / `int 21h`?

It cannot be printed; 09h counts the first '$' as the end of the string and does not write it. For '$' the 02h service is used: `mov ah, 02h`, `mov dl, '$'` (24h), `int 21h`. INT 10h/0Eh with AL = 24h does the same job.

Exercise 7. What is the offset, within segment A000h, of pixel (160, 100) in mode 13h?

100 × 320 + 160 = 32,160 = 7DA0h. The physical address is A7DA0h.

Exercise 8. INT 1Ah/02h returned DH = 59h. What is written on the screen? With what value would the same second arrive from INT 21h/2Ch?

59h is packed BCD: high nibble 5, low nibble 9; `59` is written on the screen. INT 21h/2Ch gives the second in binary: 59 = 3Bh, that is DH = 3Bh.

Exercise 9. How many ticks must be waited to wait about 5 seconds with the tick counter at 0040:006C?

5 × 18.2065 ≈ 91 ticks. At 54.9 ms per tick, 91 ticks is 4.998 seconds.

5.2.11 Summary

- The BIOS is a set of ROM routines that drive the hardware directly; DOS, on top of it, offers file, string, clock and program services. A program reaches both with `INT n`, without knowing the address of the service.
- The service-call pattern: AH ← service number, parameters in the other registers, `INT n`, result in registers or flags. Services change some registers: INT 21h/02h makes AL = DL, 09h makes AL = 24h.
- A new line takes CR LF (0Dh, 0Ah). Digit = code − 30h; uppercase and lowercase letters differ in bit 5.
- INT 10h: 00h selects the mode and clears the screen; 02h/03h set and read the cursor. 09h writes in color but does not advance the cursor; 0Eh writes teletype and advances the cursor. 13h writes the string at ES:BP of length CX; 06h scrolls or clears a window.
- Attribute = background × 16 + foreground; bit 7 is blink. 07h is the default, 1Fh is white on blue, 4Eh is yellow on red.
- In mode 03h the video memory is at B800:0000h; the cell is at address (row × 80 + column) × 2, with the character in the even byte and the attribute in the odd byte. In mode 13h, from A000:0000h there is one byte per pixel and the address is y × 320 + x.

- INT 16h/00h waits for a key and gives the scan code in AH and ASCII in AL; for extended keys AL = 00h. 01h polls without waiting: ZF = 1 means no key.
- INT 21h: 01h reads and echoes, 02h writes DL, 09h writes a '$'-terminated string, 0Ah reads a buffered line. DOS output can be redirected; BIOS output cannot.
- The 0Ah buffer: byte 0 capacity, byte 1 number read, the text starts at byte 2 and ends with 0Dh.
- INT 21h/2Ah and 2Ch give the date and time in binary (printed with AAM); INT 1Ah/02h gives the RTC time in packed BCD (printed with SHR and AND).
- The BIOS data area is in segment 0040h: keyboard flags at 0017h, buffer head and tail at 001Ah and 001Ch, tick counter at 006Ch. The tick rate is 18.2065 Hz, one tick ≈ 54.9 ms.
- A program ends with RET (only a COM program with a balanced stack), INT 20h (COM) or the recommended INT 21h/4Ch; AL is the exit code. An EXE program sets DS itself with `@data`.

5.2.12 Key terms

BIOS (Basic Input/Output System) software interrupt interrupt vector ASCII ASCII video mode cursor attribute byte pixel little-endian scan code keyboard buffer polling packed / unpacked BCD nibble real-time clock (RTC) program segment prefix (PSP) program termination emulator

---

5.2.13 Test Yourself

1. In mode 03h, with the cursor at (24, 0), INT 10h is called with AH = 09h, AL = '-', BL = 1Eh, CX = 80. What appears on the screen, and where does the cursor stay?

   Answer

   A yellow '-' on a blue background is written into all 80 cells of the last row. 09h does not advance the cursor and does not scroll the screen; the cursor stays at (24, 0). If the same row were written 80 times with 0Eh, after the last character the cursor would try to move to the next line and the screen would scroll up one line.

2. What is the attribute byte of blinking white text on a red background?

   Answer

   Bit 7 = 1 (blink), background 100b (red), foreground 1111b (white): 1100 1111b = **CFh**.

3. In mode 03h, at what offset is the character byte of the first cell (24, 0) of the last row? What is its

physical address?

Answer

(24 × 80 + 0) × 2 = 3840 = 0F00h. The character is at B800:0F00h and the attribute at B800:0F01h. The physical address is B8000h + 0F00h = **B8F00h**.

4. While reading into a buffer defined as `buffer db 4, ?, 4 dup(?)` with INT 21h/0Ah, the user types abcdef and presses Enter. What are the first six bytes of the buffer?

Answer

Because the capacity is 4 including Enter, at most 3 characters are accepted; the bell rings at d, e and f. The buffer becomes `04 03 61 62 63 0D`: the capacity, the count 3, 'a', 'b', 'c' and Enter.

5. What is AL after the instructions `mov ah, 02h`, `mov dl, 'Z'`, `int 21h`?

Answer

INT 21h/02h puts DL into AL on return: AL = 5Ah ('Z'). If a value had been kept in AL before, it is lost.

6. The time is 23:00. A program that reads the time from DOS (INT 21h/2Ch) sends CH to the routine that prints BCD (SHR/AND). What appears on the screen? What would appear if CH read from the RTC were sent to the routine that uses AAM?

Answer

DOS gives binary: CH = 23 = 17h. The BCD routine separates the nibbles as 1 and 7 and writes **17**. The RTC gives CH = 23h; AAM counts this as binary 35 (23h = 35) and **35** is written. In both cases the correct value is 23.

7. How many ticks must be waited to wait about 3 seconds with the tick counter? Can a 10 ms wait be measured this way?

Answer

3 × 18.2065 ≈ 54.6; about **55 ticks**. 10 ms cannot be measured: one tick is about 54.9 ms and the counter cannot tell apart intervals shorter than that.

8. In mode 13h, what is the offset of pixel (200, 150)? What are the offsets of the pixels one to the

right and one below?

Answer

150 × 320 + 200 = 48,200 = **BC48h**. The one to the right, (201, 150), is at BC49h; the one below, (200, 151), is at BC48h + 140h = BD88h.

9. In the BIOS data area the buffer head is read as 003Ch and the buffer tail as 0020h. How many keys are waiting?

Answer

Because the buffer is circular, the tail may have fallen behind the head. From the head to the end of the area (003Ch–003Dh) there is one word, and from the start of the area to the tail (001Eh–001Fh) one word: **2 keys**. The general formula is (tail − head + 32) mod 32, divided by 2: (20h − 3Ch + 20h) = 4 bytes.

10. An EXE program ends with `RET` instead of `MOV AX, 4C00h` · `INT 21h`. What happens?

Answer

There is no 0000h word placed by DOS on the EXE's stack and CS does not point to the PSP. RET loads whatever it finds at the top of the stack into IP and jumps to a random address in the code segment. The program may lock up. An EXE program is ended with INT 21h/4Ch.

5.2.14 Further reading

- Ralf Brown, *Ralf Brown's Interrupt List*: a reference list that describes all the services of INT 10h, 16h, 1Ah and 21h at register level, with BIOS version differences.
- IBM, *Personal Computer AT Technical Reference*, 1984: the BIOS source listing; the original definition of the BIOS data area and of the keyboard and timer interrupts.
- Peter Abel, *IBM PC Assembly Language and Programming*, 5th ed.: covers the screen, keyboard and DOS services with many example programs.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed.: treats BIOS and DOS calls together with the 8086 hardware and interrupt structure.

5.2.15 References

1. R. Brown, *Ralf Brown's Interrupt List* (detailed reference for INT 10h, 16h, 21h).
2. IBM, *Personal Computer AT Technical Reference*, BIOS section, 1984.
3. P. Abel, *IBM PC Assembly Language and Programming*, 5th ed., Prentice Hall, 2001.

4. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.

5. emu8086 help documents: supported interrupt functions.

6. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş* (in Turkish), 2009.

7. Intel, *8086 16-Bit HMOS Microprocessor* data sheet (instruction timings).

PART III

Instructions and Programming

Chapters 6–9

Arithmetic, Logic and Shift Instructions

The real job of a processor is to compute. In the 8086 this job is done by the arithmetic logic unit (ALU) inside the execution unit: it adds two numbers, subtracts one from the other, passes their bits through AND, OR and exclusive-OR gates, and shifts them left or right. At the end of every operation the ALU not only writes the result to a register or to memory; it also records how the result came out in the six status bits of the flags register. Later instructions, especially conditional jumps, make their decisions by looking at these bits. These notes take up the arithmetic and logic instructions of the 8086 one by one and show which flag each instruction changes and how. First the two readings, unsigned and signed, of the same bit pattern are explained, together with the difference between the carry and overflow flags. Then come addition, subtraction, comparison and the logic instructions, the implicit registers of multiplication and division, sign extension, the single-operand instructions, and shifts and rotates. How 32-bit addition, subtraction and multiplication are done on a 16-bit processor is traced step by step. The last parts examine array summation, binary-coded decimal (BCD) arithmetic and temperature conversion programs at register and flag level.

LEARNING OUTCOMES

Compute the flags

Find the values of CF, ZF, SF, OF, PF and AF from the result of an addition or subtraction; separate the unsigned and signed readings of the same result and explain the difference between CF and OF.

Use instructions with the right operands

Know the valid operand forms of the two-operand, multiply-divide and single-operand instruction groups; say which flags each instruction changes, which it preserves and which it leaves undefined.

Multiply and divide

Use the implicit registers (AL, AX, DX:AX) of MUL, IMUL, DIV and IDIV; prepare the dividend before a division with CBW, CWD or by zeroing the upper half; recognize the causes of a divide error.

Multiword arithmetic
Do 32-bit addition and subtraction with ADC and SBB and a 32 × 32-bit multiplication with four MULs; trace how the carry is passed from word to word.

Bit operations
Clear, set, invert and test bits with AND, OR, XOR and TEST; trace the effect of shift and rotate instructions on CF; multiply and divide with shifts and additions.

Trace application programs
Trace array summation, BCD adjustment and temperature conversion programs step by step at register, memory and flag level; find the range of numbers for which a program is valid.

6.1 Flags and the result of an operation

6.1.1 Where does the ALU write its result?

The inside of the 8086 is divided into two units. The **bus interface unit** (BIU) fetches instruction bytes from memory, places them in the prefetch queue, and reads and writes memory operands. The **execution unit** (EU) decodes and executes the instruction it takes from the queue. All calculations such as addition, subtraction, AND, OR and shifts are done in the 16-bit **arithmetic logic unit** (ALU) inside the EU. The ALU has two inputs and one output. The inputs come from registers, from the immediate value in the instruction, or from a number the BIU fetched from memory. The output goes to the destination register or, through the BIU, to memory.

The ALU has a second output as well: a few bits that describe the properties of the result. Is the result zero, is its leftmost bit 1, did a carry come out of the leftmost bit? The yes/no answers to these questions are written to the status bits of the 16-bit **flags register** (FLAGS). Figure 6.1 shows the bit layout of the flags register. Nine of the sixteen bits are used. Six are **status flags**: CF, PF, AF, ZF, SF and OF describe the result of the last arithmetic or logic operation. Three are **control flags**: TF, IF and DF govern the behavior of the processor and arithmetic instructions do not touch them.

Figure 6.1: The 16-bit flags register. The dashed bits are not used on the 8086; the six status flags (CF, PF, AF, ZF, SF, OF) hold the result of arithmetic and logic instructions.

Flags are a bridge that carries information between instructions. The `CMP AL, BL` instruction compares two numbers but writes the result to no register; it only changes the flags. A conditional jump right after it, such as `JZ`, `JC` or `JO`, makes its decision by looking at these flags. In multiword

addition, the carry of the first addition waits in CF and the second addition takes it up with ADC. So two questions are asked of every instruction: where does it write its result, and which flags does it change and how?

Table 6.1 collects the status and control flags, the condition under which each becomes 1, and a short example of each. A detailed description of each flag and how it is produced in hardware is given in the notes on the 8086 registers (Chapter 2.2). Here only the rules sufficient for computing the flags after an addition or subtraction are summarized.

Bit	Flag	Type	Set to 1 if...	Example
0	CF (carry)	status	unsigned overflow: a carry out of the most significant bit in addition, a borrow in subtraction	FFh + 01h → AL = 00h, CF = 1; 7 − 9 → CF = 1
2	PF (parity)	status	the low byte of the result has an even number of 1 bits	96h = 1001 0110b → four 1s → PF = 1
4	AF (auxiliary carry)	status	a carry or borrow from bit 3 to bit 4	08h + 08h = 10h → AF = 1
6	ZF (zero)	status	all bits of the result are 0	`SUB AL, AL` → ZF = 1
7	SF (sign)	status	the most significant bit (MSB) of the result is 1	96h → SF = 1
8	TF (trap)	control	INT 1 after every instruction (single step)	used by debuggers
9	IF (interrupt)	control	interrupts from the INTR pin are accepted	`STI` → 1, `CLI` → 0
10	DF (direction)	control	string instructions decrement SI and DI	`CLD` → 0, `STD` → 1
11	OF (overflow)	status	signed overflow: the result did not fit in the signed range	100 + 50 → AL = 96h (−106), OF = 1

Table 6.1: The flags of the 8086. Status flags describe the result; control flags govern the behavior of the processor.

The hardware counterpart of the status flags is simple. The ALU's adder adds the bits from right to left with a chain of full adders. The carry that bit 7 produces in a byte operation, or bit 15 in a word operation, goes directly to CF. The carry that bit 3 produces goes to AF. SF is a copy of the most significant bit of the result. ZF is the output of a NOR gate that takes all the bits of the result as input. PF passes the eight bits of the low byte through an exclusive-OR tree and inverts the result; even for a word result it looks only at the low byte. OF is the exclusive-OR of the carry *into* the most significant bit and the carry *out of* that bit. This last rule is seen with numbers in the examples below.

In subtraction the ALU does not use a separate subtractor. It performs A − B by adding A + (the ones' complement of B) + 1; this is addition with a two's complement. A carry out of the leftmost bit in this addition actually means that no borrow was needed. The 8086 therefore inverts the carry in subtraction and writes it to CF. As a result, CF behaves in subtraction like a **borrow** flag: in the unsigned reading, CF = 1 if a larger number was subtracted from a smaller one.

6.1.2 CF or OF?

The eight bits in a register are, by themselves, neither an unsigned nor a signed number. The sequence 1001 0110b means 150 if read as unsigned, and −106 if read as signed two's complement. The processor does the addition on the bits only and produces the same result for both readings. It is the programmer who knows which reading applies. The processor takes both readings into account and keeps two separate flags: CF says whether the result is correct in the unsigned reading, OF in the signed reading.

Definition 6.1

The **carry flag** (CF) shows that, when the operation is done with the unsigned reading, the result does not fit the size of the destination. CF = 1 if in byte addition the true sum exceeds 255, or in subtraction the result falls below 0.

The **overflow flag** (OF) shows that, when the operation is done with the signed two's complement reading, the result does not fit the range. For a byte the range is −128…127 and for a word it is −32,768…32,767. OF = 1 if the sum of two numbers of the same sign comes out with the opposite sign.

Table 6.2 puts five byte additions side by side with the two readings. In every row the result written to AL is one; the only thing that changes is how it is read.

AL + operand	Unsigned reading	Signed reading	AL	CF	OF
`FFh + 01h`	255 + 1 = 256	−1 + 1 = 0	00h	1	0
`64h + 32h`	100 + 50 = 150	100 + 50 = 150	96h	0	1
`C8h + 64h`	200 + 100 = 300	−56 + 100 = 44	2Ch	1	0
`80h + 80h`	128 + 128 = 256	−128 + (−128) = −256	00h	1	1
`05h + 0Ah`	5 + 10 = 15	5 + 10 = 15	0Fh	0	0

Table 6.2: The unsigned and signed readings of the same byte addition. CF says the unsigned result is wrong, OF the signed one.

In the first row the unsigned sum is 256 and does not fit in a byte. 256 − 256 = 0 stays in AL and the carry that comes out falls into CF. In the signed reading, however, −1 + 1 = 0 is entirely correct; OF = 0. This row shows that CF = 1 is not an error in signed arithmetic. The second row is the opposite. 150 fits in a byte as unsigned, CF = 0. But a signed byte can be at most 127; the sum of two positive numbers has come out 96h, and the sign bit of 96h is 1. In the signed reading this means −106 and is wrong: OF = 1.

In the third row the two readings again part ways. Unsigned 300 does not fit, CF = 1. In the signed reading, one of the numbers is negative (−56) and the other positive (100). The sum of two numbers with different signs is smaller than the larger of them in absolute value; so it can never fall outside the

range. The result 44 is correct and OF = 0. In the fourth row both readings are wrong: 256 does not fit in a byte, and −256 does not fit either. Both flags are 1 together. In the last row both readings are correct and both flags are 0.

The rule for OF can also be stated in terms of the carries at the most significant bit. Let us add the second row bit by bit: 0110 0100b + 0011 0010b. At bit 5, 1 + 1 = 10b; 0 is written and a carry of 1 passes to bit 6. At bit 6, 1 + 0 + 1 = 10b; 0 is written and the carry enters bit 7. At bit 7, 0 + 0 + 1 = 1 is written and no carry goes out. The carry into bit 7 is 1 and the carry out is 0. Because the two differ, OF = 1. In the fourth row, no carry enters bit 7 at all but 1 + 1 produces a carry out: carry in 0, carry out 1, again OF = 1. In the first row a carry enters bit 7 and a carry also goes out: because the two are the same, OF = 0.

Rule (overflow flag)

In addition, OF = (carry into the most significant bit) XOR (carry out of the most significant bit). In terms of signs this reads: if the sum of two numbers of the same sign comes out with the opposite sign, OF = 1; when two numbers of different signs are added, OF is never 1. In subtraction (A − B) the rule takes this form: if the difference of two numbers of different signs has a sign different from that of A, OF = 1.

Common mistake

Reading CF as "overflow" and looking at CF when working with signed numbers. In the addition −1 + 1, CF = 1, although the signed result is correct. A program that works with signed numbers looks at OF (and, in comparisons, at SF and OF); a program that works with unsigned numbers looks at CF. The processor does not know which one applies; the choice belongs to whoever writes the instructions.

6.1.3 Code trace: CF, OF and the other flags

The following short program shows how flags carry over from one instruction to the next. The program starts with 9Ch; 9Ch is 156 in the unsigned reading and −100 in the signed reading. Then come two separate additions, then adding a number to itself, INC and CMC.

```
MOV  AL, 9Ch     ; 156 or -100
ADD  AL, 70h     ; + 112
MOV  AL, 50h     ; 80
ADD  AL, 50h     ; + 80
ADD  AL, AL      ; A0h + A0h
MOV  AL, 0FFh
INC  AL          ; CF untouched
CMC              ; flip CF
RET
```

Instruction	Machine code	AL	CF	OF	SF	ZF	AF	PF
`MOV AL, 9Ch`	`B0 9C`	9Ch	0	0	0	0	0	0
`ADD AL, 70h`	`04 70`	0Ch	**1**	0	0	0	0	1
`MOV AL, 50h`	`B0 50`	50h	1	0	0	0	0	1
`ADD AL, 50h`	`04 50`	A0h	0	**1**	1	0	0	1
`ADD AL, AL`	`02 C0`	40h	**1**	**1**	0	0	0	0
`MOV AL, 0FFh`	`B0 FF`	FFh	1	1	0	0	0	0
`INC AL`	`FE C0`	00h	1	0	0	**1**	1	1
`CMC`	`F5`	00h	**0**	0	0	1	1	1

Table 6.3: Trace of CF, OF and the other flags. Each row is the state after the instruction executes; at the start all status flags are 0.

In the first addition 156 + 112 = 268. 268 does not fit in a byte; 268 − 256 = 12 = 0Ch stays in AL and CF = 1. In the signed reading −100 + 112 = 12 is correct; because two numbers of different signs are added, OF cannot be 1 anyway. 0Ch = 0000 1100b has two 1s, so PF = 1. Since the low nibbles C + 0 = C, no carry comes out of bit 3: AF = 0.

The MOV in the third row changes no flag. This is a rule that is often forgotten: data movement instructions such as MOV, LEA and XCHG do not pass through the ALU, and the flags stay as whatever the previous arithmetic instruction left. CF = 1 and PF = 1 stay in place.

In the fourth row 80 + 80 = 160 is correct in the unsigned reading; CF = 0. But the sum of two positive numbers has come out A0h = 1010 0000b and the sign bit is 1. In the signed reading this is −96 and wrong: OF = 1, SF = 1. In the fifth row A0h is added to itself. Unsigned 160 + 160 = 320 does not fit, CF = 1. Signed −96 + (−96) = −192 does not fit either, OF = 1. The 40h left in AL is a wrong result in both readings.

The `INC AL` in the seventh row adds 1 to FFh and AL = 00h. Because the result is zero, ZF = 1, and because F + 1 in the low nibble gives a carry, AF = 1. If the same operation were done with `ADD AL, 1`, a carry would come out and CF = 1. INC, however, never touches CF; the CF = 1 in the table is left over from the addition in the fifth row. The `CMC` (complement carry) in the last row only inverts CF. In the same family `STC` sets CF to 1 and `CLC` sets CF to 0.

6.1.4 The flags of ADD step by step

The most reliable way to compute flags by hand is to do the addition on paper in binary. The instructions `MOV AL, 64h` and `ADD AL, 32h` add 100 and 50. Figure 6.2 shows the two operands, the result, and from which bit each flag is read.

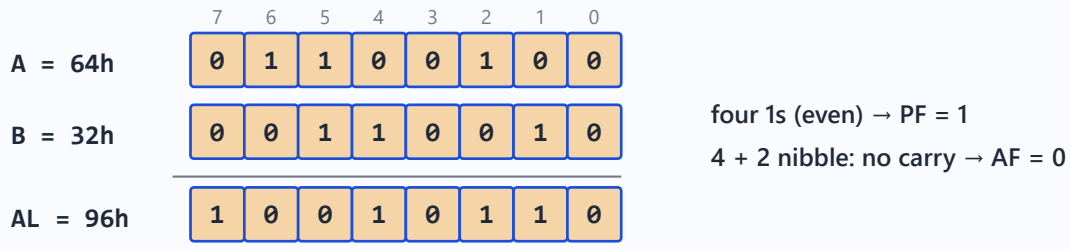


Figure 6.2: 64h + 32h = 96h. The bits are added from right to left; the flags are read from the bits of the result and from the carries.

1. **Result.** 0110 0100b + 0011 0010b = 1001 0110b = 96h. In the unsigned reading 150 is correct; in the signed reading −106 is wrong.
2. **CF.** No carry came out of the leftmost bit; 150 ≤ 255. CF = 0.
3. **OF.** The carry into bit 7 is 1, the carry out is 0. The sum of two positive numbers looked negative. OF = 1.
4. **SF and ZF.** Bit 7 of the result is 1, SF = 1. The result is not zero, ZF = 0.
5. **PF.** 1001 0110b has four 1s; four is even, PF = 1.
6. **AF.** The low nibbles are 4 + 2 = 6; no carry passes from bit 3 to bit 4, AF = 0.

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0. When this calculation is run in emu8086, the same values are seen in the flags window. The flag calculator in the slides sets up the same table for other numbers and other instructions.

Example 6.1

What are the flags after `MOV AL, 0C0h` and `ADD AL, 0C0h` ?

C0h = 1100 0000b; 192 unsigned, −64 signed. The sum is 1 1000 0000b = 180h. 80h is written to AL and the carry that comes out falls into CF: CF = 1. The carry into bit 7 is 1 (from the 1 + 1 at bit 6) and the carry out is also 1; OF = 0. Indeed −64 + (−64) = −128 fits a signed byte exactly. SF = 1, ZF = 0. 80h has a single 1 bit: PF = 0. Since the low nibbles are 0 + 0, AF = 0.

6.2 Instruction groups and operand rules

6.2.1 Three groups

The arithmetic and logic instructions of the 8086 fall into three groups according to how they take their operands. This division eases memorization: instructions in the same group accept the same operand forms and are encoded similarly in machine code. Figure 6.3 shows the operand layout of the three groups side by side.

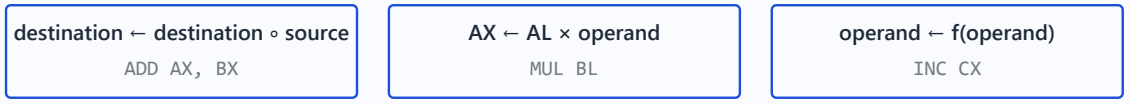


Figure 6.3: The operand forms of the three groups. In group 1 the result is written to the destination; in group 2 the implicit AL, AX and DX are used; in group 3 the single operand is both source and destination.

- **Group 1, two-operand instructions:** ADD, SUB, CMP, AND, TEST, OR, XOR. ADC (add with carry) and SBB (subtract with borrow) are also added to this list. The form is `instruction destination, source`; the operation is done between the destination and the source and the result is written to the destination.
- **Group 2, multiplication and division:** MUL, IMUL, DIV, IDIV. A single operand is written in the instruction. The other half of the operation and the place where the result goes are fixed: AL, AX or DX:AX. These are called **implicit operands**.
- **Group 3, single-operand instructions:** INC, DEC, NOT, NEG. The operand is both read and written: `INC CX` reads CX, adds 1 and writes the result back to CX.

Table 6.4 summarizes what each instruction does in one line. Shifts and rotates also take a single operand, but because they give the shift count as a second operand they are treated as a separate family.

Group	Instruction	What it does
1	`ADD`	adds two values, writes the result to the destination
	`SUB`	subtracts the source from the destination, writes the result to the destination
	`CMP`	subtracts the source from the destination, does not store the result; only sets the flags
	`AND`	bitwise AND
	`TEST`	bitwise AND; does not store the result
	`OR`	bitwise OR
	`XOR`	bitwise exclusive OR

Group	Instruction	What it does
2	`MUL`	unsigned multiplication
	`IMUL`	signed multiplication
	`DIV`	unsigned division (quotient and remainder)
	`IDIV`	signed division (quotient and remainder)
3	`INC`	increments by 1
	`DEC`	decrements by 1
	`NOT`	inverts all bits (ones' complement)
	`NEG`	negates (two's complement)

Table 6.4: The meaning of the instructions in the three groups.

6.2.2 Operands of two-operand instructions

All the instructions in group 1 accept the same five operand combinations. These are given with examples in Table 6.5. REG is one of the 8- or 16-bit general-purpose registers: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, SI, DI, BP, SP. Memory is a memory operand (`[BX]` , `[BX+SI+7]` , a variable name). Immediate is a value written inside the instruction (5, −24, 3Fh, 10001101b).

Form	Example	Machine code
REG, memory	`ADD AX, [BX]`	`03 07`
memory, REG	`SUB var1, CL`	`28 0E` + address
REG, REG	`XOR AX, AX`	`33 C0`
memory, immediate	`AND BYTE PTR [SI], 0Fh`	`80 24 0F`
REG, immediate	`CMP AL, 'A'`	`3C 41`

Table 6.5: The valid operand combinations of ADD, SUB, CMP, AND, TEST, OR and XOR.

The rules resemble the rules of MOV and come from the same hardware constraint. Both operands cannot be in memory: `ADD [BX], [SI]` is invalid, because the ModR/M byte of the instruction can describe only a single memory address. The two operands must be the same size: `ADD AL, BX` tries to add a 16-bit number to an 8-bit register and the assembler rejects it. The destination cannot be a constant: `ADD 5, AL` could not know where to write the result. Segment registers cannot be arithmetic operands: `ADD DS, AX` is invalid; DS is changed only with MOV, PUSH and POP.

When a memory operand is used with a constant, the size cannot be told from either. In the form `AND [SI], 0Fh` , 0Fh could be a byte or a word; so `BYTE PTR` or `WORD PTR` must be specified. If one of the operands is a register or a variable of known type, the size is taken from it.

Common mistake

Trying to add two variables directly: `ADD total, count`. Both operands are in memory and this instruction does not exist. One of them is first moved into a register: `MOV AX, count` and `ADD total, AX`.

6.2.3 Where is the result written, and how are the flags affected?

After a two-operand operation the result is always stored in the first operand, that is, the destination. The `SUB AX, BX` instruction computes AX − BX and writes it to AX; BX does not change. CMP and TEST are the two exceptions to this rule. CMP does the subtraction SUB does but discards the result. TEST does the bit-by-bit AND that AND does but discards the result. Both only change the flags and so make testing possible without spoiling a register.

The effect on the flags depends on whether the operation is arithmetic or logic. Table 6.6 compares the two families.

Instruction	CF	ZF	SF	OF	PF	AF
`ADD, ADC, SUB, SBB, CMP`	✓	✓	✓	✓	✓	✓
`AND, OR, XOR, TEST`	0	✓	✓	0	✓	?

Table 6.6: The flag effect of the two-operand instructions. ✓ set according to the result, 0 always cleared, ? undefined.

Arithmetic instructions set all six status flags according to the result. Logic instructions have no notion of a carry: each bit is processed on its own and nothing passes from one bit to another. So CF and OF are always set to 0. ZF, SF and PF are set according to the result. The value of AF is left undefined in Intel's documents; a program must not rely on AF after a logic instruction. Undefined does not mean the flag is random; on a real processor it takes a definite value, but that value is promised in no document and may come out differently on another x86 processor.

6.2.4 Timing: time inside the processor and on the bus

The time an instruction takes depends closely on where its operands are. Two numbers in registers go straight into the ALU; `ADD AX, BX` takes 3 clock cycles on the 8086. If there is a memory operand, the EU first computes the effective address and then asks the BIU for a read bus cycle. A bus cycle is at least 4 clock cycles: the address is put on the bus, memory is read, and the data is taken into the chip. If the destination is in memory, another bus cycle is needed to write the result back. During this read-modify-write, the 8086's RD pin becomes active first and the WR pin later. Table 6.7 gives the times from Intel's 8086 manual; EA is the time to compute the effective address (5–12 cycles).

Form	ADD, ADC, SUB, SBB, AND, OR, XOR	CMP	TEST
reg, reg	3	3	3

Form	ADD, ADC, SUB, SBB, AND, OR, XOR	CMP	TEST
reg, mem	9 + EA	9 + EA	9 + EA
mem, reg	16 + EA	9 + EA	9 + EA
reg, imm	4	4	5
mem, imm	17 + EA	10 + EA	11 + EA
AL/AX, imm	4	4	4

Table 6.7: Clock cycle counts of the two-operand instructions on the 8086 (Intel manual). For a word access at an odd address, 4 cycles are added for each transfer.

The differences in the table come from bus cycles. `ADD [BX], AX` both reads and writes memory and takes 16 + EA cycles. `CMP [BX], AX` does the same subtraction but, because it does not write the result, only reads: 9 + EA cycles. On a 5 MHz 8086 a clock cycle is 200 ns; `ADD [BX], AX` (EA = 5) takes about 21 × 200 ns = 4.2 µs. The same addition between two registers takes 0.6 µs. This is the reason for keeping frequently used values in registers throughout a loop.

6.3 Addition, subtraction and comparison

6.3.1 Code trace: ADD, SUB, CMP

The following program does an addition, a subtraction and two comparisons; the last line completes a negative number to zero. The flag effect of each instruction is traced together with the bit-level detail of the operation.

```
mov al, 0F0h      ; 240
add al, 20h       ; 240 + 32 = 272
mov bl, 7
sub bl, 9         ; 7 - 9
cmp al, 10h       ; AL - 10h
cmp bl, 0
add bl, 2         ; FEh + 2
```

Instruction	Machine code	AL	BL	CF	ZF	SF	OF	PF	AF
`mov al, 0F0h`	`B0 F0`	F0h	00h	0	0	0	0	0	0
`add al, 20h`	`04 20`	10h	00h	**1**	0	0	0	0	0
`mov bl, 7`	`B3 07`	10h	07h	1	0	0	0	0	0
`sub bl, 9`	`80 EB 09`	10h	FEh	**1**	0	1	0	0	1
`cmp al, 10h`	`3C 10`	10h	FEh	0	**1**	0	0	1	0
`cmp bl, 0`	`80 FB 00`	10h	FEh	0	0	1	0	0	0

Instruction	Machine code	AL	BL	CF	ZF	SF	OF	PF	AF
`add bl, 2`	`80 C3 02`	10h	00h	**1**	**1**	0	0	1	1

Table 6.8: Trace of ADD, SUB and CMP (all status flags 0 at the start).

Line 2. F0h + 20h = 110h. The low eight bits of this nine-bit result, 10h, are written to AL; the 1 that spills out falls into CF. In the signed reading −16 + 32 = 16 is correct and OF = 0. 10h = 0001 0000b has a single 1, PF = 0. Since the low nibbles are 0 + 0, AF = 0. The machine code is `04 20`: 04 is the short accumulator form meaning "add an immediate byte to AL".

Line 4. 7 − 9 = −2, written in 8 bits in two's complement as FEh. Because a larger number was subtracted from a smaller one, a borrow was taken: CF = 1. The most significant bit of the result is 1, SF = 1. In the low nibbles too, 7 − 9 needs a borrow, AF = 1. In the signed reading 7 − 9 = −2 is correct and OF = 0. Because this instruction uses BL, not AL, the short form cannot be chosen; the general `80 /5` form takes three bytes.

Line 5. `CMP AL, 10h` performs the subtraction 10h − 10h = 0. The result is zero, ZF = 1. The result is not stored: AL is still 10h. This is the essence of an equality test: if two numbers are equal, their difference is zero and ZF = 1. If a `JE` (or `JZ`, the other name of the same instruction) follows immediately, the jump is taken. The CF = 1 and AF = 1 left from the previous line are also erased; CMP rewrites all the status flags.

Line 6. `CMP BL, 0` is FEh − 0. The result is FEh: not zero (ZF = 0), sign bit 1 (SF = 1), no borrow (CF = 0). These three flags carry two different readings. Read as unsigned, FEh = 254 > 0; CF = 0 says so. Read as signed, FEh = −2 < 0; SF ≠ OF says so.

Line 7. FEh + 2 = 100h. 00h stays in BL; CF = 1 and ZF = 1. In the signed reading −2 + 2 = 0 is correct, OF = 0. This line shows the difference between CF and OF once more: CF = 1 does not mean the signed calculation is wrong.

6.3.2 Comparing magnitudes with CMP

The real job of CMP is to find not whether two numbers are equal but which is larger. `CMP A, B` computes A − B. For unsigned numbers, if A < B the subtraction needs a borrow and CF = 1. For signed numbers the situation is subtler. If the difference did not overflow (OF = 0), the sign of the result is correct and if A < B then SF = 1. If the difference overflowed (OF = 1), the sign bit has flipped; this time A < B is shown by SF = 0. The two cases combine into a single rule: signed A < B ⇔ SF ≠ OF.

Instructions	CF	SF	OF	Unsigned	Signed
`AL = 05h, BL = FEh` ; `CMP AL, BL`	1	0	0	5 < 254	5 > −2 (SF = OF)
`AL = 7Fh, BL = 80h` ; `CMP AL, BL`	1	1	1	127 < 128	127 > −128 (SF = OF)
`AL = 80h, BL = 01h` ; `CMP AL, BL`	0	0	1	128 > 1	−128 < 1 (SF ≠ OF)

Table 6.9: Three CMP examples. The unsigned result is read from CF, the signed result from the comparison of SF and OF.

In the second row 7Fh − 80h = FFh: SF = 1. But signed 127 − (−128) = 255 does not fit in a byte, OF = 1. The sign bit is misleading here, and because SF = OF the correct decision is "greater". Conditional jumps offer these rules ready-made. For unsigned comparison `JB` / `JA` (below/above, which look at CF and ZF) are used, and for signed comparison `JL` / `JG` (less/greater, which look at SF, OF and ZF). Which jump is chosen after the same CMP determines how the numbers are interpreted.

6.3.3 32-bit addition with ADC

The 8086's ALU adds numbers of at most 16 bits. A number that does not fit in 16 bits, such as 131,071, is stored split into two words: DX holds the high word and AX the low word, and the pair is written DX:AX. To add two 32-bit numbers, the method of addition on paper is followed: first the digits on the right are added, and the carry is added to the digits on the left. Here a "digit" is a 16-bit word and the carry is carried in CF.

Definition 6.2

ADC (add with carry) performs destination ← destination + source + CF. **SBB** (subtract with borrow) performs destination ← destination − source − CF. Their operand rules and flag effect are the same as those of ADD and SUB.

```
; DX:AX = 0001FFFFh, CX:BX = 00020001h
mov ax, 0FFFFh
mov dx, 0001h
mov bx, 0001h
mov cx, 0002h
add ax, bx      ; low words
adc dx, cx      ; high + CF
```

Instruction	Machine code	DX	AX	CX	BX	CF	ZF
`mov ax, 0FFFFh`	`B8 FF FF`	0000h	FFFFh	0000h	0000h	0	0
`mov dx, 0001h`	`BA 01 00`	0001h	FFFFh	0000h	0000h	0	0
`mov bx, 0001h` · `mov cx, 0002h`	`BB 01 00` · `B9 02 00`	0001h	FFFFh	0002h	0001h	0	0
`add ax, bx`	`03 C3`	0001h	**0000h**	0002h	0001h	**1**	1
`adc dx, cx`	`13 D1`	**0004h**	0000h	0002h	0001h	0	0

Table 6.10: Trace of the addition 0001FFFFh + 00020001h.

The first number is DX:AX = 0001FFFFh = 131,071 and the second is CX:BX = 00020001h = 131,073. `ADD AX, BX` adds the low words: FFFFh + 0001h = 10000h. 0000h is written to AX and the 1 that comes out falls into CF. This CF is the carry to be carried to the high word. That ZF = 1 is misleading

here: only the low word is zero, the 32-bit sum is not zero. `ADC DX, CX` uses this carry: 0001h + 0002h + 1 = 0004h. The result is DX:AX = 00040000h = 262,144, which agrees with 131,071 + 131,073.

The method is independent of the number of words. If a 48-bit number is held in three words, the first words are added with ADD and the next two words with ADC. The carry is carried in CF along the chain. No instruction that changes CF must come between ADD and ADC. MOV, INC, DEC and LOOP do not touch CF; so even when a counter is incremented inside a loop, the carry is preserved. In contrast, an ADD or CMP coming in between erases the carry.

6.3.4 32-bit subtraction with SBB

The same arrangement holds in subtraction. The lowest word is subtracted with SUB; if the low word is not enough, a borrow is taken from the next word up and this borrow waits in CF. SBB subtracts, from the upper word, this borrow together with the source.

```
; DX:AX = 00040000h, CX:BX = 00010001h
mov ax, 0000h
mov dx, 0004h
mov bx, 0001h
mov cx, 0001h
sub ax, bx      ; low words
sbb dx, cx      ; high - CF
```

Instruction	Machine code	DX	AX	CX:BX	CF	SF
`mov ax, 0000h` · `mov dx, 0004h`	`B8 00 00` · `BA 04 00`	0004h	0000h	0000:0000h	0	0
`mov bx, 0001h` · `mov cx, 0001h`	`BB 01 00` · `B9 01 00`	0004h	0000h	0001:0001h	0	0
`sub ax, bx`	`2B C3`	0004h	**FFFFh**	0001:0001h	**1**	1
`sbb dx, cx`	`1B D1`	**0002h**	FFFFh	0001:0001h	0	0

Table 6.11: Trace of the subtraction 00040000h − 00010001h.

The first number is 262,144 and the one subtracted is 65,537. `SUB AX, BX` performs 0000h − 0001h. The low word is not enough; a borrow is taken, AX = FFFFh and CF = 1. `SBB DX, CX` pays the borrow back: 0004h − 0001h − 1 = 0002h. The result is DX:AX = 0002FFFFh = 196,607, which agrees with 262,144 − 65,537.

Example 6.2

How is 00012345h − 0000ABCDh done with SUB and SBB? What is CF after the SUB?

The first number is loaded as DX:AX = 0001:2345h and the second as CX:BX = 0000:ABCDh. `SUB AX, BX` : 2345h − ABCDh is negative; in 16 bits 12345h − ABCDh = 7778h remains and, because a borrow was taken, CF = 1. `SBB DX, CX` : 0001h − 0000h − 1 = 0000h. The result is

00007778h = 30,584 (74,565 − 43,981).

Common mistake

Using SUB (or ADD) for both words. Written this way, the borrow or carry from the low word is lost; the result comes out wrong by exactly 10000h. The rule: ADD/SUB on the lowest word, ADC/SBB on every following word.

6.4 Logical operations

6.4.1 Gates that work bit by bit

The logic instructions are the software counterparts of the gates known from digital design. Table 6.12 gives the truth tables of the four operations. AND gives 1 only if both inputs are 1. OR gives 1 if at least one input is 1. XOR gives 1 if the inputs differ. NOT has a single input and inverts it.

A	B	A AND B	A OR B	A XOR B	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 6.12: Truth tables of AND, OR, XOR and NOT.

The logic instructions of the 8086 apply these rules not to a number as a whole but to its bits one by one. The `AND AL, BL` instruction is really eight independent AND gates: bit 0 of AL is ANDed with bit 0 of BL, bit 1 with bit 1, and the result is written to the same bit of AL. No carry or borrow passes between neighboring bits. In the ALU this operation is done with the adder's carry chain disabled and the output of the right gate selected at each bit. That is why logic instructions are as fast as addition: 3 clock cycles between registers.

A consequence of processing bit by bit is that carry and overflow are meaningless. The 8086 states this explicitly: after AND, OR, XOR and TEST, CF = 0 and OF = 0. ZF, SF and PF are set according to the result, AF is undefined. NOT is an exception: it changes no flag.

6.4.2 Masking

The most common use of logic instructions is to change some bits of a number and preserve the others. For this a **mask** is given as the second operand. Each bit of the mask chooses what happens to the corresponding bit. This method is called **masking**.

- **Clearing bits with AND.** x AND 0 = 0, x AND 1 = x. Bits that are 0 in the mask are cleared, those that are 1 are preserved.
- **Setting bits with OR.** x OR 1 = 1, x OR 0 = x. Bits that are 1 in the mask become 1 for certain, those that are 0 are preserved.
- **Inverting bits with XOR.** x XOR 1 = NOT x, x XOR 0 = x. Bits that are 1 in the mask are inverted, those that are 0 are preserved.
- **Testing bits with TEST.** TEST does an AND but does not write the result. If the bit in the mask (or one of the bits) is 1 and the number also has a 1 there, the result is nonzero: ZF = 0. The number does not change.

In ASCII encoding, uppercase and lowercase letters differ only in bit 5: 'A' = 41h = 0100 0001b, 'a' = 61h = 0110 0001b. So the conversion of a letter is done with a single logic instruction. Figure 6.4 shows the four operations with the same letters.

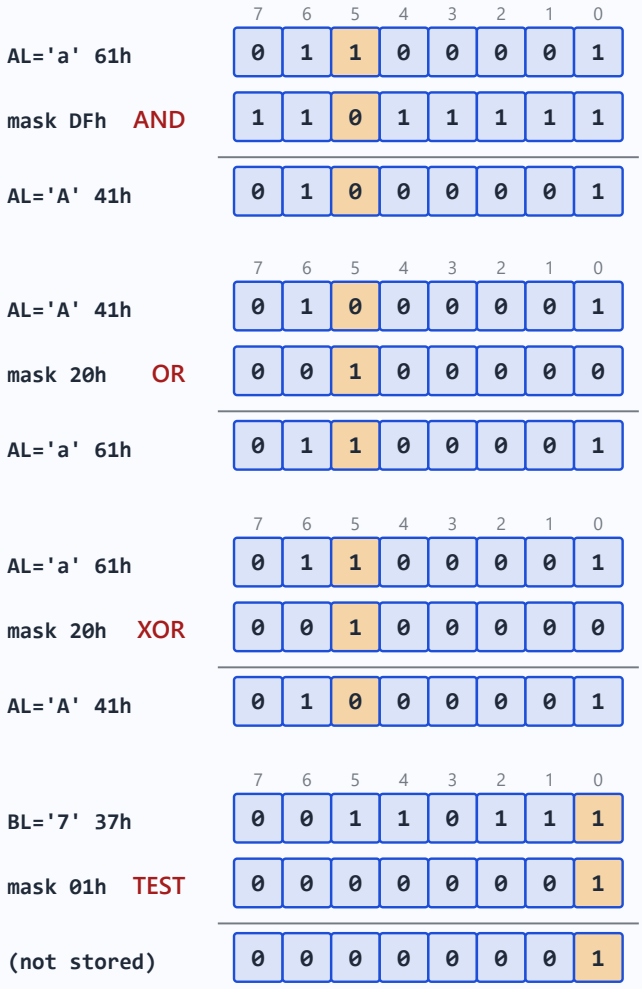


Figure 6.4: Masking. AND DFh clears bit 5 ('a' → 'A'); OR 20h sets bit 5 ('A' → 'a'); XOR 20h inverts bit 5; TEST 01h tests bit 0 and does not change BL.

The meaning of the mask values is plain when they are written in binary. DFh = 1101 1111b is a mask with only bit 5 equal to 0; with AND it clears bit 5. 20h = 0010 0000b is a mask with only bit 5 equal to 1; with OR it sets bit 5 and with XOR it inverts it. In the last figure BL = '7' = 37h = 0011 0111b is TESTed with the mask 01h. The result is 01h ≠ 0, so ZF = 0: bit 0 is one, that is, the number is odd.

6.4.3 Frequently used patterns

Some uses of logic instructions are so common that they have become almost fixed patterns in assembly programs. Table 6.13 collects the most frequently met.

Code	Effect	Why?
`XOR AX, AX`	AX = 0, ZF = 1, CF = 0	2 bytes (`33 C0`); `MOV AX, 0` is 3 bytes (`B8 00 00`) and does not touch the flags
`AND AL, 0Fh`	'7' (37h) → 7	converts an ASCII digit to a number; the high nibble is cleared
`OR AL, 30h`	7 → '7' (37h)	converts a number from 0 to 9 to an ASCII digit
`AND AL, 0DFh`	'a' → 'A'	clears bit 5: uppercase
`OR AL, 20h`	'A' → 'a'	sets bit 5: lowercase
`TEST AL, 1`	odd if ZF = 0	bit 0 is tested without changing AL
`TEST AL, 80h`	negative if ZF = 0	the sign bit is tested
`OR AX, AX`	AX unchanged	ZF and SF are set: is AX zero, is it negative?

Table 6.13: Frequently used patterns with logic instructions.

ASCII digits run from '0' = 30h to '9' = 39h. The high nibble is 3 for every digit, and the low nibble is the value of the digit. Because `AND AL, 0Fh` clears the high nibble, it converts a digit read from the keyboard to a number. `OR AL, 30h` does the reverse and prepares a character to be written on the screen. The same job can be done with `SUB AL, 30h` and `ADD AL, 30h`; the difference with the logic instructions is that they still give the right result if the input has already been converted.

`OR AX, AX` looks meaningless at first sight: ORing a number with itself does not change it. The purpose of the instruction is the flags. If AX is zero, ZF = 1; if its most significant bit is 1, SF = 1. The same job is done with `TEST AX, AX` and `CMP AX, 0`. OR and TEST take two bytes; CMP takes three because it also carries the constant.

6.4.4 Code trace: AND, OR, XOR, TEST

```
mov al, 'a'      ; 61h
and al, 0DFh     ; uppercase
or  al, 20h      ; lowercase
xor al, 20h      ; flip case
mov bl, '7'      ; 37h
and bl, 0Fh      ; number 7
test bl, 1       ; odd?
xor bl, bl       ; BL = 0
```

Instruction	Machine code	AL	BL	ZF	SF	PF	CF	OF
`mov al, 'a'`	`B0 61`	61h	00h	0	0	0	0	0
`and al, 0DFh`	`24 DF`	41h 'A'	00h	0	0	1	0	0
`or al, 20h`	`0C 20`	61h 'a'	00h	0	0	0	0	0
`xor al, 20h`	`34 20`	41h 'A'	00h	0	0	1	0	0
`mov bl, '7'`	`B3 37`	41h	37h	0	0	1	0	0
`and bl, 0Fh`	`80 E3 0F`	41h	07h	0	0	0	0	0
`test bl, 1`	`F6 C3 01`	41h	07h	**0**	0	0	0	0
`xor bl, bl`	`32 DB`	41h	00h	**1**	0	1	0	0

Table 6.14: Trace of the logic instructions. After logic instructions CF = OF = 0; AF is undefined (not shown in the table).

61h AND DFh = 41h = 'A'. 41h = 0100 0001b has two 1s, PF = 1. 41h OR 20h = 61h = 'a'; 0110 0001b has three 1s, PF = 0. 61h XOR 20h gives 41h again: bit 5 has been inverted. Applying XOR twice brings the number back to its old state; this property is used in simple encryption and bit-toggling jobs.

ASCII '7' = 37h is loaded into BL and `AND BL, 0Fh` turns it into the numeric 7. `TEST BL, 1` computes 07h AND 01h = 01h; the result is nonzero, ZF = 0 and the number is odd. BL stays 07h unchanged. On the last line BL is XORed with itself. Because no bit differs from itself, the result is 00h; ZF = 1. Zero 1 bits is an even count, PF = 1.

The machine code column of the table shows the short forms of instructions that work with AL. `AND AL, imm8` = 24, `OR AL, imm8` = 0C, `XOR AL, imm8` = 34 are two-byte instructions. The same operations with BL fall to the general `80` or `F6` form and take three bytes. This is a small but constant advantage that AL and AX have as the accumulator.

> **Common mistake**
>
> Using AND to test a bit and spoiling the number. `AND AL, 80h` looks at the sign bit but clears the other seven bits of AL. If the number will be used again, TEST should be written. The same relation exists between CMP and SUB.

6.5 Multiplication and division

6.5.1 MUL and IMUL

The product of two 8-bit numbers can be up to 16 bits, and that of two 16-bit numbers up to 32 bits: 255 × 255 = 65,025 does not fit in a byte, only in a word. The 8086 therefore always writes the result of a multiplication to a place twice the width of the operand. The other operand of the multiplication and the place of the result are fixed; only one operand is written in the instruction. Figure 6.5 shows the two sizes.

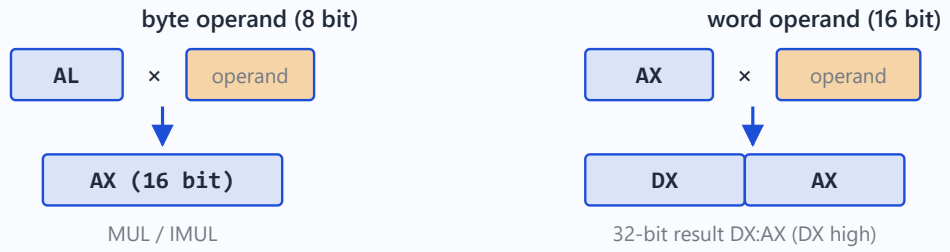


Figure 6.5: In byte multiplication AL × operand is written to AX, in word multiplication AX × operand to DX:AX.

Definition 6.3

MUL does unsigned and **IMUL** signed multiplication. If the operand is a byte, AX ← AL × operand; if the operand is a word, DX:AX ← AX × operand. The operand can be a register or a memory operand (`MUL BL` , `IMUL WORD PTR [SI]`). The 8086 has no immediate operand: `MUL 5` is invalid.

Whether the product is unsigned or signed changes the bits of the result; unlike addition, a single operation does not serve both readings. For AL = C8h and BL = 03h, `MUL BL` gives 200 × 3 = 600 = 0258h. With the same bits `IMUL BL` reads C8h as −56 and gives −56 × 3 = −168 = FF58h. The low bytes (58h) are the same, the high bytes differ. That is why the 8086 has two separate multiplication instructions.

After multiplication only CF and OF are meaningful, and the two are always equal. If the upper half of the result (AH or DX) is really needed, CF = OF = 1. For MUL "needed" means the upper half is nonzero. For IMUL it means the upper half differs from the sign extension of the lower half: if the result fits the lower half as a signed number, the upper half is only copies of the sign bit and carries no information. SF, ZF, AF and PF are undefined after multiplication.

AL, BL	Instruction	Calculation	AX	CF = OF	Why
C8h, 03h	`MUL BL`	200 × 3 = 600	0258h	1	AH = 02h ≠ 0

AL, BL	Instruction	Calculation	AX	CF = OF	Why
C8h, 03h	`IMUL BL`	−56 × 3 = −168	FF58h	1	−168 does not fit a signed byte
FEh, 05h	`IMUL BL`	−2 × 5 = −10	FFF6h	0	AH = FFh, the sign extension of AL
FEh, 05h	`MUL BL`	254 × 5 = 1270	04F6h	1	AH = 04h ≠ 0
06h, 09h	`MUL BL`	6 × 9 = 54	0036h	0	AH = 0

Table 6.15: Byte multiplication examples and CF/OF.

Multiplication is one of the slowest instructions of the 8086. There is no hardware multiplier in the ALU; the EU applies the shift-and-add method bit by bit in microcode. According to Intel's manual, `MUL` with a register operand takes 70–77 clock cycles for a byte and 118–133 for a word; `IMUL` takes 80–98 for a byte and 128–154 for a word. If there is a memory operand, a few more cycles are added with the EA time. The time is given as a range because the microcode loop passes through different paths depending on the bits of the multiplier. Since an addition takes only 3 cycles, turning multiplication by a constant into shifts and additions is often worthwhile.

6.5.2 DIV and IDIV

Division, as the inverse of multiplication, needs a **dividend** twice the width of the operand. If the divisor is a byte, the dividend is AX; if the divisor is a word, the dividend is DX:AX. The result is in two parts: the **quotient** and the **remainder**. Figure 6.6 shows word division.

Figure 6.6: In word division DX:AX is the dividend and the operand is the divisor; the quotient goes to AX and the remainder to DX.

Instruction	If the operand is a byte	If the operand is a word
`DIV` (unsigned)	AL ← AX / operand, AH ← remainder	AX ← DX:AX / operand, DX ← remainder
`IDIV` (signed)	AL ← AX / operand, AH ← remainder	AX ← DX:AX / operand, DX ← remainder

Table 6.16: The implicit operands of DIV and IDIV.

After division all the status flags are undefined; no information about the result of the division can be got from the flags. Division is slower than multiplication: `DIV` with a register takes 80–90 cycles for a byte and 144–162 for a word; `IDIV` takes 101–112 for a byte and 165–184 for a word.

IDIV truncates the quotient toward zero and leaves the remainder with the same sign as the dividend. For −23 / 5 the quotient is −4.6; truncated toward zero it becomes −4. The remainder is found from −23 = (−4) × 5 + remainder as −3. With this rule the equality dividend = quotient × divisor + remainder always holds and the absolute value of the remainder is smaller than the divisor.

Definition 6.4

A **divide error** occurs when the divisor is 0 or when the quotient does not fit in its destination (AL or AX). The processor does not complete the division; it raises interrupt number 0 (INT 0). The first entry of the interrupt vector table (0000:0000) points to the service routine of this interrupt. DOS prints the message "Divide overflow" in this routine and terminates the program.

Whether the quotient will fit is easy to test before the division. In byte division the quotient must not exceed 255; this holds only if AH < divisor. In word division the same rule takes the form DX < divisor. For AX = 1000h (4096) and BL = 2, AH = 10h ≥ 2: the quotient 2048 does not fit in AL and a divide error occurs. For DX:AX = FFFFFFFFh and BX = 1 too, although the divisor is not zero, DX = FFFFh ≥ 1 and the quotient does not fit. In IDIV the limit is slightly narrower: on the 8086 the byte quotient must be in the range −127…127 and the word quotient in −32,767…32,767; −128 and −32,768 also cause a divide error.

The return address that the 8086 pushes on the stack at a divide error carries one more detail. The IP pushed when the INT 0 service is called points to the instruction *after* the division instruction. On the 80286 and later processors it points to the division instruction itself, so that the instruction can be retried after the error is removed. This difference concerns only programs that write their own INT 0 routine.

6.5.3 Multiplication and division step by step

Table 6.17 collects the step-by-step multiplication and division examples of the slides in one table. Each row is an independent experiment; the registers start with the values given at the start of the row.

Start	Instruction	Result	Explanation
AL = C8h, BL = 03h	`MUL BL`	AX = 0258h, CF = OF = 1	200 × 3 = 600; did not fit in AL
AL = C8h, BL = 03h	`IMUL BL`	AX = FF58h, CF = OF = 1	−56 × 3 = −168; does not fit a signed byte
AL = FEh, BL = 05h	`IMUL BL`	AX = FFF6h, CF = OF = 0	−2 × 5 = −10; AH is only the sign extension
AX = 0017h, BL = 05h	`DIV BL`	AL = 04h, AH = 03h (AX = 0304h)	23 = 4 × 5 + 3
AX = FFE9h, BL = 05h	`IDIV BL`	AL = FCh, AH = FDh (AX = FDFCh)	−23 = (−4) × 5 + (−3)
AX = 1000h, BL = 02h	`DIV BL`	INT 0	the quotient 2048 does not fit in AL
DX:AX = FFFFFFFFh, BX = 0001h	`DIV BX`	INT 0	the quotient FFFFFFFFh does not fit in AX

Table 6.17: Byte and word multiplication and division examples.

In the fourth row AX = 0017h = 23 and BL = 5. The quotient 4 is written to AL and the remainder 3 to AH, and the new AX is 0304h. This value should not be read as a number; the two halves of AX now carry two separate results. In the fifth row the negative of the same number is divided. −23 = FFE9h. The quotient is −4 = FCh and the remainder −3 = FDh; check: −4 × 5 + (−3) = −23.

6.5.4 Preparing for division: CBW and CWD

Because the dividend is always twice the width of the divisor, the dividend must be extended before the division. The number to be divided is often only in AL or in AX; the programmer decides what the upper half (AH or DX) will be.

- **Before an unsigned division** the upper half is zeroed: `MOV AH, 0` or `XOR DX, DX`. The value of the number does not change; zeros are only added in front of it.
- **Before a signed division** the sign is extended: the sign bit is copied into all the bits of the upper half. The 8086 has two instructions for this.

Definition 6.5

Sign extension is moving a signed number to a wider place while preserving its value: the most significant bit is copied into all the newly added bits. **CBW** (convert byte to word) extends AL into AX: if bit 7 of AL is 0, AH = 00h; if 1, AH = FFh. **CWD** (convert word to doubleword) extends AX into DX:AX: if bit 15 of AX is 0, DX = 0000h; if 1, DX = FFFFh. Neither instruction takes an operand and neither changes any flag.

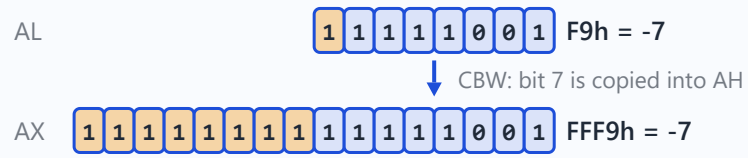


Figure 6.7: CBW: AL = F9h (−7); because the sign bit is 1, AH = FFh; AX = FFF9h is again −7.

F9h as a signed byte is −7. The same −7 written as a word is FFF9h: 65,536 − 7 = 65,529 = FFF9h. CBW does this conversion in a single instruction and takes 2 clock cycles. CWD takes 5. Both use implicit operands and are one-byte instructions (`98` and `99`).

```
; unsigned: 1000 / 7
mov ax, 1000
xor dx, dx      ; DX:AX = 1000
mov bx, 7
div bx          ; AX = 142, DX = 6
; signed: -1000 / 7
mov ax, -1000
cwd             ; DX = FFFFh
idiv bx         ; AX = -142, DX = -6
```

In the first part 1000 = 03E8h is loaded into AX and DX is zeroed; the dividend becomes DX:AX = 000003E8h. 1000 / 7 = 142, remainder 6: AX = 008Eh, DX = 0006h. In the second part −1000 = FC18h is loaded. Because bit 15 is one, CWD puts FFFFh into DX and DX:AX = FFFFFC18h, which is −1000 in 32 bits. IDIV gives the quotient −142 = FF72h and the remainder −6 = FFFAh.

Common mistake

Forgetting DX before a word division. A number left in DX from an earlier operation enlarges the dividend. If `XOR DX, DX` is written instead of CWD for signed −1000, the dividend becomes 0000FC18h = 64,536. IDIV then gives a meaningless result such as 64,536 / 7 = 9219 (2403h). If a large value is left in DX, the quotient does not fit and the program stops with a divide error.

6.5.5 Code trace: DIV, CWD and IDIV

The following program does the same division job three times: first unsigned, then signed, and last forgetting to clear DX.

```
MOV  AX, 2000
XOR  DX, DX      ; DX:AX = 2000
MOV  BX, 9
DIV  BX          ; unsigned div
MOV  AX, -2000
CWD              ; sign-extend
IDIV BX          ; signed div
MOV  AX, 2000    ; DX not cleared!
DIV  BX
RET
```

Instruction	Machine code	DX	AX	BX	DX:AX (decimal)
`MOV AX, 2000`	`B8 D0 07`	0000h	07D0h	0000h	2000
`XOR DX, DX`	`33 D2`	0000h	07D0h	0000h	2000
`MOV BX, 9`	`BB 09 00`	0000h	07D0h	0009h	2000

Instruction	Machine code	DX	AX	BX	DX:AX (decimal)
`DIV BX`	`F7 F3`	0002h	00DEh	0009h	quotient 222, remainder 2
`MOV AX, -2000`	`B8 30 F8`	0002h	F830h	0009h	0002F830h = 194,608 (wrong dividend)
`CWD`	`99`	FFFFh	F830h	0009h	FFFFF830h = −2000
`IDIV BX`	`F7 FB`	FFFEh	FF22h	0009h	quotient −222, remainder −2
`MOV AX, 2000`	`B8 D0 07`	FFFEh	07D0h	0009h	FFFE07D0h ≈ 4.29 billion
`DIV BX`	`F7 F3`	divide error: DX = FFFEh ≥ 9, the quotient does not fit in AX → INT 0			

Table 6.18: Trace of DIV, CWD and IDIV. The flags are undefined after a division.

In the first division 2000 / 9 = 222, remainder 2: AX = 00DEh, DX = 0002h. The fifth row shows why CWD is needed. −2000 = F830h has been loaded into AX but DX still holds the remainder of the previous division (2). IDIV, which reads DX:AX as a 32-bit number, would divide 194,608 in this state. Because the sign bit of AX is 1, CWD writes FFFFh into DX and makes the dividend correctly −2000. For −2000 / 9, IDIV gives the quotient −222 = FF22h and the remainder −2 = FFFEh. The remainder, like the dividend, is negative.

The last two rows act out the classic mistake. 2000 has been loaded into AX, but DX still holds the remainder of IDIV (FFFEh). The unsigned dividend FFFE07D0h is about 4.29 billion. Dividing this by 9 gives a quotient that does not fit in 16 bits; the rule says so in advance: DX = FFFEh ≥ 9. The processor raises INT 0 and the program stops. The fix is to write `XOR DX, DX` before the division.

6.5.6 Signed overflow: JO and CBW

The sum of two signed bytes may not fit in a byte. In that case the correct result is found by first extending the two numbers to words and adding them in 16 bits. The program below first tries the byte addition; if overflow occurs, it switches to the word addition. There are two jump instructions in the program. `JO label` (jump if overflow) jumps to the given label if OF = 1 and otherwise goes on to the next instruction. `JMP label` jumps unconditionally. Its opposite, `JNO`, jumps if OF = 0.

```
mov al, -92      ; A4h
mov bl, -92
add al, bl       ; 8-bit add
jo  widen        ; if OF = 1
jmp done
widen:
mov al, -92      ; first number
cbw              ; AL -> AX
mov dx, ax
mov al, bl       ; second number
cbw
add ax, dx       ; 16-bit add
done:
ret
```

Instruction	AX	DX	CF	OF	SF	Explanation
`mov al, -92` · `mov bl, -92`	00A4h	0000h	0	0	0	−92 = 256 − 92 = 164 = A4h
`add al, bl`	0048h	0000h	1	**1**	0	A4h + A4h = 148h; AL = 48h = +72
`jo widen`	0048h	0000h	1	1	0	OF = 1: the jump is taken
`mov al, -92` · `cbw`	FFA4h	0000h	1	1	0	the MSB of AL is 1 → AH = FFh
`mov dx, ax` · `mov al, bl` · `cbw`	FFA4h	FFA4h	1	1	0	both numbers are −92 as words
`add ax, dx`	**FF48h**	FFA4h	1	**0**	1	FFA4h + FFA4h = 1FF48h; AX = −184

Table 6.19: Resolving signed overflow with JO and CBW.

In the byte addition the sum of two negative numbers has come out positive: AL = 48h = +72. The correct result is −184 and is not in the range −128…127. Because OF = 1, JO jumps to the widen label. CF = 1 as well, but CF has no meaning in the signed reading. In the extension part the first number is loaded into AL again and extended to FFA4h with CBW. It is saved in DX. The second number is taken from BL and extended too. The 16-bit sum is FFA4h + FFA4h = 1FF48h; the 1 that comes out falls into CF, and FF48h = −184 stays in AX. The result fits the word, OF = 0.

6.5.7 Summing a signed byte array with CBW

The same idea can be applied to all the elements of an array. The program below accumulates four signed bytes in a 16-bit sum. The program uses a loop, and the loop is built with the `LOOP` instruction.

Definition 6.6

The **LOOP label** instruction does two things: it decrements CX by 1 and then, if CX is nonzero, jumps to the label; if CX has become zero it goes on to the next instruction. Before the loop, CX is loaded with the number of iterations. A loop body that starts with CX = 4 runs four times. LOOP changes no flag; the decrement of CX is not reflected in the flags either.

```
org 100h
jmp start
arr db 100, -30, 90, -7
total dw 0
start:
lea si, arr
mov cx, 4
xor bx, bx      ; BX = 16-bit sum
again:
mov al, [si]
cbw             ; AL -> AX (signed)
add bx, ax
inc si
loop again
mov total, bx  ; 153 = 0099h
ret
```

The `JMP start` at the beginning of the program jumps over the data definitions. In a .COM program code and data are in the same segment and the processor starts at 0100h; without the jump the bytes of the array would be taken for instructions. `LEA SI, arr` loads the offset of the array (0102h) into SI. emu8086 turns this line into the same three-byte code as `MOV SI, 0102h` (`BE 02 01`). BX holds the sum and is zeroed.

Iteration	AL	AX after CBW	BX (sum)	CX (at the start of the iteration)	Explanation
1	64h	0064h	0064h = 100	4	+100, MSB = 0 → AH = 00h
2	E2h	FFE2h	0046h = 70	3	−30, MSB = 1 → AH = FFh
3	5Ah	005Ah	00A0h = 160	2	+90
4	F9h	FFF9h	0099h = 153	1	−7; CX = 0 after LOOP

Table 6.20: The signed byte sum with CBW. The state after `cbw` and `add bx, ax` in each iteration.

In the second iteration FFE2h is added to BX = 0064h: 0064h + FFE2h = 1 0046h. The carry that comes out is discarded and BX = 0046h = 70 remains; 100 + (−30) = 70 is correct. This is a well-known property of two's complement: adding a negative number gives the same result as subtracting when the carry is discarded. In the third iteration the sum becomes 160. If the addition had been done in bytes, 70 + 90 = 160 > 127 would have given AL = A0h (−96) and OF = 1. In a word 160 fits easily. After the loop the total = 0099h = 153 is written to memory.

Common mistake

Using `MOV AH, 0` when extending signed bytes. This extends the number as unsigned: −30 = E2h is added as 00E2h = 226; −7 is counted as 249 too, and the total comes out 665 instead of 153. MOV AH, 0 is used for unsigned data and CBW for signed data.

6.5.8 A 32-bit total with CWD and ADC

When the elements are words the total may not fit in 16 bits either. The solution is to extend each element to 32 bits with CWD and accumulate in a 32-bit total with an ADD-ADC pair.

```
org 100h
jmp start
arr dw 30000, 25000, -4000
total dw 0, 0        ; 32-bit total
start:
lea si, arr
mov cx, 3
again:
mov ax, [si]
cwd                  ; AX -> DX:AX
add [total], ax      ; low word
adc [total+2], dx    ; high + CF
add si, 2
loop again
ret
```

The total is two words: the low word is at `total` and the high word at `total+2`. In a word array every element takes two bytes, so SI increases by 2 in each iteration.

Iteration	AX	CWD: DX	Total after ADD, CF	total+2 after ADC	32-bit total
1	7530h (30.000)	0000h	7530h, CF = 0	0000h	00007530h = 30.000
2	61A8h (25.000)	0000h	D6D8h, CF = 0	0000h	0000D6D8h = 55.000
3	F060h (−4000)	FFFFh	C738h, CF = 1	0000h	0000C738h = 51.000

Table 6.21: Trace of the 32-bit total with CWD and ADC (after ADD and ADC in each iteration).

In the second iteration 7530h + 61A8h = D6D8h. In the 16-bit signed reading this number is negative and OF = 1; but because the total is kept in 32 bits there is no problem: the high word is 0000h and 55,000 is correct. In the third iteration the sign bit of −4000 = F060h is 1; CWD puts FFFFh into DX and DX:AX = FFFFF060h, which is −4000 in 32 bits. In the low word D6D8h + F060h = 1C738h; the total becomes C738h and CF = 1. In the high word 0000h + FFFFh + 1 = 10000h; total+2 = 0000h remains. The result is 0000C738h = 51,000 = 30,000 + 25,000 − 4000.

The `add si, 2` in the program changes CF, but this is not a problem, because the carry of the next iteration starts afresh with ADD. The carry must be preserved only between an ADD and the ADC right after it. These two instructions use a memory operand: each reads the word, adds, and writes it back. On the 8086 each takes 16 + EA clock cycles and needs two bus cycles (a read and a write).

6.5.9 32 × 32-bit multiplication

The 8086 multiplies at most 16 × 16 bits. The product of two 32-bit numbers is done with the method of multiplication on paper, using 16-bit "digits". If A = A1:A0 and B = B1:B0 are split into halves, then

> $A \times B = A1 \cdot B1 \cdot 2^{32} + (A1 \cdot B0 + A0 \cdot B1) \cdot 2^{16} + A0 \cdot B0$

holds. Each of the four partial products is computed with one MUL and its 32-bit result lands in DX:AX. The partial products are added to the four words of the result (z3:z2:z1:z0, 64 bits in total) according to their positions. A product multiplied by 2^{16} is shifted left by one word and is added to z2:z1; the one multiplied by 2^{32} is shifted left by two words and is added to z3:z2.

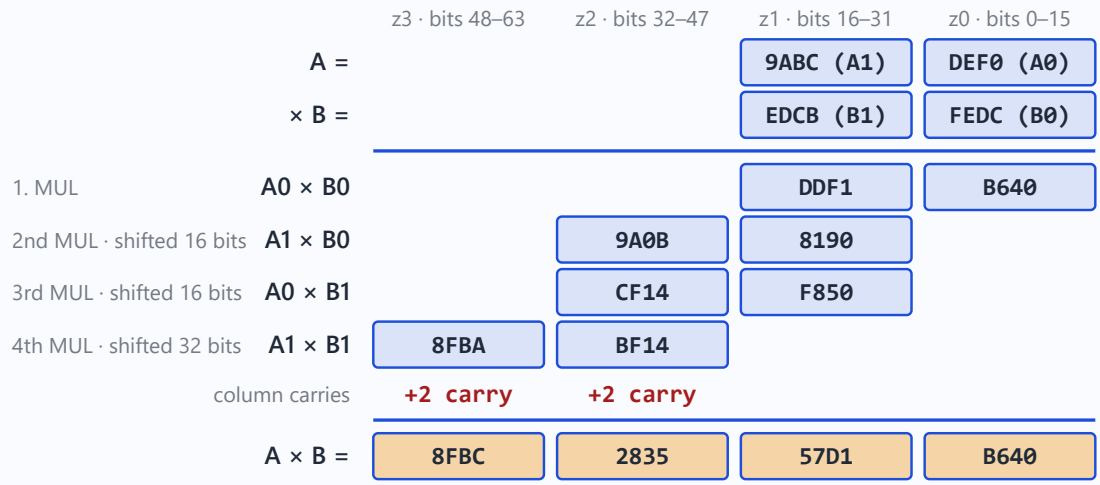


Figure 6.8: The product of A = 9ABCDEF0h and B = EDCBFEDCh. The four partial products are added by columns; the column carries pass to the next word up.

In the example A = 9ABCDEF0h and B = EDCBFEDCh. The partial products are A0 × B0 = DDF1B640h, A1 × B0 = 9A0B8190h, A0 × B1 = CF14F850h and A1 × B1 = 8FBABF14h. In the z0 column there is only B640h. The sum of the z1 column is DDF1h + 8190h + F850h = 257D1h: 57D1h is written and a carry of 2 passes to z2. The z2 column is 9A0Bh + CF14h + BF14h + 2 = 22835h: 2835h is written and a carry of 2 passes to z3. z3 = 8FBAh + 2 = 8FBCh. The result is 8FBC 2835 57D1 B640h. Decimal check: 2,596,069,104 × 3,989,569,244 = 10,357,197,452,617,037,376.

A column carry can be 2, but an ADD gives at most a carry of 1. The code therefore does the additions one at a time and carries the carry of each ADD to the next word up at once with an ADC.

```
org 100h
mov ax, [x]       ; A0
mul [y]           ; A0*B0
mov [z], ax
mov [z+2], dx
mov ax, [x+2]     ; A1
mul [y]           ; A1*B0
add [z+2], ax
adc [z+4], dx
mov ax, [x]       ; A0
mul [y+2]         ; A0*B1
add [z+2], ax
adc [z+4], dx
adc [z+6], 0
mov ax, [x+2]     ; A1
mul [y+2]         ; A1*B1
add [z+4], ax
adc [z+6], dx
ret
x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
z dw 4 dup(0)       ; 64-bit result
```

Instruction(s)	DX:AX	CF	z3	z2	z1	z0
`mov ax,[x]` · `mul [y]`	DDF1:B640h	1	0000h	0000h	0000h	0000h
`mov [z],ax` · `mov [z+2],dx`	DDF1:B640h	1	0000h	0000h	DDF1h	B640h
`mov ax,[x+2]` · `mul [y]`	9A0B:8190h	1	0000h	0000h	DDF1h	B640h
`add [z+2],ax`	9A0B:8190h	1	0000h	0000h	5F81h	B640h
`adc [z+4],dx`	9A0B:8190h	0	0000h	9A0Ch	5F81h	B640h
`mov ax,[x]` · `mul [y+2]`	CF14:F850h	1	0000h	9A0Ch	5F81h	B640h
`add [z+2],ax`	CF14:F850h	1	0000h	9A0Ch	57D1h	B640h
`adc [z+4],dx`	CF14:F850h	1	0000h	6921h	57D1h	B640h
`adc [z+6],0`	CF14:F850h	0	0001h	6921h	57D1h	B640h
`mov ax,[x+2]` · `mul [y+2]`	8FBA:BF14h	1	0001h	6921h	57D1h	B640h
`add [z+4],ax`	8FBA:BF14h	1	0001h	2835h	57D1h	B640h
`adc [z+6],dx`	8FBA:BF14h	0	**8FBCh**	**2835h**	**57D1h**	**B640h**

Table 6.22: Trace of the 32 × 32-bit multiplication. z3…z0 are the four words of the result (0 at the start).

After the first MUL, DX ≠ 0, so CF = 1; this CF only says "the result did not fit in 16 bits" and has nothing to do with the following additions. In the second partial product z1 + AX = DDF1h + 8190h = 1 5F81h; z1 becomes 5F81h and CF = 1. ADC makes z2 = 0 + 9A0Bh + 1 = 9A0Ch. Since z2 was zero before, no new carry arises here.

The third partial product is more interesting. z1 = 5F81h + F850h = 1 57D1h; the carry goes to z2. z2 = 9A0Ch + CF14h + 1 = 1 6921h; this time the carry must spill into z3. The `adc [z+6], 0` instruction does this job: it adds no number to z3, only the carry, and z3 = 0001h. If this line is forgotten, the result comes out short by exactly 2^{48}. The fourth partial product is added to z3:z2: z2 = 6921h + BF14h = 1 2835h, z3 = 0001h + 8FBAh + 1 = 8FBCh. In memory z holds, in order, the bytes `40 B6 D1 57 35 28 BC 8F`; in little-endian order this is 8FBC283557D1B640h.

Each of the four MULs takes 124–139 cycles plus EA; the whole program takes over 600 cycles on the 8086. On the 80386 the same job is done with a single 32-bit MUL. This is the price of multiword arithmetic on a 16-bit processor.

6.6 Single-operand instructions

6.6.1 INC, DEC, NOT and NEG

The four instructions in the third group take a single operand. The operand can be a register (AX, BX, …, SP or the 8-bit halves) or a memory operand; it cannot be a constant. With a memory operand the size must be specified: `INC BYTE PTR [BX]`. The instruction reads the operand, changes it, and writes it back to the same place.

- **INC** adds 1 to the operand and **DEC** subtracts 1. They set ZF, SF, OF, PF and AF according to the result, but **do not touch CF**.
- **NOT** inverts every bit. The result is the operand's **ones' complement**. It changes no flag.
- **NEG** changes the sign of the operand: it computes 0 − operand. The result is the **two's complement**: the bits are inverted and 1 is added. It sets all six status flags; CF is 1 if the operand is nonzero and 0 if it is zero.

Instruction	CF	ZF	SF	OF	PF	AF
`INC, DEC`	—	✓	✓	✓	✓	✓
`NOT`	—	—	—	—	—	—
`NEG`	✓ (1 if operand ≠ 0)	✓	✓	✓	✓	✓

Table 6.23: The flag effect of the single-operand instructions.
✓ set according to the result, — unchanged.

That INC and DEC preserve CF is a design decision. In multibyte or multiword addition, the loop does an ADC on each iteration and carries the carry to the next iteration. The same loop advances the pointer with INC and decrements the counter with DEC. If these two instructions erased CF, the carry would be lost. The 8086 designers therefore left INC and DEC independent of CF.

INC and DEC are also very short instructions. `INC SI`, which increments a 16-bit register, is a single byte (`46`) and takes 2 clock cycles; `ADD SI, 1` is three bytes and 4 cycles. On an 8-bit register INC is two bytes (such as `FE C0`) and 3 cycles, and in memory 15 + EA cycles. NEG and NOT take 3 cycles on

a register and 16 + EA in memory.

6.6.2 NEG: two's complement step by step

Two's complement is the standard way of representing negative numbers in binary. The negative of a number is found by inverting its bits and adding 1. NEG does these two steps in a single instruction. Figure 6.9 shows the negative of 5 and of −128 step by step.

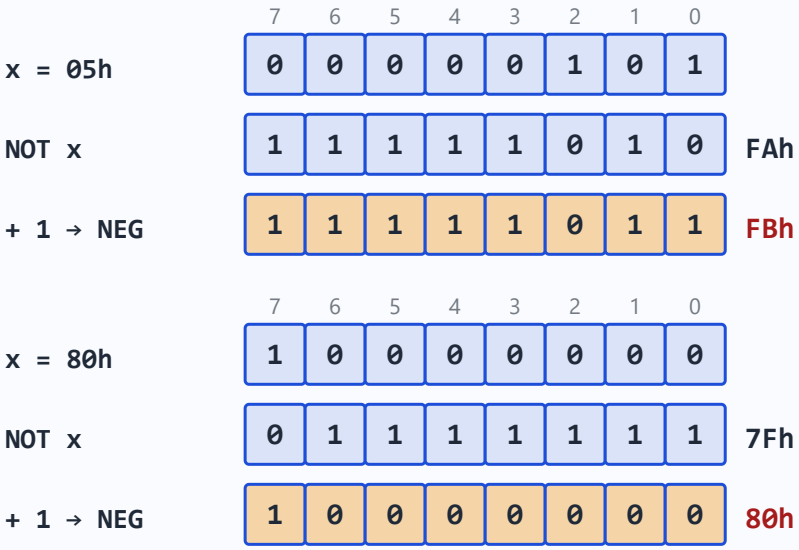


Figure 6.9: The two steps of NEG: first NOT (ones' complement), then +1. On the left 05h → FAh → FBh (−5); on the right 80h → 7Fh → 80h: the negative of −128 does not fit in a byte.

x	NOT x	NEG x	CF	OF	Explanation
05h (5)	FAh (−6)	FBh (−5)	1	0	the negative of a positive
FEh (−2)	01h	02h (2)	1	0	the negative of a negative is positive
80h (−128)	7Fh	80h	1	**1**	+128 does not fit in a byte
00h	FFh	00h	**0**	0	ZF = 1; the only case with CF = 0

Table 6.24: NEG on four byte values.

The value of the ones' complement is always 1 less than the two's complement: NOT 5 = −6, NEG 5 = −5. The case of −128 is special. The signed byte range is −128...127 and +128 is not in this range. NEG 80h gives 80h again and reports the overflow with OF = 1. The CF rule of NEG also comes from the subtraction 0 − x: if x is not zero, subtracting a positive number from 0 needs a borrow and CF = 1; if x = 0 no borrow is needed.

6.6.3 Code trace: INC, DEC, NOT, NEG

```
mov al, 0FFh
inc al          ; FFh + 1
dec al          ; 00h - 1
mov bl, 7Fh     ; +127
inc bl          ; +128 ?
not bl
neg bl
```

Instruction	Machine code	AL	BL	CF	ZF	SF	OF	PF	AF
`mov al, 0FFh`	`B0 FF`	FFh	00h	0	0	0	0	0	0
`inc al`	`FE C0`	00h	00h	**0**	1	0	0	1	1
`dec al`	`FE C8`	FFh	00h	**0**	0	1	0	1	1
`mov bl, 7Fh`	`B3 7F`	FFh	7Fh	0	0	1	0	1	1
`inc bl`	`FE C3`	FFh	80h	0	0	1	**1**	0	1
`not bl`	`F6 D3`	FFh	7Fh	0	0	1	1	0	1
`neg bl`	`F6 DB`	FFh	81h	**1**	0	1	0	1	1

Table 6.25: Trace of INC, DEC, NOT and NEG (all status flags 0 at the start).

`INC AL` makes FFh into 00h and ZF = 1. With ADD AL, 1, CF = 1 would result; INC does not change CF and CF stays 0. `DEC AL` computes 00h − 1 = FFh. A borrow was needed, but CF again does not change; only SF = 1. These two lines show that the only way to tell from the flags whether INC and DEC wrapped around is ZF and SF.

BL is loaded with the largest positive byte, 7Fh = +127. With `INC BL` BL becomes 80h: in the unsigned reading 128 is correct, in the signed reading it is −128. A signed overflow has occurred, OF = 1. `NOT BL` turns 80h into 7Fh but does not touch the flags: SF still looks 1, although the sign bit of BL is now 0. This shows that looking at the flags after NOT can be misleading. `NEG BL` takes the negative of +127: −127 = 81h. Because the operand is not zero, CF = 1; the result is in range, OF = 0.

6.7 Shifts and rotates

6.7.1 Shifts: SHL/SAL, SHR, SAR

Shift instructions move all the bits of a number in one direction. The bit that leaves at one end falls into CF, and a new bit enters at the other end. The instruction decides what the entering bit will be. It is the software counterpart of the shift register of digital design; the only difference is that the 8086 does the shift with an instruction rather than with a clock pulse. Figure 6.10 shows the bit flow of the three shifts.

Figure 6.10: SHL/SAL (0 enters from the right, b7 goes to CF), SHR (0 enters from the left, b0 goes to CF) and SAR (the sign bit stays in place and copies itself, b0 goes to CF).

Definition 6.7

SHL (shift logical left), and its other name **SAL**, shifts the bits left; 0 enters from the right and the most significant bit goes out to CF. Each shift multiplies an unsigned number by 2. **SHR** (shift logical right) is the **logical shift**: 0 enters from the left and the least significant bit goes out to CF; it divides an unsigned number by 2. **SAR** (shift arithmetic right) is the **arithmetic shift**: the most significant bit stays in place and is also copied to the right; it divides a signed number by 2.

With the example 96h = 1001 0110b the three shifts work as follows. `SHL AL, 1`: the bits move left, b7 (1) goes out to CF, 0 enters from the right: 0010 1100b = 2Ch, CF = 1. The unsigned 150 × 2 = 300 did not fit in a byte; CF shows this. `SHR AL, 1`: b0 (0) goes out to CF, 0 enters from the left: 0100 1011b = 4Bh = 75 = 150 / 2. `SAR AL, 1`: the sign bit 1 is copied: 1100 1011b = CBh = −53. In the signed reading 96h = −106 and −106 / 2 = −53 is correct.

The division by SAR rounds differently from that of IDIV. SAR always rounds toward minus infinity; IDIV truncates toward zero. For positive numbers the two give the same result; for negative odd numbers they part. For −7 = F9h, `SAR AL, 1` gives FCh = −4 (−3.5 has been rounded down). For −7 / 2, IDIV gives −3. This difference must be kept in mind when turning division into shifting.

6.7.2 Rotates: ROL, ROR, RCL, RCR

Rotate instructions resemble shifts, but the bit that leaves at one end is not lost; it comes back in at the other end. The bits turn around a ring. Figure 6.11 shows the bit flow of the four rotates.

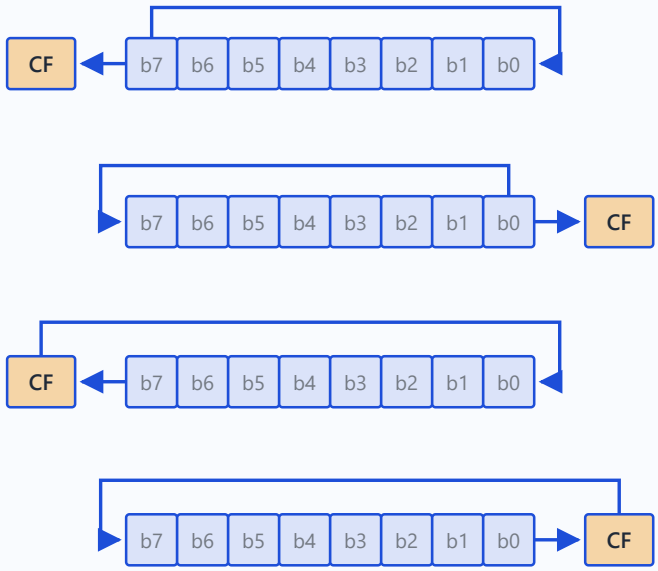


Figure 6.11: ROL and ROR rotate in an 8-bit ring and write a copy of the bit that leaves to CF. RCL and RCR include CF in the ring: a 9-bit ring for a byte, 17 bits for a word.

- **ROL** (rotate left): b7 goes both to b0 and to CF.
- **ROR** (rotate right): b0 goes both to b7 and to CF.
- **RCL** (rotate left through CF): b7 goes to CF, the old CF goes to b0.
- **RCR** (rotate right through CF): b0 goes to CF, the old CF goes to b7.

Instruction	With CF = 0	With CF = 1	Result CF	OF
`SHL AL, 1`	2Ch (0010 1100)	2Ch	1	1
`SHR AL, 1`	4Bh (0100 1011)	4Bh	0	1
`SAR AL, 1`	CBh (1100 1011)	CBh	0	0
`ROL AL, 1`	2Dh (0010 1101)	2Dh	1	1
`ROR AL, 1`	4Bh (0100 1011)	4Bh	0	1
`RCL AL, 1`	2Ch (0010 1100)	2Dh (0010 1101)	1	1
`RCR AL, 1`	4Bh (0100 1011)	CBh (1100 1011)	0	1 / 0

Table 6.26: The one-step result of seven instructions for AL = 96h (1001 0110b). For RCL and RCR the initial CF changes the result.

In a rotate no bit is lost. 1234h rotated CL = 8 times with `ROL AX, CL` becomes 3412h: the high and low bytes exchange places. Rotate instructions change only CF and (when the count is 1) OF; they do not touch SF, ZF, PF and AF. Shift instructions, however, set SF, ZF and PF according to the result; AF is undefined.

RCL and RCR are the key to multiword shifting. To multiply the 32-bit DX:AX by 2, the low word is shifted first and then the high word is rotated through CF: `SHL AX, 1` sends b15 of AX out to CF, and `RCL DX, 1` brings this bit into b0 of DX. For example with DX:AX = 0000FFFFh, after these two instructions DX:AX = 0001FFFEh. For division the reverse order is followed: `SHR DX, 1` and then `RCR AX, 1`.

6.7.3 Shift count, OF and timing

On the 8086 the shift count is given in only two forms: the constant 1 or the CL register. `SHL AX, 1` and `SHR AL, CL` are valid. A constant greater than 1 such as `SHL AL, 4` was added on the 80186 and later processors; in 8086 code one writes `SHL AL, 1` four times, or `MOV CL, 4` and `SHL AL, CL`. emu8086 accepts this form, but the code it produces does not run on a real 8086. The 8086 does not mask the number in CL; if CL = 255 the shift is done 255 times. The 80286 and later reduce the count to 5 bits (0–31).

OF is defined for shifts and rotates only when the count is 1. In a one-step left shift, OF says whether the sign bit changed: OF = (new b7) XOR CF. 40h = 0100 0000b shifted left becomes 80h; the sign bit turns from 0 to 1 and OF = 1. C0h shifted left also becomes 80h; but the sign bit stays 1 and OF = 0: −64 × 2 = −128 is correct. When the count is given in CL and is greater than 1, OF is undefined.

In terms of time, shifting is much faster than multiplication. According to Intel's manual, `SHL reg, 1` on a register takes 2 clock cycles and `SHL reg, CL` takes 8 + 4n cycles (n is the shift count). In memory a one-step shift is 15 + EA and with CL 20 + EA + 4n cycles. Considering that a byte MUL takes over 70 cycles, it is understandable why multiplication and division by powers of 2 are left to shifting.

6.7.4 Code trace: multiplying by shifting

Processors without a hardware multiplier do multiplication with the method on paper. The bits of the multiplier are examined one by one from right to left. If the bit is 1, the current multiplicand is added to the result; at each step the multiplicand is shifted left by one, that is, multiplied by 2. The program below computes 13 × 5 with this method.

```
org 100h
mov bl, mcand     ; 13
mov dl, mplier    ; 5 = 0101b
mov cx, 4         ; 4 bits
check:
shr dl, 1         ; low bit → CF
jnc skip
add result, bl    ; add if bit is 1
skip:
shl bl, 1         ; mcand × 2
loop check
ret
mcand db 13
mplier   db 5
result    db 0
```

There are two control instructions in the program. `JNC skip` (jump if no carry) goes to the skip label if CF = 0 and skips the addition. `LOOP check` decrements CX by 1 and returns to the check label if CX is not zero. With CX = 4 the loop runs four times; since the multiplier is 4 bits, this is enough.

Iteration	DL (after SHR)	CF	Addition	result	BL (after SHL)	CX (after LOOP)
1	02h	1	0 + 0Dh	0Dh (13)	1Ah (26)	3
2	01h	0	skipped	0Dh	34h (52)	2
3	00h	1	0Dh + 34h	41h (65)	68h (104)	1
4	00h	0	skipped	41h	D0h	0

Table 6.27: Computing 13 × 5 by shifting. Each row is one iteration: the bit shifted out by SHR, the addition, and the state after SHL.

The multiplier 5 = 0101b. In the first iteration SHR shifts the rightmost bit (1) out to CF and DL = 02h. Because CF = 1, JNC does not jump; the multiplicand itself, 13 × 1, is added to the result. In the second iteration the bit that comes out is 0 and the addition is skipped. In the third iteration the bit that comes out is 1 again. By then BL has been shifted twice and is 13 × 4 = 52; the result becomes 13 + 52 = 65 = 41h. In the fourth iteration the bit is 0. The loop ends and result = 41h = 65 = 13 × 5 remains. In the last iteration BL has become D0h; this value (208) is no longer used.

The program assumes that the multiplier is 4 bits and that the result fits in a byte. For larger multipliers the loop turns 8 times, and the multiplicand and result are kept in words. In the first generation of microprocessors (Intel 8080, Zilog Z80) there was no multiply instruction and every multiplication was done with a subroutine of this kind. The 8086's MUL instruction also applies the same method inside, in microcode.

6.7.5 Fast multiplication and division with shifts

Multiplication by a constant can be turned into shifts and additions according to the binary expansion of that constant. Since 3 = 2 + 1, AX × 3 = AX × 2 + AX:

```
; AX ← AX × 3 = AX×2 + AX
mov bx, ax      ; save
shl ax, 1       ; AX × 2
add ax, bx      ; ×2 + ×1
; unsigned ÷ 8 and remainder
mov dx, ax
and dx, 0007h   ; rem = low 3 bits
mov cl, 3
shr ax, cl      ; quotient
; signed ÷ 4
mov cl, 2
sar ax, cl      ; -7 → -2
```

For AX = 25 the first part gives BX = 25, AX = 50 and finally AX = 75. The total time is about 2 + 2 + 3 = 7 cycles; MUL takes over 118. The second part shows division by a power of 2. Shifting right by n bits divides by 2^n; the low n bits that go out during the shift are the remainder. The remainder is taken before the shift with AND: the mask with the low three bits 1 is 0007h. For AX = 100 the remainder is DX = 4 and the quotient AX = 12; 12 × 8 + 4 = 100.

The third part divides a signed number with SAR. For AX = −7, SAR gives −2 in two steps; with IDIV, −7 / 4 would give −1. SAR rounds toward minus infinity and IDIV toward zero. When compilers turn signed division into shifting, they first add $(2^n - 1)$ to negative numbers to remove this difference.

6.7.6 Code trace: 37 × 10 and 370 ÷ 16

Since 10 = 8 + 2, AX × 10 = AX × 8 + AX × 2. The program below first multiplies 37 by 10 this way, then divides the result by 16 and finds the remainder separately.

```
org 100h
mov ax, 37
shl ax, 1       ; AX × 2
mov bx, ax      ; save
shl ax, 1       ; AX × 4
shl ax, 1       ; AX × 8
add ax, bx      ; ×8 + ×2
mov bx, ax
and bx, 000Fh   ; rem: low 4 bits
mov cl, 4
shr ax, cl      ; quotient: ÷ 16
ret
```

Instruction	Machine code	AX	AX (binary)	BX	CL
`mov ax, 37`	`B8 25 00`	0025h = 37	`0000 0000 0010 0101`	0000h	00h
`shl ax, 1`	`D1 E0`	004Ah = 74	`0000 0000 0100 1010`	0000h	00h
`mov bx, ax`	`8B D8`	004Ah	`0000 0000 0100 1010`	004Ah = 74	00h
`shl ax, 1`	`D1 E0`	0094h = 148	`0000 0000 1001 0100`	004Ah	00h
`shl ax, 1`	`D1 E0`	0128h = 296	`0000 0001 0010 1000`	004Ah	00h
`add ax, bx`	`03 C3`	0172h = 370	`0000 0001 0111 0010`	004Ah	00h
`mov bx, ax`	`8B D8`	0172h	`0000 0001 0111 0010`	0172h	00h
`and bx, 000Fh`	`83 E3 0F`	0172h	`0000 0001 0111 0010`	0002h	00h
`mov cl, 4`	`B1 04`	0172h	`0000 0001 0111 0010`	0002h	04h
`shr ax, cl`	`D3 E8`	0017h = 23	`0000 0000 0001 0111`	0002h	04h

Table 6.28: Trace of 37 × 10 and 370 ÷ 16.

Each SHL shifts the bits one place left and doubles the number: 37, 74, 148, 296. The ×2 value saved in between (74) is added to the ×8 value and 370 = 37 × 10 is found. For the division, the remainder is separated first: the low four bits of 370 = 0172h are 0010b = 2 and `AND BX, 000Fh` leaves only them. Then `SHR AX, CL` shifts four bits right: 0172h → 0017h = 23. Check: 23 × 16 + 2 = 370.

Two details stand out in the machine codes. `shl ax, 1` (D1 E0) and `shr ax, cl` (D3 E8) use different opcodes: D1 means "once", D3 means "CL times". `and bx, 000Fh` takes three bytes with the 83 form: because the constant 0Fh fits in a single byte, the processor sign-extends it to 16 bits at run time. With the times in Intel's manual the program takes 4 + 2 + 2 + 2 + 2 + 3 + 2 + 4 + 4 + (8 + 4 × 4) = 49 clock cycles; a MUL with a word operand and a DIV together take over 260.

6.8 Example: array summation

6.8.1 The program

Adding two arrays element by element and writing the result to a third array is the simplest example in which arithmetic instructions are used together with memory. The program below does vec3[i] = vec1[i] + vec2[i] for i = 0…3.

```
org 100h
jmp start
vec1 db 1, 2, 5, 6
vec2 db 3, 5, 6, 1
vec3 db ?, ?, ?, ?
start:
lea si, vec1
lea bx, vec2
lea di, vec3
mov cx, 4
sum:
mov al, [si]
add al, [bx]
mov [di], al
inc si
inc bx
inc di
loop sum
ret
```

The structure of the program, line by line, is as follows.

- `JMP start` jumps over the data definitions. In a .COM program code and data are in the same segment and execution starts at 0100h. Without the jump the processor would try to decode the bytes of vec1 (01 02 05 06) as instructions.
- `vec3 db ?, ?, ?, ?` reserves four bytes. The question mark means "the initial value does not matter"; emu8086 writes 0 to these bytes.
- `LEA` (load effective address) loads not the value of a variable but its address (offset). SI, BX and DI become the pointers of the three arrays.
- `MOV AL, [SI]` reads the memory byte SI points to; this is register indirect addressing. `ADD AL, [BX]` adds the element of the second array, and `MOV [DI], AL` writes the result to the third array.
- The three INCs advance the pointers to the next element. Because the arrays are byte arrays each pointer increases by 1; with word arrays it would have to increase by 2.
- `LOOP sum` builds the loop.

The LOOP instruction always uses CX as the loop counter. Each time it executes it first decrements CX by 1 and then looks at CX. If CX is nonzero it jumps back to the label (sum); if zero it goes on to the next instruction (RET). Because CX was loaded with 4 before the loop, the body runs four times: after LOOP, CX becomes 3, 2, 1 and 0 in turn and on the last time the jump is not taken. LOOP does not change the flags. On the 8086 a LOOP that jumps takes 17 clock cycles and one that does not takes 5.

6.8.2 Code trace

Step	SI	BX	DI	CX	AL	vec3	Explanation
`jmp start`	0000 h	0000 h	0000 h	0000 h	00 h	0 0 0 0	the data (0102h–010Dh) is skipped
`lea si / bx / di`	0102 h	0106 h	010A h	0000 h	00 h	0 0 0 0	the addresses of the three arrays
`mov cx, 4`	0102 h	0106 h	010A h	0004 h	00 h	0 0 0 0	4 iterations
iteration 1: `mov al,[si]` · `add al,[bx]`	0102 h	0106 h	010A h	0004 h	04 h	0 0 0 0	1 + 3 = 4
iteration 1: `mov [di],al`	0102 h	0106 h	010A h	0004 h	04 h	**4** 0 0 0	vec3[0] = 4
iteration 1: three `inc` · `loop`	0103 h	0107 h	010B h	0003 h	04 h	4 0 0 0	CX = 3 ≠ 0 → back to sum
iteration 2	0104 h	0108 h	010C h	0002 h	07 h	4 **7** 0 0	2 + 5 = 7
iteration 3	0105 h	0109 h	010D h	0001 h	0B h	4 7 **11** 0	5 + 6 = 11
iteration 4	0106 h	010A h	010E h	0000 h	07 h	4 7 11 **7**	6 + 1 = 7; CX = 0, the loop ends

Table 6.29: Trace of the array summation program.

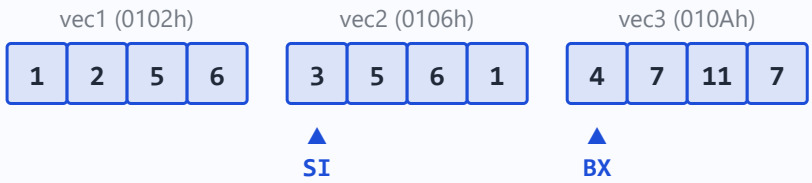


Figure 6.12: Memory when the program ends: vec3 = 4, 7, 11, 7 (04h 07h 0Bh 07h). After the last iteration SI points to the start of vec2 and BX to the start of vec3.

After the last iteration the pointers point one element past the arrays. SI = 0106h is now the first element of vec2; BX = 010Ah is the first element of vec3. This is the natural consequence of the arrays lying one after another in memory, and it is harmless since the loop has ended. Because the sums do not exceed 255, CF is never 1 in any iteration; with large elements the sum may not fit in a byte, and then CF must be looked at after the ADD.

6.8.3 Program and data in memory

Table 6.30 gives the view of the program in the emu8086 memory window (CS = DS = 0700h). Code and data are in the same segment, one after another.

Address	Bytes	Meaning
0100h	`EB 0C`	`JMP 010Eh` ; target = 0102h + 0Ch
0102h	`01 02 05 06`	vec1 (data)
0106h	`03 05 06 01`	vec2 (data)
010Ah	`04 07 0B 07`	vec3 (after the program has run)
010Eh	`BE 02 01`	`MOV SI, 0102h` (the code of LEA)
0111h	`BB 06 01`	`MOV BX, 0106h`
0114h	`BF 0A 01`	`MOV DI, 010Ah`
0117h	`B9 04 00`	`MOV CX, 0004h` ; the constant is little-endian
011Ah	`8A 04`	`MOV AL, [SI]` (sum:)
011Ch	`02 07`	`ADD AL, [BX]`
011Eh	`88 05`	`MOV [DI], AL`
0120h	`46 43 47`	`INC SI` , `INC BX` , `INC DI` (one byte each)
0123h	`E2 F5`	`LOOP 011Ah` ; F5h = −11
0125h	`C3`	`RET`

Table 6.30: The bytes of the array summation program in memory (emu8086).

Three things can be read from the table. First, emu8086 turns the directly addressed `LEA SI, vec1` line into the same three-byte code as `MOV SI, 0102h` . The two instructions do the same job: because the offset of the variable is known at assembly time, it is enough to load it as a constant. MASM, however, produces LEA with its own opcode (`8D 36 02 01`) as four bytes; in that case the following addresses shift by one byte. Second, jump instructions carry not the target address but the **distance** relative to the next instruction. The distance of `JMP` is 0Ch: 0102h + 0Ch = 010Eh. The distance of LOOP is F5h; as a signed number it is −11 and takes control back to 0125h − 0Bh = 011Ah, that is, to the sum label. Third, the constant in the MOV CX, 4 instruction sits in memory with the low byte first, as `04 00` .

6.9 BCD arithmetic

6.9.1 Binary-coded decimal numbers

People think in decimal, while the processor computes in binary. In devices that show decimal digits directly, such as calculators, cash register software, clock chips and seven-segment displays, converting the number from binary to decimal every time is expensive. Instead the number can be stored digit by digit.

Definition 6.8

In **binary-coded decimal** (BCD) representation each decimal digit is held in a separate 4-bit group with the values 0000–1001; 1010–1111 are not used. **Unpacked BCD** puts one digit in each byte; the upper nibble is 0. **Packed BCD** puts two digits in a byte: the high nibble is the tens digit and the low nibble the ones digit.

Representation	Bits	Byte(s)
Binary	`0000 1110`	`0Eh`
Unpacked BCD	`0000 0001 · 0000 0100`	`01h 04h` (AH, AL)
Packed BCD	`0001 0100`	`14h`
ASCII	`0011 0001 · 0011 0100`	`31h 34h` ('1', '4')

Table 6.31: Four representations of the number 14.

In packed BCD the number 14 is written as the byte 14h; the hexadecimal notation reads the decimal digits directly. ASCII digits are unpacked BCD with 3 added to the high nibble. Thanks to this closeness, digits read from the keyboard become unpacked BCD with `AND 0Fh`, and unpacked BCD becomes a character to write on the screen with `OR 30h`.

The processor's adder does not know BCD; it adds two BCD numbers like binary numbers. The result is often an incorrect BCD. 9 + 5 gives 0Eh in binary; the correct BCD result is "14". The 8086 therefore offers **BCD adjust** instructions that convert the result into BCD after a binary operation. All of them work only on AL (and AH when needed); BCD arithmetic is therefore done byte by byte.

6.9.2 The adjust instructions

Instruction	When	Effect	Time (cycles)
`AAA`	after unpacked addition	adjusts AL to BCD, carry into AH	4
`AAS`	after unpacked subtraction	adjusts AL, borrow from AH	4
`AAM`	after unpacked multiplication	AH ← AL / 10, AL ← AL mod 10	83
`AAD`	**before** unpacked division	AL ← AH × 10 + AL, AH ← 0	60
`DAA`	after packed addition	adjusts the two digits in AL	4
`DAS`	after packed subtraction	adjusts the two digits in AL	4

Table 6.32: The BCD adjust instructions. All are one-byte instructions with implicit operands (AAM and AAD are two bytes).

The idea of the adjustment rests on the number 6. A nibble can take 16 values while a decimal digit takes 10. When the sum of a digit exceeds 9, the binary sum falls 6 short of BCD: 9 + 5 = 0Eh, whereas "14" = 14h = 0Eh + 6. If the digit sum exceeds 15 (for example 9 + 8 = 17), a carry comes out of the

nibble and AF = 1; in this case too 6 must be added: 9 + 8 = 11h, 11h + 6 = 17h. The adjust instructions check these two conditions (nibble value > 9 or AF = 1) and add or subtract 6 if needed.

Rule (AAA and DAA)

AAA: if the low nibble of AL is greater than 9 or AF = 1, 6 is added to AL, 1 is added to AH and AF = CF = 1; otherwise AF = CF = 0. In both cases the high nibble of AL is cleared. **AAS** makes the same check, but subtracts 6 from AL and 1 from AH.

DAA: First the low nibble is checked: if it is greater than 9 or AF = 1, 06h is added to AL and AF = 1. Then the high digit is checked: if AL before the operation was greater than 99h or CF = 1, 60h is added to AL and CF = 1. **DAS** subtracts 06h and 60h with the same checks.

The application of the rules is seen in a few examples. 38h + 45h = 7Dh; because the low nibble D > 9, DAA adds 06h and AL = 83h: 38 + 45 = 83. 99h + 01h = 9Ah; the low nibble A > 9, AL + 06h = A0h; because the original AL was greater than 99h, 60h is added too and AL = 00h, CF = 1: 99 + 1 = 100, "00" and the hundreds digit in CF. In subtraction 83h − 38h = 4Bh; because the low nibble B > 9, DAS subtracts 06h and AL = 45h: 83 − 38 = 45. 25h − 48h = DDh; both adjustments are needed and DAS gives AL = 77h, CF = 1. This reads like 125 − 48 = 77: CF = 1 says a borrow was taken from the high digit.

6.9.3 Code trace: 9 + 5 with AAA

The following program adds two unpacked BCD digits, adjusts the result with AAA and prints the two digits on the screen. The printing part uses the 0Eh service (teletype) of BIOS interrupt 10h: with AH = 0Eh, `INT 10h` writes the character in AL where the cursor is and advances the cursor by one.

```
; first number '9':
mov     ah, 09h
; second number '5':
mov     al, 05h
; al = al + ah = 09h + 05h = 0eh
add     al, ah
; clear tens byte of bcd
xor     ah, ah
; adjust result to bcd form, ah = 1, al = 4 -> '14'
aaa
; --- print the result (extra) ---
or      ax, 3030h   ; to ASCII
mov     bl, al      ; save '4'
mov     al, ah      ; '1'
mov     ah, 0Eh
int     10h
mov     al, bl
int     10h
ret
```

Instruction	Machine code	AX	BL	CF	AF	Explanation
`mov ah, 09h` · `mov al, 05h`	`B4 09` · `B0 05`	0905h	00h	0	0	two digits
`add al, ah`	`02 C4`	090Eh	00h	0	0	9 + 5 = 0Eh; right in binary, not BCD
`xor ah, ah`	`32 E4`	000Eh	00h	0	?	the tens digit is cleared
`aaa`	`37`	**0104h**	00h	**1**	**1**	E > 9: AL + 6 = 14h, AH + 1, the high nibble is cleared
`or ax, 3030h`	`0D 30 30`	3134h	00h	0	?	'1', '4'
`mov bl, al` · `mov al, ah`	`8A D8` · `8A C4`	3131h	34h	0	?	'4' saved, AL = '1'
`mov ah, 0Eh` · `int 10h`	`B4 0E` · `CD 10`	0E31h	34h	0	?	'1' on the screen
`mov al, bl` · `int 10h`	`8A C3` · `CD 10`	0E34h	34h	0	?	'4' on the screen

Table 6.33: Trace of 9 + 5 with AAA. "14" appears on the screen.

09h + 05h = 0Eh, which in binary is 14 and is correct; but in BCD there is no digit E. When 9 + 5 was added in the low nibbles, no carry came out of bit 3, AF = 0. AAA sees that the low nibble E > 9. It adds 6 to AL: 0Eh + 6 = 14h. It increments AH by 1, sets AF and CF to 1, and clears the high nibble of AL: AL = 04h. The result is AX = 0104h, that is, unpacked "14". `OR AX, 3030h` puts 3 in the high nibbles of the two bytes and AX = 3134h, the ASCII '1' and '4'. The rest of the program writes these two characters to the screen in turn.

The purpose of the `XOR AH, AH` line is to have AAA write its carry into a clean tens digit. Without this line the first digit (09h) would remain in AH and AAA would make it 0Ah. AAA looks only at the low nibble of AL; so it is not necessary to subtract 30h from ASCII digits before the addition. In '7' + '8' = 37h + 38h = 6Fh the low nibble F > 9; AAA gives AH = 01h, AL = 05h, and with `OR AX, 3030h` '1' and '5' are obtained.

6.9.4 Code trace: 5 − 8 with AAS

```
; make 5 - 8, al = 0fdh (not in bcd form)
mov     al, 05h
mov     bl, 08h
sub     al, bl
; convert to bcd, al = 7
; and 1 is borrowed from ah,
; like calculating 15 - 8:
aas
; --- print ---
or      al, 30h
mov     ah, 0Eh
int     10h
mov     ah, 0
int     16h
ret
```

Instruction	AX	BL	CF	AF	SF	Explanation
`mov al, 05h` · `mov bl, 08h`	0005h	08h	0	0	0	
`sub al, bl`	00FDh	08h	1	1	1	5 − 8 = −3 = FDh
`aas`	**FF07h**	08h	**1**	**1**	?	AL − 6 = F7h, AH − 1 = FFh; the high nibble is cleared
`or al, 30h`	FF37h	08h	0	?	0	AL = '7'
`mov ah, 0Eh` · `int 10h`	0E37h	08h	0	?	0	'7' on the screen
`mov ah, 0` · `int 16h`	—	08h	0	?	0	waits for a key; AL = the code of the key pressed

Table 6.34: Trace of 5 − 8 with AAS. The program starts with AX = 0000h.

5 − 8 = −3, which is FDh in two's complement. A borrow was taken (CF = 1); in the low nibble too 5 − 8 needed a borrow (AF = 1). Because AF = 1, AAS subtracts 6 from AL: FDh − 6 = F7h. It borrows 1 from AH: 00h − 1 = FFh. It clears the high nibble of AL: AL = 07h. The result is AX = FF07h. "7" appears on the screen.

This result looks odd at first: 5 − 8 should be −3. What AAS does is take a 10 from a higher digit, as in subtraction on paper: 15 − 8 = 7. CF = 1 says that a borrow was taken and that the result is negative. 7 is the ten's complement of −3 (10 − 3 = 7). In a multidigit BCD subtraction this borrow is paid with SBB in the subtraction of the next digit; that AH is FFh reflects this. If we had started with AH = 01h (unpacked "15"), AAS would have given AX = 0007h, that is, the correct "07".

6.9.5 AAM and AAD

The adjustment for multiplication and division works differently from that for addition, because adding 6 is not enough here; the number must really be divided by 10 or multiplied by 10. The 8086 offers two instructions that do these two jobs in microcode.

Definition 6.9

AAM (ASCII adjust after multiply) divides AL by 10: the quotient is written to AH and the remainder to AL. It splits the product of two unpacked digits (at most 9 × 9 = 81) into two unpacked digits. **AAD** (ASCII adjust before division) converts the tens digit in AH and the ones digit in AL to a binary number: AL ← AH × 10 + AL, AH ← 0. It is written *before* a division instruction. Both set SF, ZF and PF according to AL; OF, AF and CF are undefined.

```
mov al, 7
mov bl, 8
mul bl      ; AX = 0038h (56)
aam         ; AH = 05h, AL = 06h
mov ax, 0504h  ; '54' unpacked
aad            ; AX = 0036h (54)
mov bl, 6
div bl         ; AL = 09h, AH = 00h
```

In the first part 7 × 8 = 56 = 0038h. AAM divides 56 by 10: AH = 5, AL = 6. AX = 0506h is now unpacked "56". In the second part AX = 0504h is unpacked "54". AAD converts it to 5 × 10 + 4 = 54 = 36h and zeroes AH. With a normal DIV, 54 / 6 = 9: AL = 09h, remainder AH = 00h. If AAD had been forgotten, DIV would have divided 0504h = 1284 and the result would have been 1284 / 6 = 214.

The machine code of AAM and AAD is interesting: `D4 0A` and `D5 0A`. The second byte is 0Ah, that is 10. On the 8086 this byte is really used as the base; `D4 10` divides AL by 16. Intel did not document this, and assemblers always produce 0Ah. `D4 00` raises INT 0 because it is a division by zero. AAM takes 83 and AAD 60 clock cycles; they are a division and a multiplication done in microcode.

6.9.6 Code trace: AAM, AAD and DAA

```
MOV  AL, 6
MOV  BL, 9
MUL  BL          ; AX = 6 × 9
AAM              ; split digits
MOV  AX, 0702h   ; '72' unpacked
AAD              ; to binary
MOV  BL, 8
DIV  BL
MOV  AL, 19h     ; packed 19
ADD  AL, 28h     ; + 28
DAA              ; adjust BCD
RET
```

Instruction	Machine code	AX	BL	AF	CF	Explanation
`MOV AL, 6` · `MOV BL, 9`	`B0 06` · `B3 09`	0006h	09h	0	0	
`MUL BL`	`F6 E3`	0036h	09h	?	0	6 × 9 = 54; fits in AL
`AAM`	`D4 0A`	**0504h**	09h	?	?	unpacked "54"
`MOV AX, 0702h`	`B8 02 07`	0702h	09h	?	?	unpacked "72"
`AAD`	`D5 0A`	**0048h**	09h	?	?	7 × 10 + 2 = 72 = 48h
`MOV BL, 8` · `DIV BL`	`B3 08` · `F6 F3`	0009h	08h	?	?	72 / 8 = 9, remainder 0
`MOV AL, 19h`	`B0 19`	0019h	08h	?	?	packed "19"
`ADD AL, 28h`	`04 28`	0041h	08h	**1**	0	9 + 8 = 17: carry out of the nibble
`DAA`	`27`	**0047h**	08h	1	0	AF = 1 → AL + 06h = 47h

Table 6.35: Trace of AAM, AAD and DAA. ? is an undefined flag.

The last three lines of the program show why DAA needs AF. 19h + 28h = 41h. The low nibble of the result is 1 and is not greater than 9; looking at it alone, no adjustment seems necessary. But in the ones digit 9 + 8 = 17 occurred and a carry came out of the nibble: AF = 1. DAA sees this flag and adds 06h: 41h + 06h = 47h. The result is "47" = 19 + 28. Because the high nibble 4 ≤ 9 and CF = 0, the second adjustment is not made.

> **Common mistake**
>
> Not writing the adjust instruction right after the addition. DAA and AAA look at AF and CF; a MOV in between causes no problem, but an instruction that changes the flags (ADD, CMP, AND, even an INC, which changes AF) spoils the adjustment. Another mistake is to use AAA with packed BCD or DAA with unpacked BCD: AAA clears the high nibble and destroys the tens digit of a packed number.

6.10 Example: temperature conversion

6.10.1 Celsius to Fahrenheit

The relation between degrees Celsius (°C) and degrees Fahrenheit (°F) is f = c × 9 / 5 + 32. The program below computes this formula with integer operations. Because temperatures can be negative, signed instructions (IMUL, IDIV) are used.

```
org 100h
start:
;celsius to fahrenheit according to f = c * 9 / 5 + 32
mov cl, 15
mov al, 9
imul cl
mov cl, 5
idiv cl
add al, 32
ret
```

Instruction	Machine code	AX	CL	CF	OF	Explanation
`mov cl, 15`	`B1 0F`	0000h	0Fh	0	0	c = 15
`mov al, 9`	`B0 09`	0009h	0Fh	0	0	
`imul cl`	`F6 E9`	0087h	0Fh	1	1	9 × 15 = 135; does not fit a signed byte, but is complete in AX
`mov cl, 5`	`B1 05`	0087h	05h	1	1	divisor
`idiv cl`	`F6 F9`	001Bh	05h	?	?	135 / 5 = 27, remainder 0
`add al, 32`	`04 20`	003Bh	05h	0	0	27 + 32 = 59

Table 6.36: Converting 15 °C to Fahrenheit.

9 × 15 = 135. 135 does not fit a signed byte (at most 127); IMUL therefore sets CF = OF = 1. But this is not an error: the result sits complete in AX as 0087h. IDIV divides AX by 5; the quotient 27 = 1Bh is written to AL and the remainder 0 to AH. 27 + 32 = 59 = 3Bh: 15 °C = 59 °F. When the program ends AX = 003Bh and CX = 0005h.

The order of operations in the formula matters. First multiply by 9, then divide by 5. If it were done the other way round, that is, 9 / 5 computed first, integer division would give 1 and the wrong formula f = c + 32 would result. In integer arithmetic division is left to the end as far as possible.

6.10.2 Fahrenheit to Celsius

```
org 100h
start:
;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
mov cl, 59
sub cl, 32
mov al, 5
imul cl
mov cl, 9
idiv cl
ret
```

Instruction	Machine code	AX	CL	CF	OF	Explanation
`mov cl, 59`	`B1 3B`	0000h	3Bh	0	0	f = 59
`sub cl, 32`	`80 E9 20`	0000h	1Bh	0	0	59 − 32 = 27
`mov al, 5`	`B0 05`	0005h	1Bh	0	0	
`imul cl`	`F6 E9`	0087h	1Bh	1	1	5 × 27 = 135
`mov cl, 9` · `idiv cl`	`B1 09` · `F6 F9`	000Fh	09h	?	?	135 / 9 = 15, remainder 0

Table 6.37: Converting 59 °F to Celsius.

59 − 32 = 27, 5 × 27 = 135 and 135 / 9 = 15: 59 °F = 15 °C. The result is in AL; AH carries the remainder (0). The two programs are inverses of each other and go back and forth exactly between 15 and 59, because the remainder is zero in both divisions.

6.10.3 Why IMUL and IDIV? The limits

Negative temperatures are the reason for the signed instructions. For c = −10, IMUL reads −10 = F6h as signed and gives AX = FFA6h = −90. IDIV computes −90 / 5 = −18 = EEh; with 32 added, 0Eh = 14: −10 °C = 14 °F is correct. If the same calculation were done with MUL and DIV, F6h would be taken as 246. MUL gives 9 × 246 = 2214 = 08A6h, and when DIV divides this number by 5 the quotient 442 does not fit in AL: the program stops with a divide error.

Integer division throws away the fractional part. Because IDIV truncates toward zero, c = 1 °C gives 33 °F with 9 / 5 = 1; the correct value is 33.8. If rounding is needed, the remainder in AH is examined: if the remainder is greater than half of the divisor, 1 is added to the quotient.

The most important limit of the program is that the result is kept in AL, a signed byte. Table 6.38 shows the limits of the Celsius to Fahrenheit conversion with numbers.

c	IMUL: AX	IDIV: AL	AL after ADD	OF	Result
−72	FD78h (−648)	quotient −129 does not fit → INT 0			divide error
−71	FD81h (−639)	81h (−127)	A1h (−95)	0	correct

c	IMUL: AX	IDIV: AL	AL after ADD	OF	Result
−10	FFA6h (−90)	EEh (−18)	0Eh (14)	0	correct
53	01DDh (477)	5Fh (95)	7Fh (127)	0	correct (the largest)
54	01E6h (486)	61h (97)	81h	1	wrong: read as −127
72	0288h (648)	quotient 129 does not fit → INT 0			divide error

Table 6.38: The limit values of the °C → °F program.

The valid range is c ∈ [−71, 53]. At c = 54 the quotient 97 fits, but 97 + 32 = 129 does not fit a signed byte; ADD sets OF = 1 and AL = 81h, read as −127. At c = 72 the quotient itself (129) does not fit in AL and a divide error occurs. In the Fahrenheit to Celsius conversion the limit is that f − 32 must fit in a byte: f ∈ [−96, 127]. For wider ranges, word operations are needed: c is taken into AX and multiplied in 16 bits with `IMUL BX`. Because the product is already in DX:AX, CWD is not needed; it is divided directly with `IDIV`.

6.11 Which instruction affects which flag?

Table 6.39 collects the flag effect of all the instructions described in these notes in one table. When tracing a program, or when looking for why a conditional jump behaves unexpectedly, this is the first place to look.

Instruction	CF	ZF	SF	OF	PF	AF
`ADD SUB CMP ADC SBB`	✓	✓	✓	✓	✓	✓
`NEG`	✓	✓	✓	✓	✓	✓
`INC DEC`	—	✓	✓	✓	✓	✓
`AND OR XOR TEST`	0	✓	✓	0	✓	?
`NOT`	—	—	—	—	—	—
`MUL IMUL`	✓	?	?	✓	?	?
`DIV IDIV`	?	?	?	?	?	?
`SHL SHR SAR`	✓	✓	✓	✓[1]	✓	?
`ROL ROR RCL RCR`	✓	—	—	✓[1]	—	—
`AAA AAS`	✓	?	?	?	?	✓
`AAM AAD`	?	✓	✓	?	✓	?
`DAA DAS`	✓	✓	✓	?	✓	✓
`CBW CWD MOV LEA LOOP`	—	—	—	—	—	—

Table 6.39: The effect of the instructions on the status flags. ✓ according to the result, 0 cleared, — unchanged, ? undefined, [1] defined only when the count is 1.

The table can be reduced to a few general rules. Data movement instructions (MOV, LEA, XCHG, CBW, CWD) and LOOP do not touch the flags. The addition and subtraction family writes all the flags; INC and DEC preserve CF of these. Logic instructions clear CF and OF. Multiplication writes a meaningful value only to CF and OF, and division to no flag. Rotates change only CF and OF. A program that relies on an undefined flag may work on one processor and break on another.

6.12 Common mistakes

Mistake	What happens?	Correct
Looking at CF with signed numbers	correct results such as −1 + 1 are taken as "overflowed"	signed: OF; in comparisons SF ≠ OF
Forgetting DX before `DIV BX`	the old DX enlarges the dividend; wrong result or INT 0	`XOR DX, DX` or `CWD`
`MOV AH, 0` for a signed byte	−30 = E2h is counted as 226	`CBW`
`MUL 10`	the 8086 has no immediate operand; the assembler rejects it	`MOV BL, 10` · `MUL BL`
`SHL AX, 4`	an 80186 instruction; does not work on the 8086	`MOV CL, 4` · `SHL AX, CL`
ADD on both words	the carry of the low word is lost	ADD, then ADC
A CMP between ADD and ADC	CF is erased	only MOV, INC, DEC, LOOP in between
AND to test a bit	the other bits of the number are erased	`TEST`
Dividing a negative number with SAR and expecting the IDIV result	−7 / 2: SAR −4, IDIV −3	choosing the rounding direction knowingly
Writing AAD after DIV	the unpacked number is taken for binary: 0702h = 1794 is divided	AAD before the division

Table 6.40: Common mistakes with arithmetic and logic instructions.

Common mistake

Choosing the wrong operand size for MUL and IMUL. `MUL BX` is a word multiplication and writes the result to DX:AX; a value kept in DX is silently erased. If a byte multiplication is wanted the operand must be 8 bits (`MUL BL`); then only AX changes.

6.13 Exercises and solutions

The eleven exercises in the slides are solved here with their intermediate steps.

1. What are AL, CF, OF, SF, ZF, PF after `MOV AL, 0C0h` and `ADD AL, 0C0h` ?

C0h + C0h = 180h; AL = 80h, CF = 1. −64 + (−64) = −128 fits a signed byte: OF = 0. SF = 1, ZF = 0. 80h has a single 1 bit: PF = 0.

2. After `CMP AL, BL`, by which flag is unsigned "AL < BL" recognized, and by which flags signed "AL < BL"?

Unsigned: CF = 1 (the subtraction needed a borrow); `JB` tests this. Signed: SF ≠ OF; `JL` tests this. SF alone is not enough, because if the difference overflows the sign bit flips.

3. With AX = 0FFFFh, BX = 0FFFFh, what are DX, AX, CF after `MUL BX`?

$65{,}535^2 = 4{,}294{,}836{,}225 = \text{FFFE0001h}$. DX = FFFEh, AX = 0001h. Since the upper half is not zero, CF = OF = 1.

4. What happens with AX = 0100h, BL = 0 on `DIV BL`?

Division by zero: the processor raises INT 0 (divide error). AX and the flags do not change; control passes to the interrupt service routine.

5. Change the array summation program for word (DW) arrays.

The data are defined with `dw`, AX is used instead of AL, and the pointers advance by 2:

```
vec1 dw 1000, 2000, 3000, 4000
vec2 dw 5, 6, 7, 8
vec3 dw 4 dup(?)
; ...
sum:
mov ax, [si]
add ax, [bx]
mov [di], ax
add si, 2
add bx, 2
add di, 2
loop sum
```

With this data vec3 = 1005, 2006, 3007, 4008. `INC SI` can be written twice; but `ADD SI, 2` changes CF and INC does not. If the loop uses ADC, the INCs are preferred.

6. Write the program that counts the 1 bits in a byte with SHL/TEST and a loop. Compare with PF.

```
mov al, 0B5h     ; the byte to count
xor bl, bl       ; counter
mov cx, 8
again:
test al, 80h     ; is the leftmost bit 1?
jz zero
inc bl
zero:
shl al, 1
loop again
```

For B5h = 1011 0101b, BL = 5. In each iteration TEST tests the leftmost bit and SHL brings the next bit to the far left. `JZ` (jump if ZF = 1) skips incrementing the counter if the bit is 0. PF says only whether the number of 1s is odd or even: for B5h PF = 0 (five is odd), but it does not give how many 1s there are.

7. What happens with DX = 0005h, AX = 0000h, BX = 0005h on `DIV BX`?

The dividend DX:AX = 50000h = 327,680. The quotient would be 65,536 and does not fit in AX. The rule says so too: DX = 5 ≥ the divisor 5. Even though the divisor is not zero, a divide error (INT 0) occurs.

8. What are AL and OF after `MOV AL, 70h` and `ADD AL, 30h`? How is the correct sum obtained in 16 bits?

AL = A0h, OF = 1: 112 + 48 = 160 > 127. The two numbers are extended separately with CBW and added in 16 bits: `MOV AL, 70h` · `CBW` · `MOV DX, AX` · `MOV AL, 30h` · `CBW` · `ADD AX, DX` → AX = 00A0h = 160.

9. Do 00012345h − 0000ABCDh with SUB/SBB. What is CF after the SUB?

2345h − ABCDh = 7778h, CF = 1 (borrow). 0001h − 0000h − 1 = 0000h. The result is 00007778h = 30,584.

10. In the 32 × 32-bit multiplication, to which words of the result are the four partial products added?

A0·B0 → z1:z0; A1·B0 and A0·B1 → z2:z1; A1·B1 → z3:z2. The carry of each ADD is carried to the next word up with ADC; after the third partial product an `ADC [z+6], 0` that adds only the carry to z3 may be needed.

11. With DX = 0000h, AX = 0FFFh, CF = 0, what is DX:AX after `SHL AX, 1` and `RCL DX, 1`?

SHL: AX = 1FFEh, the bit that came out (b15 = 0) goes to CF: CF = 0. RCL: 0 enters DX, DX = 0000h. DX:AX = 00001FFEh = 0FFFh × 2.

6.14 Summary

- **Flags**: arithmetic and logic instructions write the result to the destination and the properties of the result to the flags register. CF shows unsigned overflow (a carry in addition, a borrow in subtraction), OF shows signed overflow; OF = the XOR of the carries into and out of the most significant bit. ZF result zero, SF most significant bit, PF an even number of 1s in the low byte, AF carry from bit 3.
- **Two-operand instructions** (ADD, ADC, SUB, SBB, CMP, AND, OR, XOR, TEST) write the result to the destination; CMP and TEST change only the flags. Two memory operands, different sizes, a constant destination and a segment register cannot be used. Register-register takes 3 clock cycles, memory destination 16 + EA.
- **Comparison**: after CMP, unsigned "below" is CF = 1, signed "less" is SF ≠ OF.
- **Multiword arithmetic**: ADD/SUB on the lowest word, ADC/SBB on the upper ones; the carry is carried in CF. A 32 × 32 multiplication is done with four MULs and a chain of ADD/ADC.
- **Logic instructions** work bit by bit; they make CF = OF = 0. AND clears bits, OR sets them, XOR inverts them, TEST tests them. NOT changes no flag.
- **Multiplication and division**: in byte operations AX = AL × operand, AL = AX / operand (AH remainder); in word operations DX:AX. MUL/IMUL set only CF = OF meaningfully; DIV/IDIV leave all flags undefined. INT 0 if the divisor is 0 or the quotient does not fit. IDIV truncates toward zero.
- **Sign extension**: CBW (AL → AX) and CWD (AX → DX:AX) before signed division and signed addition; for unsigned data the upper half is zeroed.
- **INC and DEC** preserve CF; **NEG** takes the two's complement and sets CF = 1 if the operand is nonzero; NEG 80h overflows.
- **Shifts and rotates**: the count is 1 or CL. SHL ×2, SHR unsigned ÷2, SAR signed ÷2 (rounds toward minus infinity). In a rotate no bit is lost; RCL/RCR form a 9- or 17-bit ring with CF and make multiword shifting possible. Multiplying with shifts is much faster than MUL.
- **BCD**: AAA/AAS after unpacked addition/subtraction, DAA/DAS after packed addition/subtraction, AAM after multiplication, AAD before division. The adjustment rests on adding or subtracting 6 (and 60h for packed); this is why AF exists.
- **Application**: array summation is done with LOOP, temperature conversion with IMUL/IDIV; a signed byte result is correct only within a certain range.

6.15 Key terms

arithmetic logic unit (ALU) execution unit (EU) flags register (FLAGS) status / control flag carry flag (CF) overflow flag (OF) zero, sign, parity flag auxiliary carry flag (AF) carry / borrow signed / unsigned number two's / one's complement arithmetic / logical instruction implicit operand masking multiplication / division dividend / divisor / quotient / remainder

sign extension shift / rotate logical / arithmetic shift binary-coded decimal (BCD)
packed / unpacked BCD BCD adjustment nibble loop conditional jump interrupt vector
microcode clock cycle bus cycle

6.16 Test Yourself

1. What are AL and the six status flags after `MOV AL, 9Ch` and `ADD AL, 9Ch`?

   Answer

   9Ch + 9Ch = 138h; AL = 38h, CF = 1. Signed −100 + (−100) = −200 does not fit a byte and the sum of two negatives has come out positive (38h): OF = 1. SF = 0, ZF = 0. 38h = 0011 1000b has three 1s: PF = 0. The low nibbles are C + C = 18h; a carry comes out of bit 3: AF = 1.

2. What CF and OF values does `SUB AL, BL` give for AL = 30h, BL = 50h, and for AL = 80h, BL = 01h?

   Answer

   30h − 50h = E0h: unsigned 48 − 80 needs a borrow, CF = 1; signed 48 − 80 = −32 is correct, OF = 0. 80h − 01h = 7Fh: unsigned 128 − 1 = 127 is correct, CF = 0; signed −128 − 1 = −129 does not fit and a negative has turned into a positive, OF = 1.

3. With AL = F0h, BL = 10h, what are AX and CF after `IMUL BL` and after `MUL BL`? What would IMUL give if AL = F8h?

   Answer

   IMUL: −16 × 16 = −256 = FF00h; −256 does not fit a byte, CF = OF = 1. MUL: 240 × 16 = 3840 = 0F00h; AH ≠ 0, CF = 1. For AL = F8h (−8) IMUL gives −128 = FF80h; AH is only the sign extension of AL and CF = OF = 0.

4. The number −50 is divided by 7 and by 8 with IDIV (AX = −50). What are the quotients and remainders? What is found if −50 is shifted 3 bits with SAR?

   Answer

   IDIV 7: −50 / 7 = −7.14; toward zero −7 = F9h, remainder −50 − (−49) = −1 = FFh. IDIV 8: quotient −6 = FAh, remainder −2 = FEh. SAR 3 (÷ 8): CEh = 1100 1110b → 1111 1001b = F9h = −7; −6.25 has been rounded toward minus infinity to −7. IDIV and SAR give different results.

5. With CF = 1 and AL = 81h, what are AL and CF after three consecutive `RCL AL, 1` instructions?

   Answer

   1000 0001b, CF = 1 → 0000 0011b, CF = 1 → 0000 0111b, CF = 0 → 0000 1110b, CF = 0. AL = 0Eh, CF = 0. The bits have turned around the nine-bit ring and none has been lost.

6. 58 + 67 is done in packed BCD: `MOV AL, 58h`, `ADD AL, 67h`, `DAA`. What are AL, AF and CF after the ADD and after the DAA?

   Answer

   ADD: 58h + 67h = BFh; in the low nibble 8 + 7 = 15 = Fh, no carry (AF = 0), CF = 0. DAA: low nibble F > 9 → +06h = C5h, AF = 1; the original AL BFh > 99h → +60h = 125h, AL = 25h, CF = 1. The result is "25" and 1 in the hundreds digit: 58 + 67 = 125.

7. With AL = B6h, what are AL and PF after `AND AL, 0F0h`, `OR AL, 05h`, `XOR AL, 0FFh`?

   Answer

   AND: B0h (the low nibble is cleared). OR: B5h (bits 0 and 2 are set). XOR: 4Ah = 0100 1010b (all bits are inverted). There are three 1s: PF = 0. After logic instructions CF = OF = 0.

8. Which three instructions are written to add 1 to the 48-bit number DX:BX:AX = 0001 FFFF FFFFh? What is the result?

   Answer

   `ADD AX, 1` · `ADC BX, 0` · `ADC DX, 0`. AX = 0000h and CF = 1; BX = FFFFh + 0 + 1 = 0000h and CF = 1; DX = 0001h + 0 + 1 = 0002h. The result is 0002 0000 0000h. If `INC AX` had been used, CF would not have changed and the carry would have been lost.

9. What are AX, CF and OF after `MOV AX, 8000h` and `NEG AX`? Why?

   Answer

   AX stays 8000h; CF = 1 (the operand is not zero), OF = 1. 8000h is −32,768 as a signed word; +32,768 does not fit the word range (at most 32,767). NOT 8000h = 7FFFh, +1 = 8000h.

10. What is the difference between `INC AX` and `ADD AX, 1` in machine code, time and flags?

Answer

`INC AX` is a single byte (`40`) and takes 2 clock cycles; it does not touch CF. `ADD AX, 1` is three bytes (`05 01 00` , or three bytes with the 83 form as well) and takes 4 cycles; it writes six flags including CF. With AX = FFFFh, ADD sets CF = 1; INC leaves CF as it was.

6.17 Further reading

- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 5: arithmetic and logic instructions, BCD and ASCII adjustments, shifts and rotates; it extends the subjects of these notes to the processors after the 8086.
- Intel, *The 8086 Family User's Manual* (1979), the instruction set section: the flag effect, encoding and clock cycle times of each instruction are here at first hand.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), Chapter 7: integer arithmetic, multiword shifts and addition, ASCII and packed decimal arithmetic.
- emu8086 documentation, "8086 instructions": a short description of each instruction, its flag table and example code; it is the source of the BCD examples in the slides.

6.18 References

1. Intel, *The 8086 Family User's Manual*, 1979 — Instruction Set section.
2. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapter 5.
3. emu8086 help documents: "8086 instructions" and "Arithmetic and logic instructions".
4. K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — Chapter 7.
5. A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.
6. M. Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz* (in Turkish), EMO, 2014.
7. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş* (in Turkish), 2009.

Flow Control

Left to itself, the processor executes instructions in the order they sit in memory. After each instruction the instruction pointer, IP, grows by the length of that instruction and points at the next one. Any program that makes a decision, repeats a task, or branches off on an error has to break this straight line. Flow control instructions do so by loading a new value into IP. This note first covers the unconditional jump (JMP) and how the jump displacement is written into machine code. It then turns to conditional jumps that test the flags, the CMP instruction, and the reason signed and unsigned numbers need separate families of jumps. Next it builds the assembly counterparts of the if/else and while constructs of high-level languages. Finally it studies the LOOP family, which uses CX as a counter, and JCXZ. Every example program is traced step by step at the level of registers, flags, and memory; the effect of a jump on the prefetch queue and on clock cycles is explained as well.

LEARNING OUTCOMES

Jump mechanism
Direct program flow with JMP and labels; explain how IP changes and how the jump displacement (relative offset) is computed; distinguish short, near, far, and indirect jumps.

Conditional jumps
State which flag (ZF, CF, SF, OF, PF) each conditional jump tests, find its opposite and its synonyms, and compute its machine code.

Comparing correctly
After CMP, choose the JA/JB family for unsigned numbers and the JG/JL family for signed numbers; explain through overflow why the rule is SF = OF.

High-level constructs
Write if/else, while, and for constructs with patterns that jump on the opposite condition; fix an out-of-range conditional jump with the inverted condition plus JMP.

Loops
Write counted loops with LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ, and JCXZ; explain the CX = 0 trap and why a nested loop must save its counter; trace the example programs step by step.

7.1 Sequential execution and flow control

7.1.1 IP and the prefetch queue

The instructions the 8086 executes are bytes laid out one after another in the code segment. The processor knows which of these bytes starts the next instruction from two registers: **CS** (the code segment) and **IP** (the instruction pointer). CS holds the start of the segment, and IP holds the offset of the next instruction within it. As with every other memory access, the physical address of the instruction is CS × 10h + IP. In a .COM program with CS = 0700h and the first instruction at 0100h, the processor fetches the first instruction from 07000h + 0100h = 07100h.

A program cannot write to IP with MOV; there is no `MOV IP, AX`. Only the processor itself changes IP. In the ordinary case the change is simple: once an instruction is decoded, IP grows by that instruction's byte count. If the three-byte `MOV AX, 5` (`B8 05 00`) sits at 0100h, IP = 0103h when it finishes. The two-byte `MOV AL, 25` (`B0 19`) advances IP by two. This pattern is called **sequential execution**.

Sequential execution has a hardware counterpart in the 8086. The processor consists of two units. The **bus interface unit** (BIU) manages the address and data buses and reads instruction bytes from memory. The **execution unit** (EU) decodes those bytes and carries out the operation. The BIU does not wait for the EU to finish: whenever the bus is idle, it reads ahead of CS:IP and stores the bytes in a six-byte **prefetch queue**. By the time the EU moves on to the next instruction, its bytes are usually already waiting in the queue. Reading from memory and executing instructions thus overlap, which makes this a simple pipeline.

The queue rests on one assumption: the next instruction follows immediately in memory. Flow control instructions break that assumption. When a jump executes, IP moves to a different address, and the bytes in the queue now belong to the wrong instructions. The EU flushes the queue and the BIU starts reading again from the new CS:IP. Much of the clock-cycle cost of a jump comes from this refill.

Definition 7.1

The **instruction pointer** (IP) is the 16-bit register that holds the offset, within the code segment, of the next instruction to execute. The physical address of the instruction is CS × 10h + IP. A **flow control instruction** is one that loads IP (and, for a far jump, CS as well) with a value other than the one given by the instruction length.

7.1.2 Types of flow control instructions

The flow control instructions of the 8086 fall into three groups. The first is the **unconditional jump**: `JMP label` jumps in every case. The second is the **conditional jumps** (`Jcc`). These test one or more flags in the flags register: if the condition holds they jump, otherwise execution continues with the next instruction. Conditional jumps split into three subgroups: those that test a single flag (JZ, JC, JS, JO, JP and their opposites), those for signed numbers (the JG, JL family), and those for unsigned numbers (the JA, JB family). The third group is the **loop instructions**: `LOOP` , `LOOPE/LOOPZ` , `LOOPNE/LOOPNZ` , and `JCXZ` . They use CX as the counter.

Other instructions change the flow too. `CALL` and `RET` go to a subroutine and come back, keeping the return address on the stack. `INT` and `IRET` go to an interrupt service routine and return. The `RET` that ends every program in this note is also a flow control instruction. When a .COM program starts, the top of the stack holds 0000h, and `RET` loads this value into IP. At offset 0000h sits the PSP (program segment prefix) that DOS places at the start of the program. The first two bytes of the PSP are `INT 20h` , and that is how the program terminates. This is why the last row of the trace tables shows IP = 0000h and SP = 0000h.

In high-level languages, flow control appears as decision structures (if/else, switch) and loops (for, while, do-while). The compiler translates all of them into compare, conditional jump, and unconditional jump instructions. An assembly programmer does this translation by hand. Figure 7.1 compares how the same piece of program is traversed under sequential execution and under execution with jumps.

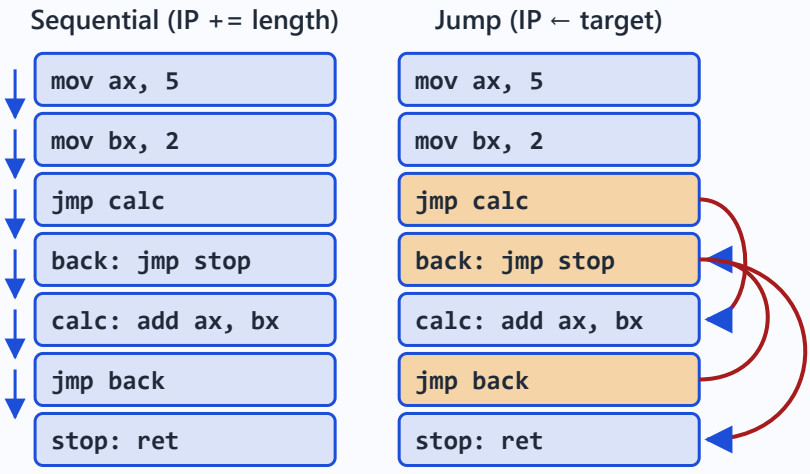


Figure 7.1: On the left, instructions execute in memory order and IP grows by the instruction length at every step. On the right, the `jmp` instructions load a new value into IP, and the flow leaps to the labels calc, back, and stop.

7.2 The unconditional jump: JMP

7.2.1 JMP and its forms

The `JMP label` instruction transfers control to another point in the program under all conditions. It can jump forward or backward. It changes no flag; the flags that held before the jump are still valid after it. This matters: the flags set by a CMP can still be read after an intervening JMP.

> **Definition 7.2**
>
> An **unconditional jump** makes execution continue at the specified target address regardless of the flags. On the 8086 it is performed by the `JMP` instruction. The target is given inside the instruction either as a **displacement**, as an absolute CS:IP pair, or indirectly through a register or memory word.

The 8086 recognizes four forms of the JMP instruction. Which one is used depends on how far away the target is and where it is written. Table 7.1 summarizes these forms.

Form	Code	Bytes	Range	Changes	Clock cycles
short (`JMP SHORT`)	`EB rel8`	2	−128…+127	IP	15
near (`JMP NEAR`)	`E9 rel16`	3	anywhere in the segment	IP	15
far (`JMP FAR`)	`EA ofs seg`	5	anywhere in 1 MB	CS and IP	15
indirect, register (`JMP BX`)	`FF E3`	2	within the segment	IP	11
indirect, memory (`JMP [BX]`)	`FF 27`	2–4	within the segment	IP	18 + EA

Table 7.1: The forms of the JMP instruction, their machine codes, and their timings on the 8086.

A **short jump** is two bytes: the opcode EBh and an 8-bit signed displacement (rel8). The displacement ranges from −128 to +127, counted from the address of the instruction immediately after the jump. Most loops and short if blocks fit in this range, so the short jump is the most common form.

A **near jump** is three bytes: E9h and a 16-bit displacement (rel16). Because 16-bit addition wraps around, rel16 reaches every offset (0000h–FFFFh) in the 64 KB of the code segment. Read as signed, the displacement means ±32 KB, but thanks to the wraparound the jump can also reach the other end of the segment. In short and near jumps only IP changes and CS stays the same, so both are **intrasegment** jumps.

A **far jump** is five bytes: EAh, followed by the 16-bit offset and the 16-bit segment value of the target. The byte order is little-endian, so the low byte of the offset comes first. For example, a far jump to 0800h:0000h is encoded as `EA 00 00 00 08`. The instruction loads 0000h into IP and 0800h into CS,

and execution continues at physical address 08000h. This is the only way to move to another segment, and it is called an **intersegment** jump.

In an **indirect jump** the target is not written in the instruction. `JMP BX` loads the value of BX into IP. `JMP [BX]` reads the word at address DS:BX and loads it into IP. Indirect jumps suit cases where the target is computed at run time. A typical example is going to a different function according to the number chosen in a menu. The programmer builds a **jump table**: `table dw task0, task1, task2`. The selection number is multiplied by 2 and placed in SI, and `JMP table[SI]` goes to the right address. The machine code of this instruction consists of `FF A4` and the 16-bit address of the table, four bytes in all.

The programmer often does not need to choose a form. If the assembler knows the target and the target is within short range, it picks the 2-byte short form itself. If the target lies further down in the code and is not yet defined, some assemblers first reserve 3 bytes. Multi-pass assemblers (emu8086 is one) shrink to the short form once the addresses are final. Writing `JMP SHORT label` forces the short form; if the target is out of range, the assembler reports an error. Writing `JMP NEAR PTR label` forces the 3-byte form even when the target is close.

> **Common mistake**
>
> The claim “JMP can go at most 127 bytes forward” is wrong. That limit applies only to the short form. A near JMP reaches anywhere in the code segment, and a far JMP reaches anywhere in the 1 MB address space. The 127-byte limit matters mainly for the 8086's *conditional* jumps, because they have no 16-bit form.

7.2.2 Labels

An assembly programmer writes a jump target not as an address but as a **label**. A label is a name given to an address. To define one, write the name followed by a colon: `loop_top:`. A label can stand on a line of its own or in front of an instruction: `back: jmp stop`. In both forms the label receives the address of the first instruction that follows it.

A label takes no space in memory. While reading the source, the assembler keeps a **location counter**. It adds each instruction's byte count to the counter, and when it meets a label it records the counter's current value as the label's address. Later, when encoding a line such as `jmp loop_top`, it uses that address. In Figure 7.2 the label `loop_top` becomes the address of the `DEC CX` instruction that follows it, 0103h. In machine code there is no such thing as `loop_top`; only the displacement byte of the `JNZ` instruction holds a number leading to that address.

Figure 7.2: The label `loop_top` becomes the address (0103h) of the `DEC CX` instruction that follows it; `JNZ loop_top` jumps to that address. The label occupies no bytes in memory.

There are a few rules for label names. A name consists of characters such as letters, digits, and underscores, but it cannot start with a digit. It cannot contain spaces. Instruction names (`mov`, `loop`) and register names (`ax`, `cx`) cannot be labels, because the assembler first tries to recognize the word at the start of a line as an instruction or a register. A label is defined once in a program; a second definition would make it ambiguous which address is meant and causes an error.

Label	Status	Reason
`loop_top:`	valid	made of letters
`next_char:`	valid	underscores are allowed
`k1:`, `calc:`	valid	a digit is not the first character
`1start:`	invalid	starts with a digit
`mov:`	invalid	instruction name
`two words:`	invalid	contains a space

Table 7.2: Examples of valid and invalid labels.

A label also has a type. Labels inside code are of type near: they carry only an offset, which is why they are used in intrasegment jumps. The names of data definitions (`msg db 'Hello', 0`) are addresses as well, but they are called variables and have data types (byte, word). The difference matters for JMP. Writing `jmp calc` makes the assembler produce a displacement leading to the address of calc. But when the name of a variable such as `table dw 0103h` is written, `jmp table` is encoded as an indirect jump (`FF 26` and the address of the table). The processor reads the word inside the table and loads it into IP. Although the syntax looks alike, one jumps to a label and the other to the contents of a variable.

7.2.3 Code trace: forward and backward with JMP

The program below shows that JMP can jump both forward and backward. It loads two numbers, jumps forward to the `calc` block that does the addition, returns from there backward to the line `back`, and finally jumps forward again to the line `stop`. In the source, the order of the lines differs from the order of execution.

```
org    100h
mov    ax, 5          ; ax = 5
mov    bx, 2          ; bx = 2
jmp    calc           ; go to 'calc'
back:  jmp stop       ; go to 'stop'
calc:
add    ax, bx         ; ax = ax + bx
jmp    back           ; back to 'back'
stop:
ret                   ; return to the OS
```

The `org 100h` directive is not an instruction. It tells the assembler to start the location counter at 0100h, because DOS loads a .COM program at offset 0100h of the segment. The first instruction, `mov ax, 5`, is therefore at 0100h. When the processor starts the program, IP = 0100h.

Table 7.3 gives each instruction of the program, its address, its machine code, and the state after the instruction completes. The IP in the last column is the address of the next instruction.

Step	Address	Instruction	Code	AX	BX	IP (after)
0	—	start	—	0000h	0000h	0100h
1	0100h	`mov ax, 5`	`B8 05 00`	0005h	0000h	0103h
2	0103h	`mov bx, 2`	`BB 02 00`	0005h	0002h	0106h
3	0106h	`jmp calc`	`EB 02`	0005h	0002h	010Ah
4	010Ah	`add ax, bx`	`01 D8`	0007h	0002h	010Ch
5	010Ch	`jmp back`	`EB FA`	0007h	0002h	0108h
6	0108h	`jmp stop`	`EB 04`	0007h	0002h	010Eh
7	010Eh	`ret`	`C3`	0007h	0002h	0000h

Table 7.3: Trace of the JMP example (CS = 0700h; values are the state after the instruction has executed).

The first two steps are sequential execution: IP grows first by 3, then by 3 again. In the third step `jmp calc` executes. With sequential execution IP would be 0108h. The jump loads IP with the address of calc, 010Ah. The `back: jmp stop` instruction at 0108h, which lies in between, is not executed on this pass. In the fourth step `add ax, bx` makes AX 5 + 2 = 7. ADD affects the flags too. The result is not zero (ZF = 0), the sign bit is 0 (SF = 0), and there is no carry or overflow (CF = OF = 0). Because 07h = 0000 0111b contains three 1 bits, the parity is odd (PF = 0).

The fifth step is the backward jump. `jmp back` sits at 010Ch and pulls IP to 0108h, an address before itself. In the sixth step the `jmp stop` at 0108h is finally executed and IP becomes 010Eh. In the seventh step `ret` takes the value 0000h from the stack; IP = 0000h and SP = FFFEh + 2 = 0000h. The `INT 20h` in the PSP ends the program. The result is AX = 0007h. In terms of source lines, the execution order is 2 → 3 → 4 → 7 → 8 → 5 → 10. Since none of the three JMPs touches the flags, the flags set by ADD stay the same until the end of the program.

The code `01 D8` given for `add ax, bx` in the table is ADD's "register/memory ← register" form. The same instruction can also be encoded in the "register ← register/memory" form as `03 C3`, and some assemblers produce that. Both encodings are two bytes and do the same job; addresses and jump displacements do not change.

7.2.4 Computing the jump displacement

In the machine code of a short or near jump, the address of the target is not written. Instead, what is written is the signed displacement of the target relative to the address of *the instruction after the jump*. This approach is called **relative addressing**. The computation is:

> displacement = target − (address of the jump instruction + length of the jump instruction)

The sum in parentheses is the value of IP at the moment the processor has read and finished the jump instruction. While executing the jump, the processor adds the displacement to this IP: IP ← IP + displacement. A negative displacement is stored in two's complement, and the addition is done by the same circuit.

Definition 7.3

The **jump displacement** is the signed difference between the jump target and the address of the instruction following the jump. It is 8 bits (rel8, −128…+127) in the short jump and in all conditional jumps of the 8086, and 16 bits (rel16) in the near jump. When executing, the processor computes IP ← IP + sign-extended displacement.

Figure 7.3 derives the code of the first jump in the JMP example, `jmp calc`. The instruction is at 0106h and is two bytes long. When the processor finishes it, IP = 0108h. The target calc = 010Ah. The displacement is 010Ah − 0108h = +2, and the machine code is `EB 02`. These two bytes mean skipping over the two-byte `jmp stop` instruction at 0108h.

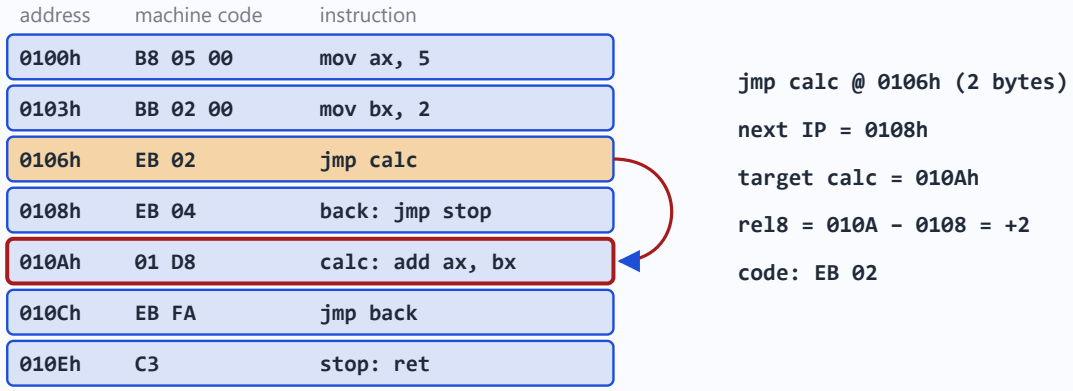


Figure 7.3: Forward jump: `jmp calc` at 0106h, next instruction at 0108h, target at 010Ah. Displacement 010Ah − 0108h = +2, code `EB 02`.

The computation for `jmp stop` at 0108h is the same. The next instruction is at 010Ah and the target stop = 010Eh. The displacement is 010Eh − 010Ah = +4 and the code is `EB 04`. During execution the processor computes IP ← 010Ah + 4 = 010Eh.

In a backward jump the displacement is negative. After the `jmp back` instruction at 010Ch, IP = 010Eh; the target back = 0108h. The displacement is 0108h − 010Eh = −6. The 8-bit two's complement of −6 is found like this: 6 = 0000 0110b; inverting the bits gives 1111 1001b; adding 1 gives 1111 1010b = FAh. The machine code is `EB FA` (Figure 7.4).

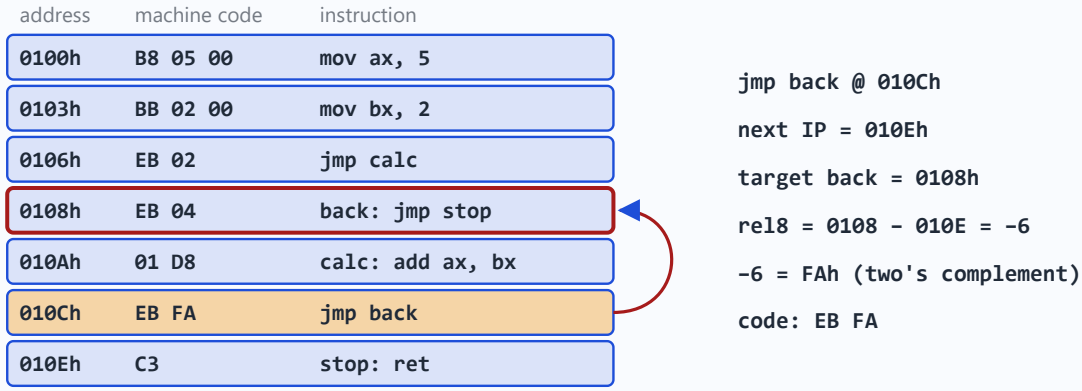


Figure 7.4: Backward jump: `jmp back` at 010Ch, next IP 010Eh, target 0108h. Displacement −6, FAh in two's complement; code `EB FA`.

Before adding this eight-bit number to IP, the processor performs **sign extension**: since the most significant bit is 1, the upper eight bits are filled with 1s and FAh becomes FFFAh. The addition is 16-bit: 010Eh + FFFAh = 1 0108h. The seventeenth bit does not fit in the register and is discarded; IP = 0108h. A positive displacement goes through the same process, except that the upper bits fill with 0s: 02h → 0002h.

Example 7.1

A short JMP at address 0150h wants to jump to 0140h. What is its machine code?

The instruction is two bytes, so the next IP = 0152h. Displacement = 0140h − 0152h = −12h = −18. 18 = 0001 0010b; inverted 1110 1101b; adding 1 gives 1110 1110b = EEh. The code is `EB EE`. Check: 0152h + FFEEh = 1 0140h → IP = 0140h.

Had the jump to the same target been made from 0100h, the displacement would be 0140h − 0102h = +3Eh and the code `EB 3E`. From 0300h the displacement would be 0140h − 0302h = −1C2h = −450. This number is less than −128, so the short form cannot be used. The assembler chooses the near form: the instruction is 3 bytes, the next IP = 0303h, the displacement is 0140h − 0303h = −1C3h = FE3Dh, and the code is `E9 3D FE`.

Relative addressing has an important benefit. When the code is moved elsewhere in memory, the displacements do not change, because both the jump instruction and its target shift by the same amount. Such code is called **position-independent code**. Whatever segment DOS loads a .COM program into, the short and near jumps within the segment work correctly. A far jump, on the other hand, contains an absolute segment value, which must be fixed up if the program is moved.

7.2.5 The hardware cost of a jump

Table 7.1 shows that JMP takes 15 clock cycles. That is a long time for a two-byte instruction; `ADD AX, BX`, which adds two registers, takes only 3. The reason is the prefetch queue. When a jump executes, the EU tells the BIU to flush the queue and start reading from the new address. The BIU puts the new CS:IP on the address bus and reads two bytes in one bus cycle (four clock cycles). The EU waits until the first byte of the new instruction arrives in the queue. A significant part of the 15 cycles is this wait.

For conditional jumps the cost is even more visible. If the condition holds and the jump is taken, the instruction takes 16 cycles. If the condition does not hold, it takes only 4, because IP points at the next instruction and the bytes in the queue remain valid. Between the two cases of the same instruction there is a fourfold difference. For this reason, in frequently executed loops, writing the jump so that it falls on the “mostly not taken” side gains speed even on the 8086.

Instruction	If taken	If not taken
`JMP` short / near / far	15	—
`JMP` register / memory	11 / 18 + EA	—
`Jcc` (all conditional jumps)	16	4
`LOOP`	17	5
`LOOPE` / `LOOPZ`	18	6
`LOOPNE` / `LOOPNZ`	19	5

Instruction	If taken	If not taken
`JCXZ`	18	6

Table 7.4: Clock cycle counts of flow control instructions on the 8086 (Intel 8086 manual).

These numbers become concrete on a 5 MHz 8086. One clock cycle is 1 / 5 MHz = 200 ns. A 15-cycle JMP takes 3 µs. A conditional jump not taken takes 0.8 µs, and one taken takes 3.2 µs. The loop instruction alone of a thousand-iteration loop can therefore take milliseconds.

A jump has one more side effect. If the target address is odd, the BIU can fetch only one byte in the first bus cycle. The 8086 data bus is 16 bits wide and can read a word in a single bus cycle only from an even address. Aligning frequently used loop heads to an even address (the `EVEN` directive) therefore gives a small gain. The 8088, with its 8-bit data bus, has no such difference, but because every byte needs its own bus cycle, filling the queue takes even longer.

7.3 Conditional jumps

7.3.1 The idea of a conditional jump

An unconditional jump cannot decide anything on its own; it goes to the same place every time. To decide, a jump must depend on a condition. On the 8086 this condition is always read from the flags in the **flags register**. The general name of the conditional jump instructions is `Jcc`; in place of cc comes the abbreviation of the condition (Z, NZ, C, A, G …). The meaning of `Jcc label` is this: if the condition is true, IP ← label; otherwise IP keeps pointing at the next instruction.

Definition 7.4

A **conditional jump** is an instruction that, depending on the state of one or more flags in the flags register, either jumps to the target or moves on to the next instruction. A conditional jump only reads the flags; it changes none of them.

The flags are set by an instruction that comes before the conditional jump. This instruction is most often `CMP`, but arithmetic and logic instructions such as `SUB`, `ADD`, `TEST`, `AND`, `INC`, and `DEC` set the flags too. Instructions that do not touch the flags (`MOV`, `JMP`, `LOOP`, `PUSH`, `POP`) may come between the flag-setting instruction and the jump that reads the flags; the decision is still made from the result of the first instruction. If an instruction that changes the flags comes in between, the decision is made from that instruction instead. This is one of the main sources of errors in flow control.

The status flags used by jumps are the following:

- **ZF** (zero flag): 1 if the result is zero.

- **CF** (carry flag): 1 if addition produces a carry out of the most significant bit, or subtraction needs a borrow.
- **SF** (sign flag): the most significant bit of the result; 1 if the result is negative in the signed interpretation.
- **OF** (overflow flag): 1 if the result does not fit in the destination's bit width in the signed interpretation.
- **PF** (parity flag): 1 if the low byte of the result has an even number of 1 bits.
- **AF** (auxiliary carry flag): 1 if there is a carry or borrow at bit 3. No jump tests this flag; AF is used only by the BCD adjust instructions (DAA, DAS, AAA, AAS).

Figure 7.5 summarizes the flow of a conditional jump. After the instruction that sets the flags, the processor reaches a fork. If the condition holds it goes to the label, and if not it goes to the next instruction. Every if in a high-level language is an instance of this fork.

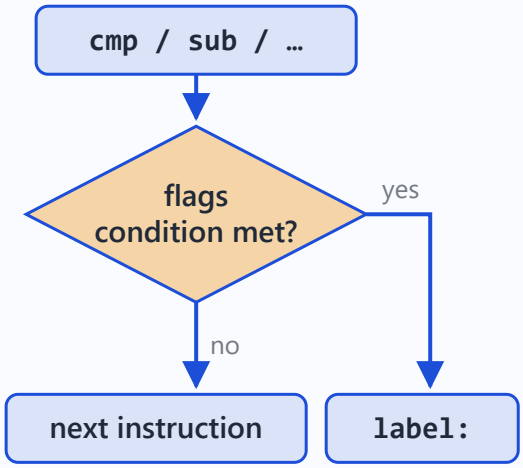


Figure 7.5: Flowchart of a conditional jump: after the instruction that sets the flags, if the condition is true the jump goes to the label; otherwise execution continues with the next instruction.

7.3.2 Jumps that test a single flag

The simplest conditional jumps look at a single flag. For each flag there are two instructions: one jumps when the flag is 1, the other when it is 0. The letter N in an instruction name means “not”: `JZ` jumps if zero, `JNZ` if not zero. Table 7.5 gives these instructions, their conditions, and their opposites.

Instruction	Meaning	Condition	Opposite
`JZ` , `JE`	jump if zero / equal	ZF = 1	`JNZ` , `JNE`
`JC` , `JB` , `JNAE`	jump if carry / below	CF = 1	`JNC` , `JNB` , `JAE`
`JS`	jump if the result is negative	SF = 1	`JNS`
`JO`	jump if signed overflow	OF = 1	`JNO`

Instruction	Meaning	Condition	Opposite
`JPE`, `JP`	jump if parity is even	PF = 1	`JPO`, `JNP`
`JNZ`, `JNE`	jump if not zero / not equal	ZF = 0	`JZ`, `JE`
`JNC`, `JNB`, `JAE`	jump if no carry / not below	CF = 0	`JC`, `JB`, `JNAE`
`JNS`	jump if the result is not negative	SF = 0	`JS`
`JNO`	jump if no overflow	OF = 0	`JO`
`JPO`, `JNP`	jump if parity is odd	PF = 0	`JPE`, `JP`

Table 7.5: Conditional jumps that test a single flag.

These instructions have different areas of use. `JZ/JNZ` are the most common: with them one tests whether a counter has reached zero, whether two values are equal, or whether the 0 byte at the end of a string has been reached. `JC/JNC` catch overflow in unsigned addition and a borrow in subtraction. Some DOS and BIOS services also report an error through CF, and `JC error` is written after the service call. `JS/JNS` look at the sign of a result. `JO/JNO` catch overflow in signed arithmetic. `JP/JNP` are rarely used; they are useful for jobs such as parity checking in serial communication.

Common mistake

The C in JC is the carry flag; overflow is OF. They are different concepts. CF reports a failure to fit in the unsigned interpretation, and OF a failure to fit in the signed interpretation. In the addition 7Fh + 01h = 80h, CF = 0 (127 + 1 = 128 fits in a byte), but OF = 1 (signed +127 + 1 = +128 does not fit in the range −128…127).

7.3.3 Encoding and synonyms

On the 8086 all conditional jumps have a fixed length: **2 bytes**. The first byte is the opcode and the second is the 8-bit signed displacement (rel8). A conditional jump can therefore go at most 128 bytes backward or 127 bytes forward relative to the instruction after it. Conditional jumps with a 16-bit displacement (the `0F 8x` opcodes) arrived only with the 80386; the 8086 does not have them.

The same condition can have more than one name. For example, the condition CF = 1 can be read in three ways: “carry set” (JC), “below” (JB, unsigned less than), and “not above or equal” (JNAE). These names are called **synonym** instructions. The assembler translates all of them to the same machine code; the processor does not know which name was written. The programmer chooses the name that best fits the meaning of the condition in that program. Writing `JE` when testing the equality of two numbers and `JZ` when testing whether a counter has reached zero keeps the code readable.

`70` JO	`71` JNO	`72` JB / JC / JNAE	`73` JAE / JNB / JNC
`74` JE / JZ	`75` JNE / JNZ	`76` JBE / JNA	`77` JA / JNBE
`78` JS	`79` JNS	`7A` JP / JPE	`7B` JNP / JPO

7C JL / JNGE	**7D** JGE / JNL	**7E** JLE / JNG	**7F** JG / JNLE
E0 LOOPNE / LOOPNZ	**E1** LOOPE / LOOPZ	**E2** LOOP	**E3** JCXZ

There is a pattern in the opcode sequence above. The conditional jumps all lie in the range 70h–7Fh and pair up two by two. In each pair the lowest bit of the opcode inverts the condition: 74h is JE while 75h is JNE, 72h is JB while 73h is JAE, 7Ch is JL while 7Dh is JGE. The upper bits select which flag or combination of flags is tested. The processor's condition circuit first computes this combination, then inverts the result if the lowest bit is 1. This pattern makes it easy to find the opposite of a condition: changing the last bit of the opcode is enough. The loop instructions also sit side by side, in the range E0h–E3h. The code of the short JMP is EBh.

7.3.4 Code trace: JNC = JNB = JAE

A small program shows that synonym instructions really produce the same machine code. The program contains three synonymous jumps in a row; all three go to the same label `a`.

```
org 100h
jnc a          ; 0100h: 73 07
jnb a          ; 0102h: 73 05
jae a          ; 0104h: 73 03
mov ax, 4      ; 0106h: B8 04 00
a: mov ax, 5   ; 0109h: B8 05 00
ret            ; 010Ch: C3
```

The opcode of all three jumps is 73h; all three test the condition CF = 0. Only their second bytes differ, because each stands at a different address and so has a different distance to the target. `jnc a` is at 0100h: the next address is 0102h, displacement 0109h − 0102h = 7. `jnb a` is at 0102h: displacement 0109h − 0104h = 5. `jae a` is at 0104h: displacement 0109h − 0106h = 3.

Step	Address	Instruction	CF	AX	IP (after)	Note
0	—	start	0	0000h	0100h	.COM start
1	0100h	`jnc a`	0	0000h	0109h	CF = 0 → jumps: 0102h + 07h
2	0109h	`mov ax, 5`	0	0005h	010Ch	the three instructions in between were not executed
3	010Ch	`ret`	0	0005h	0000h	return to the PSP

Table 7.6: Trace of the synonymous jumps program (CF = 0 at the start).

When the program starts, all the status flags are 0, CF included. The first instruction, `jnc a`, therefore jumps and IP = 0109h. `jnb`, `jae`, and `mov ax, 4` are never executed. AX = 0005h and the program ends. Had CF been 1 at the start (say, with an `STC` added at the top), none of the three

jumps would be taken. The processor would pass through the three instructions in turn, and AX would become 4 and then 5. The final value is the same either way, but the number of instructions executed differs: 3 when CF = 0, 6 when CF = 1.

7.3.5 Computing a jump code by hand

The jump code calculator in the slides finds the machine code of a jump instruction from its address and its target. The same computation can be done by hand in three steps. First determine the instruction length: JMP SHORT, all Jcc instructions, and the LOOP family are 2 bytes, and JMP NEAR is 3 bytes. Then compute the displacement: displacement = target − (address + length). Finally check whether the displacement fits the range and write it in two's complement.

Example 7.2

(a) A `JNE` at 0120h is to jump to 0100h. Next IP = 0122h; displacement = 0100h − 0122h = −22h. 22h = 0010 0010b; inverted 1101 1101b; plus 1: 1101 1110b = DEh. Code `75 DE`. The processor computes 0122h + FFDEh = 1 0100h → IP = 0100h.

(b) A `LOOP` at 0105h is to return to 0100h. Displacement = 0100h − 0107h = −7 = F9h. Code `E2 F9`.

(c) A `JE` at 0100h is to jump to 0190h. Displacement = 0190h − 0102h = 8Eh = 142. This number is greater than +127; as an 8-bit signed number, 8Eh would be read as −114. The short form cannot be used, and the 8086 has no long form of the conditional jump. The solution is to invert the condition and use a near JMP.

(d) A `JMP NEAR` at 0100h is to jump to 0190h. The instruction is 3 bytes, the next IP = 0103h, and the displacement = 008Dh. Code `E9 8D 00` (little-endian: low byte first).

If the result does not fit in the range −128…+127 in the short form, classic assemblers report a “relative jump out of range” error. This error says the displacement does not fit in one byte; the fix is to change the instruction or rearrange the code.

7.4 Comparison: CMP

7.4.1 The CMP instruction

A program often compares two values and decides according to the result: is the number greater than 100, is the character 'A', has the counter run out. The `CMP destination, source` instruction is used for this. CMP performs the subtraction that SUB does: destination − source. It does not store the result anywhere, however. It only sets the flags (CF, ZF, SF, OF, PF, AF) according to the result of the subtraction. Both operands are left unchanged.

Definition 7.5

Compare (CMP) is the instruction that performs the subtraction destination − source, discards the result, and only sets the flags. The flags are set by the same rules as in SUB. The conditional jump that follows CMP decides by looking at these flags.

CMP's operand rules are the same as SUB's. Register–register, register–memory, memory–register, and register-or-memory–constant combinations are valid. There cannot be two memory operands; `CMP [SI], [DI]` cannot be written. The two operands must be the same size. When memory is compared with a constant, the size must be specified: `CMP BYTE PTR [SI], 3`. If a variable name is used, the size follows from the variable's type: `cmp arr[si], 3` is a byte comparison because arr is defined with DB.

CMP's execution time depends on the operands. Register–register takes 3 clock cycles and register–constant takes 4. With a memory operand the effective address is computed first and memory is read in one bus cycle: register–memory takes 9 + EA, memory–constant 10 + EA cycles. A comparison and the conditional jump behind it together make up the cost of one decision: `CMP AL, BL` + a `JE` not taken is 3 + 4 = 7 cycles, and with the `JE` taken it is 3 + 16 = 19 cycles.

Instruction	Subtraction	ZF	CF	SF	OF	JE / JZ
`CMP 05h, 02h`	5 − 2 = 3 → 03h	0	0	0	0	not taken
`CMP 07h, 07h`	7 − 7 = 0 → 00h	1	0	0	0	taken

Table 7.7: Two simple CMP examples.

The notation in the table is conceptual; the 8086 has no instruction that compares two constants, so one operand must be a register or memory. In the first example the difference is not zero and ZF = 0. In the second example the difference is zero: ZF = 1, and `JE` or `JZ` is taken. That is all there is to an equality test: if two numbers are equal, their difference is zero.

7.4.2 CMP flags step by step

Equality looks only at ZF. Decisions of “greater” and “less” depend on how the numbers are interpreted. The same eight bits carry a value in the range 0–255 as an unsigned number and −128…127 as a signed number. CMP does not make this distinction; it subtracts the bits and writes into the flags the information needed for both interpretations. The answer for the unsigned interpretation is in CF and ZF, and the answer for the signed interpretation is in SF, OF, and ZF. Table 7.8 shows how `CMP AL, BL` sets the flags for five different inputs.

AL	BL	AL − BL	CF	ZF	SF	OF	PF	AF	Unsigned	Signed
05h (5)	02h (2)	03h	0	0	0	0	1	0	5 > 2: JA	5 > 2: JG
07h (7)	07h (7)	00h	0	1	0	0	1	0	equal: JE	equal: JE
05h (5)	07h (7)	FEh	1	0	1	0	0	1	5 < 7: JB	5 < 7: JL

AL	BL	AL − BL	CF	ZF	SF	OF	PF	AF	Unsigned	Signed
7Fh (127)	80h (128 / −128)	FFh	1	0	1	1	1	0	127 < 128: JB	127 > −128: JG
FBh (251 / −5)	03h (3)	F8h	0	0	1	0	0	0	251 > 3: JA	−5 < 3: JL

Table 7.8: Flags set by `CMP AL, BL` for five inputs, and the resulting jump decisions.

In the first row the difference is positive and no flag is raised except PF; PF = 1 because 03h = 0000 0011b contains two 1 bits. In the second row the difference is zero and ZF = 1. In both the signed and the unsigned view the numbers are equal.

In the third row a larger number is subtracted from a smaller one. The operation 5 − 7 needs a borrow, so CF = 1. The result is FEh: 254 unsigned, −2 signed. Because the most significant bit of the result is 1, SF = 1. Since −2 fits in eight bits, there is no overflow: OF = 0. The low nibble also borrows in 5 − 7, so AF = 1. FEh = 1111 1110b contains seven 1 bits; PF = 0.

The fourth and fifth rows are the really interesting cases. In these two rows the signed and the unsigned interpretation give different answers (Figures 7.6 and 7.7).

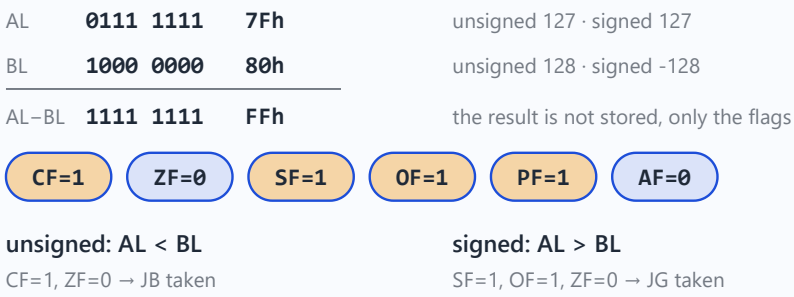


Figure 7.6: `CMP 7Fh, 80h` : in the unsigned view 127 < 128 (CF = 1, JB is taken); in the signed view 127 − (−128) = 255 does not fit in eight bits, so OF = 1, and since SF = OF, JG is taken.

In the operation 7Fh − 80h, the unsigned interpretation is 127 − 128; a borrow is taken and CF = 1. The signed interpretation is 127 − (−128) = +255. This value does not fit in the range −128…127, so OF = 1. The eight-bit result is FFh, and since its most significant bit is 1, SF = 1. SF alone would say “the result is negative, so AL is smaller”; that is wrong. The correct decision is made by reading SF and OF together.

Figure 7.7: `CMP FBh, 03h` : FBh is 251 unsigned and −5 signed. In the unsigned view 251 > 3 (CF = 0, JA is taken); in the signed view −5 < 3 (SF ≠ OF, JL is taken). The same bits lead to different decisions.

In the operation FBh − 03h = F8h there is no borrow; CF = 0. In the unsigned interpretation 251 > 3. In the signed interpretation −5 − 3 = −8. −8 fits in eight bits (F8h) and there is no overflow: OF = 0. The result is negative: SF = 1. Since SF ≠ OF, AL is smaller in the signed interpretation. The same CMP gives two opposite decisions depending on the type of the data. The processor cannot know which decision is the right one; the programmer says so by choosing the appropriate jump instruction.

7.4.3 Jumps for signed numbers

The jumps used for signed comparison are named after the English words for “greater” and “less”: **G**reater and **L**ess. These names are used on the assumption that the numbers are signed, in two's complement representation. Table 7.9 gives this family with its conditions.

Instruction	Meaning	Condition	Opposite
`JE` , `JZ`	A = B	ZF = 1	`JNE` , `JNZ`
`JNE` , `JNZ`	A ≠ B	ZF = 0	`JE` , `JZ`
`JG` , `JNLE`	A > B (greater)	ZF = 0 and SF = OF	`JNG` , `JLE`
`JL` , `JNGE`	A < B (less)	SF ≠ OF	`JNL` , `JGE`
`JGE` , `JNL`	A ≥ B	SF = OF	`JNGE` , `JL`
`JLE` , `JNG`	A ≤ B	ZF = 1 or SF ≠ OF	`JNLE` , `JG`

Table 7.9: Conditional jumps for signed numbers (after `CMP A, B`).

The conditions look confusing at a glance, but they all derive from one principle: in the signed interpretation, “A < B” is given by SF ≠ OF, and the rest follows. A ≥ B is the negation of A < B: SF = OF. A > B means “A ≥ B and A ≠ B”: SF = OF and ZF = 0. A ≤ B is the negation of A > B: ZF = 1 or SF ≠ OF. The synonyms are built with the same logic. JG is also written JNLE, meaning “not less or equal”.

7.4.4 Why SF = OF?

In the processor, `CMP A, B` is computed as A − B. If there is no overflow, the sign of the result is correct: if A < B the difference is negative and SF = 1. If there is overflow (OF = 1), the difference did not fit in eight or sixteen bits. The sign bit of a result that does not fit comes out inverted. The reason is that signed numbers are arranged on a circle (Figure 7.8): a value that goes past one end of the range enters from the other end. Adding 1 to 127 gives −128; subtracting 1 from −128 gives 127.

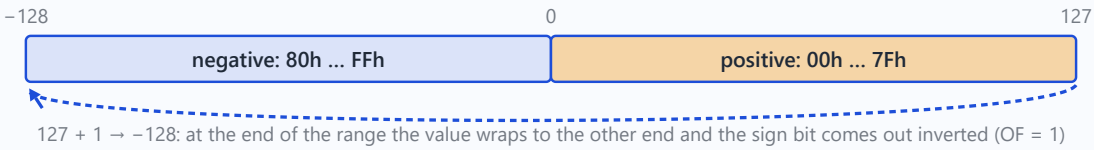


Figure 7.8: The signed byte range is −128 to 127. On overflow the value wraps to the other end of the range and the sign bit comes out inverted (OF = 1).

The true sign is therefore the exclusive OR of SF and OF: true sign = SF XOR OF. With no overflow (OF = 0) the true sign is SF itself. With overflow (OF = 1) the true sign is the inverse of SF. When SF ≠ OF the XOR gives 1: the true difference is negative and A < B. When SF = OF the XOR gives 0: the true difference is not negative and A ≥ B.

Theorem 7.1 (Signed comparison)

After `CMP A, B`, if A and B are interpreted as signed numbers in two's complement representation, then A < B ⇔ SF ≠ OF and A ≥ B ⇔ SF = OF.

Justification. The true difference d = A − B lies in the range −255…+255 for eight bits. If d is in the range −128…127, the result is shown correctly, OF = 0, and SF = (d < 0). If d > 127, the result appears as d − 256, which is negative: SF = 1, OF = 1; the true difference is positive. If d < −128, the result appears as d + 256, which is positive or zero: SF = 0, OF = 1; the true difference is negative. In all three cases (d < 0) = SF XOR OF.

A	B	A − B (8 bits)	True difference	SF	OF	Decision
5	7	FEh (−2)	−2	1	0	SF ≠ OF → 5 < 7
127 (7Fh)	−128 (80h)	FFh (−1)	+255, does not fit	1	1	SF = OF → 127 > −128
−128 (80h)	1	7Fh (+127)	−129, does not fit	0	1	SF ≠ OF → −128 < 1
−5 (FBh)	3	F8h (−8)	−8	1	0	SF ≠ OF → −5 < 3

Table 7.10: SF and OF together give the right decision (8 bits).

The second and third rows of the table involve overflow. In both, SF alone would give the wrong answer: in the second row it would say “negative, A is smaller”, and in the third row “positive, A is greater”. Evaluated together with OF, the decision is correct. In hardware this is very cheap: the

condition circuit of JL is a single XOR gate. For JLE, the output of this XOR is ORed with ZF.

> **Common mistake**
>
> Writing "jump if less" as `JS` after a signed comparison. JS looks only at SF and decides wrongly when there is overflow: after `MOV AL, 80h` / `CMP AL, 1`, SF = 0 and JS is not taken, although −128 < 1. The correct instruction is `JL`; since SF ≠ OF (0 ≠ 1), it is taken.

7.4.5 Jumps for unsigned numbers

The jumps used for unsigned comparison are named after the English words "above" and "below": **A**bove and **B**elow. In the unsigned interpretation the decision is much simpler, because there is no notion of overflow. In the subtraction A − B, if A < B a borrow is taken and CF = 1. So the "below" test is directly the carry flag: JB ≡ JC.

Instruction	Meaning	Condition	Opposite
`JE`, `JZ`	A = B	ZF = 1	`JNE`, `JNZ`
`JNE`, `JNZ`	A ≠ B	ZF = 0	`JE`, `JZ`
`JA`, `JNBE`	A > B (above)	CF = 0 and ZF = 0	`JNA`, `JBE`
`JB`, `JNAE`, `JC`	A < B (below)	CF = 1	`JNB`, `JAE`, `JNC`
`JAE`, `JNB`, `JNC`	A ≥ B	CF = 0	`JNAE`, `JB`, `JC`
`JBE`, `JNA`	A ≤ B	CF = 1 or ZF = 1	`JNBE`, `JA`

Table 7.11: Conditional jumps for unsigned numbers (after `CMP A, B`).

The condition of JA means "no borrow and the difference is not zero"; A is neither less than B nor equal to it. JBE is its opposite. JAE looks only at CF = 0; since no borrow is taken on equality either, "greater or equal" is satisfied automatically.

7.4.6 Above/Below or Greater/Less?

For each of the six comparison relations there are two instructions: one unsigned, one signed. For equality and inequality the two families use the same instruction, because equality means the bits are identical and does not depend on interpretation. Table 7.12 gives this mapping at a glance.

Comparison	Unsigned (A/B)	Signed (G/L)
A = B	`JE` / `JZ`	`JE` / `JZ`
A ≠ B	`JNE` / `JNZ`	`JNE` / `JNZ`
A > B	`JA` / `JNBE`	`JG` / `JNLE`
A ≥ B	`JAE` / `JNB`	`JGE` / `JNL`
A < B	`JB` / `JNAE`	`JL` / `JNGE`

Comparison	Unsigned (A/B)	Signed (G/L)
A ≤ B	`JBE` / `JNA`	`JLE` / `JNG`

Table 7.12: Choosing a jump by relation and data type.

The type of the data decides the choice. Addresses, counters, array indices, lengths, and ASCII codes are unsigned; use the A/B family with them. Values that can be negative, such as a temperature, the difference between two measurements, or a change of position, are signed; use the G/L family with them. An easy mnemonic: **A**bove / **B**elow is unsigned, **G**reater / **L**ess is signed. As long as the values stay in the range 0–127, the two families give the same result; the difference shows up only for values whose most significant bit is 1. That is why a wrong choice of family often goes unnoticed with small test data.

7.4.7 Code trace: same bits, JA and JG

The program below shows JA and JG making different decisions after the same CMP. Its second half also checks that JG works correctly in a comparison with overflow. The registers CL, CH, and DL serve as flags: if a jump is not taken, the corresponding register becomes 1.

```
MOV  AL, 0FBh    ; 251 or -5
MOV  BL, 03h
CMP  AL, BL      ; FBh - 03h
JA   a1          ; unsigned: 251 > 3 ?
MOV  CL, 1       ; if JA failed
a1:  JG  g1      ; signed: -5 > 3 ?
MOV  CH, 1       ; JG not taken
g1:  MOV AL, 7Fh ; 127
CMP  AL, 80h     ; 127 - (-128)
JG   g2          ; SF = OF ?
MOV  DL, 1       ; if JG failed
g2:  RET
```

Note the spelling `0FBh` in the source. A hexadecimal number that begins with a letter gets a 0 in front. If `FBh` were written, the assembler would take it for a name and report an error.

Step	Address	Instruction	Code	AL	CF ZF SF OF	Result
1	0100h	`MOV AL, 0FBh`	`B0 FB`	FBh	0 0 0 0	—
2	0102h	`MOV BL, 03h`	`B3 03`	FBh	0 0 0 0	BL = 03h
3	0104h	`CMP AL, BL`	`3A C3`	FBh	0 0 1 0	F8h, not stored
4	0106h	`JA a1`	`77 02`	FBh	0 0 1 0	taken → 010Ah
5	010Ah	`JG g1`	`7F 02`	FBh	0 0 1 0	not taken → 010Ch
6	010Ch	`MOV CH, 1`	`B5 01`	FBh	0 0 1 0	CX = 0100h
7	010Eh	`MOV AL, 7Fh`	`B0 7F`	7Fh	0 0 1 0	—

Step	Address	Instruction	Code	AL	CF ZF SF OF	Result
8	0110h	`CMP AL, 80h`	`3C 80`	7Fh	1 0 1 1	FFh, not stored
9	0112h	`JG g2`	`7F 02`	7Fh	1 0 1 1	taken → 0116h
10	0116h	`RET`	`C3`	7Fh	1 0 1 1	IP = 0000h

Table 7.13: Trace of the JA and JG program (CX and DX are 0000h at the start).

In the third step CMP computes FBh − 03h = F8h. There is no borrow (CF = 0), the result is not zero (ZF = 0), the most significant bit is 1 (SF = 1), and there is no overflow (OF = 0). In the fourth step `JA` finds the condition CF = 0 and ZF = 0 satisfied and jumps; in the unsigned interpretation 251 > 3. The line `MOV CL, 1` is skipped and CL stays 0. In the fifth step `JG` looks at the same flags. ZF = 0, but SF = 1 and OF = 0; since SF ≠ OF, it is not taken. In the signed interpretation −5 > 3 is false. In the sixth step CH becomes 1 and CX = 0100h. The same CMP has led to two different decisions.

In the second half of the program AL is set to 7Fh = 127 and compared with 80h. 7Fh − 80h = FFh. In the unsigned interpretation 127 < 128 and a borrow is taken: CF = 1. In the signed interpretation 127 − (−128) = 255 does not fit: OF = 1. The sign bit of the result is SF = 1. PF = 1, because FFh contains eight 1 bits. In the ninth step `JG` finds ZF = 0 and SF = OF (1 = 1) satisfied and jumps. 127 > −128 is true. SF alone would have said “less”. DL stays 0. When the program ends, CL = 0, CH = 1, DL = 0: in the signed interpretation −5 is not greater than 3, while 127 is greater than −128.

Figure 7.9: The ninth step of the JA and JG trace: after `CMP AL, 80h`, CF = 1, SF = 1, OF = 1. Since ZF = 0 and SF = OF, JG is taken.

7.4.8 Code trace: CMP + JE and CMP + JA

In real programs there are two separate blocks for the two outcomes of a comparison. The two short fragments below are the simplest form of this pattern. The fragments contain no `RET`, because they are meant to be placed inside a program. Instead of operations, the blocks contain only comments.

```
mov ax, 5
mov bx, 5
cmp ax, bx
je  equal_message
jmp not_equal_message
equal_message:          ; if equal
  jmp end_program
not_equal_message:      ; if not equal
end_program:
```

When the assembler places this fragment from 0100h, `cmp ax, bx` stands at 0106h (`3B C3`), `je` at 0108h (`74 02`), `jmp not_equal_message` at 010Ah (`EB 02`), and `jmp end_program` at 010Ch (`EB 00`). The trace is short:

1. `mov ax, 5` and `mov bx, 5` : AX = BX = 0005h.
2. `cmp ax, bx` : 5 − 5 = 0 → ZF = 1, CF = 0, SF = 0, OF = 0. PF = 1 (00h has zero 1 bits, and zero is even). AX does not change.
3. `je equal_message` : since ZF = 1 it jumps; IP = 010Ch.
4. The operations in the `equal_message` block are performed, and then `jmp end_program` jumps over the other block.

The JMP in the last step is critical. In assembly, blocks are separated only by labels; nothing tells the processor that a block has ended. Without `jmp end_program` , once the processor finished the equal block it would fall through to the next line, the not-equal block, and both blocks would execute. In this example both blocks are empty, so `jmp end_program` goes to the address right after itself and its code is `EB 00` : the displacement is zero. When the blocks are filled, the displacement grows by the length of the block. If the numbers were not equal, `je` would not be taken, and `jmp not_equal_message` would carry the flow to the not-equal block.

The second fragment builds the same pattern with the unsigned question “is it greater?”:

```
mov ax, 5
mov bx, 7
cmp ax, bx
ja  jump_above
jmp not_jump_above
jump_above:             ; if ax is greater
  jmp end_program
not_jump_above:         ; if ax is less/equal
end_program:
```

Here `cmp ax, bx` computes 5 − 7 = FFFEh, which is −2 in the signed interpretation. Since a borrow is taken, CF = 1; since the result is not zero, ZF = 0; since the most significant bit is 1, SF = 1. There is no overflow: OF = 0. In the low nibble 5 − 7 takes a borrow, so AF = 1. `ja` needs CF = 0 and ZF = 0; since

CF = 1 it is not taken. `jmp not_jump_above` carries the flow to the "less or equal" block. Had `jg` been written instead of `ja` in the same code, the result would not change either: SF = 1, OF = 0, and since SF ≠ OF, JG would not be taken. Because both values are positive and small, the two families give the same answer.

7.4.9 Code trace: compare and print

The program below compares two numbers. If they are equal it prints `y` on the screen, otherwise `n`. It prints the character with the DOS `INT 21h` service: when AH = 02h, DOS displays the character in DL. The structure of the program matters: the two branches only put a different value into DL and then merge at the common `prt` label. The printing code is not written twice.

```
org     100h
mov     al, 25          ; al = 25
mov     bl, 10          ; bl = 10
cmp     al, bl          ; compare al - bl
je      equal           ; jump if al = bl (zf = 1)
mov     dl, 'n'         ; al <> bl: print 'n'
jmp     prt             ; go to the print line
equal:
mov     dl, 'y'         ; al = bl: print 'y'
prt:
mov     ah, 2           ; DOS: print char in DL
int     21h
ret                     ; return to DOS
```

Step	Address	Instruction	Code	AX	DL	ZF CF	Screen
1	0100h	`mov al, 25`	`B0 19`	0019h	00h	0 0	
2	0102h	`mov bl, 10`	`B3 0A`	0019h	00h	0 0	
3	0104h	`cmp al, bl`	`3A C3`	0019h	00h	0 0	
4	0106h	`je equal`	`74 04`	0019h	00h	0 0	
5	0108h	`mov dl, 'n'`	`B2 6E`	0019h	6Eh	0 0	
6	010Ah	`jmp prt`	`EB 02`	0019h	6Eh	0 0	
7	010Eh	`mov ah, 2`	`B4 02`	0219h	6Eh	0 0	
8	0110h	`int 21h`	`CD 21`	026Eh	6Eh	0 0	n
9	0112h	`ret`	`C3`	026Eh	6Eh	0 0	n

Table 7.14: Trace of the compare-and-print program.

AL = 25 = 19h and BL = 10 = 0Ah are loaded. `cmp al, bl` computes 25 − 10 = 15 = 0Fh; ZF = 0. AF = 1 because the low nibble takes a borrow in 9 − A, and PF = 1 because 0Fh contains four 1 bits. `je equal` is not taken. `mov dl, 'n'` puts the ASCII code of the character 'n', 6Eh, into DL. `jmp prt`

skips the 'y' line; its displacement is 2, because the skipped `mov dl, 'y'` is two bytes. The two branches merge at the `prt` label.

`mov ah, 2` makes AH 02h; AX = 0219h. `int 21h` prints `n` on the screen. When returning from this service DOS also puts the character it printed into AL: AL = 6Eh, AX = 026Eh. The `ret` at the end of the program is required. Without it the processor would take the bytes after `int 21h` for instructions and keep executing them; those bytes are whatever happens to be in memory.

In this program `mov dl, 'y'` is never executed, but it still occupies memory. If the numbers were equal (for example with `mov bl, 25`), `je equal` would be taken, DL would be 79h ('y'), and `jmp prt` would never be executed. In comparison blocks, one branch is always the one that falls through, and the other is the one reached by the jump.

7.5 Counterparts of high-level constructs

7.5.1 The if/else pattern

The `if (condition) A else B` construct of a high-level language is built in assembly with two jumps. The first idea that comes to mind is to jump on the condition itself: if the condition is true, go to block A. But this reverses the order of the code; block B, which runs when the condition fails, has to be written right below the jump instruction. The tidier way is to jump on **the opposite of the condition**. If the condition fails, the jump skips over block A to block B. If the condition holds, no jump is made and the processor falls through into block A. This way the assembly code keeps the A–B order of the high-level code.

```
if (ax == bx) {
    ; block A
} else {
    ; block B
}
```

```
    cmp  ax, bx
    jne  not_eq       ; the OPPOSITE test
    ; block A
    jmp  after
not_eq:
    ; block B
after:
```

The pattern has two parts. The first is the jump on the opposite of the condition: `JNE` for `==`, `JGE` or `JAE` for `<`, `JLE` or `JBE` for `>`. The second is the `jmp after` at the end of block A. If this jump is forgotten, the processor enters block B as well when block A ends. In an if construct with no else part

the second jump is unnecessary; the opposite condition jumps straight to the end of the block.

High-level condition	Jump on unsigned data	Jump on signed data
`a == b`	`JNE`	`JNE`
`a != b`	`JE`	`JE`
`a < b`	`JAE`	`JGE`
`a <= b`	`JA`	`JG`
`a > b`	`JBE`	`JLE`
`a >= b`	`JB`	`JL`

Table 7.15: If conditions and the opposite jumps used to skip a block.

Compound conditions are built the same way. In an AND condition such as `if (a >= -10 && a <= 10)`, each subcondition is tested with its own CMP and opposite jump. When one of the subconditions fails, execution jumps straight out of the block. In an OR condition, when one of the subconditions holds, execution jumps straight into the block. This is the machine-level counterpart of "short-circuit" evaluation in high-level languages: the rest of a condition whose outcome is already settled is never tested.

7.5.2 The while, do-while, and for patterns

A loop is a jump made backward. In the `while (condition) body` construct the condition is tested before the body; if the condition is false from the start, the body never runs. In the assembly counterpart a label is placed at the top of the loop, the condition is tested with the opposite jump, and the end of the body jumps back to the top:

```
while (cx > 0) {     ; unsigned
    ; body
}
```

```
top:  cmp  cx, 0
      jbe  done       ; cx <= 0 → exit
      ; body
      jmp  top
done:
```

In the `do { body } while (condition)` construct the condition is tested after the body; the body runs at least once. Its counterpart is shorter, because a single conditional jump does both the test and the return:

```
top:  ; body
      cmp  cx, 0
      ja   top        ; if true, back to the top
```

This second form executes one fewer jump per iteration. The while form contains a conditional jump (not taken, 4 cycles) and a JMP (15 cycles) in every iteration. The do-while form contains only a conditional jump (taken, 16 cycles). That is why compilers usually turn a while loop into the form “test once first, then loop as a do-while”.

A counted loop such as `for (i = n; i > 0; i--)` is most naturally written on the 8086 with CX and the `LOOP` instruction. LOOP does the decrementing of CX, the comparison with zero, and the jump back in a single instruction. If the counter counts upward (`for (i = 0; i < n; i++)`), the counter is kept in another register. It is incremented with `INC` on each iteration and tested with `CMP` followed by `JB` or `JNE` .

7.5.3 Code trace: if/else and while

The program below uses the two patterns one after the other. First the construct `if (AX == BX) DX = 1; else DX = 2;` executes. Then, with CX = 2, the loop `while (CX > 0) { DX += CX; CX--; }` runs.

```
MOV  AX, 12
MOV  BX, 30
CMP  AX, BX
JNE  not_eq      ; OPPOSITE test
MOV  DX, 1       ; block A: AX = BX
JMP  after
not_eq: MOV DX, 2 ; block B: AX ≠ BX
after: MOV CX, 2  ; while (CX > 0)
top: CMP CX, 0
JBE  done         ; CX ≤ 0 → exit
ADD  DX, CX      ; body
DEC  CX
JMP  top
done: RET
```

Step	Address	Instruction	Code	CX	DX	ZF CF	Decision
1–2	0100h	`MOV AX, 12` / `MOV BX, 30`	`B8 0C 00` / `BB 1E 00`	0000h	0000h	0 0	—
3	0106h	`CMP AX, BX`	`3B C3`	0000h	0000h	0 1	12 − 30: not equal, borrow
4	0108h	`JNE not_eq`	`75 05`	0000h	0000h	0 1	taken → 010Fh

Step	Address	Instruction	Code	CX	DX	ZF CF	Decision
5	010Fh	`MOV DX, 2`	`BA 02 00`	0000h	0002h	0 1	block B
6	0112h	`MOV CX, 2`	`B9 02 00`	0002h	0002h	0 1	—
7–8	0115h	`CMP CX, 0` / `JBE done`	`83 F9 00` / `76 05`	0002h	0002h	0 0	2 > 0: not taken
9–10	011Ah	`ADD DX, CX` / `DEC CX`	`03 D1` / `49`	0001h	0004h	0 0	body, iteration 1
11–13	011Dh	`JMP top` / `CMP` / `JBE`	`EB F6`	0001h	0004h	0 0	1 > 0: not taken
14–15	011Ah	`ADD DX, CX` / `DEC CX`	—	0000h	0005h	1 0	body, iteration 2
16–18	011Dh	`JMP top` / `CMP` / `JBE`	—	0000h	0005h	1 0	0 = 0: taken → 011Fh
19	011Fh	`RET`	`C3`	0000h	0005h	1 0	IP = 0000h

Table 7.16: Trace of the if/else and while program (AX = 000Ch and BX = 001Eh throughout).

In the third step `CMP AX, BX` computes 12 − 30 = −18; the 16-bit result is FFEEh. The result is not zero (ZF = 0), a borrow was taken (CF = 1), and the sign bit is 1 (SF = 1). In the fourth step `JNE not_eq` finds the opposite condition satisfied and jumps over block A. `MOV DX, 1` and `JMP after` are not executed. In the fifth step block B makes DX = 2. Block B is the last block and falls straight into the `after` label; no JMP is needed on this path.

In the sixth step CX = 2 and the loop begins. `CMP CX, 0` computes 2 − 0 = 2: CF = 0, ZF = 0. `JBE done` needs CF = 1 or ZF = 1; it is not taken and the body is entered. The body makes DX = 2 + 2 = 4 and CX = 1. `JMP top` returns to the top and the condition is tested again: 1 > 0, not taken. In the second iteration DX = 4 + 1 = 5 and CX = 0. In this iteration ZF = 1 because `DEC CX` made the result zero, but the decision is still made at the top. `CMP CX, 0` computes 0 − 0 = 0: ZF = 1, CF = 0. `JBE done` is taken. The result is DX = 5: the 2 from block B, plus 2, plus 1.

The code of `CMP CX, 0` is `83 F9 00`. The opcode 83h compares a 16-bit operand with a sign-extended 8-bit constant, so the constant takes 1 byte instead of 2. The `JMP top` at the end of the loop is at 011Dh and is two bytes: displacement 0115h − 011Fh = −0Ah = F6h, code `EB F6`.

This loop is a while loop; the condition is tested before the body. Had it started with CX = 0, the body would never have run. The LOOP instruction, in contrast, tests the condition after the body; this is the important difference between the two constructs.

The first rows of Table 7.16 show CX = 0000h. The trace in the slides is made under the same assumption. In reality, when emu8086 loads a .COM program it puts the program's byte count into CX. This program is 32 bytes (0100h–011Fh), so in emu8086 the program starts with CX = 0020h. The

result is not affected, because CX is written by `MOV CX, 2` before it is first read. The general rule is: never rely on initial register values; load every register with a value before using it.

7.5.4 Conditional jumps out of range

On the 8086, conditional jumps exist only in the short form: −128…+127 bytes relative to the next instruction. When one has to skip over a long if block or return to the top of a long loop, this range may not be enough (Figure 7.10).

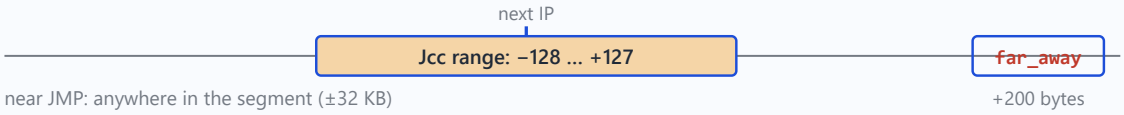


Figure 7.10: A conditional jump reaches between −128 and +127 bytes around the next IP. A target 200 bytes away lies outside this window; a near JMP, however, goes anywhere within the segment.

The solution is to invert the condition, hop over the next instruction with a short jump, and reach the far target with a near JMP:

```
; wrong: target too far
cmp al, 0
je  far_away        ; out of range!
; ... 200 bytes of code ...
far_away:
```

```
; right: opposite test + near JMP
cmp al, 0
jne cont            ; opposite test, short
jmp far_away        ; near JMP (E9)
cont:
; ... 200 bytes of code ...
far_away:
```

In the second version `jne cont` jumps forward only 3 bytes and fits the short range with ease. If AL = 0, JNE is not taken and the processor falls into the near JMP, which reaches anywhere in the segment. If AL ≠ 0, JNE passes over the JMP to the `cont` label. The behavior is the same as in the first version; only the instruction pair is 5 bytes instead of 2.

This transformation can be applied to any conditional jump, because every condition has an opposite, found by flipping the last bit of the opcode. Classic assemblers (MASM in 8086 mode, for instance) report a “relative jump out of range” error when they see an out-of-range Jcc and leave the transformation to the programmer. The emu8086 assembler, on the other hand, performs the same

transformation itself and notes it with a warning. For example, a `je far_away` at 0102h with the target at 01CFh is encoded as `75 03 E9 C8 00`: `JNE +3` (hopping over the three bytes of the JMP) and `JMP 01CFh` (0107h + 00C8h = 01CFh). Whichever assembler is used, what happens at machine level is the same. The program is also 3 bytes longer than it looks in the source.

> **Common mistake**
>
> Trying to fix the range error by writing the synonym `JZ` in place of `JE`, or by writing `JE SHORT`. A synonym translates to the same machine code and has the same range. SHORT does not widen the range; it only forces the short form. A loop instruction such as `LOOPE` is no solution either: it too is 2 bytes with rel8, and it changes CX besides.

7.6 Loop instructions

7.6.1 The LOOP family

Counted loops are written so often that the 8086 offers separate instructions for them. All of these instructions use the **CX** register as the counter. That is why CX is called the count register. Table 7.17 summarizes the family.

Instruction	Code	Operation and jump condition
`LOOP`	`E2`	Decrement CX by 1; jump to the label if CX ≠ 0.
`LOOPE`	`E1`	Decrement CX by 1; jump if CX ≠ 0 and equal (ZF = 1).
`LOOPZ`	`E1`	Same as LOOPE: jump if CX ≠ 0 and ZF = 1.
`LOOPNE`	`E0`	Decrement CX by 1; jump if CX ≠ 0 and not equal (ZF = 0).
`LOOPNZ`	`E0`	Same as LOOPNE: jump if CX ≠ 0 and ZF = 0.
`JCXZ`	`E3`	Jump to the label if CX = 0 (CX is not decremented).

Table 7.17: Loop instructions (all 2 bytes: opcode + rel8).

The family has three common properties. First, the counter is always CX; no other register can be chosen. Second, the LOOP family **does not change the flags** when it decrements CX. Even when CX falls to zero, ZF is not raised. This is an important difference from the `DEC CX` instruction, which does the same job. Third, all of them are 2 bytes and use rel8; the loop body can reach back at most 128 bytes. If the body is longer, the loop is written with `DEC CX`, `JZ`, and a near `JMP`.

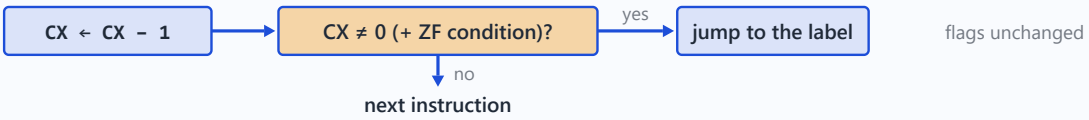


Figure 7.11: The LOOP family first decrements CX by one. If CX is not zero (and, for LOOPE/LOOPNE, the ZF condition also holds), it jumps to the label; otherwise it moves on to the next instruction. The flags do not change.

7.6.2 How LOOP works

In `LOOP label` the order of operations matters: CX is decremented first, and *then* tested for zero. LOOP therefore builds a do-while loop. The body executes at least once before reaching LOOP. In a loop that starts with CX = n, the body runs exactly n times: the jump back is taken with CX at n − 1, n − 2, …, 1, and the loop is left at 0.

```
      mov  cx, n
label:
      ; loop body
      loop label
```

In effect LOOP resembles the pair `DEC CX` and `JNZ label`, but it preserves the flags. LOOPE and LOOPNE keep the loop going only while one more condition is met. LOOPE (LOOPZ) jumps while CX ≠ 0 and ZF = 1; it continues as long as equal items are found and stops at the first difference. LOOPNE (LOOPNZ) jumps while CX ≠ 0 and ZF = 0; it continues as long as items are unequal and stops when the wanted value is found. These two instructions are designed for search loops of the form “at most n times, but earlier if the condition changes”. ZF is set by a CMP in the loop body.

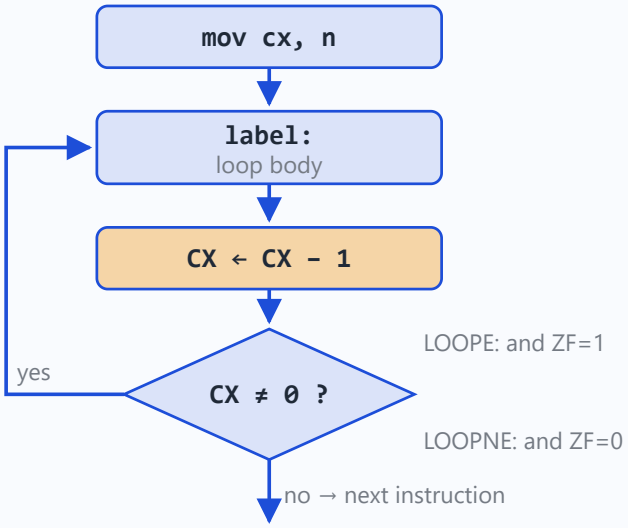


Figure 7.12: LOOP flowchart: the body executes, CX is decremented by one, and if CX is not zero control returns to the body; if it is zero, execution continues with the next instruction. For LOOPE and LOOPNE the ZF condition is also required to go back.

This order has a trap. If the loop is entered with CX = 0, the body runs once, and then LOOP decrements CX. 0 − 1 = FFFFh. Since CX is not zero, the jump back is taken. The loop continues until CX falls to zero again, and the body runs 65,536 times in all. If the iteration count is computed at run time and can be zero, this case must be prevented specifically. The remedy is called JCXZ.

Definition 7.6

A **loop** is the repetition of a group of instructions until a condition is met. In a **counted loop** the number of iterations is kept in a counter register; the LOOP family of the 8086 uses CX as the counter, decrements CX on every iteration, and leaves the loop when CX reaches zero.

7.6.3 Code trace: LOOP

The simplest LOOP example is a loop with an empty body. The program shows only the effect of LOOP on CX.

```
org 100h
mov cx, 5       ; CX = repeat count
again:
; loop body
loop again      ; if CX ≠ 0 go to again
jmp quit        ; loop done: go to quit
quit:
ret
```

`mov cx, 5` is three bytes at 0100h (`B9 05 00`). The `again` label points to 0103h. Since the body is empty, `loop again` is also at 0103h; its code is `E2 FE`. The displacement FEh = −2: the LOOP instruction jumps to itself. `jmp quit` is at 0105h and its code is `EB 00`, because `quit` is the next line anyway.

Iteration	Instruction	CX (after)	Decision	IP (after)
—	`mov cx, 5`	0005h	—	0103h
1	`loop again`	0004h	≠ 0, taken	0103h
2	`loop again`	0003h	≠ 0, taken	0103h
3	`loop again`	0002h	≠ 0, taken	0103h
4	`loop again`	0001h	≠ 0, taken	0103h
5	`loop again`	0000h	= 0, exits	0105h
—	`jmp quit`	0000h	—	0107h
—	`ret`	0000h	—	0000h

Table 7.18: Trace of the LOOP example.

The body (empty here) runs five times and LOOP is executed five times. The first four LOOPs jump and the fifth does not. The duration of the loop can be computed from Table 7.4: 4 × 17 + 5 = 73 clock cycles. When the program ends, CX = 0000h. The `jmp quit` after the loop is unnecessary but harmless; a jump with displacement 0 spends 15 cycles and flushes the queue.

7.6.4 Code trace: nested loops

Since the counter of the LOOP family is a single register, two nested LOOPs share the same CX. The moment the inner loop loads its own count into CX, the outer loop's counter is lost. The solution is to push the outer counter onto the stack before the inner loop and pop it back after the inner loop. The program below uses this method. The outer loop runs 5 times; in each outer iteration the screen gets a `1`, and then the inner loop prints five `2` characters. BX holds the total number of steps.

```
org 100h
mov bx, 0      ; total steps
mov cx, 5      ; outer counter
k1:
 add bx, 1
 mov al, '1'
 mov ah, 0eh
 int 10h       ; print '1'
 push cx       ; save outer
 mov cx, 5     ; inner counter
k2:
  add bx, 1
  mov al, '2'
  mov ah, 0eh
  int 10h      ; print '2'
  loop k2      ; inner loop
 pop cx        ; restore outer
loop k1        ; outer loop
ret
```

The characters are printed with the BIOS `INT 10h` service. When AH = 0Eh, BIOS writes the character in AL at the cursor position and advances the cursor; this service is called teletype output. The service uses BH as the page number. Since BX does not exceed 256 in this program, BH is always 0 and the output goes to page 0.

Step	Instruction	BX	CX	SP	Stack	Screen
start	`mov bx, 0` / `mov cx, 5`	0000h	0005h	FFFEh	[FFFEh] = 0000h	
outer 1	`add bx, 1` … `int 10h`	0001h	0005h	FFFEh	[FFFEh] = 0000h	1
outer 1	`push cx`	0001h	0005h	FFFCh	[FFFCh] = 0005h	1
inner 1	`mov cx, 5` … `int 10h`	0002h	0005h	FFFCh	[FFFCh] = 0005h	12
inner 1	`loop k2`	0002h	0004h	FFFCh	[FFFCh] = 0005h	12
inner 5	`loop k2` (exits)	0006h	0000h	FFFCh	[FFFCh] = 0005h	122222
outer 1	`pop cx`	0006h	0005h	FFFEh	[FFFEh] = 0000h	122222
outer 1	`loop k1`	0006h	0004h	FFFEh	[FFFEh] = 0000h	122222
end of outer 2	`loop k1`	000Ch	0003h	FFFEh	[FFFEh] = 0000h	122222 ×2
end of outer 3	`loop k1`	0012h	0002h	FFFEh	[FFFEh] = 0000h	122222 ×3
end of outer 4	`loop k1`	0018h	0001h	FFFEh	[FFFEh] = 0000h	122222 ×4
end of outer 5	`loop k1` (exits)	001Eh	0000h	FFFEh	[FFFEh] = 0000h	122222 ×5
end	`ret`	001Eh	0000h	0000h	empty	122222 ×5

Table 7.19: Trace of the nested loops (selected steps).

When the program starts, SP = FFFEh and the top of the stack holds the return address 0000h that DOS put there. In the first outer iteration BX = 1 and `1` is printed on the screen. AL = '1' = 31h and AH = 0Eh. `push cx` pushes the value of the outer counter, 5, onto the stack. SP decreases by two to FFFCh; [FFFCh] = 0005h. In little-endian order, FFFCh holds the byte 05h and FFFDh holds 00h.

The inner loop sets CX to 5 again. The value of the outer counter now lives only on the stack. The inner loop runs five times, incrementing BX and printing a `2` on each pass. The LOOP of the fifth inner iteration brings CX to zero and leaves the loop. At this point BX = 6 and CX = 0. `pop cx` restores the 0005h on the stack into CX, and SP = FFFEh. `loop k1` makes CX 4 and returns to the top of the outer loop.

Each outer iteration adds 1 + 5 = 6 to BX. After five iterations BX = 30 = 001Eh. The screen shows `122222` five times, 30 characters in all. The last LOOP brings CX to zero and the loop is left. Because the numbers of `push` and `pop` are equal, the stack is back in its initial state. `ret` takes the 0000h on top of the stack into IP, SP = 0000h, and the program ends.

The inner loop's `loop k2` is at 011Ch and its code is `E2 F5`: next address 011Eh, target k2 = 0113h, displacement −0Bh = F5h. The outer loop's `loop k1` is at 011Fh and its code is `E2 E5`: next address 0121h, target k1 = 0106h, displacement −1Bh = E5h. The body of the outer loop is 27 bytes, still far from the rel8 limit (128).

> **Common mistake**
>
> Forgetting the `push cx` / `pop cx` pair in a nested loop. When the inner loop ends, CX = 0. The outer loop's LOOP decrements this zero to FFFFh, and the outer loop runs tens of thousands of times instead of 5. Writing only one of the pair is even worse. The stack balance is broken, and the `ret` at the end of the program takes a wrong value for the return address.

7.6.5 Code trace: searching with LOOPNE

LOOPNE is made to order for searching a value in an array. The loop ends in one of two cases: when the value is found (ZF = 1), or when all elements have been examined (CX = 0). The program below searches for the value 3 in a five-element array.

```
org 100h
mov cx, 5           ; element count
mov si, -1          ; first inc makes it 0
scan:
inc si              ; next element
cmp arr[si], 3      ; is it 3?
loopne scan         ; go on if CX≠0, ZF=0
jne notfound        ; ZF=0 → not found
mov al, 1           ; found, position in SI
ret
notfound:
mov al, 0           ; not found
ret
arr db 7, 1, 3, 9, 4
```

Initializing SI to −1 (FFFFh) is a subtlety. The first instruction of the loop is `inc si`, which makes SI 0. SI thus points at the index of the element being examined on every pass, and when the loop is left it carries the position of the element found. The instruction `cmp arr[si], 3` is 5 bytes (`80 BC 16 01 03`): opcode, ModR/M, the 16-bit address of the array (0116h), and the constant. Because the array is defined with DB, the comparison is byte-sized.

Iteration	SI	arr[SI]	ZF CF SF after CMP	CX after LOOPNE	Decision
1	0000h	7	0 0 0	0004h	CX ≠ 0, ZF = 0 → continue
2	0001h	1	0 1 1	0003h	CX ≠ 0, ZF = 0 → continue
3	0002h	3	1 0 0	0002h	ZF = 1 → exit

Table 7.20: Trace of the search with LOOPNE (arr = 7, 1, 3, 9, 4; searching for 3).

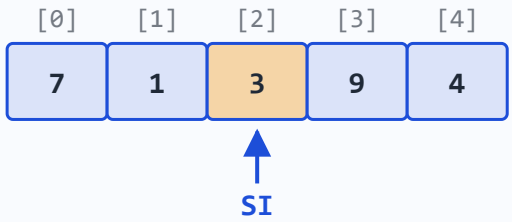


Figure 7.13: In the third iteration SI = 2 and arr[2] = 3: the comparison sets ZF = 1, and LOOPNE leaves the loop.

In the first iteration `inc si` takes SI from FFFFh to 0000h. This increment gives a zero result and so sets ZF = 1, but the CMP that follows immediately sets the flags again. 7 − 3 = 4 is not zero: ZF = 0. LOOPNE makes CX 4; since CX ≠ 0 and ZF = 0, it continues the search. In the second iteration arr[1] = 1. For 1 − 3 = −2 (FEh) a borrow is taken; CF = 1 and SF = 1. ZF is 0 again and the loop continues.

In the third iteration arr[2] = 3, and 3 − 3 = 0 sets ZF = 1. LOOPNE decrements CX again, to 2. CX is not zero, but since ZF = 1 the jump is not taken and the loop is left. Because LOOPNE does not change the flags, ZF = 1 is preserved. The `jne notfound` after the loop looks at exactly this flag. Since ZF = 1 it is not taken, and `mov al, 1` records the "found" result. SI = 2 is the position of the element found. The loop has ended after three iterations instead of five.

It matters why ZF is tested after the loop. The loop can end for two reasons, and looking at CX does not tell them apart: if the value is found at the last element, CX falls to zero too. The result of the last comparison, on the other hand, sits in ZF, and LOOPNE does not disturb it. If the array did not contain 3, after five iterations CX = 0 and ZF = 0; `jne notfound` would be taken and AL would be 0. This structure is a direct benefit of the LOOP family preserving the flags.

7.6.6 JCXZ; LOOP mu, DEC–JNZ mi?

The `JCXZ label` instruction jumps if CX = 0; it neither decrements CX nor looks at the flags. Its main use is as protection against a zero-iteration loop. If the iteration count is computed at run time, a JCXZ is placed in front of the loop:

```
      mov  cx, n      ; n known at run time
      jcxz skip       ; skip if CX = 0
again:
      ; body
      loop again
skip:
```

Without JCXZ, with CX = 0 the LOOP first makes CX FFFFh and the body runs 65,536 times. With JCXZ, the loop is jumped over in this case. The instruction tests whether CX is zero without another instruction having to look at CX; a `CMP CX, 0` that compares CX with zero would disturb the flags, while JCXZ does not.

A counted loop can also be written with the `DEC` and `JNZ` pair instead of LOOP. The differences between the two forms are collected in Table 7.21.

Feature	`LOOP`	`DEC CX + JNZ`
Size	2 bytes	1 + 2 = 3 bytes
Flags	unchanged	ZF, SF, OF, PF, AF change (CF is preserved)
Counter	CX only	any register
Extra condition	LOOPE / LOOPNE	a different Jcc can be chosen
Timing on the 8086 (taken / not taken)	17 / 5 cycles	2 + 16 = 18 / 2 + 4 = 6 cycles

Table 7.21: Comparison of LOOP and DEC CX + JNZ.

On the 8086, LOOP is both shorter and slightly faster. In a ten-iteration loop, LOOP spends 9 × 17 + 5 = 158 cycles with nine jumps and one exit. The DEC–JNZ pair spends 10 × 2 + 9 × 16 + 4 = 168 cycles. On the other hand DEC–JNZ is more flexible: a register other than CX can serve as the counter. This matters in loops that use CX for something else (for example as a shift count or the counter of a string instruction). On later x86 processors the LOOP instruction became slow and compilers stopped using it; for the 8086, however, the situation is the opposite.

The flag difference matters too. `DEC` sets the status flags other than CF according to the result. Information kept in CF throughout the loop (for example the carry of a multiword addition) is not affected by DEC. But information kept in ZF or SF is destroyed. LOOP touches no flag. In loops where the carry is carried from iteration to iteration, such as multiword addition, both forms preserve CF, so either can be used.

7.6.7 Code trace: JCXZ guard and DEC–JNZ

The program below contains two loops. The first loop is run with n = 0, and JCXZ guards it. The second loop runs with n = 3 in the DEC–JNZ form and uses BX as the counter. Before the second loop CF is set to 1; we observe that CF is preserved throughout the loop.

```
MOV  CX, 0       ; n = 0
MOV  AX, 0
JCXZ skip        ; CX = 0 → skip
again: INC AX
LOOP again
skip: MOV BX, 3   ; n = 3, counter BX
STC              ; CF = 1
again2: INC AX
DEC  BX          ; CF is kept
JNZ  again2
RET
```

Step	Instruction	AX	BX	CX	ZF	CF	Decision
1–2	`MOV CX, 0` / `MOV AX, 0`	0000h	0000h	0000h	0	0	—
3	`JCXZ skip` (`E3 03`)	0000h	0000h	0000h	0	0	CX = 0 → taken, 010Bh
4	`MOV BX, 3`	0000h	0003h	0000h	0	0	—
5	`STC`	0000h	0003h	0000h	0	1	—
6–8	`INC AX` / `DEC BX` / `JNZ`	0001h	0002h	0000h	0	1	taken (iteration 1)
9–11	`INC AX` / `DEC BX` / `JNZ`	0002h	0001h	0000h	0	1	taken (iteration 2)
12–14	`INC AX` / `DEC BX` / `JNZ`	0003h	0000h	0000h	1	1	not taken (iteration 3)
15	`RET`	0003h	0000h	0000h	1	1	IP = 0000h

Table 7.22: Trace of the JCXZ and DEC–JNZ program.

In the third step JCXZ sees that CX = 0 and jumps over the loop. `INC AX` and `LOOP again` are never executed; AX stays 0. Without the guard the body would run 65,536 times and AX would end up 0000h again (65,536 = 1 0000h does not fit in 16 bits). The result would look the same, but the processor would have gone around 65,536 times for nothing. Had the body modified a memory array instead of AX, the result would be a disaster.

In the second loop the counter is BX; LOOP could not do this. `STC` sets CF to 1. In each iteration `INC AX` and `DEC BX` set the flags, but neither touches CF. In the first iteration BX = 2 and ZF = 0, and `JNZ` jumps back. In the second iteration BX = 1. In the third iteration `DEC BX` makes BX 0 and ZF = 1. `JNZ` is not taken and the loop ends. In the end AX = 3 and CF = 1 is preserved. When `INC AX` makes AX 3 in the third iteration, PF = 1 (0000 0011b contains two 1 bits); this is set again by DEC. LOOP does not change flags either, but it works only with CX.

7.7 Example programs

7.7.1 Sum of array elements

Summing the elements of an array is the most typical use of a counted loop. The program below adds up a five-byte array and writes the result to the variable `m`. CX holds the element count, BX the index, and AL the sum.

```
org 100h
mov cx, 5             ; element count
mov al, 0             ; sum (AL) = 0
mov bx, 0             ; BX = index
next:
add al, vector[bx]    ; add element
inc bx                ; next element
loop next             ; until CX = 0
mov m, al             ; store in m
ret
vector db 5, 4, 5, 2, 1
m db 0
```

The data is defined after `ret`. Because the program ends with `ret`, the processor never reaches the data bytes and never mistakes them for instructions. When the assembler lays out the code, vector sits at 0113h and m at 0118h. The instruction `add al, vector[bx]` is 4 bytes (`02 87 13 01`): the ModR/M byte 87h selects the “[BX] + 16-bit displacement” form, and the displacement is the address of vector. `loop next` is at 010Dh; its code is `E2 F9` and the displacement is 0108h − 010Fh = −7.

Iteration	Added	AL (decimal)	AL	BX	CX	LOOP
start	—	0	00h	0000h	0005h	—
1	vector[0] = 5	5	05h	0001h	0004h	taken

Iteration	Added	AL (decimal)	AL	BX	CX	LOOP
2	vector[1] = 4	9	09h	0002h	0003h	taken
3	vector[2] = 5	14	0Eh	0003h	0002h	taken
4	vector[3] = 2	16	10h	0004h	0001h	taken
5	vector[4] = 1	17	11h	0005h	0000h	exits

Table 7.23: Trace of the array sum (each row is the end of one iteration).

Figure 7.14: The last iteration of the array sum: with BX = 4, vector[4] = 1 is added and AL = 11h.

Three instructions execute in each iteration. `add al, vector[bx]` adds the element to AL; the effective address is 0113h + BX and the segment is DS. `inc bx` increments the index by one, because the elements are single bytes. For a word array the index would advance by two. `loop next` decrements CX and returns to the top if it is not zero. After the fifth iteration CX = 0 and the loop is left. `mov m, al` writes the result to address 0118h: m = 11h = 17 = 5 + 4 + 5 + 2 + 1.

The ADD in the loop body sets the flags again on every iteration, but LOOP never reads or changes them. For example, in the fourth iteration the addition 0Eh + 02h = 10h produces a carry out of the low nibble and AF = 1. No instruction uses this information. The case where the sum does not fit in a byte is reported by CF. In this array the sum is 17, and CF is never 1 in any iteration. Had the elements been larger, AL could have overflowed; this case is examined in one of the exercises.

7.7.2 Printing Hello on the screen

The program below prints a string marked with a terminating 0 byte on the screen, character by character. The number of iterations is not known in advance; the loop ends when it meets the 0 byte. That is why CMP, JE, and JMP are used instead of LOOP. This structure is a while loop: the condition is tested at the start of every iteration.

```
org     100h
jmp     start
msg     db 'Hello', 0
start:
mov     si, 0           ; source index = 0
next_char:
mov     al, msg[si]     ; get the char
cmp     al, 0           ; compare with 0
je      stop            ; stop if 0
mov     ah, 0eh         ; teletype function
int     10h             ; print char in AL
inc     si              ; next character
jmp     next_char       ; loop back
stop:
ret
```

In this program the data sits among the code. msg, right behind the first instruction of the program, holds the bytes 48h 65h 6Ch 6Ch 6Fh 00h starting at 0102h. The first instruction, `jmp start` (`EB 06`), jumps over these six bytes to 0108h. Without the JMP the processor would decode the byte 48h at 0102h as an instruction. 48h is the code of `DEC AX`; the bytes that follow would also be decoded like other instructions, and what the program does would be unpredictable.

Iteration	SI (at start)	AL = msg[SI]	ZF	je stop	Screen
1	0000h	48h 'H'	0	not taken	H
2	0001h	65h 'e'	0	not taken	He
3	0002h	6Ch 'l'	0	not taken	Hel
4	0003h	6Ch 'l'	0	not taken	Hell
5	0004h	6Fh 'o'	0	not taken	Hello
6	0005h	00h	1	taken	Hello

Table 7.24: Trace of the Hello program (each row is the end of one iteration).

In each iteration `mov al, msg[si]` (`8A 84 02 01`) takes the next character into AL. `cmp al, 0` (`3C 00`) compares it with zero. If the character is not zero, ZF = 0 and `je stop` is not taken. With AH = 0Eh and `INT 10h` the character is printed on the screen. `inc si` increments the index and `jmp next_char` (`EB F1`) returns to the top. This JMP is at 0118h; the displacement is 010Bh − 011Ah = −0Fh = F1h.

In the sixth iteration msg[5] = 0 is read. Since 0 − 0 = 0, ZF = 1 and `je stop` is taken. The screen shows `Hello` and SI = 5. `ret` ends the program. At the end of the program AX = 0E00h: AH keeps the 0Eh of the last iteration, and AL is the 00h read last.

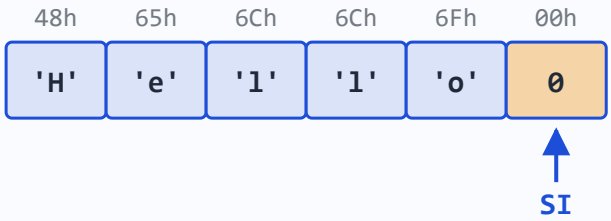


Figure 7.15: The Hello string in memory: six bytes, the last one 00h. With SI = 5, the 0 byte that is read ends the loop.

A string marked at the end with zero is called a **null-terminated string**. In this scheme the length is not stored separately; the string continues up to the 0 byte. The C language uses the same scheme. The DOS `INT 21h` /09h service, on the other hand, uses a different marker: the end of the string is the '$' character. What the two schemes have in common is that the loop ends by a condition, not by a counter.

Each iteration of the program executes seven instructions, two of which are flow control instructions. A `je` not taken spends 4 clock cycles and a `jmp` taken spends 15. Had the loop been rewritten in do-while form (with the comparison and the conditional jump back at the end), each iteration would have one JMP fewer. For a short string the difference is negligible; for a very long text it makes up a significant part of the total time.

7.7.3 Palindrome check: the idea

A palindrome is a sequence that reads the same from the front and from the back: `123321`, `kayak`, `ey edip adanada pide ye` (if spaces are not counted). To tell whether a sequence is a palindrome, two pointers are used. DI points at the beginning of the sequence and SI at its end. At each step two characters are compared; then DI moves one forward and SI one back. If one of the pairs differs, the sequence is not a palindrome. If all the pairs match, it is a palindrome.

Comparisons up to half the length are enough. In a six-character sequence there are 3 pairs: (1st, 6th), (2nd, 5th), (3rd, 4th). More comparisons would just test the same pairs again in reverse order. With an odd length the middle character has no partner and is not compared; in 5 characters 2 pairs are tested. Halving is done with `shr cx, 1`: a one-bit right shift divides an unsigned number by 2 and discards the remainder.

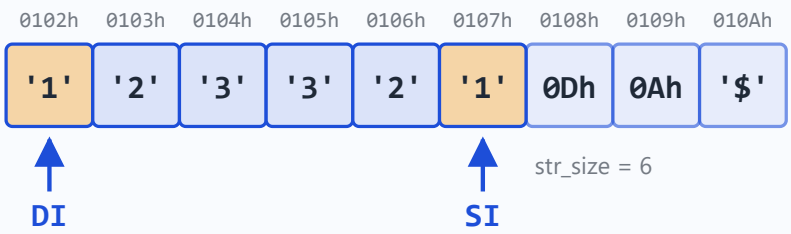


Figure 7.16: The `str` array starts at 0102h in memory; DI moves from the front and SI from the back. After the string come the line end (0Dh, 0Ah) and '$'.

In the program the length of the string is a constant: `str_size = $ - msg`. Here `$` is the assembler's location counter at that line, that is, the address right behind the string. The labels msg and str point to the same address; the difference is 6, the length of the string. The `=` directive defines a constant like EQU and takes no space in memory. If the string changes, the length updates by itself.

The program prints the string itself and the result with the DOS `INT 21h`/09h service. This service prints the string at DS:DX on the screen up to the '$' character; the '$' is not printed. The bytes 0Dh and 0Ah after the string are the carriage return and line feed characters. The result message is thereby printed one line below. The addresses in the figure assume `org 100h` and the two-byte short `jmp start` (at 0100h): str starts at 0102h.

7.7.4 Palindrome: setup

```
org 100h
jmp start
msg:
str db '123321'
str_size = $ - msg     ; = 6
db 0Dh,0Ah,'$'
start:
mov ah, 9              ; print up to '$'
mov dx, offset str
int 21h
lea di, str            ; DI = address of str
mov si, di
add si, str_size       ; SI = last + 1
dec si                 ; SI = last char
mov cx, str_size
cmp cx, 1              ; one character?
je is_palindrome       ; one char is a palindrome
shr cx, 1              ; CX = length / 2
```

The `jmp start` (`EB 09`) at the beginning of the program skips nine bytes of data (six characters, 0Dh, 0Ah, '$') and goes to 010Bh. `mov ah, 9` and `mov dx, offset str` prepare the DOS string output service: DX = 0102h. `int 21h` prints `123321` and the line end on the screen. DOS returns from this service with AL = 24h ('$'); AX = 0924h.

`lea di, str` loads into DI the address of str, 0102h. Here the effective address is a direct constant; emu8086 translates this instruction to the same 3-byte code as `mov di, offset str` (`BF 02 01`). `mov si, di` makes SI 0102h as well. `add si, str_size` adds 6: SI = 0108h. This is one past the last character of the string (the 0Dh byte). `dec si` brings SI to 0107h, the last character '1'. `mov cx, str_size` loads 6 into CX.

`cmp cx, 1` and `je is_palindrome` are a shortcut for a one-character string. A one-character string is always a palindrome, but the real reason is a different one. With length 1, `shr cx, 1` would make CX 0, and if the LOOP-based loop were entered with CX = 0, the comparison would run 65,536 times. Here CX = 6: 6 − 1 ≠ 0, ZF = 0, CF = 0; JE is not taken. `shr cx, 1` shifts CX from 0110b to 0011b: CX = 3. The bit shifted out is 0, so CF = 0.

Address	Instruction	Code	Result
0100h	`jmp start`	`EB 09`	IP = 010Bh
010Bh	`mov ah, 9`	`B4 09`	AH = 09h
010Dh	`mov dx, offset str`	`BA 02 01`	DX = 0102h
0110h	`int 21h`	`CD 21`	screen: 123321 ↵; AL = 24h
0112h	`lea di, str`	`BF 02 01`	DI = 0102h
0115h	`mov si, di`	`8B F7`	SI = 0102h
0117h	`add si, str_size`	`83 C6 06`	SI = 0108h
011Ah	`dec si`	`4E`	SI = 0107h
011Bh	`mov cx, str_size`	`B9 06 00`	CX = 0006h
011Eh	`cmp cx, 1`	`83 F9 01`	ZF = 0, CF = 0
0121h	`je is_palindrome`	`74 0E`	not taken
0123h	`shr cx, 1`	`D1 E9`	CX = 0003h, CF = 0

Table 7.25: Trace of the setup part of the palindrome program.

7.7.5 Palindrome: the comparison loop

```
next_char:
mov al, [di]          ; char at the start
mov bl, [si]          ; char at the end
cmp al, bl
jne not_palindrome    ; differ: not one
inc di                ; start: one forward
dec si                ; end: one back
loop next_char
is_palindrome:
mov ah, 9             ; print "palindrome!"
mov dx, offset msg1
int 21h
jmp stop
not_palindrome:
mov ah, 9             ; "not palindrome!"
mov dx, offset msg2
int 21h
stop:
ret
msg1 db "palindrome!$"
msg2 db "not palindrome!$"
```

The loop starts at 0125h with the label `next_char`. In each iteration `mov al, [di]` reads the character at the start and `mov bl, [si]` the character at the end. `cmp al, bl` compares the two. If they differ, `jne not_palindrome` leaves the loop at once. If they are equal, `inc di` and `dec si` bring the pointers closer together; `loop next_char` decrements CX and returns to the top.

Iteration	DI	SI	AL	BL	ZF after CMP	DI, SI, ZF after INC/DEC	CX after LOOP
1	0102h	0107h	31h '1'	31h '1'	1	0103h, 0106h, 0	0002h
2	0103h	0106h	32h '2'	32h '2'	1	0104h, 0105h, 0	0001h
3	0104h	0105h	33h '3'	33h '3'	1	0105h, 0104h, 0	0000h

Table 7.26: Trace of the palindrome comparison loop ('123321').

In every iteration CMP finds equality and sets ZF = 1; JNE is not taken. The `inc di` and `dec si` that follow set the flags too. Their results are not zero, so ZF becomes 0 again. This does not affect the correctness of the loop, because the decision was made at the JNE right behind CMP. The decision to loop belongs to LOOP, and LOOP looks only at CX. After the third iteration CX = 0 and the loop is left: all three pairs matched. At this point DI = 0105h and SI = 0104h; the pointers have crossed each other.

Leaving the loop, the processor falls through into the `is_palindrome` block. The DOS string output service is called with AH = 09h and DX = offset msg1. msg1 sits right after the code at 0142h, so DX = 0142h. `palindrome!` is printed on the screen and AL returns with 24h again. `jmp stop` (`EB 07`)

jumps over the `not_palindrome` block, and `ret` ends the program.

The data comes right after the `stop:` label. Without `ret`, the processor would take the bytes of `palindrome!$` for instructions and start executing them. The program is 94 bytes long (0100h–015Dh): msg1 takes 12 bytes and msg2 16.

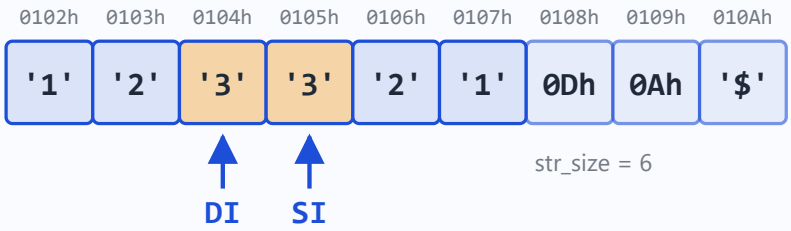


Figure 7.17: The comparison in the third iteration: DI = 0104h and SI = 0105h point at the two middle '3' characters.

7.7.6 Palindrome: different inputs

Seeing how the program behaves on different inputs is the best way to find out whether the conditions it contains are necessary. Table 7.27 compares five inputs.

str	Length	CX (after shr)	Comparisons	Output
`'123321'`	6	3	1=1, 2=2, 3=3	palindrome!
`'123421'`	6	3	1=1, 2=2, 3≠4 → jne	not palindrome!
`'12321'`	5	2	1=1, 2=2 (the middle 3 is skipped)	palindrome!
`'7'`	1	—	cmp cx,1 → je is_palindrome	palindrome!
`'ab'`	2	1	a≠b → jne	not palindrome!

Table 7.27: Behavior of the palindrome program on different inputs.

In the second input, in the third iteration '3' is compared with '4'; ZF = 0 and `jne not_palindrome` leaves the loop. LOOP is never reached and CX stays at 1. In the third input the length is 5; `shr` divides 5 by 2 and discards the remainder 1 (CF = 1). Two pairs are compared, and the middle character is left alone. In the fourth input the `cmp cx, 1` check proves useful: with length 1, `shr cx, 1` would make CX 0 and the LOOP-based loop would run 65,536 times. Length 0 carries the same danger. For an empty string CX = 0, `cmp cx, 1` finds no equality, and `shr` makes 0 into 0 again. This case can be prevented with a `jcxz is_palindrome` before the loop.

7.8 Common mistakes

Most errors in flow control reduce to a few patterns. The list below collects these patterns and how to avoid them.

Common mistake

The wrong jump family. Using JA/JB on signed data or JG/JL on unsigned data. For AL = FBh and BL = 03h, JA is taken (251 > 3) and JG is not (−5 < 3). The error shows up only for values whose most significant bit is 1, and stays hidden with small test data. Remedy: decide the type of the data for each comparison and choose the family accordingly.

Common mistake

The forgotten JMP. In the if/else pattern, if the `JMP after` at the end of block A is not written, the processor enters block B as well when block A ends. Blocks are separated by labels; a label is not a wall, only an address.

Common mistake

A flag-changing instruction between CMP and the jump. In `CMP AL, BL` / `INC SI` / `JE equal`, the JE looks at the result of INC, not of CMP. If an instruction has to go in between, choose one that does not change flags (MOV, LEA, PUSH, POP), or move the CMP right in front of the jump.

Common mistake

LOOP with CX = 0. LOOP decrements first and tests afterward. A loop entered with CX = 0 runs 65,536 times. Remedy: if the iteration count is determined at run time, put a `JCXZ` in front of the loop.

Common mistake

Not saving CX in a nested loop. The inner loop's `MOV CX, n` wipes out the outer counter. Remedy: `PUSH CX` before the inner loop and `POP CX` after it; or use another counter register with DEC–JNZ in the outer loop.

Common mistake

Executing data as code. If the data lies in front of the code, the `JMP start` is forgotten; if it lies behind the code, the final `RET` is forgotten. In both cases the processor takes data bytes for instructions. The character 'H' (48h) executes as `DEC AX`; the bytes that follow turn into random instructions.

Common mistake

Fixing an out-of-range Jcc with a synonym. Writing JZ instead of JE or adding SHORT does not change the range. The right solution is a short jump on the opposite condition plus a near JMP.

Common mistake

Changing the counter inside the loop body. An instruction that changes CX in the body of a LOOP-based loop (for example `DEC CX`) corrupts the iteration count. In a loop that starts with CX = 3 and has `DEC CX` in its body, CX decreases by two per iteration. After LOOP, CX passes through odd values: 1, FFFFh, FFFDh, … Since CX never sees zero after LOOP, the loop never ends.

7.9 Exercises and solutions

The exercises from the slides and their solutions follow. The solutions are given with their intermediate steps.

1. **A JNE instruction at address 0120h is to jump to address 0100h. What is its machine code?**

   The opcode of JNE is 75h and the instruction is 2 bytes. Next IP = 0122h. Displacement = 0100h − 0122h = −22h. Two's complement of −22h: 22h = 0010 0010b, inverted 1101 1101b, plus 1: 1101 1110b = DEh. The code is `75 DE`. Check: 0122h + FFDEh = 1 0100h → IP = 0100h.

2. **After `mov al, 80h` / `cmp al, 1`, are JA and JG taken?**

   80h − 01h = 7Fh. There is no borrow (CF = 0) and the result is not zero (ZF = 0): JA is taken; in the unsigned interpretation 128 > 1. In the signed interpretation −128 − 1 = −129 does not fit in eight bits: OF = 1. The sign bit of the result is 0: SF = 0. Since SF ≠ OF, JG is not taken; −128 < 1. AF = 1 because 0 − 1 needs a borrow in the low nibble, and PF = 0 because 7Fh contains seven 1 bits.

3. **How many times does `mov cx, 0` / `d: loop d` iterate? How is it prevented?**

   LOOP first decrements CX: 0 − 1 = FFFFh ≠ 0, so it jumps back. It jumps at every value of CX from FFFFh down to 1 and exits at 0. LOOP is executed 65,536 times in total. To prevent this, the case CX = 0 is skipped with `jcxz` before the loop.

4. **Code that sets BX = 1 if the unsigned number in AX is greater than 100, otherwise BX = 0.**

```
    mov  bx, 0
    cmp  ax, 100
    jbe  done       ; skip if AX <= 100 (unsigned)
    mov  bx, 1
done:
```

The opposite of the condition (≤) is taken from the unsigned family: JBE. Tried out, it gives BX = 0 for AX = 0, 99, and 100, and BX = 1 for AX = 101 and FFFFh (65,535). With JLE, FFFFh would count as signed −1 and BX would stay 0.

5. **In the array sum example, what would AL and CF be if the elements were 200 and 100? Propose a fix.**

   200 + 100 = 300 = 12Ch. Only the low byte fits in AL: AL = 2Ch = 44. The ninth bit goes to CF: CF = 1 (carry). The result is wrong. The fix is to keep the sum in 16 bits. In the loop, `adc ah, 0` is added after `add al, vector[bx]`, and AX is cleared before the loop. ADC adds CF to AH along with 0; every carry moves into AH. With this fix AX = 012Ch = 300. Since LOOP does not change flags, it does not disturb the carry; but because ADD sets CF again on every iteration, ADC must stand right after ADD.

6. **What happens in the Hello example if the `jmp start` line is deleted?**

   The program starts at 0100h, where the first byte of msg, 48h ('H'), now sits. The processor executes this byte as the instruction `DEC AX`. The bytes that follow, 65h, 6Ch, 6Ch, 6Fh, 00h, are also decoded and executed as instructions. The behavior of the program becomes unpredictable; Hello is not printed. The assembler does not catch this error, because the distinction between data and code exists only in the programmer's head.

7.10 Summary

- **IP and the queue**: the processor executes instructions in order from CS:IP; IP grows by the instruction length after each instruction. The BIU fetches the next bytes ahead of time into the six-byte prefetch queue. A jump flushes the queue; that is why a jump that is taken needs 15–19 clock cycles.
- **JMP** is unconditional and does not change the flags. Its forms are short (EB rel8, −128…+127), near (E9 rel16, within the segment), far (EA ofs seg, CS:IP), and indirect (FF /4, register or memory).
- **Displacement** = target − (instruction address + instruction length). A negative displacement is stored in two's complement; the processor sign-extends it and adds it to IP.
- A **label** takes no space in memory; the assembler turns it into an offset. It cannot start with a digit, cannot contain spaces, and cannot be an instruction or register name.

- **Jcc** looks at the flags and does not change them. On the 8086 all are 2 bytes (70h–7Fh + rel8); the last bit of the opcode inverts the condition. Synonyms assemble to the same code.
- **CMP** is a SUB that does not store its result; the Jcc that follows makes the decision.
- **Unsigned** comparison: JA, JAE, JB, JBE → CF and ZF. JB ≡ JC.
- **Signed** comparison: JG, JGE, JL, JLE → ZF, SF, and OF. A < B ⇔ SF ≠ OF; because the sign bit comes out inverted on overflow, SF alone is not enough.
- **if/else**: jump over block A on the opposite condition, and at the end of block A jump over block B with a JMP. **while**: the condition is tested at the top, and a JMP at the end of the body returns to the top.
- **Out-of-range Jcc**: short jump on the opposite condition + near JMP. emu8086 makes this transformation itself.
- **The LOOP family**: CX decrements, flags are preserved; LOOPE/LOOPNE also look at ZF; JCXZ jumps if CX = 0. All are 2 bytes with rel8.
- **Traps**: LOOP with CX = 0 (65,536 iterations), not saving CX in a nested loop, a forgotten JMP, a flag-changing instruction between CMP and Jcc, executing data as code.

7.11 Key terms

flow control instruction pointer (IP) prefetch queue bus interface unit (BIU) execution unit (EU) jump unconditional / conditional jump short / near / far jump jump displacement relative addressing label offset opcode two's complement sign extension flag zero flag (ZF) carry flag (CF) sign flag (SF) overflow flag (OF) parity flag (PF) borrow comparison (CMP) signed / unsigned number above / below (unsigned) greater / less (signed) loop nested loop count register branch stack string clock cycle program segment prefix (PSP)

7.12 Test Yourself

1. A short JMP at address 0150h jumps to 0140h. What is its machine code? Which form would be used if the instruction were at 0300h?

   Answer

   Next IP = 0152h; displacement 0140h − 0152h = −12h → EEh. Code `EB EE`. From 0300h the displacement would be −1C2h (−450) and would not fit the short range. The near form is chosen: next IP 0303h, displacement FE3Dh, code `E9 3D FE`.

2. With AX = 8000h and BX = 0001h, which flags does `CMP AX, BX` set? Are JB and JL taken?

   Answer

   8000h − 0001h = 7FFFh. CF = 0 (no borrow), ZF = 0, SF = 0 (the most significant bit of the result is 0), OF = 1 (signed −32,768 − 1 does not fit). JB is not taken: in the unsigned interpretation 32,768 > 1. JL is taken: SF ≠ OF, and in the signed interpretation −32,768 < 1.

3. The opcode of JL is 7Ch. What is the opcode of its opposite? What code is needed to make a JL at 0108h jump back to 0100h?

   Answer

   The last bit changes: 7Dh, that is JGE (JNL). Backward jump: next IP = 010Ah, displacement 0100h − 010Ah = −0Ah = F6h; code `7C F6`.

4. A `dec cx` has been left in the body of the loop below by mistake. What is AX when the program ends if it starts with CX = 4? What happens if it starts with CX = 3?

```
    mov  cx, 4
    mov  ax, 0
d:  inc  ax
    dec  cx
    loop d
```

   Answer

   CX = 4: in iteration 1, DEC gives 3 and LOOP 2 (taken); in iteration 2, DEC gives 1 and LOOP 0 (exits). AX = 2. CX = 3: in iteration 1, DEC gives 2 and LOOP 1 (taken); in iteration 2, DEC gives 0 and LOOP FFFFh (taken); after that CX always passes through odd values and is never 0 after LOOP. The loop is infinite.

5. How can code that sets BL = 1 if the signed number in AL is negative and BL = 0 otherwise be

written with CMP and a single conditional jump? Which jump should be chosen?

Answer

```
    mov  bl, 0
    cmp  al, 0
    jge  done       ; skip if AL >= 0
    mov  bl, 1
done:
```

The opposite of the condition (AL ≥ 0) is taken from the signed family: JGE. For AL = 80h and FFh, BL = 1; for 00h, 05h, and 7Fh, BL = 0. Since a comparison with zero cannot overflow, `JNS` works correctly too.

6. A ten-iteration loop is written with LOOP and with DEC CX + JNZ. How many clock cycles does the loop control alone take on the 8086?

Answer

LOOP: 9 jumps taken × 17 + 1 exit × 5 = 158 cycles. DEC CX + JNZ: 10 × 2 (DEC) + 9 × 16 + 1 × 4 = 168 cycles. On the 8086, LOOP is both 1 byte shorter and 10 cycles faster.

7. The target of the `je far_away` instruction at 0102h is at 01CFh. How does emu8086 encode it? How many bytes in total?

Answer

The displacement 01CFh − 0104h = CBh = 203 > 127; the short form is not enough. The condition is inverted: `75 03` (JNE; if the condition holds it hops over the three bytes of the JMP) and `E9 C8 00` (near JMP; next IP 0107h, 01CFh − 0107h = 00C8h). 5 bytes in total.

8. If the array in the LOOPNE search example were `7, 1, 8, 9, 3`, in how many iterations would the loop end? What are CX, ZF, and AL after the loop?

Answer

3 is the last element. In the fifth iteration CMP sets ZF = 1; LOOPNE takes CX from 1 to 0 and exits for both reasons. CX = 0 and ZF = 1. Since `jne notfound` looks at ZF, it is not taken and AL = 1, SI = 4. Had CX been checked, it would have been mistaken for “not found”.

7.13 Further reading

- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 6 (Program Control Instructions): the forms of JMP, conditional jumps, the LOOP family, and the assembly counterparts of high-level constructs; this is the closest source to this note.
- Intel, *The 8086 Family User's Manual* (1979): the instruction encoding tables and clock cycle counts are first-hand here; the operation of the prefetch queue is also described.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), Chapter 6 (Conditional Processing): CMP, signed and unsigned jumps, compound conditions, and loop examples.
- emu8086 documentation, *8086 assembler tutorial for beginners*, Part 7 (Program Flow Control): the source of the examples in the slides; the out-of-range conditional jump trick is also explained here.

7.14 References

1. emu8086 documentation: *8086 Assembler Tutorial for Beginners* (Part 7: Program Flow Control).
2. Intel, *The 8086 Family User's Manual*, 1979: instruction set and encoding tables.
3. Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009: Chapter 6, “Program Control Instructions”.
4. Kip R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014: conditional processing and loops.

PART III · CHAPTER 8

The Stack and Procedures

The 8086 stack is addressed by the SS:SP pair and is a LIFO structure that grows downward in memory. The chapter covers PUSH and POP, how SP changes, saving and swapping registers, preserving the flags with PUSHF and POPF, and stack safety. The procedures part explains defining procedures with PROC and ENDP, how CALL and RET push and pop the return address, passing parameters in registers, in memory and on the stack, and setting up a stack frame with BP. Recursive examples and common mistakes close the chapter.

NOTES IN THIS CHAPTER

8.1 The Stack

8.2 Procedures

PART III · CHAPTER 8.1

The Stack

While a program runs, many values have to be set aside for a short time: the old value of a register, the return address of a procedure, the flags of an interrupted program, to name the first. For these jobs the 8086 uses a region of memory as the stack. The value put on the stack last is the first to be taken back. This note begins with the stack concept and the last-in, first-out discipline. It then explains how the stack is addressed through the SS and SP registers, why it grows downward, and which bus cycles the 8086 uses to reach it. Next it takes up, one by one, the effect of PUSH, POP, PUSHF, and POPF on registers, flags, and memory. Every example in the slides is traced step by step together with the registers and the stack: saving a value, swapping, rotating three registers, and reversing an array with the stack. The last section covers errors such as unbalanced PUSH and POP, stack overflow, and underflow, and the ways to guard against them.

LEARNING OUTCOMES

Explain the stack model
Explain the last-in, first-out (LIFO) discipline of the stack, how the 8086 stack is addressed with SS:SP, and why it grows downward; compute the physical address of the top of the stack.

Trace PUSH and POP
Know the valid operand types for PUSH and POP, and show step by step, at byte level, how SP and stack memory change with each instruction.

Save and swap registers
Use the stack in the right order to preserve and restore a register's value, swap two registers, and reverse an array.

Preserve the flags
Save the flags register on the stack with PUSHF and POPF, restore flags that an operation has destroyed, and interpret the flags word bit by bit.

Recognize stack errors
Recognize the symptoms of unbalanced PUSH and POP, wrong ordering, stack overflow, and underflow, and apply the coding rules that prevent them.

Trace the stack in emu8086
Interpret the register panel, the stack window, and the memory view while single-stepping a program.

8.1.1 The stack concept

8.1.1.1 What is a stack?

The 8086 has eight general-purpose registers. A program often uses all of them, and at some point space runs short. For example, AX may hold a sum while AX is also needed for something else. One solution is to write the sum to a variable and read it back later. But defining a separate variable for every intermediate value clutters the program. Besides, it is not always known in advance how many values will have to be saved. One procedure can call another, and that one yet another. The return address of every call has to be kept somewhere, and the depth of calls depends on the flow of the program.

These needs share one property: the value saved last is the first one asked back. The innermost procedure finishes first, and its return address is the address saved last. When a register is borrowed temporarily, too, the value saved last is the first one restored. The 8086 supports a memory region that follows this discipline at the hardware level.

Definition 8.1.1

A **stack** is an area of memory set aside for temporary data and used in last-in, first-out order. On the 8086 the location of the stack is determined by the registers SS (stack segment) and SP (stack pointer). Putting a value on the stack is called a **push**; taking a value from the stack is called a **pop**.

The programmer reaches the stack with two instructions. `PUSH` puts a 16-bit value on the stack. `POP` takes a 16-bit value off the stack and writes it to a destination. The stack is an area used not only by the programmer but by the processor as well. When the `CALL` instruction calls a procedure, it puts the return address on the stack. The `RET` at the end of the procedure takes this address off the stack and returns there. When `INT` calls an interrupt, similar work is done. The flags register, the code segment (CS), and the instruction pointer (IP) are saved on the stack; with `IRET` the interrupt is returned from and the three values are restored. The interrupt routines of the operating system and the BIOS use the same stack to preserve registers.

Having so many users of the stack also brings a responsibility. Every value the programmer puts on the stack sits in the same area as the processor's return addresses. If the programmer leaves a value on the stack or takes one extra, the processor's return address shifts and the program returns to a wrong address. Stack safety is therefore treated in a section of its own.

On the 8086 the unit of the stack is the word. Always 2 bytes are written to the stack and always 2 bytes are read from it. An 8-bit operand such as in `PUSH AL` is therefore invalid. If a byte has to be saved, the 16-bit register that contains it is pushed; `PUSH AX` puts both AL and AH on the stack. The stack also has a direction. It grows from high addresses toward low ones, that is, downward. These two rules determine every detail of the PUSH and POP instructions.

User	Instruction	Put on the stack	Taken back by
programmer	`PUSH`, `PUSHF`	register, segment register, memory word, flags	`POP`, `POPF`
processor (procedure)	`CALL`	return address (IP for a near call, CS and IP for a far call)	`RET`, `RETF`
processor (interrupt)	`INT n`, hardware interrupt	FLAGS, CS, and IP	`IRET`
operating system, BIOS	interrupt routines	the registers the routine uses	the POPs at the end of the routine

Table 8.1.1: Who uses the stack on the 8086, and what they put on it.

Table 8.1.1 summarizes the four users of the stack. The subject of this note is the first row: the programmer's use of the stack with PUSH, POP, PUSHF, and POPF. The effect of CALL and RET on the stack is mentioned only briefly, from the stack's point of view. The definition of procedures, parameter passing, and the stack frame are the subject of the Procedures note.

8.1.1.2 Last in, first out (LIFO)

The word stack describes this discipline well in its everyday meaning, too. Think of plates piled on a table. A new plate is always put on top. When a plate is to be taken, the top one is taken. To reach a plate in the middle, all those above it have to be lifted off first. The plate put down last is taken first, and the plate put down first is taken last.

Definition 8.1.2

LIFO (last in, first out) is the discipline “the last one in is the first one out”. The last element added to a structure is the first element removed from it. The stack works in LIFO order; only the topmost element is accessible. The topmost element is called the **top of the stack**.

The opposite of LIFO is FIFO (first in, first out): the first one in is the first one out. A queue at a bank or a keyboard buffer works in FIFO order. The difference between the two disciplines is easiest to see with a sequence of numbers. Let 1, 2, 3, 4, and 5 be put on the stack in that order. The first value taken is 5. Then 4, 3, and 2 are taken, and 1 last. Had the same values been put in a FIFO queue, they would come out in the order 1, 2, 3, 4, 5.

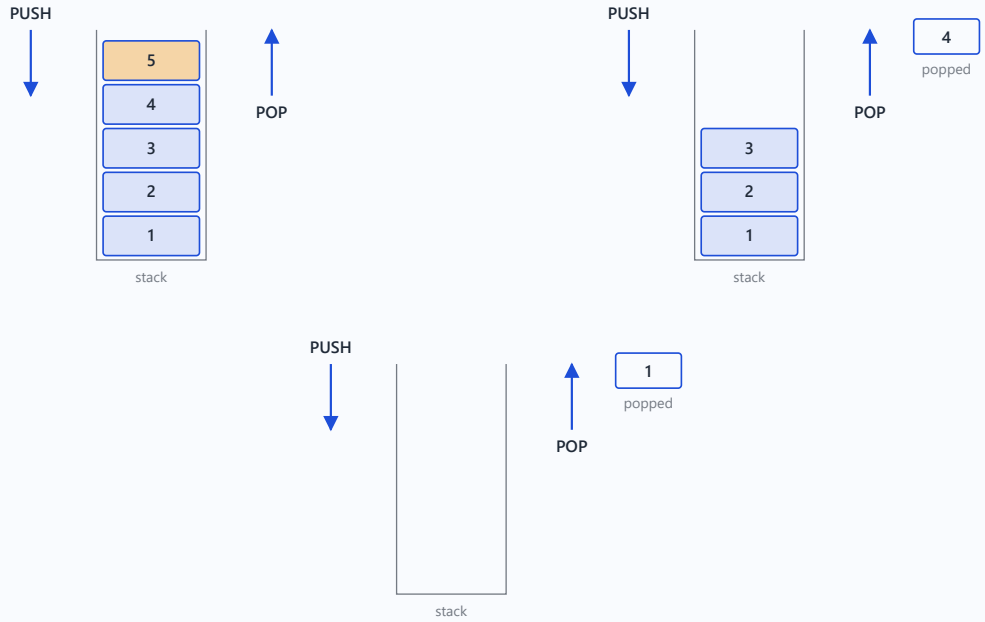


Figure 8.1.1: The LIFO discipline. On the left, five values from 1 to 5 have been pushed, and 5 is at the top of the stack. In the middle, after two POPs, 5 and 4 have come out and 3 is on top. On the right, after five POPs the stack is empty; the order of output has been 5, 4, 3, 2, 1.

Figure 8.1.1 shows this process at three moments. The stack in the figure is a conceptual drawing; there are no memory addresses yet. What matters is that each POP takes the value that is on top at that moment. The third POP takes 3, because 5 and 4 have been taken earlier and 2 and 1 remain on the stack.

The LIFO discipline has two consequences in programming. The first is that the stack gives values back in reverse order. This property can be used deliberately: pushing an array onto the stack in order and then popping it reverses the array. The second is that if several registers are saved and later restored to their old places, the POP order must be the reverse of the PUSH order. If AX, BX, and CX were pushed in this order, the restore order is CX, BX, AX. This rule is called **mirror order**. Written one below the other, the PUSHes and POPs stand like an image in a mirror.

Common mistake

Pushing registers in the order `PUSH AX`, `PUSH BX` and restoring them in the order `POP AX`, `POP BX`. This sequence does not preserve the registers; it swaps their values: since BX was pushed last, the first POP writes it into AX. The error is not noticed at assembly time, because all four instructions are valid. The program just produces a wrong result.

8.1.1.3 The stack segment: SS and SP

The memory of the 8086 is divided into segments. In every memory access a 16-bit segment value and a 16-bit offset are combined. For code, CS and IP are used; for data, most often DS and an addressing expression. There is a dedicated register pair for the stack as well.

Definition 8.1.3

SS (stack segment) is the segment register that holds the start of the 64 KB segment in which the stack lives. **SP** (stack pointer) is the 16-bit register that holds the offset, within that segment, of the word on top of the stack. SS:SP always points at the top of the stack, that is, at the word put there last. The physical address of the top of the stack is SS × 10h + SP.

The physical address computation is the same as for code and data accesses. The segment value is multiplied by 16; in hexadecimal this means appending a 0 to the right of the number. Then the offset is added and the 20-bit address is found.

Example 8.1.1

When emu8086 loads a .COM program, it sets all the segment registers to 0700h and SP to FFFEh. The physical address of the top of the stack is found as follows:

1. Stack segment: SS = 0700h.
2. The segment value is multiplied by 16: 0700h × 10h = 07000h.
3. The offset is added: 07000h + FFFEh = 16FFEh.

The top of the stack is at physical address 16FFEh. The addition produces a carry: 7000h + FFFEh = 16FFEh, so the result exceeds 16 bits. Because the address bus is 20 bits wide, this overflow is not lost; it appears as bit 16 of the 20-bit address.

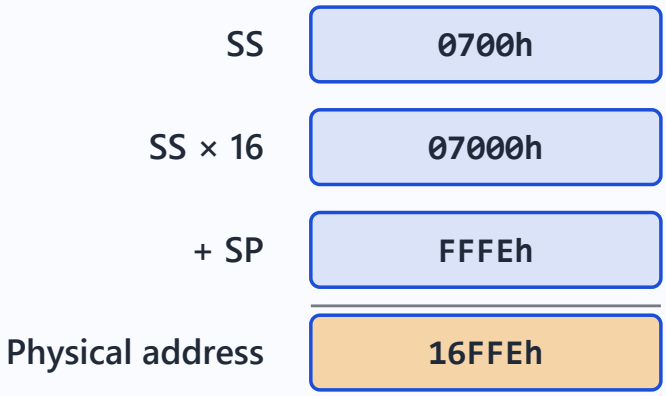


Figure 8.1.2: The physical address of the top of the stack. SS = 0700h is multiplied by 16 (07000h), SP = FFFEh is added, and the 20-bit address 16FFEh results.

The initial values of SS and SP are usually determined not by the programmer but by the operating system that loads the program into memory (Figure 8.1.2). In a .COM program, code, data, and stack share a single 64 KB segment. The loader makes SS the same as CS and sets SP to the end of the segment, FFFEh. In an .EXE program, on the other hand, the stack is a separate segment. The location of the segment and the initial value of SP are written in the header of the .EXE file; the loader puts these values into SS and SP. In the source code, a directive such as `.STACK 100h` reserves a 256-byte stack segment. In emu8086, an .EXE program written with `.stack 100h`, `.data`, and `.code` in that order starts with SS = 0710h and SP = 0100h. The top of the stack is then at physical address 07100h + 0100h = 07200h; the first PUSH writes to 071FEh.

The stack is not reached through SP alone. BP (the base pointer) also uses the SS segment by default. An instruction such as `MOV AX, [BP+4]` reads address SS:BP+4 even though no segment prefix is written. BP is therefore the right register for reaching data inside the stack, for example parameters passed to a procedure on the stack. SP, on the other hand, cannot be used in an addressing expression on the 8086: `MOV AX, [SP]` is invalid. To read the value on top of the stack without removing it, SP is first copied into BX or BP.

It is also possible for a program to move the stack itself. For this, new values are loaded into SS and SP. If an interrupt arrives between these two loads, the processor writes FLAGS, CS, and IP to an address that is half old and half new. The 8086 contains a safeguard against this danger. After an instruction that writes to a segment register with `MOV` or `POP`, no interrupt is accepted until the next instruction finishes. That is why, when changing the stack, SS is loaded first and SP immediately after:

```
MOV  AX, 2000h
MOV  SS, AX        ; interrupts held off one instruction
MOV  SP, 0100h     ; SS:SP = 2000:0100, consistent
```

If the order of the two lines is swapped, the protection does not work. The deferral covers only the instruction following the one that writes to the segment register.

8.1.1.4 The stack grows downward

In a .COM program there are three regions inside the single 64 KB segment. The first 256 bytes of the segment (0000h–00FFh) are the program segment prefix (PSP). DOS fills this area when it loads the program; the command line and return information are here. The program starts at address 0100h, as the `ORG 100h` directive demands; code and data are laid out upward. The stack starts at the very top of the segment, at FFFEh, and grows downward.

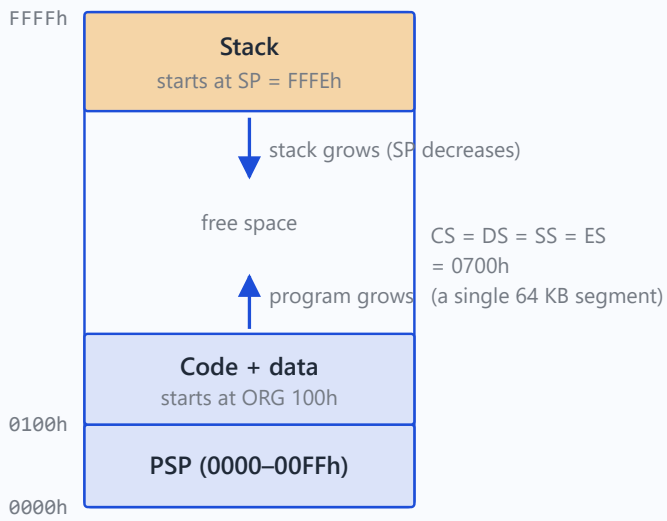


Figure 8.1.3: Layout of the 64 KB segment in a .COM program. The PSP is at the bottom, code and data go upward from 0100h, and the stack goes downward from FFFEh. Both use the free space in between.

The logic of this layout is clear in Figure 8.1.3. The program and the stack grow toward each other from the two ends of the segment. How much room the program takes is known at assembly time. How deep the stack will get is determined at run time, by the number of calls and PUSHes. Placing the two regions at opposite ends lets whichever needs it use all of the free space in between. Had the stack grown upward, a fixed size would have had to be reserved after the end of the program, and that size would have been either too much or too little.

The downward growth of the stack shows up in the instructions in a simple way. Each PUSH decreases SP by 2. Each POP increases SP by 2. As the value of SP gets smaller, the stack gets deeper. The number of words on the stack can also be computed from SP. For example, if SP = FFFEh at the start and SP = FFF8h now, (FFFEh − FFF8h) / 2 = 3 words have been pushed onto the stack.

How deep the stack can get in a .COM program can also be computed. The example program that reverses an array with the stack is 26 bytes and occupies addresses 0100h–0119h. The first free byte after the program is 011Ah. The stack starts at FFFEh; after n PUSHes SP = FFFEh − 2n. For the stack

not to overwrite the program, SP ≥ 011Ah must hold. From this, n ≤ (FFFEh − 011Ah) / 2 = (65,534 − 282) / 2 = 32,626. In this program up to 32,626 words can be put on the stack. In practice this limit is hardly approached, but an infinite loop or infinite recursion exceeds it in no time.

Common mistake

Understanding “top of the stack” as “the highest address”. The top of the stack is the place where the word put last is found and where SP points. Because the stack grows downward, this place is the *lowest* address among the items on the stack. If small addresses are drawn at the top in figures, the word “top” is also visually right; the stack figures in this note use this layout.

8.1.1.5 Stack access in hardware

PUSH and POP are a single instruction from the software point of view. In hardware, two units of the 8086 work together. The execution unit (EU) decodes the instruction and decreases or increases SP by 2 in its own arithmetic-logic unit. The bus interface unit (BIU) computes the physical address and accesses memory. The address adder in the BIU shifts SS left by 4 bits and adds SP. The resulting 20-bit address is put out on the address/data lines in a memory write (PUSH) or read (POP) bus cycle.

The data bus of the 8086 is 16 bits wide. If a word sits at an even address, it is transferred in a single bus cycle, that is, in four clock cycles (T1–T4). In this cycle A0 = 0 and BHE = 0; the low byte passes on lines D0–D7 and the high byte on D8–D15. As long as SP is even, every PUSH is a single write cycle and every POP a single read cycle. If SP falls to an odd value, the word is transferred in two separate cycles and each access takes four clock cycles longer. Loaders therefore always start SP with an even value, and programs never change SP by an odd amount.

There is one more detail that tells outside circuits about stack access. In the T2–T4 interval of each bus cycle, the 8086 puts status bits on lines A16/S3–A19/S6. The bits S4 and S3 tell which segment register is used in that cycle. If S4 = 0 and S3 = 1, that cycle is an access to the SS segment, that is, to the stack. A designer can decode these bits to route stack accesses to a separate memory bank and thus use a memory area larger than 1 MB. In ordinary PCs these bits are not used, but on a logic analyzer they are useful for seeing when a program accesses the stack.

S4	S3	Segment used	Example access
0	0	ES (extra segment)	destination of string instructions, access with an ES prefix
0	1	SS (stack segment)	PUSH, POP, CALL, RET, BP-based access
1	0	CS or no segment	instruction fetch, I/O and interrupt acknowledge cycles
1	1	DS (data segment)	ordinary variable access

Table 8.1.2: The S4 and S3 status bits of the 8086 (during T2–T4 of the bus cycle).

On the 8088 the situation is different. The external data bus of the 8088 is 8 bits wide; every word is transferred in two separate byte cycles. So on the 8088 every PUSH and POP takes two memory cycles even when SP is even. From the programming point of view the two processors are the same; only the timings differ. All the examples in this note are for the 8086.

8.1.2 PUSH and POP

8.1.2.1 The PUSH instruction

The `PUSH source` instruction puts a copy of the 16-bit source on top of the stack. The source does not change; a copy of it appears on the stack. The source can be of four kinds: a general-purpose or pointer register, a segment register, a word in memory, and, from the 80186 on, an immediate value. Table 8.1.3 gives these kinds with examples.

Form	Valid operands	Example
PUSH register	AX, BX, CX, DX, SI, DI, BP, SP	`PUSH AX`
PUSH segment register	CS, DS, SS, ES	`PUSH DS`
PUSH memory	`[BX]`, `[BX+SI+7]`, 16-bit variable	`PUSH [BX]`, `PUSH num`
PUSH immediate	5, –24, 3Fh, 10001101b	`PUSH 3Fh` (not on the 8086)

Table 8.1.3: Operand types of the PUSH instruction.

The last row of the table carries a warning. PUSH with an immediate value is not in the instruction set of the 8086 and 8088; it arrived with the 80186. emu8086 accepts and runs this instruction. But in a program written for a real 8086, a constant is first loaded into a register, and then the register is pushed:

```
MOV  AX, 5
PUSH AX          ; instead of PUSH 5 on the 8086
```

For PUSH with a memory operand, the size of the operand must be a word. For a variable defined as `num DW 0BEEFh`, `PUSH num` is valid; the two bytes of the variable are copied to the stack. A byte variable defined with `DB`, however, cannot be pushed. The same rule applies to registers: `PUSH AL` and `PUSH byte_var` give an assembly error. The flags register cannot be written as an ordinary operand; it has a separate instruction, `PUSHF`.

Pushing SP itself carries an interesting detail. The `PUSH SP` instruction both decrements SP and writes it to the stack, and which happens first changes the result. The 8086 first decrements SP and then writes the *decremented* value. With SP = FFFEh, `PUSH SP` puts FFFCh on the stack. The 80286

and later processors write the value from before the decrement, FFFEh. This difference was used in old programs to tell whether the processor was an 8086 or an 80286. In a portable program `PUSH SP` is not relied on; if the value of SP is needed, it is first copied into another register.

> **Common mistake**
>
> Taking `PUSH 5` for an 8086 instruction because it worked in emu8086. emu8086 accepts some 80186 instructions and only gives a warning. When the same code runs on a real 8086 or 8088, the processor interprets these bytes as some other instruction and the program behaves unexpectedly.

8.1.2.2 The POP instruction

The `POP destination` instruction reads the word on top of the stack, writes it to the destination, and moves the top of the stack up by one word. The destination types resemble the source types of PUSH, but with two omissions (Table 8.1.4).

Form	Valid operands	Example
POP register	AX, BX, CX, DX, SI, DI, BP, SP	`POP BX`
POP segment register	DS, SS, ES (not CS)	`POP DS`
POP memory	`[BX]`, `[BX+SI+7]`, 16-bit variable	`POP result`

Table 8.1.4: Operand types of the POP instruction.

Why is there no POP CS? CS determines the segment from which instructions are fetched. Had `POP CS` changed CS and left IP as it was, the next instruction would be fetched from the same offset in a completely different segment. The program flow would jump to a random place. That is why CS changes only together with IP: far `JMP` and `CALL`, `RETF`, `INT`, and `IRET` load CS and IP in the same instruction. Assemblers do not accept `POP CS`. On the first 8086s the code of this instruction (0Fh) worked in an undocumented way; Intel used this code on the 80286 as the prefix of new two-byte instructions.

Why is there no POP constant? `POP 5` is meaningless. POP takes a value from the stack, and a place is needed to write that value to. A constant is not a place a value can be written to. If the value on top of the stack is not needed and is only to be discarded, there are two ways. The value is taken into a register that is not in use, or 2 is added directly to SP (`ADD SP, 2`). The second way changes the flags; the first does not.

Flags. PUSH and POP do not touch the flags. Even if 0000h arrives in AX with `POP AX`, ZF does not change; what changes ZF is an arithmetic or logic operation, not a data transfer. The only exception to this rule is `POPF`. POPF loads the word it takes from the stack into the whole flags register.

Copying between segment registers. On the 8086 there is no direct `MOV` from one segment register to another; `MOV DS, CS` is invalid. This copy is done in two ways. Either the value is passed through a general-purpose register (`MOV AX, CS` and `MOV DS, AX`), or the stack is used:

```
PUSH CS          ; value of CS onto the stack
POP  DS          ; DS ← CS
PUSH DS          ; value of DS onto the stack
POP  ES          ; ES ← DS
```

The advantage of copying through the stack is that it disturbs no general-purpose register. If the value of AX is to be preserved, the first way requires saving AX separately; the second does not. The idiom `PUSH CS` with `POP DS` is common in programs whose data sits in the code segment and at the start of interrupt routines. In emu8086, if DS is set to 1234h and these four instructions are run, ES = 1234h and DS = 0700h; SP returns to FFFEh.

8.1.2.3 How PUSH and POP work

Both instructions consist of two steps. The order of the steps determines what SP points at at each moment.

Definition 8.1.4

PUSH source: (1) SP ← SP − 2; (2) [SS:SP] ← source. **POP destination**: (1) destination ← [SS:SP]; (2) SP ← SP + 2. PUSH first decrements and then writes; POP first reads and then increments. SP therefore always points at the topmost occupied cell.

This ordering is called the “full stack” convention. The cell SP points at always holds a valid value. Some processors use the opposite convention: SP points at the next free cell, and PUSH first writes and then decrements. Whichever convention is chosen, PUSH and POP must be exact opposites. On the 8086, PUSH is “decrement, write” and POP is “read, increment”.

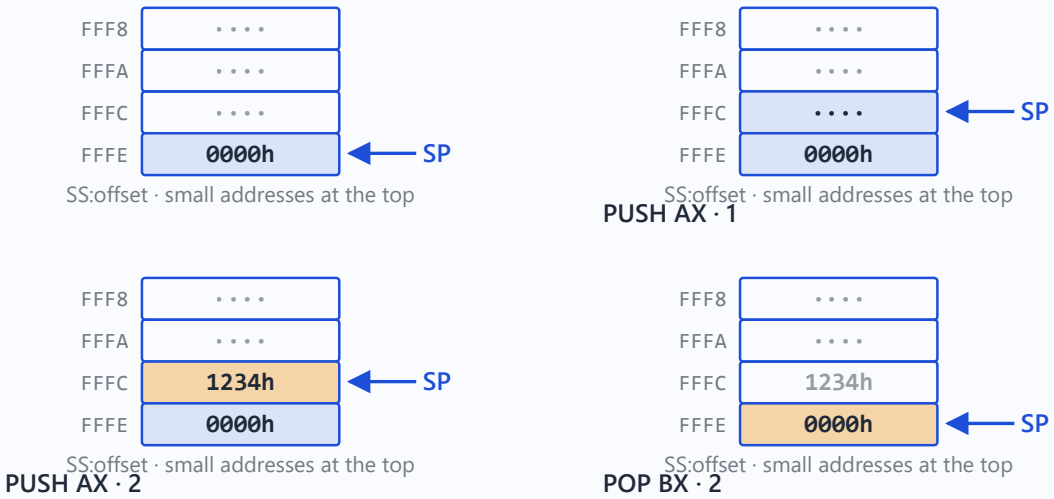


Figure 8.1.4: PUSH AX and POP BX with AX = 1234h. Top left, the start: SP = FFFEh, where the 0000h put by DOS sits. Top right, the first step of PUSH: SP = FFFCh, the new cell not yet written. Bottom left, the second step: [FFFCh] ← 1234h. Bottom right, POP BX has finished: BX = 1234h, SP = FFFEh; the 1234h at FFFCh has not been erased, it has only fallen outside the stack.

Figure 8.1.4 shows the two instructions at four moments. At the start SP = FFFEh and AX = 1234h. At address FFFEh sits the word 0000h that DOS put for the .COM program; the job of this word is explained at the end of the program.

1. **PUSH AX, step 1.** SP ← FFFEh − 2 = FFFCh. The top of the stack moves down by one word. The cell at FFFCh holds, for now, an old, meaningless value.
2. **PUSH AX, step 2.** [SS:FFFCh] ← 1234h. A copy of AX is written into the new top cell. AX does not change.
3. **POP BX, step 1.** BX ← [SS:FFFCh] = 1234h. The value is read from the top of the stack.
4. **POP BX, step 2.** SP ← FFFCh + 2 = FFFEh. The top of the stack moves up by one word.

The last step has an important consequence. POP does not erase the value it reads from memory. The 1234h at FFFCh stays there; it is just now above SP, that is, outside the stack. In the figure this cell is drawn faded. The next PUSH overwrites the same cell. For this reason, trying to reach a value that has been taken off the stack by reading memory again is not reliable. If an interrupt arrives in between, the processor writes FLAGS, CS, and IP to exactly those cells, and the old value is lost.

Common mistake

Thinking that after a POP the value is erased from memory and 0 is written in its place. POP only reads and increments SP. In the memory window of emu8086 the old values remain visible. These values are not “on the stack”; the stack is only the region from SP up to FFFEh.

8.1.2.4 Byte order on the stack

The stack works with words, while memory is addressed in bytes. When a word is written to memory, its two bytes get two separate addresses. For this the 8086 uses the little-endian convention: the low byte of the word is written to the lower address and the high byte to the next address. The rule is the same for the stack.

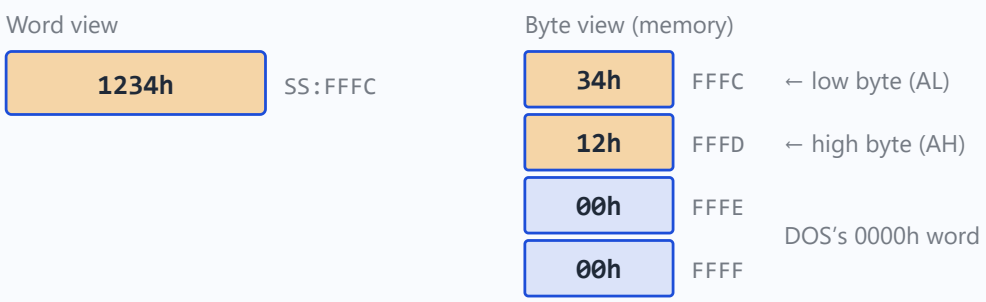


Figure 8.1.5: The stack after PUSH AX (AX = 1234h). On the left, the word view: 1234h at SS:FFFC. On the right, the byte view: [FFFCh] = 34h (AL), [FFFDh] = 12h (AH); the 0000h word of DOS at FFFEh and FFFFh.

When `PUSH AX` runs with AX = 1234h, SP = FFFCh. The low byte of the word, 34h (AL), is written to address FFFCh, and the high byte, 12h (AH), to address FFFDh (Figure 8.1.5). SP always points at the low byte of the word, that is, at the smaller address. The stack window of emu8086 shows words and writes 1234h on this line. The memory window, however, shows bytes in order: first 34, then 12. The two windows present the same data in different ways. That the value looks "reversed" in the memory window is not an error but a natural result of the little-endian convention.

This convention can also be tested with a program. To read the bytes on top of the stack, SP is copied into a base register, because the 8086 has no `[SP]` addressing expression:

```
ORG 100h
MOV  AX, 1234h
PUSH AX
MOV  BX, SP        ; BX = FFFCh
MOV  CL, [BX]      ; CL = 34h, low byte
MOV  CH, [BX+1]    ; CH = 12h, high byte
POP  DX            ; DX = 1234h
```

The program runs as follows. `PUSH AX` makes SP FFFCh and writes the bytes 34h and 12h to FFFCh and FFFDh. `MOV BX, SP` makes BX FFFCh. `[BX]` uses the DS segment by default. In a .COM program DS = SS, so DS:FFFCh is the same physical address as the top of the stack: 07000h + FFFCh = 16FFCh. `MOV CL, [BX]` reads 34h from there, and `MOV CH, [BX+1]` reads 12h from 16FFDh. CX = 1234h. Finally `POP DX` reads the two bytes again as a word: DX = 1234h. In an .EXE program DS and SS may differ. Then `[BX]` reads the data segment, not the stack; in that case one must write `SS:[BX]` or use BP instead of BX.

SP staying even is related to this byte-level structure too. As long as SP is even, every stack word starts at an even address and the 8086 moves it in a single bus cycle. If SP were odd, the low byte of the word would fall on an odd address and the high byte on an even one, and two separate cycles would be needed. The program would still run correctly, just more slowly.

8.1.2.5 What changes and what does not?

All instructions that use the stack change SP. Whether they write to memory, whether they change a destination, and their effect on the flags differ, however. Table 8.1.5 gives these effects side by side.

Instruction	SP	Memory	Destination	Flags
`PUSH src`	−2	[SS:SP] ← source	—	unchanged
`POP dest`	+2	read, not erased	destination ← [SS:SP]	unchanged
`PUSHF`	−2	[SS:SP] ← FLAGS	—	unchanged
`POPF`	+2	read	FLAGS ← [SS:SP]	all loaded
`CALL` (near)	−2	[SS:SP] ← return IP	IP ← target	unchanged
`RET` (near)	+2	read	IP ← [SS:SP]	unchanged
`INT n`	−6	FLAGS, CS, IP	CS:IP ← vector	IF = TF = 0
`IRET`	+6	read	IP, CS, FLAGS	all loaded

Table 8.1.5: Effect of the stack instructions on SP, memory, the destination, and the flags.

The most important column of the table is the last one. PUSH, POP, and PUSHF do not touch the flags. This means the result of a comparison, that is, the flags, can be carried safely across PUSH and POP. In the code below, `CMP` sets ZF to 1. Then AX is put on the stack, changed, and taken back. Since these three instructions do not touch ZF, `JE` is still taken:

```
MOV  AX, 5
CMP  AX, 5          ; ZF = 1
PUSH AX
MOV  AX, 7          ; MOV does not change flags
POP  AX             ; AX = 5, ZF still 1
JE   equal          ; taken
```

Had an instruction that changes the flags come in between, an `ADD` for instance, the result of the comparison would have been lost. In that case `PUSHF` and `POPF` are used to save the flags. The stack effect of CALL and RET is very similar to that of PUSH and POP: CALL pushes the return address and RET pops it. The difference is that the destination is IP, not a register. INT pushes three words, that is, 6 bytes, and is the only push instruction among these that changes the flags: on entering the interrupt routine, IF and TF are cleared.

8.1.2.6 Machine codes and clock cycles

The stack instructions are among the shortest on the 8086. Pushing or popping a register takes a single byte. Five bits of the opcode determine the instruction and the remaining three the register. The code is 50h + r for `PUSH` and 58h + r for `POP`. The register numbers follow the general order of the 8086: AX = 0, CX = 1, DX = 2, BX = 3, SP = 4, BP = 5, SI = 6, DI = 7. So `PUSH AX` = 50h, `PUSH BX` = 53h, `POP BX` = 5Bh, and `POP DI` = 5Fh.

Segment registers are single bytes too. The push code is 06h + 8 × s and the pop code is 07h + 8 × s, where s is the number ES = 0, CS = 1, SS = 2, DS = 3. `PUSH ES` = 06h, `PUSH CS` = 0Eh, `PUSH SS` = 16h, `PUSH DS` = 1Eh; `POP ES` = 07h, `POP SS` = 17h, `POP DS` = 1Fh. In this scheme the code of `POP CS` would have been 0Fh; this code fell out of use along with POP CS. The forms with a memory operand use a ModR/M byte: `PUSH memory` = FFh /6, `POP memory` = 8Fh /0. For example, for a variable at address 0109h, `PUSH num` = FF 36 09 01, four bytes. `PUSHF` = 9Ch and `POPF` = 9Dh.

Instruction	Code	Bytes	Clock cycles
`PUSH register`	50h + r	1	11
`PUSH seg`	06h + 8s	1	10
`PUSH memory`	FFh /6	2–4	16 + EA
`POP register`	58h + r	1	8
`POP seg`	07h + 8s	1	8
`POP memory`	8Fh /0	2–4	17 + EA
`PUSHF`	9Ch	1	10
`POPF`	9Dh	1	8

Table 8.1.6: Codes and 8086 clock cycles of the stack instructions (Intel 8086 manual; EA = time of the effective address computation).

The timings in Table 8.1.6 are for the case where the words sit at even addresses. A PUSH takes 11 clock cycles; on a 5 MHz 8086 one clock cycle is 200 ns, so this is 2.2 µs. Four cycles of this time are the memory write cycle; the rest is decoding the instruction and decrementing SP. The forms with a memory operand have two memory accesses: one to the variable and one to the stack. The time is therefore longer, and the time of the effective address computation is added. On the 8088 four more cycles are added to each word access.

8.1.3 Examples: code traces

All the programs in this section were run in emu8086 as .COM programs. At the start CS = DS = SS = ES = 0700h, IP = 0100h, and SP = FFFEh; unless stated otherwise the other registers are 0000h. In the trace tables each row gives the state *after* that instruction has executed. The values that change in that row are highlighted. The stack column shows only the occupied cells that SP points at and below; the 0000h at FFFEh is the word DOS put for the .COM program.

8.1.3.1 First example: PUSH AX / POP BX

The first program copies a value from AX to BX through the stack. This could also have been done with `MOV BX, AX`; the aim is to see the working of the stack in its plainest form.

```
ORG     100h
MOV     AX, 42      ; AX = 42
PUSH    AX          ; push AX
POP     BX          ; pop into BX
RET                 ; return to the OS
END
```

The program consists of four instructions and 6 bytes: `MOV AX, 42` is 3 bytes (B8 2A 00, 0100h–0102h), `PUSH AX` is 1 byte (50, 0103h), `POP BX` is 1 byte (5B, 0104h), and `RET` is 1 byte (C3, 0105h).

In the line `MOV AX, 42` the number has no h at the end. The assembler reads it as a decimal number: 42 = 2 × 16 + 10 = 2Ah. AX = 002Ah, not 0042h. This small detail is one of the points beginners stumble on most often in emu8086. `PUSH AX` makes SP FFFEh − 2 = FFFCh and writes [FFFCh] ← 002Ah; at byte level [FFFCh] = 2Ah and [FFFDh] = 00h. `POP BX` takes the 002Ah at FFFCh into BX and brings SP back to FFFEh. The value has been copied from AX to BX through the stack; AX keeps its value too.

Instruction	IP (next)	AX	BX	SP	Stack (from the top)
start	0100h	0000h	0000h	FFFEh	0000h
`MOV AX, 42`	0103h	**002Ah**	0000h	FFFEh	0000h
`PUSH AX`	0104h	002Ah	0000h	**FFFCh**	**002Ah, 0000h**
`POP BX`	0105h	002Ah	**002Ah**	**FFFEh**	0000h
`RET`	**0000h**	002Ah	002Ah	**0000h**	(empty)

Table 8.1.7: Trace of the PUSH AX / POP BX program.

The last row shows the role of the stack at the end of the program (Table 8.1.7). `RET` takes the word on top of the stack as the return address. The stack holds only the 0000h put by DOS; IP = 0000h and SP = FFFEh + 2 = 0000h. SP wraps around in 16 bits. The processor goes to address 0700:0000, the start of the PSP. The first two bytes of the PSP are CD 20, which is the `INT 20h` instruction that terminates the program. The program thus ends properly. This mechanism works only if the stack is balanced: when RET is reached, DOS's 0000h must be on top of the stack.

8.1.3.2 LIFO order: 1, 2, 3 go in, 3, 2, 1 come out

The second program shows the LIFO discipline with real registers. AX is loaded with 1, 2, and 3 in turn, and each value is pushed onto the stack. Then `POP BX` is done three times.

```
MOV     AX, 1       ; AX = 1
PUSH    AX          ; push 1
MOV     AX, 2       ; AX = 2
PUSH    AX          ; push 2
MOV     AX, 3       ; AX = 3
PUSH    AX          ; push 3
POP     BX          ; BX ← 3
POP     BX          ; BX ← 2
POP     BX          ; BX ← 1
```

Instruction	AX	BX	SP	Stack (from the top, DOS word excluded)
start	0000h	0000h	FFFEh	(empty)
`MOV AX, 1`	**0001h**	0000h	FFFEh	(empty)
`PUSH AX`	0001h	0000h	**FFFCh**	**0001h**
`MOV AX, 2`	**0002h**	0000h	FFFCh	0001h
`PUSH AX`	0002h	0000h	**FFFAh**	**0002h, 0001h**
`MOV AX, 3`	**0003h**	0000h	FFFAh	0002h, 0001h
`PUSH AX`	0003h	0000h	**FFF8h**	**0003h, 0002h, 0001h**
`POP BX`	0003h	**0003h**	**FFFAh**	0002h, 0001h
`POP BX`	0003h	**0002h**	**FFFCh**	0001h
`POP BX`	0003h	**0001h**	**FFFEh**	(empty)

Table 8.1.8: Trace of the LIFO order program.

Two points stand out in the trace (Table 8.1.8). First, when a new value is loaded into AX, the old value is not lost; the preceding PUSH has copied it onto the stack. When AX = 0002h, 0001h is safe on the stack. Second, the values arriving in BX are in the reverse of the order they went in: first 3, then 2, then 1. Since three PUSHes and three POPs were done, SP returns to where it started, FFFEh. At the end of the program AX = 0003h remains, because the last value loaded into AX is 3 and the POPs change only BX.

8.1.3.3 Saving a register's value

The most common use of the stack is to borrow a register temporarily for another job and then return it to its old value. The program first loads AX with 1234h. Then it saves AX, writes another value into AX, and restores the saved value.

```
ORG   100h
MOV   AX, 1234h
PUSH  AX          ; save AX
MOV   AX, 5678h   ; change AX
POP   AX          ; restore old value
RET
END
```

Instruction	AX	SP	[FFFCh]	Note
`MOV AX, 1234h`	**1234h**	FFFEh	—	value to be preserved
`PUSH AX`	1234h	**FFFCh**	**1234h**	a copy of AX on the stack
`MOV AX, 5678h`	**5678h**	FFFCh	1234h	AX is being used for another job
`POP AX`	**1234h**	**FFFEh**	1234h	the old value is back; the cell was not erased
`RET`	1234h	**0000h**	1234h	IP ← 0000h, INT 20h in the PSP

Table 8.1.9: Trace of the value-saving program.

The program is 9 bytes: two three-byte MOVs and one-byte PUSH, POP, and RET. The fourth column of Table 8.1.9 shows once more that POP does not erase memory: [FFFCh] = 1234h after the POP too. This pattern is the same as procedures PUSHing the registers they use on entry and POPping them on exit. If a procedure is going to destroy AX, it writes `PUSH AX` on entry and `POP AX` on exit. The program that called the procedure does not notice that AX was changed. This method is called preserving registers.

Example 8.1.2

AX holds the counter of a loop and BX the address of an array. In the middle of the loop a character is to be printed on the screen by loading AH = 02h and DL = the character and calling `INT 21h`. Function 02h of INT 21h also changes AL: when the function ends, AL is the character printed. Since both halves of AX will be disturbed, the printing is wrapped like this:

```
PUSH AX          ; save the counter
PUSH DX          ; DX will be disturbed too
MOV  AH, 2
MOV  DL, '*'
INT  21h         ; AL = '*'
POP  DX          ; mirror order: DX first
POP  AX          ; the counter is back
```

Since the PUSH order is AX, DX, the POP order is DX, AX. BX is not touched, so it is not saved. Each register saved costs two instructions and 19 clock cycles (11 + 8); only registers that are really disturbed are saved.

8.1.3.4 Swapping two registers with the stack

The opposite of mirror order, that is, POPping in the same order, swaps the values. This is sometimes a mistake and sometimes a method used deliberately.

```
ORG   100h
MOV   AX, 1212h  ; AX ← 1212h
MOV   BX, 3434h  ; BX ← 3434h
PUSH  AX         ; push AX
PUSH  BX         ; push BX
POP   AX         ; AX ← old BX
POP   BX         ; BX ← old AX
RET
END
```

When emu8086 loads this program, CX shows 000Bh. This value is the byte count of the program: the two MOVs are 3 bytes each, the four stack instructions and RET one byte each, 3 + 3 + 1 + 1 + 1 + 1 + 1 = 11 bytes in all. After loading, emu8086 writes the size of the .COM program into CX; the DOS DEBUG program likewise writes the length of the file it loaded into BX:CX. This is a custom, not a rule. A program must not trust the initial value of CX and must load CX itself before using it. In the other traces of this note the initial value of CX is taken as 0000h; since the swap program does not use CX, the result does not change.

Instruction	AX	BX	SP	Stack (from the top)
start	0000h	0000h	FFFEh	0000h
`MOV AX, 1212h`	**1212h**	0000h	FFFEh	0000h
`MOV BX, 3434h`	1212h	**3434h**	FFFEh	0000h
`PUSH AX`	1212h	3434h	**FFFCh**	**1212h, 0000h**
`PUSH BX`	1212h	3434h	**FFFAh**	**3434h, 1212h, 0000h**
`POP AX`	**3434h**	3434h	**FFFCh**	1212h, 0000h
`POP BX`	3434h	**1212h**	**FFFEh**	0000h
`RET`	3434h	1212h	**0000h**	(empty); IP = 0000h

Table 8.1.10: Trace of the swap program (CX = 000Bh throughout).

The secret of the swap is LIFO (Table 8.1.10). Since BX was pushed last, it is on top of the stack and the first `POP AX` takes it. Then `POP BX` takes the next value, AX's old value 1212h. For a moment AX and BX are both 3434h; during this time the stack holds 1212h safely. Had the POPs been done in mirror order, that is, `POP BX` first and then `POP AX`, the values would have stayed where they were.

8.1.3.5 Three registers: POP in the same order

With three registers the same experiment gives a more interesting result. This program was loaded as a .BIN file in the slides. emu8086 starts a .BIN file with CS = SS = 0100h and IP = 0000h; SP is again FFFEh. Because the segment value is different, the physical addresses change, but the offsets and the SP values are the same as in the .COM program.

```
mov ax, 2
mov bx, 4
mov cx, 8
push ax
push bx
push cx
pop  ax
pop  bx
pop  cx
```

Instruction	AX	BX	CX	SP	Stack (from the top)
`mov ax, 2`	**0002h**	0000h	0000h	FFFEh	(empty)
`mov bx, 4`	0002h	**0004h**	0000h	FFFEh	(empty)
`mov cx, 8`	0002h	0004h	**0008h**	FFFEh	(empty)
`push ax`	0002h	0004h	0008h	**FFFCh**	**0002h**
`push bx`	0002h	0004h	0008h	**FFFAh**	**0004h, 0002h**
`push cx`	0002h	0004h	0008h	**FFF8h**	**0008h, 0004h, 0002h**
`pop ax`	**0008h**	0004h	0008h	**FFFAh**	0004h, 0002h
`pop bx`	0008h	0004h	0008h	**FFFCh**	0002h
`pop cx`	0008h	0004h	**0002h**	**FFFEh**	(empty)

Table 8.1.11: Trace of the three-register stack program.

In the end AX = 0008h, BX = 0004h, and CX = 0002h (Table 8.1.11). AX and CX have changed places and BX stayed where it was. The reason is that the middle value is in the middle both when pushing and when popping. In general, if n registers are pushed in one order and popped in the same order, the register sequence is reversed: the first register gets the value of the last, and the last gets the value of the first. With three registers, the middle one is the fixed point. At the line `pop bx`, 0004h is written to BX again; since the value does not change, it is not highlighted in the table, but the instruction was executed and SP increased.

8.1.3.6 Reversing an array with the stack

The reversing property of LIFO can be applied to a string as well. The program pushes the characters of the string 'ABC' onto the stack in order, then pops them from the stack and prints them on the screen. CBA appears on the screen.

```
ORG  100h
MOV  SI, OFFSET msg
MOV  CX, 3
put: MOV  AL, [SI]
     PUSH AX       ; save the char
     INC  SI
     LOOP put
MOV  CX, 3
prt: POP  DX       ; DL = char
     MOV  AH, 2
     INT  21h      ; print DL
     LOOP prt
RET
msg DB 'ABC'     ; 0117h
```

First the layout of the program in memory is worked out, because the address of the variable `msg` comes after the code. Table 8.1.12 gives the address and machine code of every instruction. The code takes 23 bytes, from 0100h to 0116h. The following `msg` is at addresses 0117h–0119h; the bytes 41h, 42h, and 43h are the ASCII codes of the characters 'A', 'B', and 'C'. The program is 26 bytes in all.

Line	Address	Code	Instruction
2	0100h	BE 17 01	`MOV SI, OFFSET msg`
3	0103h	B9 03 00	`MOV CX, 3`
4	0106h	8A 04	`put: MOV AL, [SI]`
5	0108h	50	`PUSH AX`
6	0109h	46	`INC SI`
7	010Ah	E2 FA	`LOOP put`
8	010Ch	B9 03 00	`MOV CX, 3`
9	010Fh	5A	`prt: POP DX`
10	0110h	B4 02	`MOV AH, 2`
11	0112h	CD 21	`INT 21h`
12	0114h	E2 F9	`LOOP prt`
13	0116h	C3	`RET`
14	0117h	41 42 43	`msg DB 'ABC'`

Table 8.1.12: Addresses and machine codes of the reversal program.

The second byte of the LOOP instructions is the relative jump displacement. `LOOP put` is at 010Ah and is two bytes; the next instruction is at 010Ch. To return to 0106h the displacement must be 0106h − 010Ch = −6; the 8-bit two's complement of −6 is FAh. For `LOOP prt` the displacement is 010Fh − 0116h = −7 = F9h.

First loop: pushing the characters. SI holds the string and CX the loop counter. In each iteration `MOV AL, [SI]` reads a character into AL. Since a byte cannot be pushed onto the stack, `PUSH AX` pushes AL together with AH as a word. Because AH = 00h at the start of the program, 0041h, 0042h, and 0043h go onto the stack. `INC SI` points at the next character; `LOOP put` decrements CX by 1 and returns to the top if CX ≠ 0. After three iterations SI = 011Ah and SP = FFF8h. On top of the stack is the 'C' pushed last.

Line	Instruction	AX	CX	SI	SP	Stack (from the top, DOS word excluded)
2	`MOV SI, OFFSET msg`	0000h	0000h	**0117h**	FFFEh	(empty)
3	`MOV CX, 3`	0000h	**0003h**	0117h	FFFEh	(empty)
4	`MOV AL, [SI]`	**0041h**	0003h	0117h	FFFEh	(empty)
5	`PUSH AX`	0041h	0003h	0117h	**FFFCh**	**0041h**
6	`INC SI`	0041h	0003h	**0118h**	FFFCh	0041h
7	`LOOP put`	0041h	**0002h**	0118h	FFFCh	0041h
4	`MOV AL, [SI]`	**0042h**	0002h	0118h	FFFCh	0041h
5	`PUSH AX`	0042h	0002h	0118h	**FFFAh**	**0042h, 0041h**
6	`INC SI`	0042h	0002h	**0119h**	FFFAh	0042h, 0041h
7	`LOOP put`	0042h	**0001h**	0119h	FFFAh	0042h, 0041h
4	`MOV AL, [SI]`	**0043h**	0001h	0119h	FFFAh	0042h, 0041h
5	`PUSH AX`	0043h	0001h	0119h	**FFF8h**	**0043h, 0042h, 0041h**
6	`INC SI`	0043h	0001h	**011Ah**	FFF8h	0043h, 0042h, 0041h
7	`LOOP put`	0043h	**0000h**	011Ah	FFF8h	0043h, 0042h, 0041h

Table 8.1.13: Trace of the first loop (DX = 0000h).

There is also a small detail about the flags in the first loop. `INC SI` updates the arithmetic flags other than CF. When SI = 0118h, the low byte 18h = 0001 1000b contains two 1 bits; PF = 1. At SI = 0119h and 011Ah the number of 1s in the low bytes is odd and PF = 0. These flags do not affect the result of the program, because `LOOP` does not look at the flags; it tests only CX.

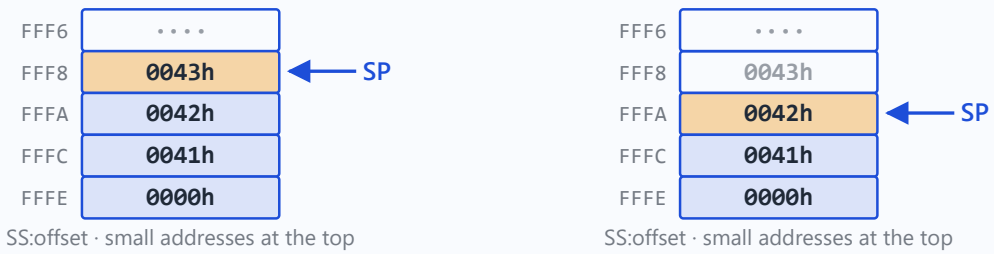


Figure 8.1.6: On the left, the stack when the first loop ends: 0043h ('C') on top, SP = FFF8h. On the right, after the first POP DX of the second loop: 'C' has been taken into DX, SP = FFFAh; the 0043h at FFF8h is now outside the stack.

Second loop: popping and printing. CX is set to 3 again. In each iteration `POP DX` takes the word on top of the stack into DX; DL is the character and DH is 00h. `MOV AH, 2` selects the DOS character output function, and `INT 21h` prints the character in DL (Figure 8.1.6). Function 02h of INT 21h has a side effect: when the function ends, AL is the character printed. AX therefore changes after every INT 21h. In the first iteration AL is already 43h, so no difference shows; in the following iterations AL = 42h and AL = 41h. The first to come off the stack is 'C', then 'B', and 'A' last.

Line	Instruction	AX	CX	DX	SP	Screen
8	`MOV CX, 3`	0043h	**0003h**	0000h	FFF8h	
9	`POP DX`	0043h	0003h	**0043h**	**FFFAh**	
10	`MOV AH, 2`	**0243h**	0003h	0043h	FFFAh	
11	`INT 21h`	0243h	0003h	0043h	FFFAh	**C**
12	`LOOP prt`	0243h	**0002h**	0043h	FFFAh	C
9	`POP DX`	0243h	0002h	**0042h**	**FFFCh**	C
10	`MOV AH, 2`	0243h	0002h	0042h	FFFCh	C
11	`INT 21h`	**0242h**	0002h	0042h	FFFCh	**CB**
12	`LOOP prt`	0242h	**0001h**	0042h	FFFCh	CB
9	`POP DX`	0242h	0001h	**0041h**	**FFFEh**	CB
10	`MOV AH, 2`	0242h	0001h	0041h	FFFEh	CB
11	`INT 21h`	**0241h**	0001h	0041h	FFFEh	**CBA**
12	`LOOP prt`	0241h	**0000h**	0041h	FFFEh	CBA
13	`RET`	0241h	0000h	0041h	**0000h**	CBA

Table 8.1.14: Trace of the second loop (SI = 011Ah).

In the last row of Table 8.1.14, SP = 0000h and IP = 0000h; RET has taken DOS's 0000h word. The program executes 28 instructions in all and ends with the INT 20h in the PSP. Because the two loops run the same number of iterations, three POPs correspond to three PUSHes and the stack is balanced. Had the loop counts been different, for example had the second loop mistakenly started with CX = 2, 'A' would have stayed on the stack and RET would have taken it for the return address: IP = 0041h. The program would jump into the middle of the PSP and not terminate properly.

The same method can also be used to reverse a string in place in memory without printing it. In the second loop the character is taken with `POP AX` and written back into the string from the start with `MOV [SI], AL`; for this, SI is first brought back to the start of the string. The stack acts here as an intermediate buffer. There is no need to define a separate buffer array, but each character takes 2 bytes on the stack.

Common mistake

Assuming AX does not change after an INT 21h call. Function AH = 02h makes AL equal to the printed character; function AH = 09h leaves 24h ('$') in AL. If a loop counter or an intermediate result is kept in AL, write `PUSH AX` before the call and `POP AX` after it.

8.1.3.7 What happens to the registers?

The prediction question in the slides is a piece of code in which PUSHes and POPs are used in a mixed order. At the start SP = FFFEh.

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

The solution is found by writing down the contents of the stack step by step. After the three PUSHes the stack, from top to bottom, is 3333h, 2222h, 1111h and SP = FFF8h. `POP AX` takes the top 3333h: AX = 3333h. `POP CX` takes the next 2222h: CX = 2222h, SP = FFFCh. Only 1111h is left on the stack. `PUSH AX` pushes 3333h again and `POP BX` takes it right away: BX = 3333h. The last `POP BX` takes the 1111h on the stack and overwrites BX's value from a moment ago: BX = 1111h.

The result is AX = 3333h, BX = 1111h, CX = 2222h, and SP = FFFEh. Since four PUSHes and four POPs were done, the stack is balanced. The values have gone one full turn among the registers: AX took CX's old value, CX took BX's, and BX took AX's. In questions like this the most reliable method is to draw the stack as a column on paper and mark the top cell at every instruction.

8.1.3.8 Comparing swap methods

There are three ways to swap two registers. Swapping with the stack takes four instructions: `PUSH AX`, `PUSH BX`, `POP AX`, `POP BX`. Swapping with a temporary register takes three `MOV`s: `MOV CX, AX`, `MOV AX, BX`, `MOV BX, CX`. The third way is the 8086's swap instruction: `XCHG AX, BX`.

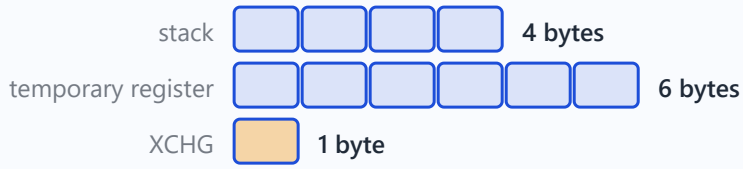


Figure 8.1.7: Code size of the three swap methods: 4 bytes with the stack, 6 bytes with a temporary register, 1 byte with XCHG.

Method	Bytes	Memory accesses	Clock cycles	Side effect and scope
with the stack	4	4	2 × 11 + 2 × 8 = 38	no register is disturbed; also works for segment registers
with a temporary register	6	0	3 × 2 = 6	the value of CX is destroyed
`XCHG AX, BX`	1 (93h)	0	3	shortest and fastest; cannot be used with segment registers

Table 8.1.15: Comparison of swap methods (8086 clock cycles).

Table 8.1.15 gives a clear conclusion: the best way to swap two general-purpose registers is `XCHG`. The instruction is one byte. For `XCHG` swapping AX with another 16-bit register there are short codes of the form 90h + r; since the number of BX is 3, the code is 93h. At 5 MHz, swapping with the stack takes 38 × 200 ns = 7.6 µs, and XCHG takes 0.6 µs. The real value of the stack method is not swapping but saving. Preserving registers during a procedure or an interrupt, holding intermediate results temporarily, and copying segment registers (`PUSH DS` , `POP ES`) are the real jobs of the stack. Since segment registers cannot be swapped with `XCHG` , swapping them is also done with the stack: `PUSH DS` , `PUSH ES` , `POP DS` , `POP ES` .

8.1.3.9 The interactive stack simulator

The interactive simulator in the slides shows the registers AX, BX, CX, and SP and the stack cells around SS:SP. The register values can be edited in hexadecimal. Six buttons run the instructions `PUSH AX` , `PUSH BX` , `PUSH CX` , `POP AX` , `POP BX` , and `POP CX` , and the instructions run accumulate in a list. With the simulator all the ordering examples of this note can be tried again: mirror order, swapping, rotating three registers.

An important case the simulator shows is POP from an empty stack. The real 8086 does not prevent it. If POP is done when the stack is empty, SP still increases by 2 and whatever is at that address in the segment is read. There is no way for the processor to know whether the stack is empty or full; SP is just a number. Keeping track of the limits of the stack is entirely the programmer's job.

8.1.4 Saving the flags: PUSHF and POPF

8.1.4.1 The flags word

The flags register (FLAGS) of the 8086 is a 16-bit register, but only nine of its bits are used. Six status flags describe the result of the last arithmetic or logic operation: CF (carry, bit 0), PF (parity, bit 2), AF (auxiliary carry, bit 4), ZF (zero, bit 6), SF (sign, bit 7), and OF (overflow, bit 11). Three control flags determine the behavior of the processor: TF (trap, bit 8), IF (interrupt, bit 9), and DF (direction, bit 10). The remaining bits are unused. On the 8086, bit 1 and bits 12–15 always read as 1; bit 3 and bit 5 are 0.

Definition 8.1.5

PUSHF pushes the 16-bit word of the flags register onto the stack: SP ← SP − 2, [SS:SP] ← FLAGS. **POPF** loads the word on top of the stack into the flags register: FLAGS ← [SS:SP], SP ← SP + 2. PUSHF does not change the flags; POPF can change all the flags at once.

In a program the flags are read one at a time: `JC` looks at CF and `JZ` at ZF. There is no `MOV` that accesses the whole flags register directly. `LAHF` copies only the low byte (SF, ZF, AF, PF, CF) into AH; it does not reach OF, DF, IF, and TF. The only way to save and restore the whole flags register is the stack. The stack is also used to bring the flags word into a register: `PUSHF` followed by `POP AX`. At the start of a .COM program these two instructions bring F202h into AX. F000h is the four constant 1 bits, 0200h is IF = 1, and 0002h is the constant 1 in bit 1. After `STC` the same two instructions give F203h; the difference is bit 0, CF.

8.1.4.2 PUSHF and POPF: code trace

The program below saves the flags of an addition, puts in between another addition that destroys the flags, and restores the saved flags.

```
ORG   100h
MOV   AX, 0FFFFh
ADD   AX, 1       ; CF=1, ZF=1
PUSHF             ; save flags
MOV   BX, 5
ADD   BX, 3       ; CF=0, ZF=0
POPF              ; restore flags
RET
```

The number in the first line is written as 0FFFFh. If a hexadecimal number starts with a letter, a 0 is added in front so the assembler does not take it for a name; writing FFFFh gives an error. `ADD AX, 1` computes FFFFh + 1 = 1 0000h. The result is 17 bits; the low 16 bits, 0000h, are written to AX and the

1 carried out goes to CF. Since the result is zero, ZF = 1. The low byte of 0000h has no 1 bits, that is, an even number of them, and PF = 1. In the low nibble F + 1 = 10h produces a carry, so AF = 1. SF = 0 and OF = 0: −1 + 1 = 0 is correct as signed, too.

`PUSHF` pushes the flags word at this moment. The value of the word is computed from the bits: the constant bits F000h + 0002h, 0200h for IF, 0040h for ZF, 0010h for AF, 0004h for PF, and 0001h for CF. The total is F257h. [FFFCh] = 57h and [FFFDh] = F2h. `MOV BX, 5` does not change the flags. `ADD BX, 3` computes 5 + 3 = 8: no carry, the result is not zero. CF, ZF, PF, and AF all become 0. 8 = 0000 1000b has a single 1 bit, so PF = 0. The flags of the first addition are lost. `POPF` loads F257h into the flags register, and CF = 1, ZF = 1, PF = 1, AF = 1 come back. AX and BX are not touched; AX = 0000h and BX = 0008h remain.

Instruction	AX	BX	SP	Flags	Stack (from the top)
`MOV AX, 0FFFFh`	**FFFFh**	0000h	FFFEh	odItszapc	0000h
`ADD AX, 1`	**0000h**	0000h	FFFEh	**odItsZAPC**	0000h
`PUSHF`	0000h	0000h	**FFFCh**	odItsZAPC	**F257h, 0000h**
`MOV BX, 5`	0000h	**0005h**	FFFCh	odItsZAPC	F257h, 0000h
`ADD BX, 3`	0000h	**0008h**	FFFCh	**odItszapc**	F257h, 0000h
`POPF`	0000h	0008h	**FFFEh**	**odItsZAPC**	0000h
`RET`	0000h	0008h	**0000h**	odItsZAPC	(empty); IP = 0000h

Table 8.1.16: Trace of the PUSHF/POPF program (IF = 1 throughout; flags in the order ODITSZAPC, capital letter = 1).

The final state of the program is seen in Table 8.1.16. What would have happened had `POP BX` been written instead of POPF? F257h would again have been taken from the stack, but this time written to BX: BX = F257h. The flags would have stayed CF = 0 and ZF = 0 as after the second addition. This experiment shows the difference between POPF and POP clearly. Both instructions take the same word from the stack; only their destinations differ.

The program is 15 bytes. `ADD BX, 3` is encoded with three bytes (83 C3 03): the opcode 83h says the 8-bit immediate is sign-extended to 16 bits. `ADD AX, 1`, on the other hand, is encoded with the short form special to AX as 05 01 00. PUSHF (9Ch) and POPF (9Dh) are one byte each.

Common mistake

Pushing another value onto the stack before restoring the flags saved with PUSHF, and then loading that value into the flags register with POPF. In the sequence `PUSHF`, `PUSH AX`, …, `POPF`, `POP AX`, POPF loads the value of AX as flags, and the old flags word arrives in AX. Flags and registers obey mirror order too: `PUSH AX`, `PUSHF`, …, `POPF`, `POP AX`.

8.1.4.3 Changing the flags word

There are direct instructions for CF, DF, and IF: `STC` / `CLC` / `CMC`, `STD` / `CLD`, `STI` / `CLI`. There is no instruction that changes TF. The only way to set TF to 1 goes through the stack. The flags word is pushed onto the stack, taken into a register, bit 8 is changed, the word is pushed again, and it is loaded into the flags register with POPF:

```
PUSHF
POP  AX
OR   AX, 0100h     ; bit 8 = TF
PUSH AX
POPF               ; TF = 1
```

Debuggers use this pattern to run a program one step at a time. When TF = 1, the processor generates a type 1 interrupt after every instruction. OF can be changed the same way; there is no instruction that sets OF directly either. In this pattern AX is destroyed; if AX is to be preserved, the pattern is placed between `PUSH AX` and `POP AX`.

Another use of changing the flags word on the stack is in interrupt routines. If an interrupt routine wants to return error information to the calling program through CF, it changes bit 0 of the FLAGS copy on the stack. When IRET restores this copy, the calling program sees CF. Most DOS functions report errors this way, as CF = 1.

8.1.4.4 Instructions that use the stack automatically

Some instructions do by themselves what PUSHF and POPF do. `INT n` first pushes the flags word, then CS and IP; it then clears IF and TF. `IRET` restores IP, CS, and the flags word. Interrupt routines therefore do not save the flags themselves; INT does this automatically. In ordinary code, PUSHF and POPF are needed to preserve a result, because CALL does not save the flags.

`CALL` and `RET` also use the stack automatically. A near CALL pushes the offset of the instruction after it, that is, the return address, and decrements SP by 2. RET pops this address into IP and increments SP by 2. A far CALL pushes CS first and then IP; RETF takes the two back in reverse order. From the stack's point of view, these instructions are PUSHes and POPs whose destination is IP. If a PUSH inside a procedure does not have a matching POP, RET takes the forgotten value instead of the return address. All the stack safety rules rest on this simple observation.

Instruction	Pushes (in order)	Popped by	SP difference
`CALL` (near)	IP	`RET`	2 bytes
`CALL` (far)	CS, IP	`RETF`	4 bytes
`INT n`, hardware interrupt	FLAGS, CS, IP	`IRET`	6 bytes

Table 8.1.17: The words pushed and popped by instructions that use the stack automatically.

8.1.5 Stack safety

8.1.5.1 The balance rule and the end of a .COM program

The PUSH and POP instructions hold temporary data while the program runs. The integrity of the stack is kept by an equal number of PUSHes and POPs. This rule must hold not on just one path of the program but on every path and in every branch. If the numbers of PUSHes and POPs are not equal, the stack is corrupted and it may not be possible to return to the operating system.

Definition 8.1.6

A piece of code is **stack-balanced** if the value of SP at its exit equals the value of SP at its entry. In a balanced piece every PUSH (and PUSHF) has a matching POP (or POPF). If the piece has more than one exit, the balance must hold for each exit separately.

Why balance matters so much shows at the end of a .COM program. RET expects a return address on top of the stack. When DOS starts a .COM program, it puts 0000h on the stack. If the program is balanced, the final RET takes this word and goes to CS:0000h, the start of the PSP. The `INT 20h` there terminates the program. If the program is not balanced, RET takes some other word.

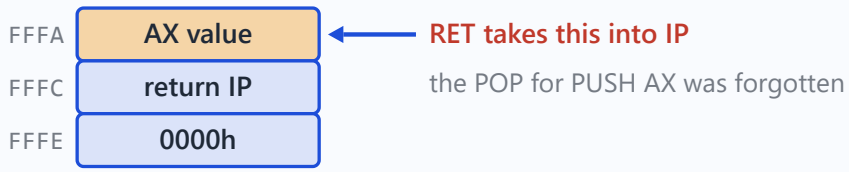


Figure 8.1.8: An extra PUSH: instead of the return address, the value of AX is left on top of the stack, and RET takes it into IP.

Stack errors come in three forms, and each gives different symptoms (Figure 8.1.8). With an **extra PUSH**, RET takes the data put last instead of the return address and jumps to a random address. With an **extra POP**, the return address is "eaten" into a register; RET takes the word after it and again returns to the wrong place. With the **wrong order**, the stack is balanced and the program returns to the right place, but the values get mixed up. The program silently runs wrong. The third is the hardest to find, because the program does not crash.

8.1.5.2 Error example: an unbalanced PUSH

In the program below AX is put on the stack but never taken back.

```
ORG   100h
MOV   AX, 1234h
PUSH  AX          ; no POP!
RET               ; IP ← ?
```

Instruction	IP (next)	AX	SP	Stack (from the top)
start	0100h	0000h	FFFEh	0000h
`MOV AX, 1234h`	0103h	**1234h**	FFFEh	0000h
`PUSH AX`	0104h	1234h	**FFFCh**	**1234h, 0000h**
`RET`	**1234h**	1234h	**FFFEh**	0000h

Table 8.1.18: Trace of the unbalanced PUSH program.

RET takes the word on top of the stack for the return address (Table 8.1.18). IP = 1234h and SP goes up to FFFEh. The processor starts executing the bytes at address 0700:1234h as instructions. This program is only 5 bytes; there is no program at address 1234h, only whatever was left in that region of memory. The program crashes or freezes. The odd thing is that DOS's 0000h word is still on the stack. The information needed for the return has not been lost; one word has just been left below.

The fix is easy: add a `POP AX` before RET, or do not write the PUSH at all. The real difficulty is that the imbalance often hides in one branch. In the code below, AX is pushed on both branches but popped on only one:

```
        PUSH AX
        CMP  BX, 0
        JE   done         ; no POP on this branch!
        ; … work using AX …
        POP  AX
done:   RET
```

As long as BX ≠ 0 the program works correctly. The first time BX = 0, RET jumps to the value of AX. Errors of this kind are not easily caught in testing. The rule is: the number of PUSHes and POPs of each branch (JMP, Jcc) is counted separately. The safest solution is to put the POP under a single exit label and direct all branches there. In this example the fix is to jump to a label before the POP instead of `JE done`.

8.1.5.3 Extra POP and wrong order

The effect of an extra POP can also be seen in a trace:

```
ORG  100h
MOV  AX, 5
PUSH AX
POP  BX
POP  CX            ; extra POP
RET
```

`PUSH AX` and `POP BX` are balanced; BX = 0005h and SP = FFFEh. The extra `POP CX` takes DOS's 0000h word into CX and makes SP FFFEh + 2 = 0000h. SP has wrapped around in 16 bits. RET now reads the word at SS:0000h. This address is the start of the PSP, and its first two bytes are CD 20. In little-endian order this word is 20CDh. IP = 20CDh and the processor jumps to an address unrelated to the program. The return address, 0000h, has been "eaten" by CX.

The wrong order gives a different symptom. The sequence `PUSH AX`, `PUSH BX`, ..., `POP AX`, `POP BX` is balanced; RET returns to the right place. But the values of AX and BX have been swapped. The program that called the procedure expects to find in AX the value it left there and continues with a wrong value. The error shows up only when this value is used, and often far from the procedure.

8.1.5.4 Stack overflow and underflow

If the imbalance is repeated continuously instead of once, the stack goes beyond its limits. The 8086 does not check these limits; there is no error interrupt, only corruption.

Definition 8.1.7

Stack overflow is the pushing of more values than the area set aside for the stack can hold, so that SP goes below the stack area. **Stack underflow** is the popping of a value that was never pushed, so that SP goes above the stack area.

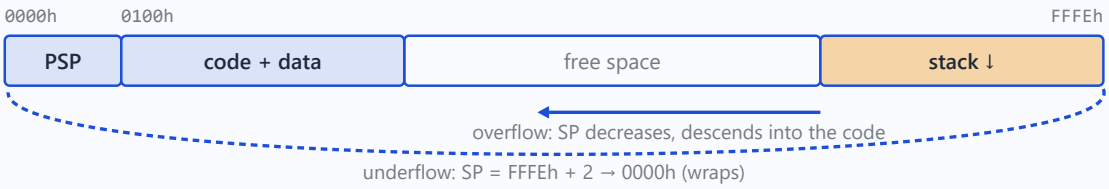


Figure 8.1.9: In a .COM program the PSP, code, and data are at the bottom of the segment; the stack grows downward from FFFEh. On overflow SP gets smaller and descends into the code; on underflow SP wraps from FFFEh to 0000h.

Typical causes of **overflow** are infinite recursion and a PUSH without a POP inside a loop. As SP gets smaller the stack descends and, in a .COM program, overwrites the data in the same segment and then the code (Figure 8.1.9). After the code is overwritten, in the next round the processor executes

stack data as instructions. The symptom is often seen far from where the overflow began.

A small example of a PUSH without a POP in a loop is:

```
        ORG  100h
        MOV  CX, 3
        MOV  AX, 0100h
again:  PUSH AX
        INC  AX
        LOOP again
        RET
```

The loop runs three iterations and pushes 0100h, 0101h, and 0102h onto the stack. SP = FFF8h. RET takes the 0102h on top of the stack and IP = 0102h. This address is inside the program but is not the start of an instruction: `MOV CX, 3` occupies 0100h–0102h and 0102h is its last byte. The processor takes the 00h byte for the start of a new instruction and begins executing a meaningless sequence. Instead of three iterations, a loop that starts with CX = 0 runs 65,536. In that case about 128 KB of values are pushed onto the stack. The 64 KB segment cannot hold that much: SP wraps around and the stack overwrites the code of the program.

Underflow is popping a value that was never pushed. If POP is done with SP = FFFEh, SP = 0000h; the 16-bit addition wraps. The value read is garbage or, in a .COM program, DOS's 0000h. The next PUSH takes SP down to FFFEh and overwrites DOS's word. The program then cannot terminate properly with RET, because the 0000h needed for the return has been overwritten.

The way to prevent overflow is to choose a stack area of sufficient size. The size of the stack area is chosen from the deepest call chain and the largest number of PUSHes of the program. Each CALL takes 2 bytes (4 bytes for a far call), each PUSH 2 bytes, and each interrupt 6 bytes. Interrupts can arrive at any moment of the program; hardware interrupt routines use the same stack and need room for the registers they save. In .EXE programs a separate stack segment of sufficient size is reserved with a directive such as `.STACK 100h`. A separate segment has another advantage: even if the stack overflows it does not overwrite the code, it only corrupts the memory below its own segment.

Example 8.1.3

A program makes at most four levels of nested near CALLs. Each procedure pushes three registers on entry. A hardware interrupt can arrive at any moment of the program, and the interrupt routine pushes four registers. How deep can the stack get at most?

1. Calls: 4 levels × (2 bytes return address + 3 × 2 bytes registers) = 4 × 8 = 32 bytes.
2. Interrupt: 6 bytes (FLAGS, CS, IP) + 4 × 2 bytes of registers = 14 bytes.
3. Total: 32 + 14 = 46 bytes.

The stack gets at most 46 bytes deep. The 256 bytes reserved with `.STACK 100h` are more than enough for this program; the margin is left for procedures added later and for the BIOS routines' own use.

8.1.5.5 Good practices

Most stack errors are prevented with a few coding habits. The six rules in the slides are:

1. **Mirror order.** The reverse of the order `PUSH AX`, `PUSH BX`, `PUSH CX` is `POP CX`, `POP BX`, `POP AX`. Written one below the other, the code should look like a mirror image.
2. **Single exit.** Use a single exit label in a procedure instead of several RETs. The POPs are written once and all branches pass through them.
3. **Do not touch SP.** SP is changed directly for only two jobs: clearing the parameters passed on the stack (`ADD SP, n`) and reserving room for local variables (`SUB SP, n`). n must always be even.
4. **The BP frame.** Data on the stack cannot be reached with `[SP+n]`, because on the 8086 SP is not a base register. Do `MOV BP, SP` and use `[BP+n]`; BP addresses the SS segment by default.
5. **Protect the segments.** Routines that change DS or ES must PUSH and POP them as well. The calling program cannot know that its data segment has changed.
6. **Check with emu8086.** Open the stack window and check that SP returns to its value at entry before RET.

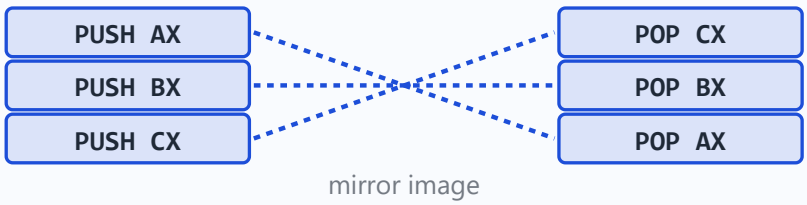


Figure 8.1.10: Mirror order. The reverse of the order PUSH AX, BX, CX is POP CX, BX, AX; the lines join the same register and do not cross.

Figure 8.1.10 gives mirror order visually. The lines between the PUSHes and POPs form nested arcs. If the lines cross, the order is wrong. This picture resembles the nesting of opening and closing parentheses: the parenthesis opened last is closed first.

The skeleton below shows all these rules together:

```
my_proc PROC
        PUSH AX          ; registers in use
        PUSH BX
        PUSH DS          ; a segment will change too
        ; … work …
        JC   done          ; the error branch also leaves through the exit
        ; … work …
done:   POP  DS            ; mirror order
        POP  BX
        POP  AX
        RET
my_proc ENDP
```

8.1.5.6 Tracing the stack in emu8086

All the trace tables of this note can be reproduced step by step in emu8086. The method has four steps.

1. **Compile and load.** The program is assembled with the compile button and loaded into the emulator with emulate. The register panel shows the initial values of CS, IP, SS, and SP: for a .COM program 0700h, 0100h, 0700h, and FFFEh.
2. **Single step.** The single step button runs one instruction per press, and the registers that change are highlighted. Step back undoes the last step; it is handy for seeing the effect of a PUSH again.
3. **The stack window.** The stack button at the bottom opens a window that shows the words around SS:SP. The row SP points at is marked. With every PUSH the marker moves up a row (to a smaller address), and with every POP down a row.
4. **The memory view.** The list in the middle of the emulator shows the bytes in hexadecimal, decimal, and ASCII. The bytes `B8 12 12` are the instruction `MOV AX, 1212h`. The little-endian convention shows up here: `MOV AX, 1234h` is encoded as B8 34 12.

After loading, CX shows the byte count of the program; in the swap example this value is 000Bh, that is, 11 bytes. A program must not rely on this value and must load CX itself before using it. The 0000h seen on the FFFEh row of the emu8086 stack window is not a coincidence either; it is the return address for the RET at the end of the program. Looking at this row before every RET of a program is the shortest way to check the stack balance.

8.1.6 Exercises and solutions

The five exercises from the slides are solved here with intermediate steps.

1. With SS = 2000h and SP = 0100h, what are the physical addresses of the bytes written after PUSH AX?

PUSH first decrements SP: SP = 0100h − 2 = 00FEh. The new top of the stack is 2000h × 10h + 00FEh = 20000h + 00FEh = 200FEh. In little-endian order AL is written to 200FEh and AH to 200FFh. With AX = 1234h, [200FEh] = 34h and [200FFh] = 12h.

2. After MOV AX,1 / MOV BX,2 / MOV CX,3 / PUSH AX / PUSH BX / PUSH CX / POP BX / POP CX / POP AX, what are AX, BX, CX?

After the three PUSHes the stack, from top to bottom, is 3, 2, 1. `POP BX` takes 3, `POP CX` takes 2, and `POP AX` takes 1. The result is AX = 1, BX = 3, CX = 2. AX stays where it was, and BX and CX are swapped. AX stays fixed because it is both the first register pushed and the last one popped.

3. Copy the value of DS into ES using only PUSH and POP.

`PUSH DS` followed by `POP ES`. The two instructions are two bytes (1Eh, 07h) and disturb no general-purpose register. The stack stays balanced.

4. With SP = FFFEh, PUSH AX / PUSH BX / PUSHF / POP CX are executed. What is SP, and what arrives in CX?

Three pushes decrease SP by 6, and one pop increases it by 2: FFFEh − 6 + 2 = FFFAh. CX receives the word pushed last, that is, the flags word at the moment of PUSHF. At the start of a .COM program this value is F202h. Copies of AX and BX remain on the stack; if the program were ended with RET in this state, it would return to the value of BX.

5. Write a PUSH/POP sequence that preserves AX and the flags during an operation.

`PUSH AX`, `PUSHF`, … operation …, `POPF`, `POP AX`. Since the flags were pushed last, they are popped first; the sequence obeys mirror order. For example, with AX = 0000h and CF = ZF = 1, let `MOV AX, 7` and `ADD AX, 1` run inside the wrapper. At the end of the wrapper AX = 0000h, CF = 1, ZF = 1, and SP = FFFEh.

8.1.7 Summary

- The **stack** is an area of memory set aside for temporary data and working in last-in, first-out (**LIFO**) order. The programmer (PUSH, POP), the processor (CALL, INT), and the operating system use the same stack.
- The top of the stack is **SS:SP**; its physical address is SS × 10h + SP. In a .COM program SS = CS and SP = FFFEh; in an .EXE program the values in the header are used. BP also addresses the SS segment by default; SP cannot be used in an addressing expression.
- The stack **grows downward**: PUSH decreases SP by 2 and POP increases it by 2. The unit is always the word; `PUSH AL` is invalid. In a .COM program the code grows up and the stack down, and they share the free space in between.
- **PUSH**: first SP ← SP − 2, then [SS:SP] ← source. **POP**: first destination ← [SS:SP], then SP ← SP + 2. POP does not erase memory. SP always points at the topmost occupied cell.

- PUSH takes a register, a segment register, or a memory word; PUSH with an immediate exists only on the 80186 and later. There are no POP CS and no POP constant. On the 8086, `PUSH SP` writes the decremented value.
- A word is written to the stack **little-endian**: the low byte at the address SP points at, the high byte one above. SP is kept even; an odd SP makes every access take two bus cycles.
- PUSH, POP, and PUSHF **do not change the flags**; POPF loads all the flags. The result of a CMP is carried safely across PUSH and POP.
- When saving registers, the POP order is the reverse of the PUSH order (**mirror order**); POPping in the same order swaps or reverses the values. An array can be reversed with the stack. The best way to swap two registers is XCHG; the real job of the stack is saving.
- **PUSHF/POPF** save and restore the whole flags register. The flags word is brought into a register with PUSHF and POP; TF is changed this way. INT pushes the flags automatically; CALL does not.
- An **equal number of PUSHes and POPs** is done on every path and in every branch. In a .COM program the final RET takes the 0000h DOS put there, and the INT 20h in the PSP ends the program. With an extra PUSH, RET returns to data; with an extra POP, to beyond the stack.
- The 8086 does **not check overflow and underflow**: SP wraps in 16 bits and the stack can overwrite the code. The stack area is chosen taking into account the deepest call chain, the largest number of PUSHes, and interrupts.

8.1.8 Key terms

stack LIFO (last in, first out) push / pop (PUSH / POP) stack segment (SS) pointer register (SP, BP) base pointer (BP) flags register (FLAGS) stack overflow / underflow little-endian / big-endian segment:offset (logical address) physical address default segment return address register preservation data transfer instruction

---

8.1.9 Test Yourself

1. With SS = 3000h and SP = 0000h, `PUSH AX` runs. What is SP? To which physical addresses are the two bytes of AX written?

   Answer

   SP = 0000h − 2 = FFFEh; the 16-bit subtraction wraps. The top of the stack is 30000h + FFFEh = 3FFFEh. AL is written to 3FFFEh and AH to 3FFFFh. The initial value SP = 0000h can be used so that the stack starts at the very top of the segment; the first PUSH writes to the last word of the segment.

2. After the sequence `MOV AX, 1A2Bh`, `PUSH AX`, `MOV BX, SP`, `MOV DL, [BX+1]`, `MOV DH, [BX]`, `POP AX` (.COM program, starting with SP = FFFEh), what are BX and DX?

   Answer

   BX = FFFCh. [FFFCh] = 2Bh (low byte), [FFFDh] = 1Ah (high byte). DL = 1Ah, DH = 2Bh; DX = 2B1Ah. Because the bytes were read into the opposite places, DX is AX with its bytes exchanged.

3. With AX = AAAAh, BX = BBBBh, CX = CCCCh, `PUSH AX`, `PUSH BX`, `PUSH CX`, `POP BX`, `POP AX`, `POP CX` run. What are the three registers in the end?

   Answer

   The stack, from top to bottom, is CCCCh, BBBBh, AAAAh. BX = CCCCh, AX = BBBBh, CX = AAAAh. Each register has taken the value of another; the stack is balanced.

4. After the sequence `STC`, `PUSHF`, `CLC`, `POPF`, what is CF? If `POP AX` had been written instead of `POPF`, what would CF and AX be (at the start of the program IF = 1, the other flags 0)?

   Answer

   POPF restores the word saved after STC: CF = 1. With POP AX, CF stays 0 as CLC left it, and AX = F203h (constant bits F002h, IF 0200h, CF 0001h).

5. A .COM program ends with `MOV AX, 5`, `PUSH AX`, `PUSH AX`, `POP BX`, `RET`. What is IP after RET? Does the program terminate properly?

   Answer

   Two PUSHes are matched by one POP; the second 0005h is left on top of the stack. RET takes this value: IP = 0005h. The processor jumps to an address inside the PSP; the program never reaches INT 20h and does not terminate properly.

6. The 26-byte program that reverses an array with the stack occupies 0100h–0119h. How many words can be pushed at most onto a stack starting from SP = FFFEh without overwriting the program?

   Answer

   After n pushes SP = FFFEh − 2n, and SP ≥ 011Ah must hold. n ≤ (FFFEh − 011Ah) / 2 = (65,534 − 282) / 2 = 32,626. At most 32,626 words can be pushed.

7. A student puts `PUSH CX`, `INC CX`, `POP CX` between `CMP AX, BX` and `JE`. Does JE jump on the

result of the comparison?

Answer

No. PUSH and POP do not change the flags, but `INC CX` sets ZF again according to the new value of CX. JE now looks at the result of INC, not of CMP. The fix is to put PUSHF/POPF in between, or to move the INC before the comparison.

8. How many clock cycles does swapping with the stack (two PUSHes, two POPs) take on the 8086, compared with `XCHG AX, BX`? What is the difference in µs at 5 MHz?

Answer

With the stack: 2 × 11 + 2 × 8 = 38 cycles. XCHG AX, BX: 3 cycles. The difference is 35 cycles; at 5 MHz one cycle is 200 ns, so 35 × 200 ns = 7 µs.

8.1.10 Further reading

- Intel, *The 8086 Family User's Manual* (1979): the exact definitions, encodings, and clock cycle counts of PUSH, POP, PUSHF, POPF, CALL, RET, INT, and IRET; the meaning of the S3–S6 status bits. The values in Tables 8.1.2 and 8.1.6 are from this source.
- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 4: the stack instructions among the data transfer instructions, initializing the stack, and detailed examples of PUSH/POP.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), the section on stack operations: the runtime stack, preserving registers, and the example of reversing a string with the stack.

8.1.11 References

1. Intel, *The 8086 Family User's Manual*, 1979 — instruction definitions of PUSH, POP, CALL, RET, INT, IRET.
2. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapters 4 and 6.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — stack operations and procedures.
4. emu8086 help documentation: 8086 Assembler Tutorial, Part 9 (The Stack).

PART III · CHAPTER 8.2

Procedures

The same job is needed in many places of a program: printing a piece of text on the screen, splitting a number into decimal digits, searching for a value in an array. Rewriting the code for this job everywhere fills memory and multiplies errors. A procedure lets such a job be written once and called from any place. The 8086 does this with two instructions: CALL writes the return address onto the stack and jumps to the procedure, and RET takes that address from the stack and returns. This note first explains how a procedure is defined and the effect of CALL and RET on registers, the stack, and machine code. It then covers near and far calls, the indirect call, and the clock cycle cost of a call. Next it turns to how parameters are passed through registers, memory, addresses, and the stack, to preserving registers, to the stack frame, and to local variables. The last sections cover example programs, recursion, and common mistakes in procedures. Every piece of code is traced step by step together with registers, flags, and the stack.

LEARNING OUTCOMES

Define a procedure
Write a procedure with PROC/ENDP, place it in the right spot in the program, and tell an assembler directive from a processor instruction.

Trace CALL and RET
Show step by step, with the values of IP, SP, and the stack, how the return address is pushed and how RET takes it back; compute the relative displacement in the machine code of CALL.

Distinguish call types
Explain what near, far, and indirect calls push onto the stack and what RET, RETF, and RET n pop; show that the procedure type and the return instruction must match.

Pass parameters
Send data to a procedure through registers, memory variables, addresses, and the stack; get the result back through a register, a flag, or memory; compare passing by value with passing by address.

Build a stack frame
Reach parameters with fixed offsets such as [BP+4] and [BP+6] through BP, open room for local variables with SUB SP, and close the frame in the right order.

Write safe procedures
Preserve registers with PUSH/POP, keep the stack balanced, use a single exit point, compute the stack need of recursion, and avoid common mistakes.

8.2.1 The procedure concept and its definition

8.2.1.1 What is a procedure?

Assembly programs grow long quickly. Jobs such as printing a string on the screen, splitting a number into decimal digits, or searching for a value in an array are needed in many places of a program. Rewriting the same sequence of instructions wherever it is needed creates two problems. First, the program grows needlessly; on small 8086-based systems memory is a limited resource. Second, an error in one copy of the code is present in all copies, and each has to be fixed separately. The function and subroutine concepts of high-level languages solve these problems. Their counterpart in assembly language is the procedure.

Definition 8.2.1

A **procedure** (also called a subroutine) is a piece of code that performs a specific task and can be called from any place in the program. A procedure is called with the `CALL` instruction. When it finishes its task, it returns with the `RET` instruction to the instruction right after the one that called it. The address written to the stack at the moment of the call is called the **return address**.

The most important concept in the definition is the return address. A procedure can be called from different places in the program. The place to return to differs with each call, but the code of the procedure is the same. A procedure cannot keep in its own code where to return to. On every call the CALL instruction writes this information to the stack, and RET reads it from there. So the same RET instruction returns to one address on one call and to another address on another call.

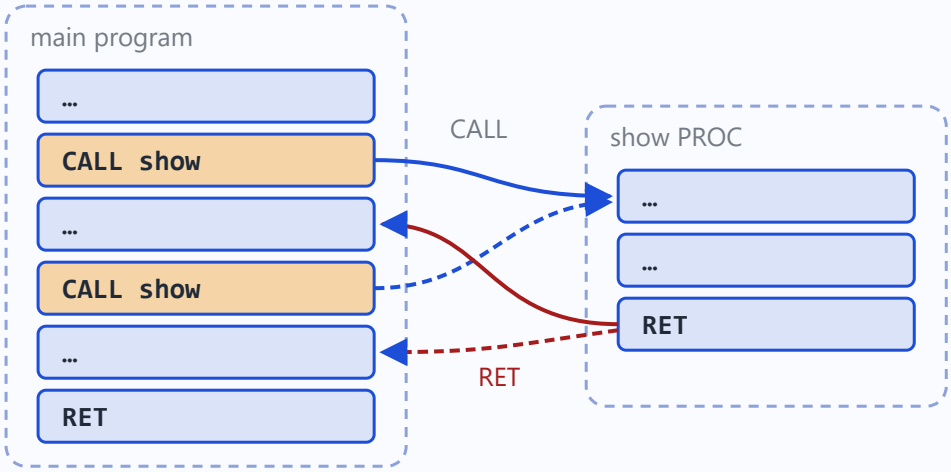


Figure 8.2.1: The main program calls the procedure `show` twice. Each RET returns to the instruction right after the call that started it.

Figure 8.2.1 shows this situation. The main program calls the procedure `show` from two separate places. The return of the first call goes behind the first CALL, and the return of the second call behind the second CALL. Only a single copy of the procedure is loaded in memory.

Using procedures has three benefits. **Reuse**: the code is written once and called as many times as wanted; code duplication and memory use go down. **Readability**: the program is divided into small tasks, each with a name. The line `CALL print_num` is much easier to read than the fifteen instructions that would take its place. **Maintenance**: an error is fixed in one place, and a procedure can be tested separately from the rest of the program. These properties make a program more structured and understandable.

A procedure has a cost, too. On every call the processor executes the extra CALL and RET instructions and writes the return address to memory and reads it back. This overhead can be computed in clock cycles. It is small, but it shows up in very short jobs that are called very often.

8.2.1.2 Defining with PROC and ENDP

In 8086 assembly language a procedure is written between a `PROC` line that starts with its name and an `ENDP` line that ends with the same name. The general form is:

```
name PROC     ; start (NEAR by default)
    ; procedure body goes here
    RET
name ENDP     ; end
```

The pattern has three rules. First, `name` is the name of the procedure and must be written the same at the start and at the end. A definition that starts with `p1 PROC` and ends with `p2 ENDP` is an error for the assembler. Second, the name follows the label rules: it starts with a letter or an underscore, cannot start with a digit, and cannot be the same as an instruction name. The assembler binds the name to the address of the first instruction of the procedure. When `CALL name` is written, this address is used. Third, there must be at least one `RET` inside the procedure. The processor does not see ENDP. If there is no RET, the flow continues after the last instruction of the procedure with whatever is in memory there.

The type can also be written on the PROC line: `name PROC NEAR` or `name PROC FAR`. If no type is written, the procedure is taken as **near** (NEAR). A near procedure is in the same code segment as the code that calls it. Most of the examples in this note are single-segment .COM programs and all their procedures are near. The effect of a far procedure on the stack is treated together with the types of CALL and RET.

The place of a procedure in the program matters too. The processor executes instructions in memory order and does not stop at a PROC line. If the procedure is written in the middle of the main program, when the main program gets there it executes the instructions of the procedure even though it has not been called. This situation is called **falling into the procedure**. The safe place is after the final RET of the main program:

```
ORG  100h
     CALL p1
     RET        ; main program ends
p1   PROC       ; after the RET:
     ; ...      ; never fallen into
     RET
p1   ENDP
END
```

Here the main program consists of two instructions. `CALL p1` calls the procedure, and the `RET` that follows ends the program. Since the instructions of the procedure sit after the RET of the main program, the flow reaches them only through CALL. If the procedure is to be written before the main program, a `JMP` that jumps over the procedure is put at the start of the main program. In emu8086 examples the second form is also common: `JMP start`, then the procedures, then the `start:` label.

Common mistake

Forgetting the RET at the end of the main program. In the program below there is no RET after `CALL twice`:

```
ORG 100h
MOV AX, 2
CALL twice
twice PROC
ADD AX, AX
RET
twice ENDP
```

CALL pushes 0106h as the return address, the procedure makes AX 4, and RET returns to 0106h. But 0106h is the first instruction of the procedure itself. The processor "falls into" the procedure once more, AX becomes 8, and this time RET takes the 0000h on the stack and ends the program. Although the procedure was called once, it ran twice, and the result is 8 instead of 4. The assembler cannot catch this error, because the code is syntactically correct.

8.2.1.3 Directives and instructions

Two different kinds of element stand side by side in a procedure definition. `PROC`, `ENDP`, and `END` are addressed to the assembler. `CALL` and `RET`, on the other hand, are instructions the processor executes. This distinction is the key to understanding how procedures work.

Definition 8.2.2

A **directive** is a line that tells the assembler what to do during translation. A directive is not translated to machine code and takes no space in memory. An **instruction** is an operation that is translated to one or more bytes and executed by the processor at run time.

Element	Kind	Job
`PROC` / `ENDP`	assembler directive	Tells the assembler the start address and the type (NEAR/FAR) of the procedure. It is not translated to machine code.
`CALL`	processor instruction	Pushes the return address onto the stack and jumps to the address of the procedure. The code of the near direct form is E8.
`RET`	processor instruction	Takes the return address from the stack and jumps there. The code of the near form is C3. It is used to return both from a procedure and from a .COM program.
`END`	assembler directive	Tells the assembler that the source file has ended.

Table 8.2.1: Elements used in a procedure definition.

Figure 8.2.2: PROC, ENDP, and END are used only during assembly and produce no bytes; CALL and RET are the instructions the processor runs.

Table 8.2.1 and Figure 8.2.2 summarize this distinction. Its consequence is: at run time there is no such object as a "procedure". In memory there are only instruction bytes. The name, start, and end of a procedure live only in the source code and in the assembler's symbol table. For the processor a procedure is nothing but an address range that is jumped into with CALL and left with RET. That is why the ENDP line does not do the work of an "end". What ends a procedure is the RET instruction inside it.

The second use of RET is also understood from this viewpoint. A .COM program ends like a procedure called by DOS. Before starting the program, DOS writes the `INT 20h` instruction into the first 256 bytes of the program's segment, the PSP (program segment prefix). Then it pushes the word 0000h onto the stack and starts the program at address 0100h. The final RET of the program takes the 0000h on the stack into IP. The processor executes the `INT 20h` at CS:0000h, and the program terminates. In emu8086, when a .COM program starts, SP = FFFEh and [FFFEh] = 0000h. In all the trace tables of this note, the 0000h at the bottom of the stack is this word pushed by DOS.

8.2.1.4 Modular programming

In large programs the procedures do not fit in one source file either. The program is divided into source files, each collecting related procedures. These files are called **modules**. Two directives are used to call a procedure in one module from another module:

```
; main.asm
EXTRN show:NEAR     ; in another file
      ...
      CALL show
; tools.asm
PUBLIC show         ; exported
show  PROC NEAR
      ...
      RET
show  ENDP
```

`PUBLIC show` opens the name `show` to other modules. `EXTRN show:NEAR` tells the assembler that this name is defined in another module and that its type is NEAR. When assembling main.asm, the assembler does not know the address of `show`. It leaves the address field of the CALL instruction blank and leaves a note in the object file saying "the address of show goes here".

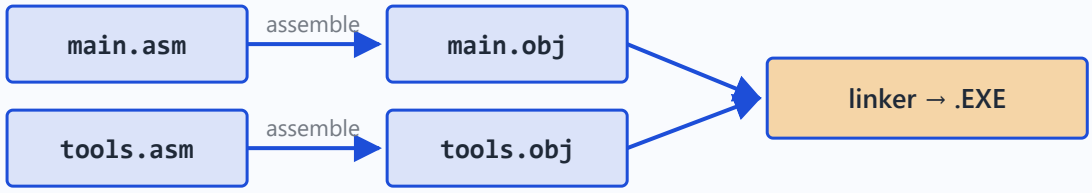


Figure 8.2.3: Two source files are assembled separately into object files; the linker combines them into a single program.

Each module is assembled separately and becomes an object file (.OBJ) (Figure 8.2.3). The linker places the object files one after another, finds the final address of each procedure, and fills in the blank fields in the CALL instructions. Frequently used object files can be gathered in a library (.LIB) file. The linker takes from the library only the procedures the program actually calls. The benefit of this method is that unchanged modules are not reassembled and the same library is used in many programs.

emu8086 does not do separate assembly and linking. In this environment shared code is added with the `INCLUDE` directive. INCLUDE inserts the text of the named file into the source as it is, and the program is assembled as a single piece. The `emu8086.inc` file that comes with emu8086 is used this way. The difference between the two methods is that the linker combines object code, while INCLUDE combines source text.

8.2.2 CALL and RET

8.2.2.1 CALL and RET step by step

The job of CALL and RET is in short “call and return”. Inside the processor this job is split into a few small steps. The smallest example is used to see the steps: the main program calls the procedure `m1`, m1 writes 5 to BX and returns, and the main program writes 2 to AX and ends.

```
ORG     100h
CALL    m1          ; 0100h: E8 04 00
MOV     AX, 2       ; 0103h: B8 02 00
RET                 ; 0106h: C3 → DOS
m1 PROC
MOV     BX, 5       ; 0107h: BB 05 00
RET                 ; 010Ah: C3 → caller
m1 ENDP
END
```

The addresses and bytes in the comments are how the program looks in memory. `ORG 100h` says the first instruction will be placed at address 0100h. CALL is 3 bytes and occupies 0100h–0102h. The next instruction starts at 0103h. The first instruction of the procedure, `MOV BX, 5`, is at 0107h; the value of the label m1 is therefore 0107h.

Definition 8.2.3

A **near CALL** does two things: (1) SP ← SP − 2 and [SS:SP] ← IP, where IP is the address of the instruction after CALL. (2) IP ← target address. A **near RET** does the opposite: IP ← [SS:SP] and SP ← SP + 2. Neither instruction touches the flags.

The phrase “IP is the address of the instruction after CALL” in the definition comes from how the processor works. In the 8086 the bus interface unit (BIU) fetches instruction bytes from memory and puts them in the 6-byte instruction queue. The execution unit (EU) takes an instruction from the queue and decodes it. When the instruction has been decoded, IP points right behind that instruction's bytes. At the moment CALL starts executing, IP = 0103h. This is the value written to the stack as the return address. The processor does not do an extra addition.

Step	Operation	IP	SP	Stack
0	start, the next instruction is `CALL m1`	0100h	FFFEh	[FFFEh] = 0000h
1	CALL decoded: IP points at the next instruction	0103h	FFFEh	unchanged
2	SP ← SP − 2, [SP] ← IP	0103h	**FFFCh**	**[FFFCh] = 0103h**
3	IP ← 0103h + 0004h	**0107h**	FFFCh	unchanged
4	`MOV BX, 5` executed	010Ah	FFFCh	unchanged
5	`RET` : IP ← [SP], SP ← SP + 2	**0103h**	**FFFEh**	the 0103h at [FFFCh] is now invalid
6	`MOV AX, 2` executed	0106h	FFFEh	[FFFEh] = 0000h

Table 8.2.2: The internal steps of `CALL m1` and `RET` (SS = 0700h, SP = FFFEh at the start).

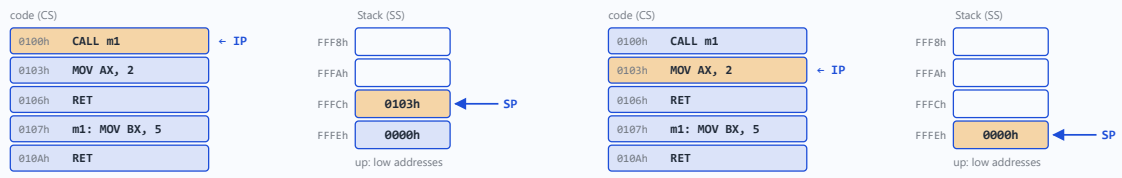


Figure 8.2.4: On the left, after CALL has pushed the return address: [FFFCh] = 0103h, SP = FFFCh. On the right, after RET: IP = 0103h, SP = FFFEh; the stack is back in its state before the call.

Table 8.2.2 gives the steps in order. Three points deserve attention. First, the stack of the 8086 grows downward. CALL first decrements SP and then writes. The return address is therefore written one word below FFFEh, at FFFCh. Second, the target address in step 3 is found with an addition: 0103h + 0004h = 0107h. The value 0004h is the relative displacement inside the CALL instruction; it is

explained in detail when the machine code is examined. Third, RET does not erase the value it reads from the stack. 0103h stays at address FFFCh, but since SP now points above it, it is considered invalid. The next PUSH or CALL overwrites this word.

The hardware counterpart of these steps can also be followed. CALL's write to the stack is a memory write bus cycle. The physical address is found as SS × 16 + SP: 07000h + FFFCh = 16FFCh. Since the address is even, the 16-bit word is written in a single bus cycle. In this cycle A0 = 0 and BHE = 0, and the two data bytes pass together on lines D0–D7 and D8–D15. Had SP been at an odd value, the same word would have needed two bus cycles. That is why SP is always kept even. RET does a read bus cycle from the same address.

CALL and RET have one more hardware cost. The BIU may already have fetched into the queue the bytes after CALL, that is, the bytes of `MOV AX, 2` at 0103h. When IP changes to 0107h these bytes are useless. The processor flushes the queue and the BIU starts fetching from the new address. The next instruction waits until its bytes arrive from memory. The same happens at RET. This is one reason the clock cycle counts of jump and call instructions are high.

8.2.2.2 A simple procedure: code trace

The same program is now traced instruction by instruction from beginning to end. Each row of Table 8.2.3 shows the state after that instruction has executed. It is the same as the values obtained by running the program in emu8086.

Line	Instruction	IP (next)	SP	AX	BX	Stack
—	start	0100h	FFFEh	0000h	0000h	[FFFEh] = 0000h
2	`CALL m1`	**0107h**	**FFFCh**	0000h	0000h	[FFFCh] = 0103h
6	`MOV BX, 5`	010Ah	FFFCh	0000h	**0005h**	[FFFCh] = 0103h
7	`RET`	**0103h**	**FFFEh**	0000h	0005h	[FFFEh] = 0000h
3	`MOV AX, 2`	0106h	FFFEh	**0002h**	0005h	[FFFEh] = 0000h
4	`RET` (DOS'a)	**0000h**	**0000h**	0002h	0005h	empty

Table 8.2.3: Trace of the m1 program (each row is the state after execution).

The line order is 2, 6, 7, 3, 4. The flow does not follow the order in the source code: CALL jumps to the procedure and RET brings it back. In the last row the main program's RET takes the 0000h on the stack into IP, and SP becomes 0000h, the part of FFFEh + 2 = 10000h that fits in 16 bits. The processor executes the `INT 20h` at 0700:0000 and the program ends. The result is AX = 0002h, BX = 0005h.

It is also instructive to look at the rows of the table where the registers do not change. CALL and RET do not touch AX and BX. The BX written by the procedure is still 0005h after the return. The procedure and the main program share the same set of registers; the procedure has no registers of its

own. This makes passing parameters easy. But the danger of the procedure destroying the caller's registers arises from the same fact. Parameter passing and register preservation deal with these two sides.

8.2.2.3 Machine code in memory

When the assembler translates the source code to bytes, the m1 program takes only 11 bytes in memory: `E8 04 00 B8 02 00 C3 BB 05 00 C3`. No trace of ORG, PROC, ENDP, and END is left.

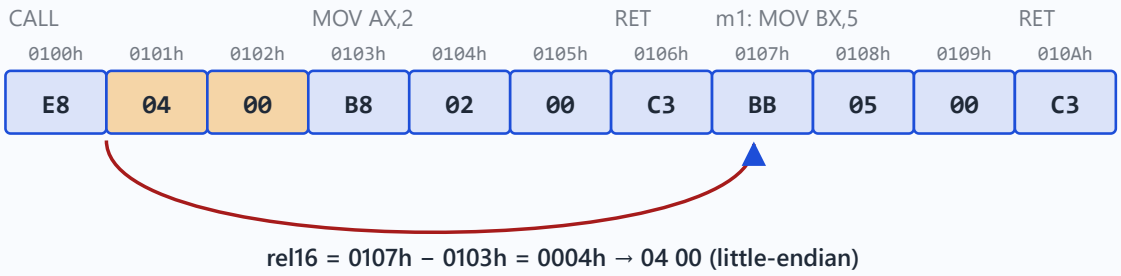


Figure 8.2.5: The machine code bytes between 0100h and 010Ah. The 16-bit displacement field of the CALL instruction (rel16) is 0004h and is stored as 04 00 in little-endian order.

The first three bytes in Figure 8.2.5 are the CALL instruction. E8 is the opcode of "near, direct CALL". The two bytes after it are not the absolute address of the target. They are the distance of the target relative to the instruction after CALL.

Definition 8.2.4

The **relative displacement** (rel16) of a near direct CALL is the difference between the target address and the address of the instruction after the CALL: rel16 = target − (address of CALL + 3). While running, the processor computes IP ← IP + rel16. The difference is a 16-bit two's complement number; if the target is behind, rel16 is negative.

For m1, rel16 = 0107h − (0100h + 3) = 0107h − 0103h = 0004h. The 8086 stores multibyte values in little-endian order: the low byte first (04), then the high byte (00). The instruction is therefore `E8 04 00`.

Using a displacement has a benefit. If the code is moved elsewhere in memory, the difference between the CALL and its target does not change and the instruction keeps working correctly. This property is called **position-independent code**. Had an absolute address been stored, the address would have had to be fixed up every time the code was moved.

Example 8.2.1

Let us find the machine code of two CALLs: (a) the second `CALL m2` at 0107h in the `m2` example, m2 = 010Bh; (b) the `CALL p` at 0120h, p = 0100h.

1. **Next instruction.** (a) 0107h + 3 = 010Ah. (b) 0120h + 3 = 0123h. These addresses are also the return addresses to be pushed onto the stack.
2. **Difference.** (a) 010Bh − 010Ah = 0001h. (b) 0100h − 0123h = −23h. In 16-bit two's complement −23h = 10000h − 23h = FFDDh.
3. **Bytes.** The low byte is written first. (a) `E8 01 00`. (b) `E8 DD FF`.
4. **Check.** For (b) the processor computes IP ← 0123h + FFDDh. The sum is 10100h; the carry that does not fit in 16 bits is discarded and IP = 0100h.

The name of the procedure does not appear in the machine code either. `m1` exists only in the assembler's symbol table; of it, the machine code keeps only the address 0107h stored as a displacement. The debugger of emu8086 can show names because it stores the symbol table separately.

8.2.2.4 Types of CALL and RET

The 8086 has several forms of CALL and RET. The forms are distinguished by two questions. Is the target in the same code segment (near) or in another one (far)? Is the target inside the instruction (direct) or in a register or memory (indirect)?

Instruction	Code	Bytes	Pushes / pops	Description
CALL near	E8 rel16	3	IP	Direct call within the same segment (default).
CALL register/memory	FF /2	2–4	IP	Indirect near call: `CALL BX`, `CALL [table+SI]`.
CALL FAR	9A ofs seg	5	CS, then IP	Procedure in another segment; CS:IP change together.
RET	C3	1	IP	Near return.
RETF	CB	1	IP, then CS	Far return (FAR procedure).
RET n	C2 n16	3	IP; SP += n	On return also discards n bytes of parameters from the stack.

Table 8.2.4: Forms of CALL and RET.

The notation “FF /2” in Table 8.2.4 says that the opcode is FF and that the middle field of the ModR/M byte that follows is 2. The ModR/M byte indicates whether the operand is a register or a memory address. The length of an indirect CALL therefore varies from 2 to 4 bytes depending on the operand. `CALL BX` is 2 bytes (FF D3). `CALL table[SI]` carries a 16-bit displacement, so it is 4 bytes (FF 94 10 01). The far RET also has a `RETF n` form (CA n16).

The assembler encodes the `RET` inside a procedure according to the type of the procedure. In a procedure declared `name PROC NEAR` or with no type written, RET becomes C3. Inside `name PROC FAR` the same word RET is encoded as CB, that is, RETF. The programmer writes the type once on the PROC line, and the assembler fits the return instructions. CALL also looks at the type of the target: if the target is a FAR procedure, the assembler produces the 5-byte far call that starts with 9A.

Figure 8.2.6: A near CALL pushes only IP onto the stack (SP − 2); a far CALL pushes CS first and then IP (SP − 4).

8.2.2.5 Far procedure and far return: code trace

A far (FAR) procedure is in a code segment other than the caller's. To reach it, both CS and IP must change. To be able to return, the old values of both must be saved. A far CALL therefore pushes two words onto the stack: first CS, then IP. RETF pops them in reverse order: first IP, then CS. The order on the stack matches RETF's popping order. The IP pushed last is on top of the stack and is the first one popped.

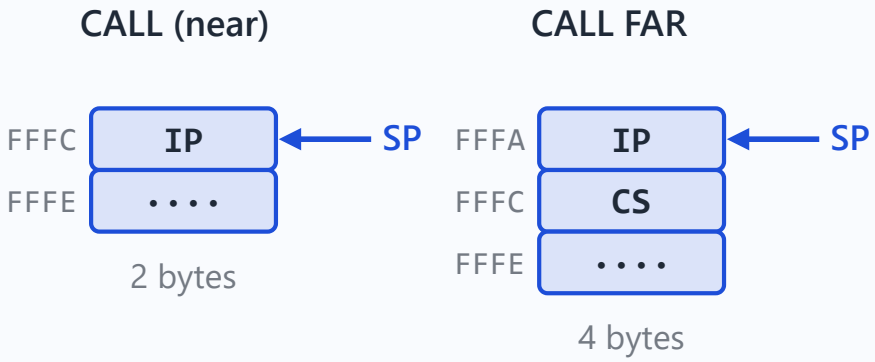


Figure 8.2.7: After a near CALL there is only IP on the stack (2 bytes). After CALL FAR, IP is on top and CS beneath it (4 bytes).

Which call is needed depends on the structure of the program. A .COM program runs in a single 64 KB segment; CS, DS, ES, and SS carry the same value. In such a program a near call is enough. In large .EXE programs with more than one code segment, a far call is needed for procedures in other segments. A far call is longer (5 bytes) and slower, because it writes two words to the stack and changes CS.

To see how a far return relates to the stack, a small trick can be used in a single-segment program. The two pushes of CALL FAR are done with two separate instructions: first `PUSH CS`, then a near CALL. The stack takes the layout that follows a far call, and the procedure can return with RETF.

```
ORG 100h
MOV AX, 5
PUSH CS         ; first half of a far call
CALL farp       ; pushes IP
ADD AX, 1
RET
farp PROC
SHL AX, 1
RETF            ; IP first, then CS
farp ENDP
```

The byte layout of the code is: `MOV AX, 5` 3 bytes (0100h–0102h), `PUSH CS` 1 byte (0103h), `CALL farp` 3 bytes (0104h–0106h), `ADD AX, 1` 3 bytes (0107h–0109h), `RET` 1 byte (010Ah). The procedure starts at 010Bh: `SHL AX, 1` 2 bytes, `RETF` 1 byte (010Dh, code CB). Since no type is written on the `farp` PROC line, it is NEAR; the programmer has deliberately written RETF inside it.

Line	Instruction	AX	CS:IP (next)	SP	Stack (from the top)
2	`MOV AX, 5`	0005h	0700:0103	FFFEh	0000h
3	`PUSH CS`	0005h	0700:0104	**FFFCh**	[FFFCh] = 0700h, 0000h
4	`CALL farp`	0005h	**0700:010B**	**FFFAh**	[FFFAh] = 0107h, 0700h, 0000h
8	`SHL AX, 1`	**000Ah**	0700:010D	FFFAh	0107h, 0700h, 0000h
9	`RETF`	000Ah	**0700:0107**	**FFFEh**	0000h
5	`ADD AX, 1`	**000Bh**	0700:010A	FFFEh	0000h

Table 8.2.5: Trace of PUSH CS + CALL with RETF (each row is the state after execution).

Two rows of the trace (Table 8.2.5) are important. **After line 4** the top of the stack holds the return IP (0107h) and beneath it CS (0700h). This is exactly the layout a real CALL FAR would leave. **In line 9** RETF pops two words: IP = 0107h, CS = 0700h. SP grows by 4 to FFFEh. Since CS is already 0700h the change is not visible, but the processor has reloaded CS from the stack. The program continues from `ADD AX, 1` and ends with AX = 000Bh. A real far call (9A, 5 bytes) does the same two pushes with a single instruction.

Figure 8.2.8: On the left, after the near CALL: IP (0107h) on top and the 0700h pushed by PUSH CS beneath it. On the right, after RETF: two words popped, SP = FFFEh.

Common mistake

Not matching the type of the procedure with the return instruction. If a procedure called with a near CALL ends with RETF, RETF pops a CS that is not on the stack. If a procedure called with CALL FAR ends with a near RET, CS is left on the stack. In both cases the stack is off by 2 bytes and the return goes to a wrong address. This error is traced step by step among the common mistakes. Writing the type correctly on the PROC line and always using `RET` as the return instruction prevents this error; the assembler encodes RET according to the type.

8.2.2.6 Indirect CALL: code trace

In the CALLs so far the target was written inside the instruction. Which procedure would be called was known when the program was written. In some cases this decision is made at run time. For example, in a menu the operation the user chose is called. For this job the target address is read from a register or from memory. Such a call is called an **indirect call**.

```
MOV AX, 6
MOV SI, 2               ; 2nd op (× 2)
CALL table[SI]          ; indirect, memory
MOV BX, OFFSET add_one
CALL BX                 ; indirect, register
RET
table DW add_one, twice
add_one: INC AX
RET
twice: SHL AX, 1
RET
```

`table` is an array of words holding the addresses of two procedures. The assembler writes the addresses of the two labels into the line `DW add_one, twice`: 0114h and 0116h. The table starts at 0110h; the first element is at 0110h and the second at 0112h. Since each element is 2 bytes, the offset

of the element with index i is 2 × i (the index starts at 0). SI = 2 selects the second element of the table, that is, the address of `twice`.

`CALL table[SI]` computes the effective address as table + SI = 0110h + 2 = 0112h and reads the word there: 0116h. It pushes the return address (010Ah) onto the stack and loads 0116h into IP. There are two memory accesses here: a read from the DS segment to fetch the target, and a write to the SS segment to store the return address. `CALL BX`, on the other hand, takes the target straight from BX and reads no target from memory.

Line	Instruction	AX	BX	IP (next)	SP	[FFFCh]
1	`MOV AX, 6`	0006h	0000h	0103h	FFFEh	—
2	`MOV SI, 2`	0006h	0000h	0106h	FFFEh	—
3	`CALL table[SI]`	0006h	0000h	**0116h**	**FFFCh**	**010Ah**
10	`twice: SHL AX, 1`	**000Ch**	0000h	0118h	FFFCh	010Ah
11	`RET`	000Ch	0000h	**010Ah**	**FFFEh**	(invalid)
4	`MOV BX, OFFSET add_one`	000Ch	**0114h**	010Dh	FFFEh	—
5	`CALL BX`	000Ch	0114h	**0114h**	**FFFCh**	**010Fh**
8	`add_one: INC AX`	**000Dh**	0114h	0115h	FFFCh	010Fh
9	`RET`	000Dh	0114h	**010Fh**	**FFFEh**	(invalid)

Table 8.2.6: Trace of the indirect CALL (each row is the state after execution).

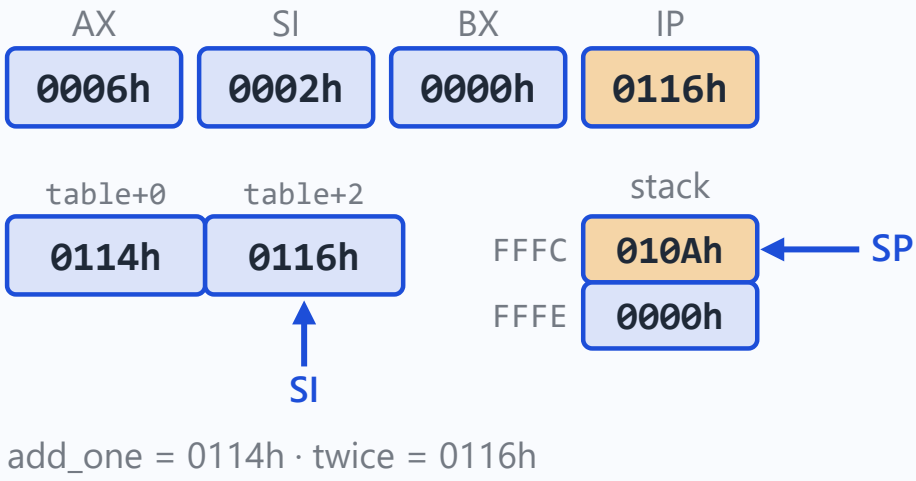


Figure 8.2.9: At the moment of `CALL table[SI]`: the target is read from [table+2] = 0116h, and the return address 010Ah is pushed onto the stack.

In the end AX = 13 (000Dh): first 6 × 2 = 12, then 12 + 1 = 13. In the trace (Table 8.2.6) the return addresses of the two calls differ: the first is 010Ah, the second 010Fh. Both are written to the same stack word, [FFFCh], because SP has returned to FFFEh between the two calls. Which procedure is called was chosen at run time through SI or BX. Had SI been 0, the first call would have gone to `add_one` and the same code would have produced a different result.

There is one more detail in the code. `add_one` and `twice` are defined not with PROC/ENDP but with ordinary labels. For the processor there is no difference: CALL jumps to an address and RET returns. PROC/ENDP gives the programmer and the assembler structure and type information.

8.2.2.7 A menu with a jump table

The most common use of the indirect call is the jump table. In a menu each option is a procedure, and their addresses sit in order in a table. The key the user presses is converted to a table index, and the matching procedure is called with a single indirect CALL.

```
MOV AH, 1
INT 21h           ; AL = '1'..'3'
SUB AL, '1'       ; 0..2
CMP AL, 2
JA invalid        ; range check
MOV BL, AL
MOV BH, 0
SHL BX, 1         ; 2-byte entries
CALL ops[BX]
invalid: RET
ops DW sum, diff, prod
```

The code runs line by line as follows. Function 01h of `INT 21h` reads a character from the keyboard, echoes it on the screen, and puts its ASCII code in AL. `SUB AL, '1'` converts the character to a number: '1' (31h) → 0, '2' → 1, '3' → 2. `CMP AL, 2` and `JA invalid` are the range check; if the value is greater than 2, no call is made. The next three instructions move the 8-bit index into the 16-bit BX and multiply it by 2. Since the addresses in the table are 2 bytes, the byte offset of index i is 2i. Shifting left by one bit is the fastest way to multiply by 2. Finally `CALL ops[BX]` calls the selected procedure.

The range check has a subtlety. JA makes an unsigned comparison. If the '0' key is pressed, the result of `SUB AL, '1'` is 30h − 31h = FFh. Viewed as signed this is −1 and is less than 2. Viewed as unsigned it is 255 and greater than 2, so JA is taken and '0' is rejected as well. A single unsigned check closes both the lower and the upper bound. The '9' key gives 8 and is rejected too.

Key	AL (after SUB)	BX	Called
'1'	00h	0000h	`sum`
'2'	01h	0002h	`diff`
'3'	02h	0004h	`prod`
'0'	FFh	—	none (RET)
'9'	08h	—	none (RET)

Table 8.2.7: BX and the procedure called, by key.

The values in Table 8.2.7 are the same as those obtained by adding the three procedures with short bodies and running with five keys in emu8086. Without the range check the '9' key would give BX = 16, and CALL would take the two bytes 16 bytes beyond the table for an address and go there. There is program code or data there; the processor jumps to a meaningless address. In every program that uses a jump table the index must be checked against the table size.

The counterpart of a jump table is a decision chain built with successive CMP and JE instructions. As the number of options grows the chain gets longer, and to reach the last option all the comparisons are executed. A table, on the other hand, reaches the target with a fixed number of instructions regardless of the number of options. Compilers often translate the `switch` construct of high-level languages this way.

8.2.2.8 The timing of CALL and RET

Calling a procedure has a time cost. The instruction timings of the 8086 are given in clock cycles in the Intel manual. Table 8.2.8 gathers the timings of instructions related to procedures.

Instruction	Clock cycles	Instruction	Clock cycles
CALL near	19	RET	8
CALL register	16	RETF	18
CALL FAR	28	RET n	12
PUSH register	11	POP register	8

Table 8.2.8: Timings of call-related instructions on the 8086 (clock cycles).

The values in the table are for the register and direct forms. For forms with a memory operand the time to compute the effective address is added. For example, for `CALL table[SI]` the manual gives 21 + EA cycles; EA is 9 cycles for displacement-plus-index addressing. CALL is more expensive than RET because it both writes to the stack and computes the target address. In both instructions the instruction queue is flushed and refilled.

Example 8.2.2

How large is the overhead, apart from its real work, of a procedure that saves two registers? If the procedure is called 1000 times on a 5 MHz 8086, how long does this overhead take?

1. **Instructions.** On every call one CALL (19), one RET (8), two PUSHes (2 × 11), and two POPs (2 × 8) are executed.
2. **Total.** 19 + 8 + 2 × (11 + 8) = 27 + 38 = 65 cycles.
3. **Time.** At 5 MHz one cycle is 1 / 5 MHz = 200 ns. 65 × 200 ns = 13,000 ns = 13 µs.
4. **1000 calls.** 1000 × 13 µs = 13 ms.

13 ms looks small, but if the real work of the procedure is only a few instructions, the overhead can exceed the work itself. At this point a macro is an option. A macro is a text pattern whose body the assembler copies into every place of use. Since there is no CALL and RET, it carries no call overhead, but the code takes space again at every use. The practical rule is: short jobs that run very often (such as printing a single character) suit a macro; long or rarely called jobs suit a procedure.

8.2.3 Passing parameters

8.2.3.1 Four ways

A procedure often works on an input and produces a result. A procedure that prints on the screen must know the text to be printed, and a procedure that adds must know the numbers to be added. The values the caller gives to the procedure are called **parameters**, and getting the values to the procedure is called **parameter passing**. In high-level languages the compiler arranges this itself. In assembly language the programmer chooses the way and follows the same rule in both the caller and the procedure.

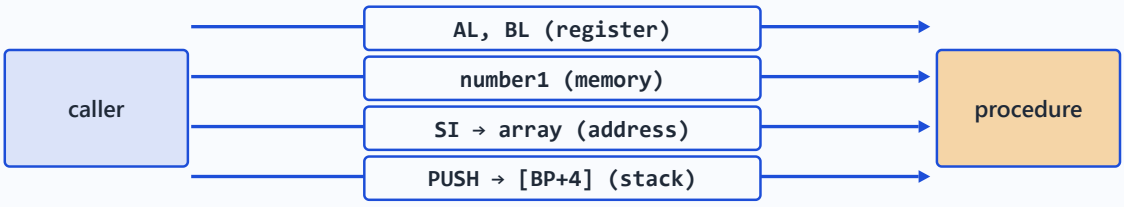


Figure 8.2.10: A parameter reaches the procedure in four ways: register, memory variable, address (pointer), and stack.

Each of the four ways in Figure 8.2.10 has its place.

1. **Through registers.** This is the easiest and fastest way; the parameter is already inside the processor and no memory access is needed. For example AL and BL carry two parameters and AX the result. Since the number of registers is limited, it does not suffice for more than a few parameters.
2. **Through memory variables.** The procedure reads and writes shared variables such as `number1` and `result`. It is easy, but the procedure stays dependent on those variables. For it to work with other variables, its code has to change.
3. **By address (pointer).** Not the data itself but its start address is sent in SI, DI, or BX. A long array or string is processed without being copied. The procedure works with whichever array it is given the address of.

4. **Through the stack.** The parameters are pushed onto the stack with PUSH before CALL. The procedure reads them through the BP register. On return they are discarded from the stack with `RET n` or, in the caller, with `ADD SP, n`. The compilers of high-level languages such as C and Pascal use this way.

Whichever way is chosen, a comment block is written at the head of the procedure. The block states the procedure's inputs, outputs, and the registers it destroys. In assembly language there is no compiler check for the name, type, or number of parameters. This comment is the only written record of the contract between the procedure and its caller.

8.2.3.2 By value or by address?

A second question, independent of how parameters are passed, is what is given to the procedure: a copy of the data, or the location of the data?

Definition 8.2.5

In **passing by value** the procedure is given a copy of the data. Even if the procedure changes the copy, the original variable in memory does not change. In **passing by address** (by reference) the procedure is given the address of the data. Through this address the procedure can read and change the original variable.

```
MOV  AX, a          ; value: AX holds a copy of a
CALL twice          ; AX doubles, a stays the same
LEA  SI, a          ; address: SI = OFFSET a
CALL twice_m        ; [SI] doubles, a changes
```

In the first two lines the contents of `a` are copied into AX. `twice` doubles AX; the result is in AX and `a` stays in memory with its old value. In the last two lines the address of `a` is loaded into SI. `twice_m` doubles the memory word `[SI]`; what changes is `a` itself.

Passing by value is fast and safe for small data such as a single number. The procedure cannot accidentally corrupt the caller's variable. Passing by address is needed in three cases: data that does not fit in a register, such as arrays and strings; variables the procedure has to change; and cases where several results have to be written to memory.

`LEA SI, a` and `MOV SI, OFFSET a` give the same result: SI carries the offset of `a`. emu8086 translates this direct-address LEA to the same 3-byte code as MOV (`BE` + a 16-bit address). In the address computations of this note LEA is counted as 3 bytes. Real 8086 assemblers may also produce this form with the 4-byte encoding `8D 36`. `MOV SI, a`, on the other hand, is a completely different instruction: it loads into SI not the address but the word inside `a`. This confusion is the most common error of passing by address.

8.2.3.3 Swapping by address: code trace

The typical example of passing by address is a procedure that exchanges (swaps) the values of two variables. The procedure takes not two values but two locations. Had it taken values, it could have changed only its own copies.

```
ORG 100h
LEA SI, a
LEA DI, b
CALL swap
RET
swap PROC          ; In: SI, DI (addrs)
PUSH AX
MOV AX, [SI]
XCHG AX, [DI]
MOV [SI], AX
POP AX
RET
swap ENDP
a DW 1234h
b DW 00FFh
```

First the layout of the program in memory is worked out. The two LEAs are 3 bytes each (0100h and 0103h). `CALL swap` is at 0106h and is 3 bytes, so the return address is 0109h. The main program's RET occupies 0109h, and the procedure starts at 010Ah. The procedure's instructions are 1, 2, 2, 2, 1, and 1 bytes, ending with the RET at 0112h. The data follows right behind: `a` = 0113h, `b` = 0115h.

The procedure uses AX as an intermediate variable, but AX is neither its input nor its output. It is therefore saved with `PUSH AX` on entry and restored with `POP AX` on exit. `XCHG AX, [DI]` is the heart of the swap: it exchanges AX with the memory word in a single instruction. Three instructions (MOV, XCHG, MOV) complete the swap.

Line	Instruction	SI	DI	AX	SP	a [0113h]	b [0115h]
2–3	`LEA SI, a` · `LEA DI, b`	**0113h**	**0115h**	0000h	FFFEh	1234h	00FFh
4	`CALL swap`	0113h	0115h	0000h	**FFFCh**	1234h	00FFh
7	`PUSH AX`	0113h	0115h	0000h	**FFFAh**	1234h	00FFh
8	`MOV AX, [SI]`	0113h	0115h	**1234h**	FFFAh	1234h	00FFh
9	`XCHG AX, [DI]`	0113h	0115h	**00FFh**	FFFAh	1234h	**1234h**
10	`MOV [SI], AX`	0113h	0115h	00FFh	FFFAh	**00FFh**	1234h
11	`POP AX`	0113h	0115h	**0000h**	**FFFCh**	00FFh	1234h
12	`RET`	0113h	0115h	0000h	**FFFEh**	00FFh	1234h

Table 8.2.9: Trace of the swap by address (each row is the state after execution).

Figure 8.2.11: On the left, after `XCHG AX, [DI]` : b = 1234h, AX = 00FFh. On the right, after POP and RET: AX is back to its old value, and a and b stay exchanged.

Four points can be seen in the trace (Table 8.2.9). SI and DI do not change from start to end; they are the addresses sent to the procedure. AX first carries the value of `a` (1234h) and, after XCHG, the old value of `b` (00FFh). In memory first `b` and then `a` change. Finally POP AX returns AX to its value before the call (0000h). The only visible effect for the caller is that `a` and `b` have exchanged places.

The memory-operand form of XCHG performs a read and a write bus cycle on the 8086. Doing the same job with two MOVs would need one more register. In cases where no other bus master must slip in between the read and the write, the LOCK prefix is put in front of the instruction. The prefix keeps the LOCK pin of the 8086 active for the duration of the instruction. In multiprocessor systems, shared variables such as semaphores are therefore changed with `LOCK XCHG`.

8.2.3.4 Parameters in registers: code trace

As an example of passing parameters in registers, the procedure `m2` that multiplies two numbers is used. AL and BL stand for the two parameters; m2 multiplies them and puts the result in AX.

Figure 8.2.12: AL and BL are the input; the procedure m2 does `MUL BL`, and the result is in AX.

The procedure's only instruction is MUL. MUL does unsigned multiplication and works in two forms depending on the operand size (Table 8.2.10). The source can be a register or a memory operand but not a constant: `MUL 2` is invalid. If the upper half of the product (AH or DX) is nonzero, CF = OF = 1; these flags report whether the result fits in the lower half. SF, ZF, AF, and PF are undefined after MUL.

Source	Operation	Result
8 bits	AL × source	AX

Source	Operation	Result
16 bits	AX × source	DX:AX

Table 8.2.10: The two forms of the MUL instruction.

```
ORG 100h
MOV AL, 1       ; 1st parameter
MOV BL, 2       ; 2nd parameter
CALL m2         ; 0104h: E8 04 00
CALL m2         ; 0107h: E8 01 00
RET             ; 010Ah: to DOS
m2 PROC         ; 010Bh
MUL BL          ; AX = AL * BL
RET             ; to caller
m2 ENDP
END
```

`MOV AL, 1` and `MOV BL, 2` are 8-bit constant loads of 2 bytes each (B0 01, B3 02). The two CALLs are at 0104h and 0107h. The return address of the first is 0107h and that of the second 010Ah. m2 starts at 010Bh. The machine codes of the two CALLs differ (E8 04 00 and E8 01 00), because the displacement is computed relative to each CALL's own position. The target is 010Bh in both.

Line	Instruction	IP (next)	SP	AX	BL	CF	[FFFCh]
2	`MOV AL, 1`	0102h	FFFEh	0001h	00h	0	—
3	`MOV BL, 2`	0104h	FFFEh	0001h	02h	0	—
4	`CALL m2` (1.)	**010Bh**	**FFFCh**	0001h	02h	0	**0107h**
8	`MUL BL`	010Dh	FFFCh	**0002h**	02h	0	0107h
9	`RET`	**0107h**	**FFFEh**	0002h	02h	0	(invalid)
5	`CALL m2` (2.)	**010Bh**	**FFFCh**	0002h	02h	0	**010Ah**
8	`MUL BL`	010Dh	FFFCh	**0004h**	02h	0	010Ah
9	`RET`	**010Ah**	**FFFEh**	0004h	02h	0	(invalid)
6	`RET` (DOS'a)	**0000h**	**0000h**	0004h	02h	0	—

Table 8.2.11: Trace of the m2 program (each row is the state after execution).

The most instructive aspect of the trace (Table 8.2.11) is that the same procedure returns to two different return points. The procedure's RET is the same instruction (010Dh) in both calls. In the first call it finds 0107h on the stack, in the second 010Ah, and returns there.

The second important point is that MUL changes AL. In the first call AX = 1 × 2 = 0002h. AL is now 02h. Since the second call is made without reloading AL, the result of the first call becomes the first parameter of the second: AX = 2 × 2 = 0004h. The program uses this behavior deliberately, but the

same situation can also cause an error unnoticed. The “Out: AX” in a procedure's description shows that AL changes as well. In both multiplications AH = 0, so CF = OF = 0.

8.2.3.5 Returning the result in a flag: code trace

The result of some procedures is not a number but a yes/no answer: could the file be opened, is the character a digit, was the value found? For such an answer a flag is used instead of spending a register. The carry flag (CF) is the most suitable flag for this. It can be set directly with STC, CLC, and CMC and tested with JC and JNC. DOS services also report an error with CF = 1.

```
MOV AL, '7'
CALL digit       ; CF = 0 → AL = value
MOV BL, AL
MOV AL, 'x'
CALL digit       ; CF = 1 → not a digit
RET
digit PROC       ; In: AL, Out: CF
SUB AL, '0'
CMP AL, 10       ; 0..9 → borrow, CF = 1
CMC              ; digit: CF = 0
RET
digit ENDP
```

The `digit` procedure converts the ASCII character in AL to a number and reports through CF whether the character is a digit. `SUB AL, '0'` subtracts 30h from the character. If the character is in the range '0'–'9', the result is 0–9. `CMP AL, 10` makes an unsigned comparison. If AL < 10, the subtraction needs a borrow and CF = 1. If AL ≥ 10, CF = 0. This is the opposite of the meaning wanted. `CMC` (complement carry) inverts CF: CF = 0 for a digit and CF = 1 for a non-digit. For a character smaller than '0', the SUB result wraps into the range D0h–FFh and counts as greater than 10; that too is rejected.

Line	Instruction	AL	BL	CF	SP	Note
1	`MOV AL, '7'`	37h	00h	0	FFFEh	'7' = 37h
2, 8	`CALL digit` · `SUB AL, '0'`	**07h**	00h	0	FFFCh	return 0105h; 37h − 30h = 7
9	`CMP AL, 10`	07h	00h	**1**	FFFCh	7 − 10 needs a borrow
10	`CMC`	07h	00h	**0**	FFFCh	“is a digit”
11, 3	`RET` · `MOV BL, AL`	07h	**07h**	0	FFFEh	value stored
4, 5, 8	`MOV AL, 'x'` · `CALL` · `SUB`	**48h**	07h	0	FFFCh	78h − 30h = 48h (72); return 010Ch
9	`CMP AL, 10`	48h	07h	0	FFFCh	72 ≥ 10, no borrow
10	`CMC`	48h	07h	**1**	FFFCh	“not a digit”
11	`RET`	48h	07h	1	**FFFEh**	result in CF

Table 8.2.12: Trace of the return in a flag (each row is the state after execution).

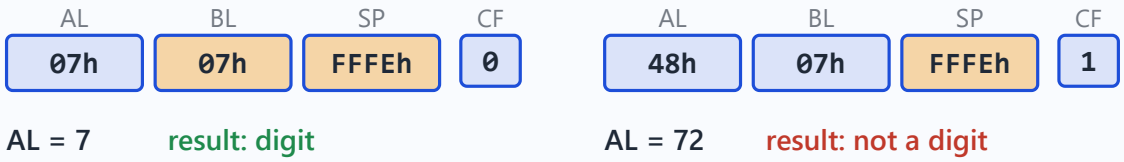


Figure 8.2.13: On the left, after the first return: CF = 0, BL = 7. On the right, after the second return: CF = 1, AL = 48h must not be used.

Two conclusions follow from the trace (Table 8.2.12). First, RET does not touch the flags. The CF set inside the procedure is still valid in the caller after the return. The caller can test the result with `JC error` or `JNC ok`. Second, when CF = 1 AL has been corrupted (48h). The contract of the procedure is: the value in AL is meaningful only if CF = 0. The caller must look at the flag first and use the register afterward.

Common mistake

Executing an instruction that changes the flags after STC, CLC, or CMC in a procedure that returns its result in CF. For example, if `ADD SP, 2` or `CMP` is done at the exit after CMC, CF is recomputed and the result is lost. POP and RET do not touch the flags; the instruction that sets the flag must be the last flag-changing instruction before RET.

8.2.3.6 Returning several results

A procedure can produce more than one result. Division gives both a quotient and a remainder; a search gives both the "found?" answer and the position. There are three ways for this.

- **A register pair.** A 32-bit result returns in DX:AX (like the 16-bit MUL). The quotient and remainder return in AL and AH (like the 8-bit DIV).
- **A flag and a register.** CF reports success and the register holds the value. The caller first looks at the flag with JC.
- **Memory.** The results are written to an area whose address is given. The number of results is not limited by the number of registers.

The `divmod` procedure below uses the first two ways together. It divides AX by BL and returns the quotient in AL and the remainder in AH. If the divisor is zero it does not divide and reports an error with CF = 1.

```
divmod PROC     ; In: AX, BL (divisor)
CMP BL, 0          ; Out: AL = quotient
JE error           ;      AH = remainder
DIV BL             ;      CF = 1: BL is 0
CLC
RET
error: STC
RET
divmod ENDP
```

The 8-bit `DIV BL` divides the 16-bit number in AX by BL. It writes the quotient to AL and the remainder to AH. After DIV all the flags are undefined. That is why, in the success case, CF is cleared explicitly with `CLC`. There are two RETs in the procedure. This is safe because nothing was pushed onto the stack: both RETs find the return address on top.

Input	Result	Note
AX = 100, BL = 7	AL = 0Eh (14), AH = 02h, CF = 0	100 = 7 × 14 + 2; AX = 020Eh
AX = 100, BL = 0	CF = 1, AX = 0064h (unchanged)	DIV is never executed
AX = 1000, BL = 2	divide error (INT 0)	quotient 500 does not fit in AL

Table 8.2.13: Results of the divmod procedure on three inputs.

The last row of Table 8.2.13 shows a gap in the procedure. Although the divisor is not zero, the result of 1000 / 2 = 500 does not fit in the 8-bit AL (the largest value is 255). In this case too the 8086 generates a type 0 interrupt (divide error). Since the procedure checks only that the divisor is zero, it cannot catch this case. For a complete check the condition AH < BL must also be tested; the quotient fits in AL only under this condition.

A structure with two RETs is safe only if nothing has been pushed onto the stack. In a procedure that does PUSH, all paths must be merged at a single exit label where the POPs are. This rule is shown with the trace of a search procedure with a single exit point.

8.2.4 Preserving registers

8.2.4.1 The problem and documentation

The 8086 has eight general-purpose registers, and a procedure and its caller share them. If a procedure changes a register the caller is using without telling it, the caller's code breaks. The most typical example is a loop counter. The caller sets up a loop with CX and calls a procedure from inside the loop. If the procedure also uses CX for its own loop, on return CX holds not the caller's counter but whatever the procedure left.

The first remedy for the problem is documentation. At the head of every procedure its input, output, and the registers it destroys are written:

```
; m2: computes AL * BL
; In     : AL, BL (unsigned)
; Out    : AX = AL * BL
; Clobber: CF, OF
```

Someone who reads this comment knows that after calling m2 the old value in AX will be lost and CF and OF will change. That no other register changes is also a promise of this comment. If the promise is not kept, that is, if the procedure changes a register not named in the comment, the error most often shows up not in the procedure but in the code that calls it. The location and the cause of the error being different makes it hard to find.

Definition 8.2.6

Register preservation is a procedure's pushing the registers it uses onto the stack on entry and popping them back in reverse order on exit. The registers then appear to the caller as if they had never changed. Registers that return results are not preserved.

```
show PROC
    PUSH AX     ; save what
    PUSH SI     ; you use
    ; ... AX and SI are used
    POP SI      ; in REVERSE
    POP AX      ; order
    RET
show ENDP
```

The reverse order comes from the last-in, first-out (LIFO) structure of the stack. Since AX was pushed first and SI after it, the value of SI is on top of the stack. The first POP takes that value and must write it to SI. If the order is mixed up, that is, if `POP AX` is written first, AX is loaded with SI's old value and SI with AX's. The stack is still balanced and RET returns correctly, but the two registers have taken each other's values.

The last sentence of the definition matters too. If a procedure that returns its result in AX pushes AX on entry and pops it back on exit, it erases with its own hand the result it computed. Since the stack is balanced the program does not crash; only the result comes out wrong. The rule is: a procedure preserves the registers it uses and does not preserve the result registers.

8.2.4.2 Saving the counter with PUSH CX: code trace

The program below wants to print two lines on the screen. The main loop runs with CX = 2 and calls the procedure `stars` on every iteration. The procedure uses CX as the inner loop counter to print three `#` characters. The first and last instructions of the procedure preserve the caller's counter.

```
MOV CX, 2          ; main loop: 2 rows
row: CALL stars
LOOP row
RET
stars PROC
PUSH CX            ; caller's counter
MOV CX, 3          ; inner counter
MOV AX, 0E23h      ; AH = 0Eh, AL = '#'
inner: INT 10h
LOOP inner
POP CX             ; counter back
RET
stars ENDP
```

`MOV AX, 0E23h` fills two register halves with a single instruction. AH = 0Eh is the “teletype” function of the BIOS screen service (INT 10h): it writes the character in AL at the cursor position and advances the cursor. AL = 23h is the ASCII code of the character '#'. The LOOP instruction first decrements CX by 1 and then jumps to the label if CX ≠ 0. The inner loop therefore runs exactly three iterations.

Line	Instruction	CX	SP	Stack (from the top)	Screen
1	`MOV CX, 2`	0002h	FFFEh	0000h	
2	`CALL stars`	0002h	**FFFCh**	0106h, 0000h	
6	`PUSH CX`	0002h	**FFFAh**	0002h, 0106h, 0000h	
7–8	`MOV CX, 3` · `MOV AX, 0E23h`	**0003h**	FFFAh	0002h, 0106h, 0000h	
9–10	inner loop, 3 iterations	**0000h**	FFFAh	0002h, 0106h, 0000h	###
11	`POP CX`	**0002h**	**FFFCh**	0106h, 0000h	###
12	`RET`	0002h	**FFFEh**	0000h	###
3	`LOOP row`	**0001h**	FFFEh	0000h	###
2–12	second call	0001h	FFFEh	0000h	######
3	`LOOP row`	**0000h**	FFFEh	0000h	######

Table 8.2.14: Trace of saving the counter with PUSH CX (each row is the state after execution).

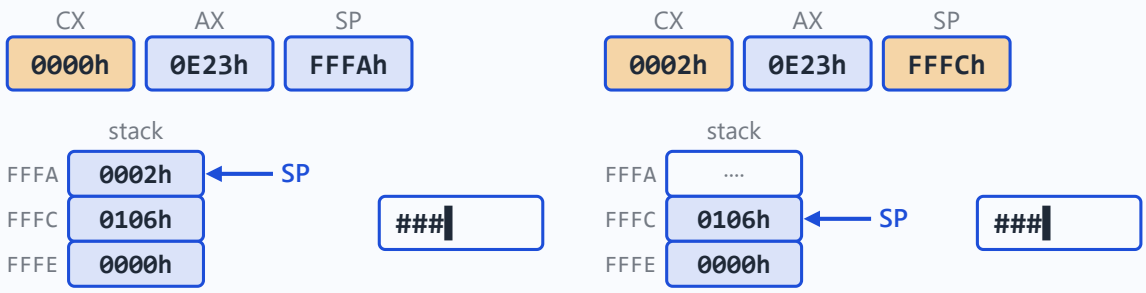


Figure 8.2.14: On the left, when the inner loop ends: CX = 0, and the stack holds the caller's counter (2) and the return address. On the right, after POP CX: CX = 2 and the return address is on top.

In the trace (Table 8.2.14) the journey of CX is clearly visible. The caller's counter (2) goes onto the stack with PUSH. The inner loop takes CX from 3 down to 0. POP CX brings back the caller's value. The main LOOP takes this value down to 1 and calls the procedure again for the second row. In the second call a 1 is pushed onto the stack and comes back. The last LOOP makes CX 0 and the loop ends. There are 6 characters on the screen.

At the deepest moment of the stack inside the procedure (during the inner loop), there are two words on the stack: the return address and the saved CX. SP = FFFAh. This layout also explains why the RET at the exit works correctly: after POP CX has taken the saved value, the return address is again on top.

Common mistake

Calling a procedure that uses CX from inside a LOOP-based loop without preserving CX. Without PUSH CX / POP CX the procedure would return with CX = 0. The LOOP in the main loop first decrements CX by 1: 0 − 1 = FFFFh. Since FFFFh is not zero, it jumps. Because every call makes CX 0 again, the loop never ends and characters are printed on the screen without stopping. The fix goes in one of two places: either PUSH CX / POP CX in the procedure, or `PUSH CX / CALL stars / POP CX` in the caller.

8.2.4.3 Does the caller or the callee preserve?

The job of saving registers can be given to one of two places: the procedure itself (the callee preserves) or the code that calls it (the caller preserves). The strengths and weaknesses of the two approaches are in Table 8.2.15.

Criterion	Callee preserves (PUSH/POP in the procedure)	Caller preserves (around the CALL)
Where is the code?	once in the procedure	at every call site
Unnecessary saving	saved even if the caller does not use that register	only the registers really needed
Risk of forgetting	low: written in one place	high: needed again at every call

Criterion	Callee preserves (PUSH/POP in the procedure)	Caller preserves (around the CALL)
Suited when	library procedures	the old value in the output register is needed

Table 8.2.15: Comparison of the callee-preserves and caller-preserves approaches.

An example of a case where the caller has to preserve is this. The procedure `read_num` returns the number it reads from the keyboard in CX. CX is its output, and the procedure cannot preserve it. The caller, however, uses CX as a loop counter. In this case saving the counter is the caller's job:

```
again: PUSH CX        ; caller: counter
       CALL read_num  ; result in CX
       ADD total, CX
       POP CX         ; counter back
       LOOP again
```

Between PUSH CX and POP CX, CX carries two different meanings. First it is the number read, which is added to `total`; then it is the loop counter that comes back from the stack. The rule in practice combines the two approaches: a procedure preserves the registers it uses and does not preserve its output registers. If the old value in an output register is needed, the caller saves it.

8.2.4.4 A single exit point: code trace

If a procedure that saves registers has more than one return path, a danger arises. If one of the paths skips the POPs and goes straight to RET, RET finds on top of the stack not the return address but a saved register value. That is why in procedures that PUSH, all paths merge at a single exit label where the POPs are.

```
ORG 100h
LEA SI, arr
MOV CX, 5
MOV AL, 7
CALL find          ; CF = 1: found
RET
find PROC          ; In: SI, CX, AL
PUSH CX
PUSH SI
XOR BX, BX         ; Out: BX = index
scan: CMP [SI], AL
JE found
INC SI
INC BX
LOOP scan
CLC                ; not found
JMP done
found: STC
done: POP SI       ; single exit point
POP CX
RET
find ENDP
arr DB 3, 9, 7, 1, 4
```

The `find` procedure searches for the value in AL among CX elements of the byte array SI points at. It returns the result in two ways: if CF = 1 the value was found and BX is its index. If CF = 0 the value is not in the array. During the search the procedure changes SI and CX. Both are inputs and the caller may want to use them on the next call, so they are preserved. BX, on the other hand, is an output and is not preserved. `XOR BX, BX` is the short way to clear a register (2 bytes).

The layout of the program is: LEA 3 bytes (0100h), `MOV CX, 5` 3 bytes (0103h), `MOV AL, 7` 2 bytes (0106h), CALL 3 bytes (0108h). The return address is 010Bh. The procedure starts at 010Ch and ends with the RET at 011Eh. The array is at 011Fh; this is the initial value of SI.

Line	Instruction	SI	CX	BX	CF	SP	Stack (from the top)
2–4	LEA, MOV, MOV	011Fh	0005h	0000h	0	FFFEh	0000h
5	`CALL find`	011Fh	0005h	0000h	0	**FFFCh**	010Bh
8–10	PUSH CX, PUSH SI, XOR	011Fh	0005h	0000h	0	**FFF8h**	011Fh, 0005h, 010Bh
11–15	arr[0] = 3 ≠ 7	**0120h**	**0004h**	**0001h**	1	FFF8h	011Fh, 0005h, 010Bh
11–15	arr[1] = 9 ≠ 7	**0121h**	**0003h**	**0002h**	0	FFF8h	011Fh, 0005h, 010Bh
11–12	arr[2] = 7: ZF = 1, JE taken	0121h	0003h	0002h	0	FFF8h	011Fh, 0005h, 010Bh
18	`found: STC`	0121h	0003h	0002h	**1**	FFF8h	011Fh, 0005h, 010Bh
19–20	`POP SI` · `POP CX`	**011Fh**	**0005h**	0002h	1	**FFFCh**	010Bh
21	`RET`	011Fh	0005h	0002h	1	**FFFEh**	0000h

Table 8.2.16: Trace of the single exit point (each group of rows is the state after execution).

It may be striking that the CF column of the table is 1 in the first iteration and then 0 (Table 8.2.16). The reason is CMP. `CMP [SI], AL` computes 3 − 7 and needs a borrow, so CF = 1. In the second iteration 9 − 7 needs no borrow, and CF = 0. LOOP and INC do not touch CF. The CF values during the search carry no meaning. What is meaningful is the value the procedure sets at the exit with STC or CLC.

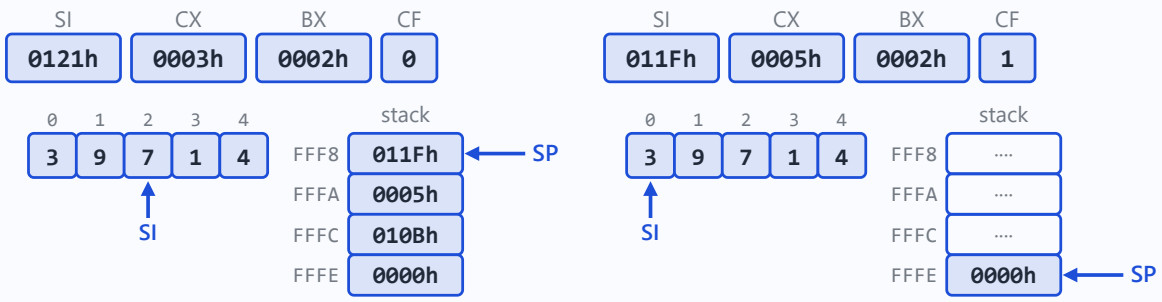


Figure 8.2.15: On the left, at the moment the value is found: the loop ends early, but there are still two saved values on the stack. On the right, after RET: SI and CX are at their old values, CF = 1, BX = 2.

The critical moment is the third iteration, where the value is found. JE jumps to `found` and the loop ends early. At that moment the top of the stack holds not the return address but the saved SI (011Fh) (Figure 8.2.15). `found` sets CF with STC and the flow moves on to the `done` label. The not-found path (CLC and JMP done) also arrives at the same label. Since both paths pass through POP SI and POP CX, RET finds 010Bh on top in every case.

Had the programmer used a direct RET at the target of JE, that is, written `found: STC` / `RET`, RET would have taken the 011Fh on top for the return address. The processor would have tried to execute the bytes 03h, 09h, 07h at address 011Fh, where the array begins, as instructions. The result would be unpredictable. The program would work correctly only when the search failed, and the error would show up only for some inputs.

8.2.5 Parameters on the stack and the stack frame

8.2.5.1 Passing parameters on the stack: code trace

Passing parameters in registers has two limits: there are few registers, and every procedure has to document separately which register carries what. Passing parameters on the stack overcomes both. The caller pushes the parameters onto the stack in order before CALL. The procedure reads them from their fixed places on the stack. The number of parameters is limited by the size of the stack, and the rule is the same for every procedure.

How does the procedure find the parameters? It cannot use SP: on the 8086 SP is not an addressing register, and `MOV AX, [SP+4]` is invalid. Besides, SP changes every time the procedure does a PUSH or POP; the offset of the same parameter would differ at different places in the procedure. The solution is the BP register. Addresses that include BP use the SS segment by default. On entry the procedure copies the value of SP into BP and thus sets up a fixed reference point on the stack.

```
ORG 100h
MOV AX, 7
PUSH AX          ; 1st parameter
MOV AX, 3
PUSH AX          ; 2nd parameter
CALL SUM2
ADD SP, 4        ; drop parameters
RET
SUM2 PROC
PUSH BP
MOV BP, SP
MOV AX, [BP+6]   ; 1st parameter
ADD AX, [BP+4]   ; 2nd parameter
POP BP
RET
SUM2 ENDP
```

The layout of the program explains the IP values in the trace table. The main program occupies 0100h–010Eh: `MOV AX, 7` (3 bytes), `PUSH AX` (1), `MOV AX, 3` (3), `PUSH AX` (1), `CALL SUM2` (3, at 0108h), `ADD SP, 4` (3, at 010Bh), and `RET` (1). SUM2 starts at 010Fh. `PUSH BP` is 1 byte, `MOV BP, SP` 2, the two instructions with memory operands 3 each, and `POP BP` and `RET` 1 each.

Line	Instruction	AX	BP	SP	IP (next)	Written to the stack
2	`MOV AX, 7`	0007h	0000h	FFFEh	0103h	
3	`PUSH AX`	0007h	0000h	**FFFCh**	0104h	[FFFCh] = 0007h
4	`MOV AX, 3`	0003h	0000h	FFFCh	0107h	
5	`PUSH AX`	0003h	0000h	**FFFAh**	0108h	[FFFAh] = 0003h
6	`CALL TOPLA`	0003h	0000h	**FFF8h**	**010Fh**	[FFF8h] = 010Bh
10	`PUSH BP`	0003h	0000h	**FFF6h**	0110h	[FFF6h] = 0000h
11	`MOV BP, SP`	0003h	**FFF6h**	FFF6h	0112h	
12	`MOV AX, [BP+6]`	**0007h**	FFF6h	FFF6h	0115h	
13	`ADD AX, [BP+4]`	**000Ah**	FFF6h	FFF6h	0118h	
14	`POP BP`	000Ah	**0000h**	**FFF8h**	0119h	
15	`RET`	000Ah	0000h	**FFFAh**	**010Bh**	
7	`ADD SP, 4`	000Ah	0000h	**FFFEh**	010Eh	

Line	Instruction	AX	BP	SP	IP (next)	Written to the stack
8	`RET` (DOS'a)	000Ah	0000h	**0000h**	**0000h**	

Table 8.2.17: Trace of passing parameters on the stack (each row is the state after execution).

The first five rows of Table 8.2.17 are the caller's preparation. The two PUSHes write 7 and 3 to the stack, and CALL puts the return address (010Bh) on top of them. When the procedure is entered, the stack, going up from SP, is: [FFF8h] the return address, [FFFAh] the second parameter (3), [FFFCh] the first parameter (7).

The first two instructions of the procedure, `PUSH BP` and `MOV BP, SP`, set up the frame. The caller's BP is saved on the stack, because the caller may also be using BP for its own frame. Then BP = SP = FFF6h. From this point on the offset of every parameter relative to BP is fixed:

- `[BP+0]` = [FFF6h]: the saved old BP (0000h),
- `[BP+2]` = [FFF8h]: the return address (010Bh),
- `[BP+4]` = [FFFAh]: the parameter pushed last (3),
- `[BP+6]` = [FFFCh]: the parameter pushed first (7).

The offset of the parameters relative to BP stays the same however much SP changes inside the procedure. Even if the procedure pushes other registers in the meantime, `[BP+6]` is always the first parameter. `MOV AX, [BP+6]` computes the physical address as SS × 16 + BP + 6 = 07000h + FFFCh = 16FFCh. No segment prefix is needed, because addresses with BP use SS by default. Had the same address been computed with DS, it would still have come out right in a single-segment .COM program, but would be wrong in .EXE programs where DS ≠ SS.

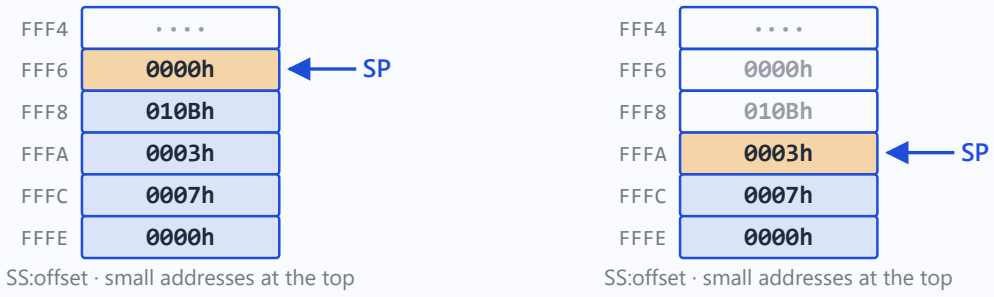


Figure 8.2.16: On the left, at the moment of `MOV AX, [BP+6]`: BP = FFF6h, the parameters at [BP+4] and [BP+6]. On the right, after the procedure's RET: SP = FFFAh, the parameters still on the stack.

The exit is the reverse of the entry. `POP BP` brings back the caller's BP and SP = FFF8h; the return address is on top. RET returns to 010Bh and SP = FFFAh. At this point the two parameters are still on the stack (Figure 8.2.16). The caller's `ADD SP, 4` instruction discards them: SP returns to its value before the PUSHes, FFFEh. The stack is balanced, and the main program's RET correctly finds 0000h. AX = 000Ah = 10 carries the result.

8.2.5.2 Who discards the parameters? RET n

The parameters pushed onto the stack have to be discarded somewhere; otherwise every call leaves a few words on the stack. Either the caller or the procedure does this.

Definition 8.2.7

RET n is the return instruction that, after taking the return address from the stack and loading it into IP, additionally adds n to SP: IP ← [SP], SP ← SP + 2 + n. n is stored in the instruction as a 16-bit constant (C2 n16, 3 bytes) and is the number of parameter bytes.

Had the last instruction of the procedure SUM2 been `RET 4`, the caller would not have needed to write `ADD SP, 4`. While RET 4 runs, SP grows by 6 in all: 2 for the return address and 4 for the two parameters. SP returns to its value before the PUSHes (FFFEh) in a single instruction.

The difference between the two methods is who knows the number of parameters. In RET n the number is fixed inside the procedure. The procedure must always take the same number of parameters. Pascal compilers use this method. With ADD SP, n, on the other hand, the caller knows the number. The same procedure can take a different number of parameters in different calls. Functions of the C language that take a variable number of parameters, such as `printf`, therefore work with the method where the caller cleans up. Whichever method is chosen, the caller and the procedure must follow the same rule. If both clean up, 4 extra bytes are given back from the stack; if neither does, 4 bytes remain on the stack.

Example 8.2.3

With SP = FFFEh the caller pushes three parameters and makes a near CALL. The procedure starts with `PUSH BP` and `MOV BP, SP`. What is BP, where is the first parameter pushed, and how should the procedure return?

1. **Pushes.** Three parameters 6 bytes, CALL 2 bytes, PUSH BP 2 bytes: FFFEh − 10 = FFF4h. BP = FFF4h.
2. **Offsets.** [BP] is the old BP, [BP+2] the return address, [BP+4] the parameter pushed last (the third), [BP+6] the second, and [BP+8] the parameter pushed first.
3. **Return.** After POP BP, `RET 6`: SP = FFF6h + 2 + 6 = FFFEh. The stack returns to its state before the call.

8.2.5.3 The stack frame and local variables

While running, a procedure may need a place to keep its intermediate results. Registers may not suffice. Keeping intermediate results in global memory variables binds the procedure to those variables and, in a recursive call, makes every level corrupt the same variable. The right place is again

the stack.

Definition 8.2.8

A **stack frame** is the region of the stack belonging to one procedure call: the parameters, the return address, the saved old BP, and the **local variables**. A local variable is a variable that occupies space on the stack only while the procedure runs. BP is the fixed reference point of the frame: parameters are at positive offsets above BP and local variables at negative offsets below it.

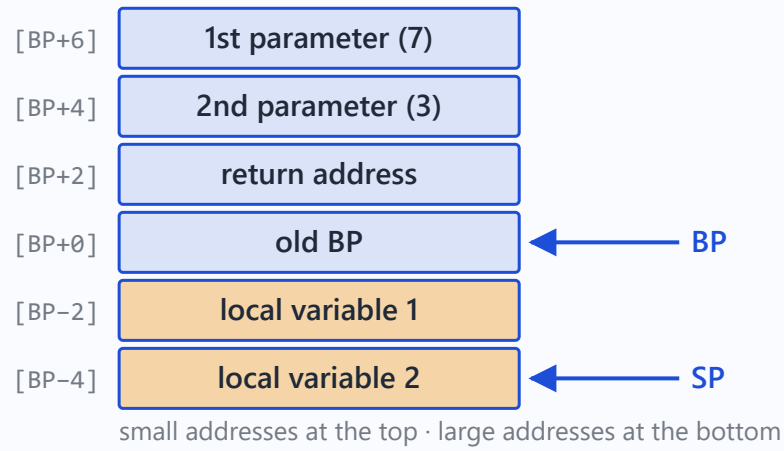


Figure 8.2.17: The stack frame of a procedure with two parameters and two local variables. Small addresses are at the top and large addresses at the bottom.

The instructions that build and tear down the frame follow the same pattern in every procedure:

```
PUSH BP
MOV  BP, SP
SUB  SP, 4        ; local space
MOV  [BP-2], AX   ; local 1
...
MOV  SP, BP       ; drop locals
POP  BP
RET  4            ; drop 2 params
```

`SUB SP, 4` opens room for two words on the stack. Nothing is written to this area; SP is just pulled down. The procedure's later PUSHes and calls therefore do not overwrite the local area. The two local variables are at addresses `[BP−2]` and `[BP−4]` (Figure 8.2.17). The initial value of the opened area is whatever was left at that address before. Local variables must be given a value before they are used.

On exit `MOV SP, BP` gives the local area back in a single instruction. It does not matter how many local variables were opened; SP returns to its value at the moment BP was set up. POP BP and RET n follow. This order cannot be broken: a POP BP done without closing the local area loads a local variable into BP instead of the old BP.

The lifetime of a local variable is the lifetime of the call. After the procedure returns, those stack words are overwritten by the next PUSH or CALL. A procedure therefore must not return the address of its local variable as a result. If the same procedure calls itself, every call builds its own frame, and so its own local variables. The levels do not touch each other's variables.

On the 80186 and later processors there is a two-instruction shorthand for this pattern: `ENTER` builds the frame, and `LEAVE` does `MOV SP, BP` and `POP BP` together. The 8086 does not have these instructions.

8.2.5.4 A frame with a local variable: code trace

The procedure below computes x·x + x from the value x in AX. Since MUL overwrites AX, a copy of x is kept in a local variable.

```
MOV AX, 9
CALL sq_plus       ; AX = x·x + x
RET
sq_plus PROC
PUSH BP
MOV BP, SP
SUB SP, 2          ; local [BP-2]
MOV [BP-2], AX     ; local = x
MUL AX             ; AX = x·x
ADD AX, [BP-2]     ; + x
MOV SP, BP         ; drop local
POP BP
RET
sq_plus ENDP
```

Line	Instruction	AX	DX	BP	SP	Stack
1	`MOV AX, 9`	0009h	0000h	0000h	FFFEh	
2	`CALL sq_plus`	0009h	0000h	0000h	**FFFCh**	[FFFCh] = 0106h
5	`PUSH BP`	0009h	0000h	0000h	**FFFAh**	[FFFAh] = 0000h
6	`MOV BP, SP`	0009h	0000h	**FFFAh**	FFFAh	
7	`SUB SP, 2`	0009h	0000h	FFFAh	**FFF8h**	[FFF8h] = local space
8	`MOV [BP-2], AX`	0009h	0000h	FFFAh	FFF8h	**[FFF8h] = 0009h**
9	`MUL AX`	**0051h**	0000h	FFFAh	FFF8h	
10	`ADD AX, [BP-2]`	**005Ah**	0000h	FFFAh	FFF8h	
11	`MOV SP, BP`	005Ah	0000h	FFFAh	**FFFAh**	local space released

Line	Instruction	AX	DX	BP	SP	Stack
12	`POP BP`	005Ah	0000h	**0000h**	**FFFCh**	
13	`RET`	005Ah	0000h	0000h	**FFFEh**	IP = 0106h

Table 8.2.18: Trace of the frame with a local variable (each row is the state after execution).

In the trace (Table 8.2.18), after `SUB SP, 2`, BP = FFFAh and SP = FFF8h. The local variable is at [BP−2] = [FFF8h]; SP points at it and a later PUSH writes below it. `MOV [BP-2], AX` copies x = 9 into the local. The 16-bit `MUL AX` computes DX:AX ← AX × AX: 9 × 9 = 81 = 0051h, DX = 0000h. The x in AX is gone, but it sits in the local. `ADD AX, [BP-2]` gives 81 + 9 = 90 = 005Ah.

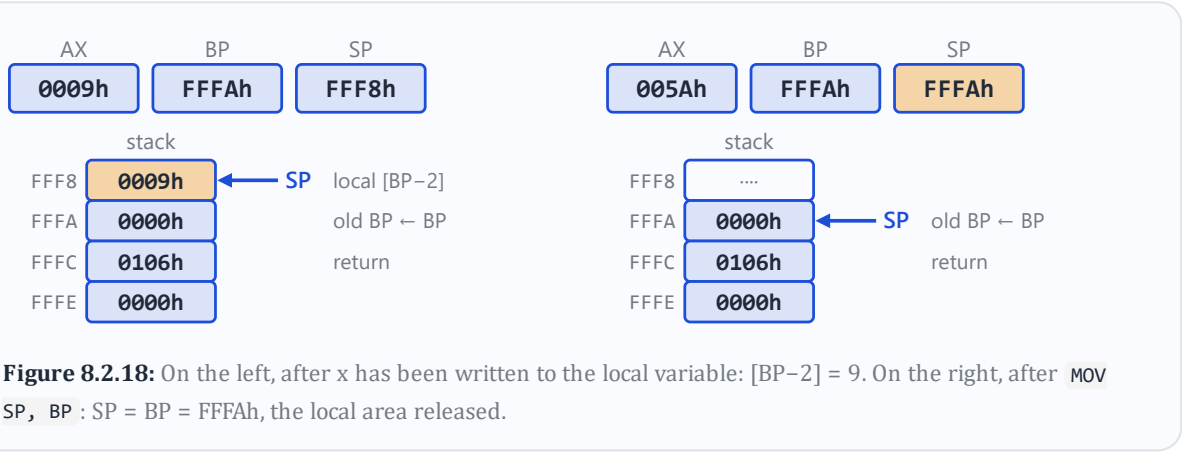


Figure 8.2.18: On the left, after x has been written to the local variable: [BP−2] = 9. On the right, after `MOV SP, BP`: SP = BP = FFFAh, the local area released.

On exit three instructions tear down the frame in reverse order. `MOV SP, BP` pulls SP to FFFAh and the local area is released. 0009h is still written at [FFF8h], but it is now outside the stack. `POP BP` brings back the caller's BP (0000h). RET returns to 0106h. The result is AX = 90.

There is a gap in the documentation of this procedure. The 16-bit MUL always writes to DX too: for x = 9, DX = 0000h. The value the caller keeps in DX is therefore lost. In this trace the difference is not visible because DX is 0 anyway. When x ≥ 256, the product does not fit in 16 bits and DX takes a nonzero value. “Clobber: DX” should be written in the comment, or DX should be preserved in the procedure with PUSH/POP.

8.2.6 Example programs

8.2.6.1 Hello World: print_me

The first example is the procedure `print_me`, which prints a string terminated by zero on the screen character by character. The parameter is the address of the string and is passed in SI. The length of the string is not given separately; the procedure stops when it sees the zero byte. Strings in the C language are stored in this form too.

```
ORG 100h
LEA SI, msg       ; SI = address of msg
CALL print_me
RET               ; to DOS
print_me PROC
next_char:
CMP b.[SI], 0     ; zero?
JE stop           ; yes: done
MOV AL, [SI]      ; next ASCII char
MOV AH, 0Eh       ; teletype function
INT 10h           ; print AL
ADD SI, 1         ; next character
JMP next_char     ; loop back
stop:
RET               ; to caller
print_me ENDP
msg DB 'Hello World!', 0
END
```

The main program is three instructions. `LEA SI, msg` loads the address of the string into SI; this is the parameter of the procedure. `CALL print_me` calls the procedure, and `RET` ends the program. In the program layout, LEA is 3 bytes (BE 18 01), CALL is at 0103h, and the return address is 0106h. The procedure starts at 0107h and ends with the RET at 0117h. The string is at 0118h: 12 characters and a zero byte.

The first instruction of the procedure is `CMP b.[SI], 0`. `b.` is an emu8086 abbreviation of `BYTE PTR`. Had `CMP [SI], 0` been written, the assembler could not know the size of the operation: [SI] can be a byte or a word, and the constant 0 does not indicate a size either. The size is therefore written explicitly. In MASM the same line is written as `CMP BYTE PTR [SI], 0`.

The loop executes six instructions per iteration: the comparison, the conditional jump, taking the character into AL, writing 0Eh to AH, INT 10h, and advancing SI; then JMP returns to the top. `MOV AH, 0Eh` could be moved out of the loop, but if it is not certain that INT 10h preserves AH, leaving it inside is safe.

An	SI	AL	ZF	SP	Screen
`LEA SI, msg`	msg+0	—	0	FFFEh	
`CALL print_me`	msg+0	—	0	**FFFCh**	
iteration 1, CMP: 'H' ≠ 0	msg+0	—	0	FFFCh	
iteration 1, INT 10h	msg+0	**48h**	0	FFFCh	H
iteration 1, ADD SI, 1	**msg+1**	48h	0	FFFCh	H
end of iteration 2	msg+2	65h	0	FFFCh	He
end of iteration 5	msg+5	6Fh	0	FFFCh	Hello
end of iteration 11	msg+11	64h	0	FFFCh	Hello World

An	SI	AL	ZF	SP	Screen
end of iteration 12	msg+12	21h	0	FFFCh	Hello World!
CMP: [SI] = 0, JE stop	msg+12	21h	**1**	FFFCh	Hello World!
the procedure's `RET`	msg+12	21h	1	**FFFEh**	Hello World!
the main program's `RET`	msg+12	21h	1	**0000h**	Hello World!

Table 8.2.19: Trace of print_me (selected moments; msg = 0118h).

The loop runs 12 iterations and prints one character in each (Table 8.2.19). The sixth character is a space (20h); a space is a character too and is printed on the screen. In the thirteenth comparison [SI] = 0, ZF = 1, and JE jumps to `stop`. The zero byte is not printed. Throughout the loop SP = FFFCh: the procedure pushes nothing, and the stack holds only the return address.

The procedure changes SI and does not preserve it. On return SI points at the zero byte at the end of the string (msg + 12 = 0124h). This is a side effect and "Clobber: SI, AX" should be written in the comment. The same procedure can be called for another text by changing only SI: `LEA SI, other_msg` / `CALL print_me`.

8.2.6.2 Adding two variables

The second example is a procedure that finds its parameters and its result in memory variables.

```
ORG 100h
start:
CALL AddNumbers     ; call it
RET                 ; to DOS
AddNumbers PROC
MOV AX, number1     ; AX = first number
ADD AX, number2     ; AX += second number
MOV result, AX      ; store the result
RET                 ; end of procedure
AddNumbers ENDP
number1 DW 5        ; first number 5
number2 DW 3        ; second number 3
result DW ?         ; holds the result
```

The first instruction of the program is CALL at 0100h; the return address is 0103h. The procedure starts at 0104h. The instruction lengths differ from one another. `MOV AX, number1` is 3 bytes with the short encoding special to AX (A1 + address). `ADD AX, number2` is 4 bytes. `MOV result, AX` is 3 bytes, again with the AX-specific encoding. The variables are behind the procedure: number1 = 010Fh, number2 = 0111h, result = 0113h.

Line	Instruction	AX	SP	ZF	CF	result
3	`CALL AddNumbers`	0000h	**FFFCh**	0	0	?

Line	Instruction	AX	SP	ZF	CF	result
6	`MOV AX, number1`	**0005h**	FFFCh	0	0	?
7	`ADD AX, number2`	**0008h**	FFFCh	0	0	?
8	`MOV result, AX`	0008h	FFFCh	0	0	**0008h**
9	`RET`	0008h	**FFFEh**	0	0	0008h
4	`RET` (DOS'a)	0008h	**0000h**	0	0	0008h

Table 8.2.20: Trace of adding two variables (each row is the state after execution).

Table 8.2.20 also shows the flags of ADD. 5 + 3 = 8 is not zero (ZF = 0) and produces no carry in 16 bits (CF = 0). MOV does not touch the flags. `result` is stored in memory as `08 00`, in little-endian order. `DW ?` says the initial value is undefined. emu8086 puts 0 in this area, but the program must not rely on this value.

This procedure is short and clear, but it works only with `number1`, `number2`, and `result`. To add two other numbers, either those numbers have to be copied into these variables first or a new procedure has to be written. This is the weakness of passing parameters in memory variables.

8.2.6.3 An array sum procedure

The third example adds the elements of an array. It takes two things as parameters: the address of the array (SI) and the number of elements (CX). It returns the result in AL. The procedure works with whichever array it is given the address of.

```
ORG 100h
LEA SI, vector     ; param 1: address
MOV CX, 5          ; param 2: count
CALL sum
MOV m, AL          ; result is in AL
RET
sum PROC           ; In: SI, CX
MOV AL, 0          ; Out: AL = sum
again:
ADD AL, [SI]
INC SI
LOOP again
RET
sum ENDP
vector DB 5, 4, 5, 2, 1
m DB 0
```

Line	Instruction / iteration	SI	CX	AL	SP	m
2–3	`LEA SI, vector` · `MOV CX, 5`	0115h	0005h	00h	FFFEh	00h
4	`CALL sum`	0115h	0005h	00h	**FFFCh**	00h

Line	Instruction / iteration	SI	CX	AL	SP	m
8	`MOV AL, 0`	0115h	0005h	00h	FFFCh	00h
10–12	iteration 1: +5	0116h	0004h	**05h**	FFFCh	00h
10–12	iteration 2: +4	0117h	0003h	**09h**	FFFCh	00h
10–12	iteration 3: +5	0118h	0002h	**0Eh**	FFFCh	00h
10–12	iteration 4: +2	0119h	0001h	**10h**	FFFCh	00h
10–12	iteration 5: +1	011Ah	0000h	**11h**	FFFCh	00h
13	`RET`	011Ah	0000h	11h	**FFFEh**	00h
5	`MOV m, AL`	011Ah	0000h	11h	FFFEh	**11h**

Table 8.2.21: Trace of the array sum (vector = 0115h, m = 011Ah).

The sum 5 + 4 + 5 + 2 + 1 = 17 = 11h (Table 8.2.21). After the last iteration SI = 011Ah and points at the variable `m` right behind the array. When LOOP brings CX down to 0 it is not taken and the flow moves on to RET. The caller writes the result in AL to `m`.

The two limits of the procedure should be written in its description. First, the sum is kept in the 8-bit AL. If the sum of the elements exceeds 255, the result overflows and only the low 8 bits remain. Second, the procedure changes SI and CX. On return SI points at the end of the array and CX is 0. The same procedure can be called for another array by changing only SI and CX.

8.2.6.4 A parameter block

If a procedure has many parameters, registers do not suffice. One solution is to write the parameters in a block that sits one after another in memory and give the procedure only the address of the block. A single register carries as many parameters as wanted.

```
LEA BX, block
CALL calc             ; result = a × b + c
RET
calc PROC             ; In: BX = block address
PUSH AX
PUSH DX               ; MUL writes DX too
MOV AX, [BX]          ; a
MUL WORD PTR [BX+2]   ; × b
ADD AX, [BX+4]        ; + c
MOV [BX+6], AX        ; result
POP DX
POP AX
RET
calc ENDP
block DW 12, 7, 5, ?
```

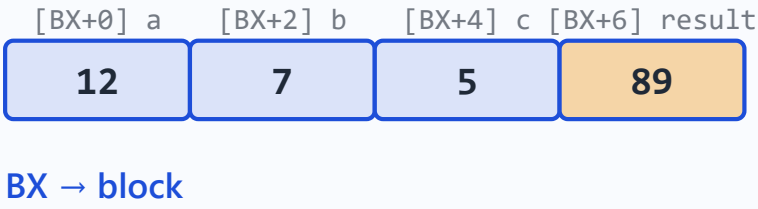


Figure 8.2.19: The parameter block: [BX+0] a = 12, [BX+2] b = 7, [BX+4] c = 5, [BX+6] result = 89.

The block consists of four words (Figure 8.2.19). The procedure reads each parameter at a fixed offset from BX: a at [BX], b at [BX+2], c at [BX+4]. It writes the result to the fourth word of the block, [BX+6]. In the line `MUL WORD PTR [BX+2]` the size is written explicitly; since MUL's only operand is in memory, the assembler cannot know the size any other way.

The computation goes like this: AX = 12, with MUL DX:AX = 12 × 7 = 84 = 0054h, with ADD AX = 84 + 5 = 89 = 0059h. The result is written to [BX+6]. When run in emu8086 the block starts at 0117h and the result is written to the bytes 011Dh–011Eh as `59 00`. Although the procedure uses AX and DX, it preserves both, because it returns its result in memory, not in a register. On return AX and DX hold their values from before the call.

The same procedure is called with another block by changing only BX. The block layout, that is, which parameter sits at which offset, must be documented in the procedure's description. The file and program loading services of DOS use this method too: the caller prepares a parameter block and gives its address in a register.

8.2.6.5 Testing a procedure

If a procedure is tried only with ordinary input, its hidden errors do not show up. Testing must cover boundary cases as well: the first and last element, a value that is not present, a one-element input, and an empty input. When the single-exit `find` procedure (array: 3, 9, 7, 1, 4) is run with these cases, the results in Table 8.2.22 are obtained.

Case	AL	CX	CF	BX	Assessment
first element	3	5	1	0	correct
last element	4	5	1	4	correct
not in the array	8	5	0	5	correct; BX is meaningless, only CF is looked at
search in a one-element range	9	1	0	1	correct: 9 is the second element, outside the range searched
empty array	7	0	1	2	wrong: with CX = 0, LOOP goes around 65,536 times

Table 8.2.22: Testing the find procedure with boundary cases.

In all cases CX and SI are at their input values on return; the preservation works correctly. The error shows up only with the empty array. Its cause is how LOOP works: LOOP first decrements CX by 1 and then looks at whether it is zero. In a loop entered with CX = 0, the first LOOP makes CX 0 − 1 = FFFFh and jumps. The loop may go around 65,536 times. The procedure, which should have scanned no elements at all, starts scanning the bytes of the array and “finds” 7 at the third byte.

The fix is to test CX before entering the loop. The `JCXZ` (jump if CX zero) instruction jumps if CX = 0 and does not look at the flags:

```
find PROC
PUSH CX
PUSH SI
XOR BX, BX
JCXZ notfound      ; empty array: leave without searching
scan: CMP [SI], AL
JE found
INC SI
INC BX
LOOP scan
notfound: CLC      ; not found
JMP done
found: STC
done: POP SI
POP CX
RET
find ENDP
```

The corrected procedure returns CF = 0 and BX = 0 for CX = 0. When 7 is searched with CX = 5, it gives CF = 1, BX = 2 as before. Both results were obtained by running in emu8086. This example shows why testing is prepared as a list for every procedure: the error would not have shown up in any trial with ordinary input.

8.2.6.6 Nested calls

A procedure can call another procedure. In this case the stack holds more than one return address at the same time. The LIFO structure of the stack automatically makes the returns happen in the right order: the one called last returns first.

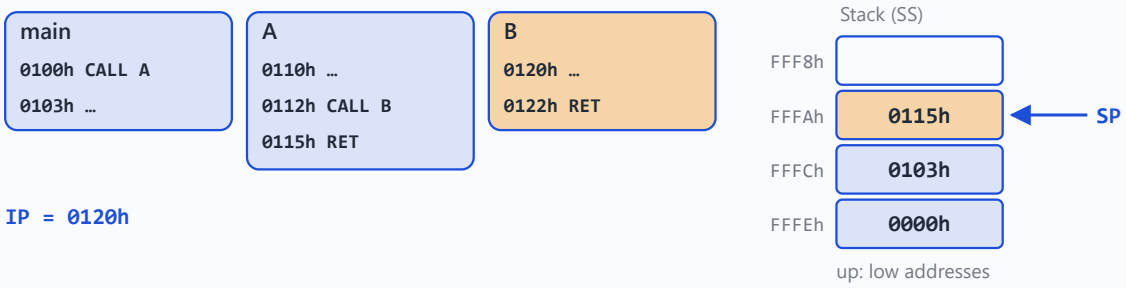


Figure 8.2.20: main has called A, and A has called B. The stack holds two return addresses: 0115h on top (return to A), and 0103h beneath it (return to main).

An	IP	SP	Stack (from the top)
main, before CALL A	0100h	FFFEh	0000h
`CALL A` (0100h)	0110h	**FFFCh**	0103h, 0000h
`CALL B` from inside A (0112h)	0120h	**FFFAh**	0115h, 0103h, 0000h
B's `RET`	0115h	**FFFCh**	0103h, 0000h
A's `RET`	0103h	**FFFEh**	0000h

Table 8.2.23: The stack in nested calls.

In the table (Table 8.2.23), B's RET takes the 0115h on top of the stack and returns to A. When A's job is done, its RET takes the 0103h one level below and returns to main. Each RET knows only the address on top; it is the structure of the stack that puts the right address on top. Procedures therefore do not have to know how many levels deep they are called.

The same layout can also be followed with a more concrete program. main calls A with AX = 1; A adds 2 to AX, calls B, then adds 4 to AX; B doubles AX. The sequence is 1 → 3 → 6 → 10 and the program ends with AX = 000Ah. At most two return addresses accumulate on the stack.

8.2.6.7 Computing the stack need

In nested calls the stack grows by the bytes that accumulate along the deepest call chain. Knowing how much stack a program needs is necessary for reserving enough room for the stack and for preventing the stack from colliding with data or code.

Definition 8.2.9

The **stack requirement** of a program is the sum of the bytes that accumulate along the deepest path of the call tree. Each near CALL adds a 2-byte return address, each far CALL 4 bytes, each PUSH 2 bytes, and each local area its own size.

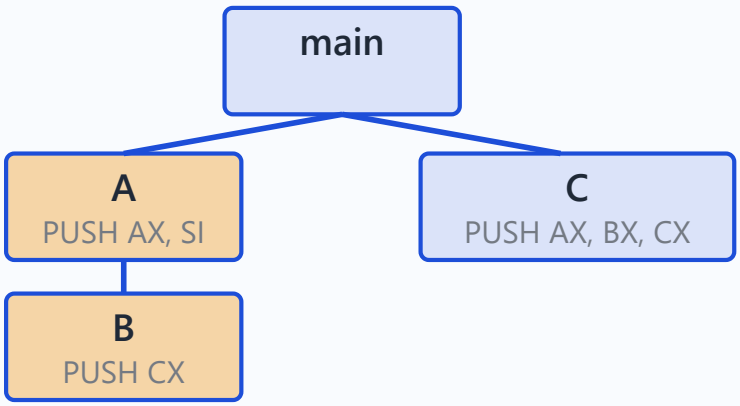


Figure 8.2.21: The call tree: main calls A and C; A also calls B. A saves two registers, C three, and B one.

Path	Computation	Bytes
main → A → B	2 (CALL A) + 4 (PUSH AX, SI) + 2 (CALL B) + 2 (PUSH CX)	10
main → C	2 (CALL C) + 6 (PUSH AX, BX, CX)	8

Table 8.2.24: The stack need of the call paths.

The deepest path is main → A → B and it requires 10 bytes (Table 8.2.24). In a program that starts with SP = FFFEh, SP goes down at most to FFFEh − 10 = FFF4h. Since C saves three registers, it does more PUSHes than A, but its depth is smaller. A and C are never on the stack at the same time: when C is called, A's bytes have long been taken back. The requirement is therefore not the sum of the paths but the largest one.

A margin must be added to the computation. Hardware interrupts can arrive at any moment of the program, at the deepest moment of the stack as well. Each interrupt pushes 6 bytes onto the stack (FLAGS, CS, and IP), and the interrupt service routine adds its own saves. A margin is therefore always left above the computed value. In a .COM program the stack grows downward from the top of the segment, and code and data grow upward from the bottom. When the gap between them is used up, the stack overwrites the data. This is called **stack overflow**, and the processor does not report it.

8.2.7 Recursion

8.2.7.1 A procedure that calls itself

We have seen that a procedure can call another procedure. The same mechanism also lets a procedure call itself. For the processor it does not matter which procedure the target of CALL is; it pushes the return address and jumps.

Definition 8.2.10

Recursion is a procedure's calling itself, directly or through another procedure. Each call puts a new return address and its own saved values on the stack; the values of each level stay separate from one another. The case in which the procedure returns without calling itself again is called the **base case** (the stopping condition).

Recursion is a natural solution when the problem becomes smaller at each step and resembles itself. Printing a number in its decimal digits is a good example. To print 472 is to print 47 first and then '2'. To print 47 is to print 4 first and then '7'. 4 has a single digit; nothing else has to be printed to print it. This is the base case.

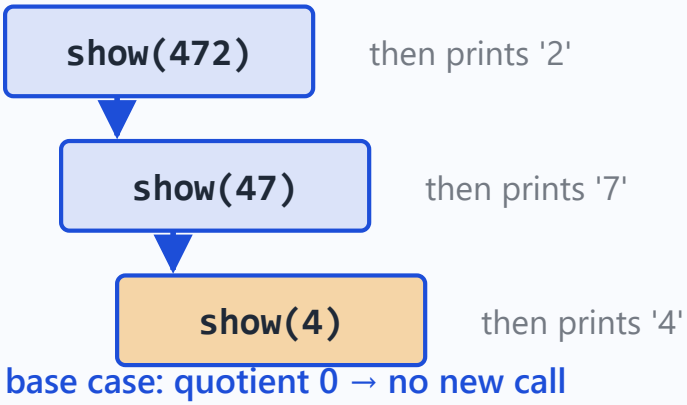


Figure 8.2.22: show(472) calls show(47), which calls show(4). On the way back 4, 7, 2 are printed in that order.

The base case is essential. A procedure with no base case calls itself forever. Each call adds at least 2 bytes to the stack. SP keeps decreasing, the stack grows down onto the data and code of the program, and the program crashes. Besides printing numbers, computing a factorial, walking a tree, and walking a directory are typical examples of recursion.

8.2.7.2 Recursive number printing: code trace

```
ORG 100h
MOV AX, 472
CALL show
RET
show PROC       ; In: AX (unsigned)
XOR DX, DX
MOV BX, 10
DIV BX          ; AX = quot., DX = digit
PUSH DX
OR AX, AX       ; more digits?
JZ done
CALL show       ; upper digits first
done: POP AX
ADD AL, '0'
MOV AH, 0Eh
INT 10h
RET
show ENDP
```

The procedure prints the unsigned number in AX on the screen. The first three instructions divide the number by 10. The 16-bit `DIV BX` divides the 32-bit number in DX:AX, so DX is cleared first with `XOR DX, DX`. After the division AX = number / 10 and DX = number mod 10. The remainder is the last digit of the number and is saved on the stack with `PUSH DX`. `OR AX, AX` sets ZF without changing AX: if the quotient is zero there is no other digit to print and JZ jumps to the base case, `done`. If the quotient is not zero, the procedure calls itself to print the quotient. On the way back the four instructions at `done` take the saved digit, convert it to ASCII, and print it.

In the program layout the CALLs are in two places. The main program's `CALL show` is at 0103h and its return address is 0106h. The `CALL show` inside the procedure is at 0113h and its return address is 0116h, that is, the `done` label. When the procedure calls itself, the machine code of this CALL is `E8 F1 FF`: 0107h − 0116h = −0Fh = FFF1h. Since the target is behind, the displacement is negative.

Line	Event	AX	DX	SP	Stack (from the top)	Screen
2–3	AX = 472, `CALL show`	01D8h	0000h	**FFFCh**	0106h	
6–9	level 1: 472 / 10, PUSH DX	**002Fh**	**0002h**	**FFFAh**	2, 0106h	
10–12	AX ≠ 0: `CALL show`	002Fh	0002h	**FFF8h**	0116h, 2, 0106h	
6–12	level 2: 47 / 10, PUSH, CALL	**0004h**	**0007h**	**FFF4h**	0116h, 7, 0116h, 2, 0106h	
6–11	level 3: 4 / 10, PUSH; AX = 0, JZ done	**0000h**	**0004h**	**FFF2h**	4, 0116h, 7, 0116h, 2, 0106h	
13–16	POP AX = 4, print	**0E34h**	0004h	**FFF4h**	0116h, 7, 0116h, 2, 0106h	4
17, 13–16	RET → 0116h; POP AX = 7, print	**0E37h**	0004h	**FFF8h**	0116h, 2, 0106h	47
17, 13–16	RET → 0116h; POP AX = 2, print	**0E32h**	0004h	**FFFCh**	0106h	472

Line	Event	AX	DX	SP	Stack (from the top)	Screen
17	last RET → 0106h	0E32h	0004h	**FFFEh**	0000h	472

Table 8.2.25: Trace of the recursive number printing (each group of rows is the state after execution).

The trace (Table 8.2.25) splits into two phases. **In the descent phase** the procedure separates one digit at each level and calls itself again. Two words are added to the stack at each level: the digit and the return address. But the digits are separated from last to first: first 2, then 7, then 4. At the third level the quotient becomes 0 and the base case is reached. At this moment SP is at its deepest point: FFF2h. The stack holds three digits and three return addresses (Figure 8.2.23).

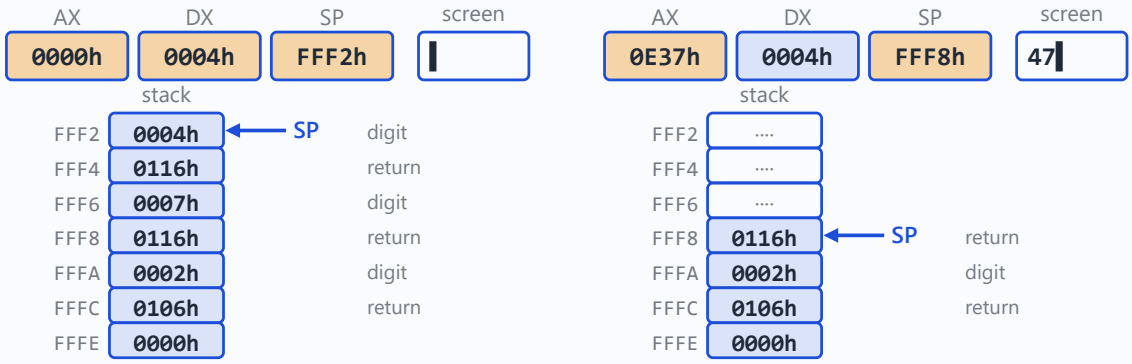


Figure 8.2.23: On the left, at the base case: SP = FFF2h, with three digits and three return addresses on the stack. On the right, after returning to the second level: '4' and '7' have been printed, SP = FFF8h.

In the ascent phase each level takes its own digit from the stack and prints it. The digit pushed last (4) comes out first. The digits are thus printed in the reverse of the order they were separated, that is, in the right order. The third level's RET returns to 0116h, to the `done` line of the second level. The second level prints 7, and its RET also returns to 0116h, this time to the first level. That the same return address appears twice on the stack is natural: both calls were made from the same CALL instruction. The first level prints 2, and the last RET returns to 0106h, to the main program. SP = FFFEh and the stack is balanced.

The last row of the trace is the state right after the outermost RET of the procedure: IP = 0106h, SP = FFFEh. Then the main program's RET takes 0000h and the program ends with SP = 0000h.

Since each level keeps its own digit on the stack, no extra variable or array was needed. The stack gave each of the levels a separate storage area. A loop-based solution also reverses the digits by pushing them onto the stack and then popping them; recursion does this job with the natural layout of CALL and RET.

Example 8.2.4

What does the procedure print when called with AX = 0? How low does SP go at most?

1. **Division.** 0 / 10: AX = 0, DX = 0. The digit 0 is pushed onto the stack.
2. **Base case.** AX = 0, JZ jumps to `done`; no new call is made.
3. **Printing.** POP AX = 0, ADD AL, '0' → 30h; '0' is printed on the screen.
4. **Stack.** The main program's CALL (2 bytes) and one digit (2 bytes): SP goes down at most to FFFEh − 4 = FFFAh.

Because the base case check is made after the division, the number 0 is also printed as '0' without any extra code.

8.2.7.3 Recursion or a loop?

The same job can be done with a loop too: the number is divided by 10 repeatedly, the digits are pushed onto the stack and counted with a counter; a second loop pops the digits and prints them. The comparison of the two solutions is in Table 8.2.26.

Criterion	Recursion (show)	Loop (division + LOOP)
Code	short; close to the definition of the problem	needs a counter and a separate printing loop
Stack	return address + digit at each level: 4 bytes	2 bytes per digit
Time	CALL + RET at each level (19 + 8 cycles)	no call overhead
Risk	if the depth grows without bound, the stack overflows	stack use is limited

Table 8.2.26: Comparison of recursive and loop-based number printing.

Example 8.2.5

How deep does the stack get at most if the `show` procedure is called with 65,535, the largest 16-bit unsigned number?

1. **Number of levels.** The number of levels equals the number of digits: 65,535 has five digits, so five levels.
2. **Bytes.** Each level adds a return address (2 bytes) and a digit (2 bytes): 5 × 2 + 5 × 2 = 20 bytes.
3. **SP.** FFFEh − 20 = FFFEh − 14h = FFEAh.
4. **Time.** Five CALLs and five RETs: 5 × (19 + 8) = 135 cycles, 27 µs of call overhead at 5 MHz.

The depth of this procedure is limited to 5, because AX carries at most five digits. When run with 65,535 in emu8086, the smallest value of SP is FFEAh.

The place of the base case matters too. The check must be made before the new CALL. In `show` the division comes first, then the quotient is checked for zero, and only if it is not zero is the CALL made. Had the check been put after the CALL, no condition would have been tied to the CALL and the

procedure would call itself without stopping.

8.2.8 Common mistakes

8.2.8.1 An unbalanced stack: code trace

Most procedure errors have the same root: when RET executes, the return address is not on top of the stack. The most common cause is a forgotten POP for a PUSH.

```
ORG 100h
MOV AX, 1234h     ; 0100h: B8 34 12
CALL bad          ; 0103h: ret = 0106h
RET               ; 0106h
bad PROC
PUSH AX           ; AX saved...
; ... work ...
RET               ; forgot POP AX!
bad ENDP
```

Line	Instruction	AX	IP (next)	SP	Stack (from the top)
2	`MOV AX, 1234h`	1234h	0103h	FFFEh	0000h
3	`CALL bad`	1234h	0107h	**FFFCh**	0106h, 0000h
6	`PUSH AX`	1234h	0108h	**FFFAh**	1234h, 0106h, 0000h
8	`RET`	1234h	**1234h**	**FFFCh**	0106h, 0000h

Table 8.2.27: Trace of the unbalanced stack (each row is the state after execution).

RET takes whatever is on top of the stack for the return address (Table 8.2.27). On top is the saved value of AX, 1234h. After RET, IP = 1234h. The processor starts executing the bytes at address CS:1234h as instructions. There is no program code at that address; whatever was left in memory gets executed. The program crashes or behaves unpredictably. The real return address, 0106h, stays on the stack, at the place SP = FFFCh points at, never used.

The rule is: when leaving a procedure, SP must equal its value at entry. Every PUSH must have a POP in reverse order, and on every path of the procedure at that. The error is easier to make as the code between PUSH and RET gets longer and branches multiply. That is why a single exit point is recommended.

8.2.8.2 Type mismatch: code trace

The second typical error is a mismatch between the type of the call and the type of the return. In the program below the procedure `farp` ends with RETF but is called with a near CALL.

```
ORG 100h
MOV AX, 5
CALL farp       ; near CALL: IP only
ADD AX, 1
RET
farp PROC
SHL AX, 1
RETF            ; pops IP and CS!
farp ENDP
```

Line	Instruction	AX	CS:IP (next)	SP	Stack (from the top)
2	`MOV AX, 5`	0005h	0700:0103	FFFEh	0000h
3	`CALL farp`	0005h	0700:010A	**FFFCh**	0106h, 0000h
7	`SHL AX, 1`	**000Ah**	0700:010C	FFFCh	0106h, 0000h
8	`RETF`	000Ah	**0000:0106**	**0000h**	empty

Table 8.2.28: Trace of a near CALL with a far return (each row is the state after execution).

A near CALL pushes only IP (0106h) onto the stack. RETF, however, pops two words. The first becomes IP: 0106h, which is right. The second becomes CS. Since CS is not on the stack, RETF takes the word one below, the 0000h that DOS pushed for the .COM program. SP grows by 4 and becomes 0000h. The processor jumps to address 0000:0106h (Table 8.2.28).

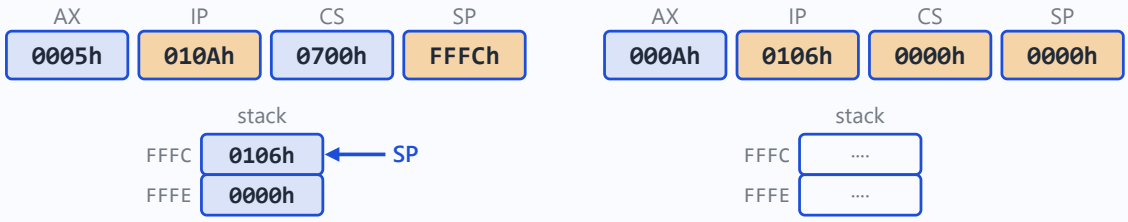


Figure 8.2.24: On the left, after the near CALL: only IP (2 bytes) on the stack. On the right, after RETF: CS = 0000h, IP = 0106h, SP = 0000h.

The physical address 0000:0106h is 00106h. This address is inside the interrupt vector table (00000h–003FFh); the vector of type 41h is in the bytes 00104h–00107h. There is not program code there but the addresses of interrupt routines. The processor takes these bytes for instructions and executes them, and the program crashes. `ADD AX, 1` never runs. In the opposite mismatch, that is, when a procedure called with CALL FAR ends with a near RET, CS is left on the stack and the return is again wrong.

8.2.8.3 List of mistakes

The most frequent mistakes in procedures are collected in Table 8.2.29. They are mistakes that most assemblers cannot catch and that show up only at run time.

Mistake	What happens?	Remedy
Forgetting RET	The processor does not see ENDP; the code or data after the procedure is executed.	Every path of every procedure ends with RET.
PUSH / POP imbalance	RET takes a wrong value into IP.	Every PUSH has a POP in reverse order.
Falling into a procedure	A procedure in the middle of the main code is executed without being CALLed; its RET at the end ends the program.	Procedures are written after the main program's RET or jumped over with JMP.
JMP instead of CALL	JMP pushes no return address; RET returns to an unrelated value on the stack.	A procedure is entered only with CALL.
Name mismatch	`p1 PROC` … `p2 ENDP` : assembler error.	The same name at the start and at the end.
Early RET	A conditional RET after a PUSH takes the saved value for the return address.	All paths go to a single exit label.
MOV instead of LEA	`MOV SI, arr` loads not the address but the first word of the array; the procedure reads from an unrelated address.	For addresses, `LEA` or `MOV SI, OFFSET arr` .
DX destroyed silently	The 16-bit MUL and DIV write DX too; the caller's DX is erased.	Write “Clobber: DX” in the comment, or PUSH/POP DX.
Type mismatch	Near CALL + RETF or CALL FAR + RET: the stack is off by 2 bytes.	Type on the PROC line; always RET for the return.
Disturbing the flag	After STC/CLC, an instruction such as ADD or CMP changes CF.	The instruction that sets the flag is the last flag-changing instruction before RET.
Preserving the result register	PUSH AX / POP AX erases the result in AX.	Output registers are not preserved.
Infinite recursion	If there is no stopping condition, or it is checked after the CALL, each call takes 2 bytes of space and the stack overflows.	The base case is checked before the CALL.

Table 8.2.29: Common mistakes in procedures.

Common mistake

Using JMP instead of a procedure call. In the program below `JMP add` has been written instead of `CALL add` :

```
ORG 100h
MOV AX, 1
JMP add
MOV BX, 9      ; does it run?
RET
add PROC
ADD AX, 5
RET
add ENDP
```

JMP pushes nothing onto the stack. The RET of `add` takes DOS's 0000h on top of the stack, and the program silently ends there, with AX = 6. `MOV BX, 9` never runs. Since the program does not crash, the error may go unnoticed. Had there been another value on the stack, RET would have returned to a random address.

8.2.9 Exercises and solutions

The six exercises from the slides are solved here with intermediate steps.

1. A `CALL p` at address 0120h calls p at 0100h. What is its machine code? What is pushed onto the stack?

CALL is 3 bytes; the next instruction is at 0123h. Displacement = 0100h − 0123h = −23h. In 16-bit two's complement −23h = FFDDh. The low byte is written first: `E8 DD FF`. 0123h is pushed onto the stack as the return address. Check: 0123h + FFDDh = 10100h, reduced to 16 bits 0100h.

2. In the m2 example, what would the final AX be if `MOV AL, 3` were added before the second `CALL m2`?

The first call gives AX = 1 × 2 = 0002h. The added instruction makes AL 3; AH stays 00h. The second call gives AX = 3 × 2 = 0006h. This time the first parameter of the second call does not depend on the result of the first call. Since the added instruction is 2 bytes, the address of the second CALL and its return address shift by 2 bytes, but the result does not change.

3. With SP = FFFEh, 3 CALLs are made in a row (no RET at all). What is SP?

Each near CALL pushes 2 bytes onto the stack: FFFEh − 3 × 2 = FFFEh − 6 = FFF8h. There are three return addresses on the stack. Had the calls been far, each would have pushed 4 bytes and SP would be FFF2h.

4. Why is it wrong to call a procedure that destroys CX from inside a LOOP? Propose two fixes.

LOOP keeps its counter in CX. If the procedure changes CX, on return LOOP decrements the value the procedure left instead of the caller's counter. The loop runs the wrong number of times; if the procedure leaves CX = 0, LOOP makes it FFFFh and the loop almost never ends. The first fix is for the

procedure to do `PUSH CX` on entry and `POP CX` on exit. The second fix is for the caller to write `PUSH CX` / `CALL` / `POP CX` around the CALL. If CX is the output of the procedure, only the second fix is applicable.

5. Write a procedure `upper` that converts the lowercase letter in AL to uppercase (input/output AL).

```
upper PROC        ; In/Out: AL
cmp al, 'a'
jb done           ; below 'a': leave it
cmp al, 'z'
ja done           ; above 'z': leave it
sub al, 20h       ; 'a' (61h) → 'A' (41h)
done: ret
upper ENDP
```

In the ASCII table there is a constant difference of 20h between lowercase and uppercase letters. The procedure changes only characters in the range 'a'–'z'; JB and JA are unsigned comparisons. Since it uses no other register, no PUSH/POP is needed. Tried with the characters 'a', 'Z', '{', and '`', the results are 'A', 'Z', '{', and '`'; the characters on both sides of the range do not change.

6. Change print_me so that it returns the number of characters it printed in CX.

`MOV CX, 0` is added at the start of the procedure and `INC CX` next to the `ADD SI, 1` in the loop. Since CX is an output, it is not preserved. For 'Hello World!' CX = 12 on return. INC CX changes the flags, but since the CMP at the top of the loop sets ZF again on every iteration, the result is not affected.

8.2.10 Summary

- A **procedure** is a piece of code that performs a specific task, is called with CALL, and returns with RET to right behind the call. The code is written once and called from many places.
- **PROC / ENDP** and END are directives; they produce no machine code. CALL and RET are processor instructions. Procedures are written after the main program's RET; otherwise the flow falls into the procedure.
- **CALL** pushes the address of the next instruction onto the stack (SP − 2) and changes IP to the target. A near direct CALL stores the target as a relative displacement: rel16 = target − (address of CALL + 3).
- **RET**: IP ← [SP], SP + 2. In a .COM program the final RET takes the 0000h pushed by DOS, and the INT 20h in the PSP ends the program.

- A **far CALL** pushes CS first and then IP (4 bytes); **RETF** pops IP first and then CS. The type of the call and of the return must match. An **indirect CALL** takes the target from a register or memory; it is the basis of jump tables.
- Call overhead: CALL 19, RET 8, PUSH 11, POP 8 cycles. The overhead of a procedure that saves two registers is 65 cycles (13 µs at 5 MHz). For very short and frequent jobs a macro is more suitable.
- **Parameters** are passed in registers, in memory variables, by address (SI/DI/BX), or on the stack. In passing by value a copy is given, in passing by address a location. The result returns in a register, in CF, or in memory.
- **Preserving registers**: a procedure saves the registers it uses with PUSH and POPs them in reverse order; output registers are not preserved. All paths of a procedure that does PUSH merge at a single exit point.
- **The stack frame**: `PUSH BP` · `MOV BP, SP` · `SUB SP, n`. Parameters are at [BP+4], [BP+6]..., local variables at [BP−2], [BP−4]... Exit: `MOV SP, BP` · `POP BP` · `RET n`.
- **Recursion**: a procedure calls itself; each level keeps its own return address and values on the stack. The base case is checked before the CALL. The stack requirement is computed from the deepest call path and a margin is left for interrupts.
- Most errors have their root in the return address not being on top of the stack at the moment of RET: a forgotten POP, an early RET, JMP instead of CALL, a type mismatch.

8.2.11 Key terms

procedure (subroutine) procedure call return address directive label stack stack pointer (SP) base pointer (BP) push / pop (PUSH / POP) LIFO (last in, first out) parameter passing pass by value / by reference register preservation stack frame local variable recursion stack overflow / underflow offset segment flag little-endian

---

8.2.12 Test Yourself

1. A `CALL p` at address 0150h calls p at 0180h. What are the bytes of the instruction, and what value is pushed onto the stack?

   Answer

   The next instruction is at 0153h. Displacement 0180h − 0153h = 002Dh: `E8 2D 00`. 0153h is pushed onto the stack.

2. With SP = FFFEh, the caller pushes two parameters and makes a near CALL. The procedure starts with `PUSH BP`, `MOV BP, SP`, `SUB SP, 4`. What are BP and SP? At which addresses are the first

parameter pushed and the two local variables?

Answer

Two parameters (4 bytes), CALL (2), and PUSH BP (2): BP = FFFEh − 8 = FFF6h. After SUB SP, 4, SP = FFF2h. The first parameter is at [BP+6] = FFFCh and the second at [BP+4] = FFFAh. The locals are at [BP−2] = FFF4h and [BP−4] = FFF2h. After `MOV SP, BP`, `POP BP`, and `RET 4`, SP = FFFEh.

3. A procedure called with AX = 1111h and BX = 2222h executes `PUSH AX`, `PUSH BX`, `MOV AX, 0`, `MOV BX, 0`, `POP AX`, `POP BX`, `RET`. What are AX and BX on return? Does RET return to the right address?

Answer

The value of BX (2222h) is on top of the stack; the first POP takes it into AX. AX = 2222h and BX = 1111h: the two registers have exchanged places. Since the numbers of PUSHes and POPs are equal, the stack is balanced and RET returns correctly. The error shows up only in the register values.

4. A procedure that returns its result in CF executes `STC`, then `OR AX, AX`, `POP CX`, and `RET`. What CF does the caller find?

Answer

CF = 0. OR is a logic instruction and always clears CF and OF. The value set by STC is lost. POP and RET do not touch the flags; the instruction that sets the flag must be the last flag-changing instruction before RET.

5. A `CALL FAR PTR fp` at 0700:0103 calls the FAR procedure at 0700:0109. How many bytes is the instruction? If SP = FFFEh, what is written to the stack? With which code is the procedure's RET produced?

Answer

A far direct CALL is 5 bytes: `9A 09 01 00 07` (offset first, then segment). The return address is 0103h + 5 = 0108h. First CS is pushed: [FFFCh] = 0700h; then IP: [FFFAh] = 0108h. SP = FFFAh. The RET in the FAR procedure is encoded as CB (RETF), and on return SP = FFFEh.

6. main calls P; P pushes AX, BX, and CX and calls Q; Q does `PUSH BP` and `SUB SP, 4`. If the program starts with SP = FFFEh, how low does SP go at most? If a margin is left for one interrupt, down to

what address must room be reserved?

Answer

2 (CALL P) + 6 (three PUSHes) + 2 (CALL Q) + 2 (PUSH BP) + 4 (local area) = 16 bytes. SP goes down at most to FFFEh − 10h = FFEEh. An interrupt pushes 6 bytes more: FFE8h. The interrupt routine's own saves are added to the account as well.

7. The recursive `show` procedure is called with AX = 1000. What appears on the screen? How low does SP go at most?

Answer

1000 appears on the screen; zeros are digits too. 1000 has four digits, four levels: 4 × (2 + 2) = 16 bytes. SP goes down at most to FFFEh − 16 = FFEEh.

8. The variable `a` is at address 0110h and holds 0005h. A programmer writes `MOV SI, a` instead of `LEA SI, a` to pass by address. What is SI? What does the procedure reach when it reads from `[SI]`?

Answer

`MOV SI, a` loads the contents of the variable: SI = 0005h. The procedure reads from DS:0005h, that is, in a .COM program from inside the PSP. The value read has nothing to do with `a`. If the procedure writes, it corrupts the PSP. The assembler does not catch this error, because both instructions are valid.

9. The `sq_plus` procedure with a local variable (x·x + x) is called with AX = 300. What are AX and DX on return? Is the result correct?

Answer

300 × 300 = 90,000 = 15F90h. The 16-bit MUL gives DX = 0001h, AX = 5F90h; CF = OF = 1. ADD AX, [BP−2]: 5F90h + 012Ch = 60BCh. On return AX = 60BCh and DX = 0001h. The correct result is 90,300 = 160BCh and does not fit in 16 bits. Since the procedure looks only at AX, the result is wrong; moreover the caller's DX has been corrupted.

8.2.13 Further reading

- Intel, *The 8086 Family User's Manual* (1979): the operation definitions, machine codes, and clock cycle counts of the CALL, RET, PUSH, and POP instructions. The values in Table 8.2.8 are from this source.

- B. B. Brey, *The Intel Microprocessors* (8th ed.), the "Procedures" section: near and far procedures, the forms of CALL and RET, passing parameters on the stack.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th ed.), the "Procedures" chapter: stack frames, local variables, recursion, and calling conventions with detailed examples.
- emu8086 documentation, *8086 Assembler Tutorial for Beginners*, Part 8 "Procedures": writing procedures in emu8086 and the print_me example.

8.2.14 References

1. emu8086 documentation: *8086 Assembler Tutorial for Beginners* (Part 8: Procedures).
2. Intel, *The 8086 Family User's Manual*, 1979: CALL, RET, and stack operations.
3. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009: Section 6.3 "Procedures".
4. K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014: Chapter 5 "Procedures".

PART III · CHAPTER 9

Macros and the emu8086.inc Library

A macro gives a name to a repeated piece of code; unlike a procedure, the assembler expands it as text at every place it is used. This chapter covers defining parameterized macros with MACRO and ENDM, expansion, parameter pitfalls, avoiding label clashes with LOCAL, and comparing macros with procedures in size and speed, running the PUTC and POWER macros step by step. The emu8086.inc part presents the INCLUDE directive, macros such as PUTC, GOTOXY, PRINT and PRINTN, and the procedures for reading and printing numbers. Sample programs built from the library's macros and procedures complete the chapter.

NOTES IN THIS CHAPTER

9.1 Macros

9.2 The emu8086.inc Library

PART III · CHAPTER 9.1

Macros

The same few instructions keep turning up side by side in assembly programs: print a character, save a few registers on the stack, add two numbers and store the sum. A procedure removes this repetition at run time, with CALL and RET. A macro does the same job at assembly time: wherever the assembler sees the macro's name, it writes the macro's body again. The processor never sees the macro; it executes only the instructions that result. These notes first show how a macro is defined and expanded and why its parameters behave like text. They then weigh the trade-off in size and speed between macros and procedures, and the problem that the LOCAL directive solves. The PUTC and POWER macros serve to treat register preservation and boundary cases. The last sections cover repeat blocks and conditional assembly, the more advanced text processing the assembler performs on the source. Every example is shown at the level of registers and memory, with a step-by-step trace of the expanded code.

LEARNING OUTCOMES

Macro definition
Write a parameterized macro with MACRO and ENDM; explain why a macro must be defined before the line that uses it.

Expansion
Work out by hand which instructions a macro call expands to, by substituting the parameters as text.

Macro or procedure?
Compute code size byte by byte and the extra time in clock cycles, and choose a macro or a procedure for a given job with a justification.

LOCAL
Explain the cause of the *Duplicate declaration* error in macros that contain labels, and give each expansion a unique label with LOCAL.

Side effects and boundary cases
Protect the registers a macro clobbers with PUSH and POP; spot parameter pitfalls, operator-precedence errors and the CX = 0 trap of LOOP.

Advanced macro features
Generate code and tables at assembly time with REPT, IRP and IRPC; assemble conditionally with IF, IFB and IFIDN.

9.1.1 The macro concept

9.1.1.1 The text processing done by the assembler

An assembly program goes through two separate stages. First the assembler reads the source file and translates it into machine code. Then the processor fetches this machine code from memory and executes it. Everything written in the source has a meaning in the first stage, but not every line becomes machine code. Directives such as `ORG 100h`, `DB` and `EQU` give information to the assembler; some of them emit bytes and some emit none. A macro is also a tool of this first stage.

A macro gives a name to a piece of code that repeats. In this respect it resembles a procedure. The difference is where the name is processed. The name of a procedure becomes an address in the machine code; at run time the processor goes to that address with `CALL` and comes back with `RET`. The name of a macro never enters the machine code. The assembler deletes every line that uses the macro and writes the macro's body in its place. This operation is called macro expansion. After expansion the source no longer contains macros, only ordinary instructions.

Definition 9.1.1

A **macro** is a code template, written between the `MACRO` and `ENDM` directives, that has a name and optional parameters. The assembler replaces every line that uses the macro's name like an instruction with a copy of the body in which the parameters have been substituted. This replacement is called **macro expansion**; it takes place at assembly time, before the program runs.

Figure 9.1.1 summarizes the process. The single source line `PUTC 'A'` turns into five instructions after it passes through the assembler. The machine code contains no call, no return address and no jump. To the processor these five instructions are no different from five instructions written by hand.

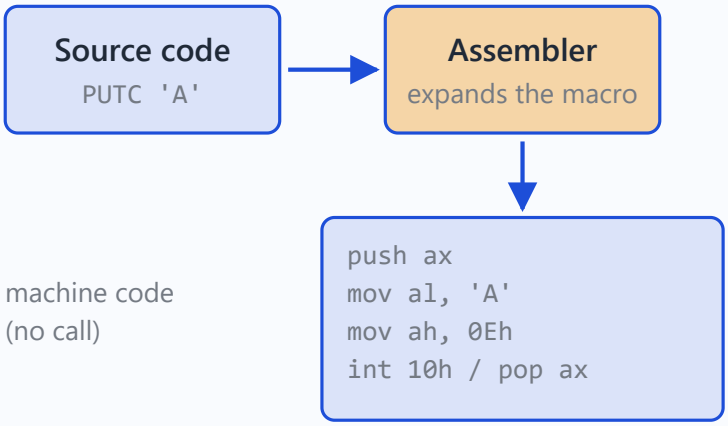


Figure 9.1.1: A macro call is expanded at assembly time. In place of the source line `PUTC 'A'` the assembler writes the body; the machine code contains no call.

Three important consequences follow from this view. First, a macro that is never used adds not a single byte to the machine code. The assembler reads the definition and records the name and body in a table, but it generates no code from the body. Code is produced only when the macro is used, and at the place where it is used. For this reason, including a library file with hundreds of macros does not enlarge the program.

Second, a macro must be defined before the line that uses it. The assembler reads the source file from top to bottom. If it reaches the line `PUTC 'A'` and does not yet know PUTC, it treats the line as an unknown instruction and reports an error. A procedure has no such ordering restriction. Because the assembler makes two passes, it finds the address of a procedure in the second pass even if the procedure is written at the end of the file. A macro, however, is not an address but a text, and the assembler cannot substitute a text it does not know.

Third, every use of a macro enlarges the program. If the same macro is used in ten places, the body is written ten times. The body of a procedure, by contrast, sits in memory once. This difference is the basis of the choice between a macro and a procedure.

9.1.1.2 Defining a macro

The definition of a macro has three parts: a header line, a body and an end line. The general form is:

```
name    MACRO  [parameter1, parameter2, ...]
        <instructions>
        ENDM
```

`name` is the name of the macro. When the macro is used, this name is written in the place of an instruction: `PUTC` is written where `MOV` would be. The name is followed by comma-separated parameter names. These are formal names; in the definition they are merely placeholders. The values given in the call replace these names as text. A macro may have no parameters at all: some macros just give a name to a pattern.

The body consists of the lines that are rewritten each time the macro is used. It may contain instructions, directives, calls to other macros, even data definitions. `ENDM` (end macro) ends the body. Unlike the `name ENDP` pattern of a procedure, the macro's name is not written before `ENDM`.

The parameter names mark the text to be substituted in the body. The assembler compares every word in the body with the parameter names. In place of a matching word it writes the corresponding text from the call. Figure 9.1.2 shows this mapping for the call `MyMacro 4, 5, DX`: the first parameter p1 maps to 4, the second p2 to 5, and the third p3 to DX. The line `MOV CX, p3` in the body thus becomes `MOV CX, DX`.

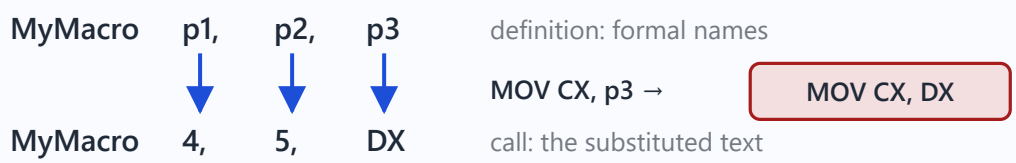


Figure 9.1.2: Parameter mapping. The formal names in the definition are paired in order with the texts in the call; the line `MOV CX, p3` becomes `MOV CX, DX`.

There is a critical point here. The assembler does not care whether a parameter is a register, a memory operand or a constant. It only replaces text. Whether the resulting line is a valid 8086 instruction is checked *after* expansion, like any ordinary line. A macro definition may therefore be free of errors and yet a particular call may produce one.

Common mistake

Writing the macro at the end of the program, next to the procedures. The placement that is right for a procedure causes an error for a macro: when the assembler reaches the call line it does not know the macro, so it treats the line as an unknown instruction. Macro definitions are written at the beginning of the file or in a library file added with INCLUDE.

9.1.1.3 Expansion step by step

Expansion is best understood through a concrete example. The macro MyMacro, which has three parameters, loads a value into each of three registers:

```
MyMacro  MACRO  p1, p2, p3
         MOV AX, p1
         MOV BX, p2
         MOV CX, p3
ENDM

ORG 100h
         MyMacro 1, 2, 3
         MyMacro 4, 5, DX
         RET
```

When the assembler reads the first four lines it generates no code. It records the name MyMacro, the three parameter names and the three body lines in a macro table. `ORG 100h` sets the location counter to 0100h. Then comes the line `MyMacro 1, 2, 3`. The assembler finds MyMacro in its table and pairs up the parameters: p1 = 1, p2 = 2, p3 = 3. Then it copies the body line by line:

1. In the line `MOV AX, p1`, the word p1 is replaced by 1: `MOV AX, 1`.
2. The line `MOV BX, p2` becomes `MOV BX, 2`.
3. The line `MOV CX, p3` becomes `MOV CX, 3`.

These three lines take the place of the call line and are translated into machine code like ordinary instructions. For the second call, `MyMacro 4, 5, DX`, the same process repeats with a new mapping: p1 = 4, p2 = 5, p3 = DX. The result is `MOV AX, 4`, `MOV BX, 5` and `MOV CX, DX`. The DX in the last line is not a number but a register name. The assembler does not notice this; it simply substitutes the text "DX". The resulting `MOV CX, DX` is a valid register-to-register copy.

emu8086 shows the expanded code in the emulator window like this:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
RET
```

The constants are written in hexadecimal with five digits; this is merely a display habit. What matters is that the two macro calls expand into six instructions and that the macro name never appears in the listing. Single-stepping in the emulator also advances over these six instructions. The programmer wrote two lines in the source; the processor executed six instructions.

The machine code of these instructions can also be computed. The `MOV r16, imm16` instruction, which loads an immediate value into a 16-bit register, is 3 bytes: the opcode B8h + register number and two data bytes. `MOV AX, 1` = `B8 01 00`, `MOV BX, 2` = `BB 02 00`, `MOV CX, 3` = `B9 03 00`. The

register-to-register `MOV CX, DX` is 2 bytes (`8B CA`). `RET` is a single byte (`C3`). The total size of the program is 5 × 3 + 2 + 1 = 18 bytes, that is, 12h bytes.

9.1.1.4 Code trace: running the expanded code

Running the expanded code is no different from running an ordinary program. Still, one detail matters for understanding the trace table. When emu8086 loads a .COM program into memory, it writes the program's size in bytes into register CX. This behavior comes from the DOS program loader. Since the size of this program is 18 bytes, CX = 0012h at load time. The other general-purpose registers are zero, and SP is FFFEh. The word 0000h sits at address FFFEh, which SP points to; the RET at the end of the program will pop this word.

Table 9.1.1 gives the trace of the expanded code. The first three steps are the instructions produced by the first call, the next three those of the second call. The small text under the instruction column shows which macro expansion the instruction comes from.

Step	Address	Instruction	AX	BX	CX	DX	SP	Explanation
1	0100h	`MOV AX, 1` MyMacro expansion	0001h	0000h	0012h	0000h	FFFEh	1st expansion, p1 = 1: AX = 0001h
2	0103h	`MOV BX, 2` MyMacro expansion	0001h	0002h	0012h	0000h	FFFEh	p2 = 2: BX = 0002h
3	0106h	`MOV CX, 3` MyMacro expansion	0001h	0002h	0003h	0000h	FFFEh	p3 = 3: CX = 0003h; the load value 0012h is overwritten
4	0109h	`MOV AX, 4` MyMacro expansion	0004h	0002h	0003h	0000h	FFFEh	2nd expansion, p1 = 4
5	010Ch	`MOV BX, 5` MyMacro expansion	0004h	0005h	0003h	0000h	FFFEh	p2 = 5
6	010Fh	`MOV CX, DX` MyMacro expansion	0004h	0005h	0000h	0000h	FFFEh	p3 = DX: register-to-register copy, CX = DX = 0000h
7	0111h	`RET` main program	0004h	0005h	0000h	0000h	0000h	0000h at the top of the stack → IP; SP = 0000h
8	0000h	`INT 20h (PSP:0000h)` PSP	0004h	0005h	0000h	0000h	0000h	the INT 20h at PSP:0000h ends the program

Table 9.1.1: Trace of the expanded code of the two MyMacro calls. Each row shows the state after the instruction has executed; at load time CX = 0012h (the program size).

A few points in the table deserve attention. The load value of CX, 0012h, stays unchanged in the first two steps and is overwritten with 0003h in the third. The last instruction of the second call copies DX into CX; since DX is initially zero, CX becomes 0000h. SP does not change in any step. As long as the macro body contains no PUSH or POP, the macro does not touch the stack. If the same job were done with a procedure, every CALL would decrease SP by 2 and write the return address to the stack. The last RET does not belong to the macro; it is the main program's RET. It takes the word 0000h from the top of the stack into IP and goes to the `INT 20h` instruction in the program segment prefix (PSP).

The running time of this program can also be computed in clock cycles. On the 8086, `MOV r16, imm16` takes 4 cycles and `MOV r16, r16` takes 2. The six instructions take 5 × 4 + 2 = 22 cycles. In a version written with a procedure, the CALL and RET times would be added to this for every call.

9.1.2 Parameters

9.1.2.1 A parameter is text: the swap trap

In a high-level language, the value of a parameter passed to a function is computed at the moment of the call, and that value is what the function receives. Nothing of the kind happens in a macro. A macro parameter carries not a value but a text. The assembler writes this text at every place where the parameter occurs in the body. The value of the register or variable that the text refers to is read only at run time, when the line executes. This distinction looks small, but a large share of macro bugs come from it.

Definition 9.1.2

A **macro parameter** is a formal name written as a placeholder in the macro definition. Its counterpart in the call is not a value but a text. The assembler writes this text at every place in the body where the parameter occurs. If the text is a register or variable name, its value is read when the expanded line executes at run time.

Suppose a programmer wants to use MyMacro to swap AX and BX. Since the first line of the macro puts p1 into AX and the second puts p2 into BX, the call `MyMacro BX, AX, 0` looks as if it does "AX ← BX, BX ← AX". After this call the programmer expects AX = 2222h and BX = 1111h. Writing out the expansion changes the picture:

```
ORG 100h
MOV  AX, 1111h
MOV  BX, 2222h
MOV  AX, BX      ; MyMacro BX, AX, 0
MOV  BX, AX      ; p2 = AX (broken!)
MOV  CX, 0
RET
```

Step	Address	Instruction	AX	BX	CX	Explanation
1	0100h	`MOV AX, 1111h` main program	1111h	0000h	0000h	AX = 1111h
2	0103h	`MOV BX, 2222h` main program	1111h	2222h	0000h	BX = 2222h
3	0106h	`MOV AX, BX` MyMacro expansion	2222h	2222h	0000h	p1 = BX: AX ← BX; 1111h is lost here

Step	Address	Instruction	AX	BX	CX	Explanation
4	0108h	`MOV BX, AX` MyMacro expansion	2222h	2222h	0000h	p2 = AX: the **current** value of AX is read, BX = 2222h
5	010Ah	`MOV CX, 0` MyMacro expansion	2222h	2222h	0000h	p3 = 0
6	010Dh	`RET` main program	2222h	2222h	0000h	result: AX = BX = 2222h
7	0000h	`INT 20h (PSP:0000h)` PSP	2222h	2222h	0000h	the program ends

Table 9.1.2: Trace of the swap attempt with `MyMacro BX, AX, 0`.

In the third step AX is filled with the value of BX, that is, 2222h. The old value of AX, 1111h, no longer exists anywhere. The `MOV BX, AX` of the fourth step does not read the "old AX" the programmer had in mind but the current value of AX. As a result both registers hold 2222h. In a high-level language this error could not occur, because parameters are evaluated at the moment of the call. In a macro, p2 = AX does not mean "the value of AX at the time of the call"; it is just the text "AX".

The general rule behind this trap is: a register that the macro body modifies must not be passed as a parameter that the body reads after that modification. MyMacro first writes AX and then reads p2, so passing AX as p2 is wrong. For a real swap the 8086 has the single instruction `XCHG AX, BX`. A macro that swaps must either use XCHG or save one of the values on the stack first.

Common mistake

Treating a macro parameter like a by-value parameter of a high-level language. In a call such as `ADD2 CX, AX`, if the body first writes CX into AX and then does `ADD AX, AX`, the result is 2 × CX rather than the expected CX + AX. The registers that a macro uses for itself are documented, and they are not passed as parameters.

9.1.2.2 Kinds of parameters and their validity

Because the assembler gives a parameter no type, the same macro can be called in very different ways. Which call is valid is decided not by the macro itself but by the instruction that results from the expansion. The ADD3 macro below shows this well:

```
ADD3 MACRO x, y, z
  MOV AX, x
  ADD AX, y
  MOV z, AX
ENDM
```

The macro adds x and y and writes the result to z. One may ask why it does not simply write something like `ADD z, x`. The 8086 instruction set does not combine two memory operands in one instruction. In instructions such as `ADD`, `MOV` and `CMP`, at most one operand can be in memory. This

restriction comes from the hardware: the ModR/M byte of an instruction can specify only one memory address. The macro therefore does the addition through AX. AX is an intermediate register here, and the macro clobbers it.

Call	Expansion (problem line)	Status
`ADD3 a, b, c`	`MOV AX, a` · `ADD AX, b` · `MOV c, AX`	valid; three variables
`ADD3 BX, 5, CX`	`MOV AX, BX` · `ADD AX, 5` · `MOV CX, AX`	valid; register, constant, register
`ADD3 a, b, 7`	`MOV 7, AX`	error: the destination cannot be a constant
`ADD3 k, b, c` (k DB)	`MOV AX, k`	error: size mismatch (16 bits ← 8 bits)
`ADD3 a, b`	`MOV , AX`	error: missing operand

Table 9.1.3: Expansions and validity of different calls of the ADD3 macro.

Table 9.1.3 compares five calls. In the first two calls the parameters are of different kinds, but the expansion produces valid instructions. In the third call z = 7, so the last line becomes `MOV 7, AX`. A constant cannot be written to; a constant is a number embedded in the instruction, not a memory cell or register that can be written. In the fourth call k is a byte variable. `MOV AX, k` tries to copy an 8-bit memory cell into a 16-bit register. The assembler catches this mismatch because it knows the variable's size from its definition. In the fifth call the third parameter is missing. The assembler replaces the missing parameter with empty text, and the last line turns into the meaningless `MOV , AX`.

All of these errors are reported at the line where the macro is used, although their cause is a line in the body. The way to understand the error message is to write out the expansion, in your head or on paper. If the same error message appears at several different lines of the program, the cause is usually in the body of the macro that all those lines use.

MASM offers two features for catching the missing-parameter problem at the definition level. If `:REQ` is written after a parameter name (for example `z:REQ`), that parameter becomes required, and a clear error message is produced when it is omitted. The form `y:=<1>` gives the parameter a default value; if y is left out of the call, 1 is used. These features are specific to MASM; in simpler assemblers a missing parameter always remains empty text.

9.1.2.3 Code trace: two expansions of ADD3

In the following program the ADD3 macro is called in two different ways. In the first call all three parameters are variables. In the second call a constant is given in place of y, and the result of the first call is used as input:

```
org 100h
ADD3 a, b, c        ; c = a + b
ADD3 c, 5, d        ; d = c + 5
ret
a dw 1200h
b dw 0345h
c dw ?
d dw ?
```

The expanded code is six instructions and a RET. The addresses of the variables depend on the size of the code, so the byte count must be worked out first. The instruction `MOV AX, a` is encoded in a short form specific to AX: the opcode `A1` and a two-byte address, 3 bytes in all. `ADD AX, b` uses the general memory-operand form: `03 06` and a two-byte address, 4 bytes in all. `MOV c, AX` is again the short form (`A3` + address), 3 bytes. The first expansion is 3 + 4 + 3 = 10 bytes.

In the second expansion the middle line is `ADD AX, 5`. The 8086 has a short form that adds an immediate value to AX: the opcode `05` and two data bytes, 3 bytes in all. The second expansion is therefore 3 + 3 + 3 = 9 bytes. The same macro has produced code of different lengths depending on the parameter. The code is 10 + 9 = 19 bytes, 20 bytes with the RET. Since the program starts at 0100h, a is at address 0100h + 14h = 0114h. b is at 0116h, c at 0118h and d at 011Ah.

Figure 9.1.3: Memory and AX after the last instruction of the second expansion. The words a, b, c, d lie one after another starting at 0114h; 154Ah has been written to d.

Step	Address	Instruction	AX	a	b	c	d	Explanation
1	0100h	`MOV AX, a` ADD3 expansion	1200h	1200h	0345h	0000h	0000h	x = a: AX ← [0114h]
2	0103h	`ADD AX, b` ADD3 expansion	1545h	1200h	0345h	0000h	0000h	y = b: 1200h + 0345h = 1545h
3	0107h	`MOV c, AX` ADD3 expansion	1545h	1200h	0345h	1545h	0000h	z = c: [0118h] ← 1545h
4	010Ah	`MOV AX, c` ADD3 expansion	1545h	1200h	0345h	1545h	0000h	2nd expansion, x = c: AX ← [0118h]
5	010Dh	`ADD AX, 5` ADD3 expansion	154Ah	1200h	0345h	1545h	0000h	y = 5 is a constant: short AX form `05 05 00`
6	0110h	`MOV d, AX` ADD3 expansion	154Ah	1200h	0345h	1545h	154Ah	z = d: [011Ah] ← 154Ah

Step	Address	Instruction	AX	a	b	c	d	Explanation
7	0113h	`ret` main program	154Ah	1200h	0345h	1545h	154Ah	the program ends

Table 9.1.4: Trace of the expansions of `ADD3 a, b, c` and `ADD3 c, 5, d` (a = 0114h, b = 0116h, c = 0118h, d = 011Ah).

Table 9.1.4 gives the details of the trace. In the first step AX is loaded with the value of a, 1200h. In the second step 0345h is added: 1200h + 0345h = 1545h. In the third step this sum is written to c. The second expansion reads c, adds 5 and writes the result to d: 1545h + 5 = 154Ah. Although the two expansions come from the same body, in one y is a memory address and in the other a constant. The assembler infers this difference from the text alone and chooses the appropriate machine code.

The memory accesses can also be examined at the hardware level. While `MOV AX, a` executes, the BIU reads a word from DS:0114h, that is, from physical address 07114h. Since a is at an even address, the read finishes in a single bus cycle. Had the address been odd, the 8086 would have had to fetch the word with two separate byte reads. The macro does not change this detail. Every instruction in the expansion makes exactly the memory accesses that its hand-written equivalent would.

Another lesson from this example is that the macro clobbers AX. A calling program that expects an old value in AX after ADD3 is mistaken. Which registers a macro modifies must be stated explicitly in the macro's documentation. Register preservation is treated separately with the PUTC macro.

Example 9.1.1

For `s1 dw 1234h` and `s2 dw 0F00Fh`, what are s3 and CF after the call `ADD3 s1, s2, s3`?

The expansion is `MOV AX, s1`, `ADD AX, s2`, `MOV s3, AX`. 1234h + F00Fh = 10243h, a 17-bit result. The low 16 bits, 0243h, fit in AX and the top bit goes to CF: CF = 1. Because MOV does not change flags, CF is still 1 after the macro. s3 = 0243h. Since the macro writes the carry nowhere, the correct result of the unsigned addition has been lost. The calling program can catch this situation with `JC` right after the macro.

9.1.2.4 Expressions and operator precedence

Since a parameter is text, an expression can also be given in the call: for instance `AREA 2+3, 4`. The assembler does not evaluate this expression first; it substitutes it as it stands. Operator precedence is applied again on the line where the expression has been substituted. This leads to unexpected results:

```
AREA MACRO wd, ht
  MOV AX, wd*ht
ENDM

DIFF MACRO p, q
  MOV CX, p-q
ENDM
```

In the call `AREA 2+3, 4`, wd = "2+3" and ht = "4". The expansion is the line `MOV AX, 2+3*4`. The assembler evaluates this expression at assembly time, and since multiplication comes before addition it finds 2 + 12 = 14. The value (2 + 3) × 4 = 20 that the programmer expected does not appear. The DIFF macro has the same problem: the call `DIFF 10, 4+1` becomes `MOV CX, 10-4+1`, and evaluated from left to right it gives CX = 7, not the expected 5.

Call	Expansion	Machine code	Result
`ALAN 2+3, 4`	`MOV AX, 2+3*4`	`B8 0E 00`	AX = 000Eh (14)
corrected `AREA 2+3, 4`	`MOV AX, (2+3)*(4)`	`B8 14 00`	AX = 0014h (20)
`FARK 10, 4+1`	`MOV CX, 10-4+1`	`B9 07 00`	CX = 0007h
corrected `DIFF 10, 4+1`	`MOV CX, (10)-(4+1)`	`B9 05 00`	CX = 0005h
`ALAN BX, 2`	`MOV AX, BX*2`	–	error: * works only on constants

Table 9.1.5: Expansion of expression parameters and their value at assembly time.

The solution is to put every parameter in parentheses in the body: `MOV AX, (wd)*(ht)` and `MOV CX, (p)-(q)`. Then whatever text arrives is evaluated within itself first. Table 9.1.5 compares the two forms with their machine codes. Note that the machine code contains no multiply or add instruction. The expression was evaluated at assembly time, and the result is embedded in the `MOV` instruction as an immediate value. The processor merely loads the number 000Eh or 0014h into AX.

The last row shows the limit of this point. The call `AREA BX, 2` produces the line `MOV AX, BX*2`. The value of BX is not known at assembly time; it is known only at run time. The assembler cannot multiply a register, so it reports an error. Multiplying a register's value takes an instruction that runs at run time: `MUL`, or `SHL` for powers of 2. This distinction between an expression evaluated at assembly time and an instruction executed at run time is the key to understanding macros.

Common mistake

Using parameters without parentheses in the body: write `(wd)*(ht)` instead of `wd*ht`. If the macro is tried only with parameters that consist of a single number, no error shows; at the first call with an expression the result is silently wrong. The assembler gives no warning, because `2+3*4` is a valid expression.

9.1.3 Macros and procedures

A macro and a procedure are two different solutions to the same problem: gathering repeated code under a name. One does the job at assembly time, the other at run time. This difference shows up in code size, speed, stack usage and the behavior of labels. This section first defines the two methods separately and then compares them with byte and clock-cycle calculations.

9.1.3.1 Properties of a procedure

A procedure is a block of code written between the `PROC` and `ENDP` directives, called with `CALL` and left with `RET`. Its body sits at one address in memory as a single copy. The PRINT_A procedure below writes 'A' to the screen:

```
PRINT_A PROC
      PUSH AX
      MOV  AL, 'A'
      MOV  AH, 0Eh
      INT  10h
      POP  AX
      RET
PRINT_A ENDP

      CALL PRINT_A  ; 3 bytes
      CALL PRINT_A  ; 3 bytes
```

When a near `CALL` instruction executes, the processor does three things. First it decreases SP by 2. Then it writes the address of the instruction after the CALL, the return address, to SS:SP. Finally it loads IP with the address of the procedure. `RET` does the opposite: it takes the word at SS:SP into IP and increases SP by 2. Because the return address is kept on the stack, the same procedure can be called from different places in the program; each time it returns to the right place.

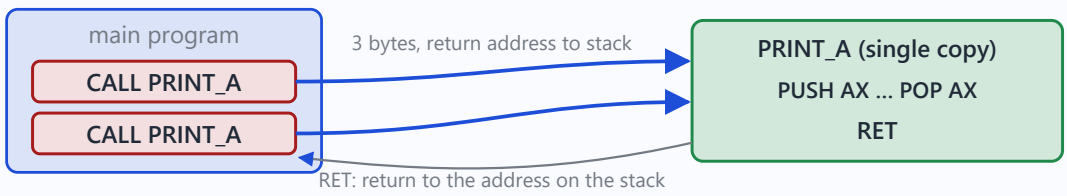


Figure 9.1.4: A procedure is a single copy in memory. Two CALLs go to the same body; each time, RET returns to the return address on the stack, that is, to the caller.

A near CALL instruction is 3 bytes: the opcode `E8` and a two-byte relative displacement. The displacement is found by subtracting the address of the instruction after the CALL from the address of the procedure. Even if the same procedure is called a hundred times, the body is written once; each

call adds only 3 bytes. Parameters are passed in registers, in memory or on the stack. Because the name of a procedure turns into an address in the machine code, a procedure can be defined anywhere in the file, even after the call.

9.1.3.2 Properties of a macro

The PUTC macro that does the same job is written as follows:

```
PUTC MACRO char
     PUSH AX
     MOV  AL, char
     MOV  AH, 0Eh
     INT  10h
     POP  AX
ENDM

     PUTC 'A'  ; 8 bytes
     PUTC 'B'  ; 8 bytes
```

A macro is used just by writing its name; there is no CALL. The parameter is written after the macro name. Because the assembler rewrites the body at every use, two uses of PUTC produce two separate 8-byte copies in the program (Figure 9.1.5). The bytes of the body are: `PUSH AX` 1 byte (`50`), `MOV AL, 'A'` 2 bytes (`B0 41`), `MOV AH, 0Eh` 2 bytes (`B4 0E`), `INT 10h` 2 bytes (`CD 10`), `POP AX` 1 byte (`58`). The total is 8 bytes. Note that the macro contains no RET; a macro is not a piece of code that is called but one that is written in place.

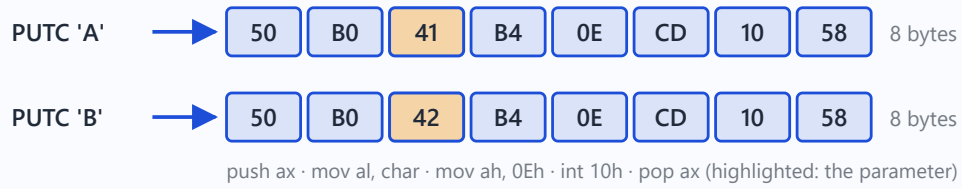


Figure 9.1.5: A macro is expanded in place at every use. Two PUTC uses produce two separate 8-byte copies one after the other; the stack is not used.

There is also an important difference with labels. The body of a procedure is a single copy, so the labels inside it are defined once. The body of a macro is rewritten at every use. If the body contains a label and the macro is used twice, the same label is defined twice and the assembler reports a *Duplicate declaration* error. The LOCAL directive, which solves this problem, is treated in its own section.

Criterion	Macro	Procedure
When is it processed?	at assembly time (text substitution)	at run time (CALL/RET)

Criterion	Macro	Procedure
>Use	`PUTC 'A'`	`CALL PRINT_A`
Code size	grows by the body at every use	body once + 3 bytes per call
Speed	no CALL/RET and no stack access	CALL + RET time at every call
Parameters	text after the name	register, memory or stack
Stack	not used (unless the body uses it)	used for the return address
Where defined	before use	anywhere
End	`ENDM`	`name ENDP`
Labels	LOCAL needed	no problem (single copy)

Table 9.1.6: Comparison of a macro and a procedure.

Table 9.1.6 puts the two methods side by side on nine criteria. The first row is the reason for all the others. Because a macro is processed at assembly time, it leaves no cost at run time, but it takes space at every use. Because a procedure is processed at run time, it saves space, but it spends time at every call.

9.1.3.3 Size calculation and the break-even point

Two simple formulas suffice to compute code size as a function of the number of uses. Let a job whose body is n bytes be used in k places. In a macro the body is written k times: the size is k · n bytes. In a procedure the body is written once, a 1-byte RET is added at the end, and every use is a 3-byte CALL: the size is (n + 1) + 3k bytes.

$$\text{macro} = k \cdot n \qquad \text{procedure} = (n + 1) + 3k$$

The body of PUTC is n = 8 bytes. For a single use the macro takes 8 bytes and the procedure 8 + 1 + 3 = 12 bytes; the macro is smaller. For ten uses the macro grows to 80 bytes and the procedure to 9 + 30 = 39 bytes. For a hundred uses the gap widens: the macro is 800 bytes, the procedure 9 + 300 = 309 bytes. Figure 9.1.6 shows these three cases with bars. The macro's bar grows in direct proportion to the number of uses; the procedure's bar adds only 3 bytes per use on top of a fixed 9-byte block.

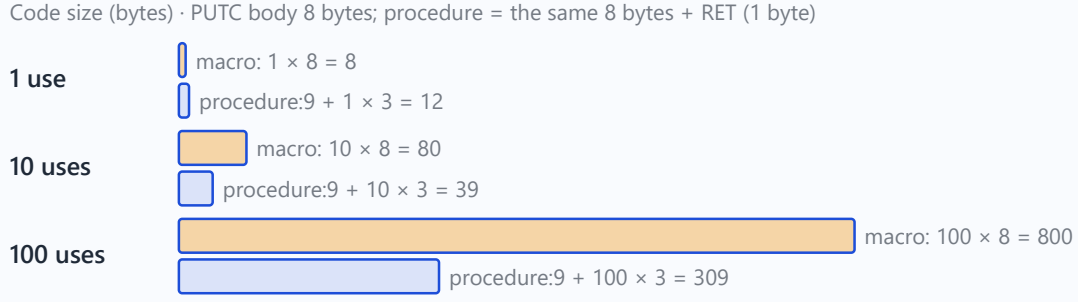


Figure 9.1.6: Code size for the PUTC body with 1, 10 and 100 uses: macro (8k bytes) versus procedure (9 +

3k bytes). The gap grows with the number of uses.

The number of uses at which a macro starts to be larger than a procedure can be called the break-even point. For the procedure to be smaller we need $k \cdot n > (n + 1) + 3k$. Collecting terms gives $k \cdot (n - 3) > n + 1$. Two conclusions follow from this inequality.

- If $n \le 3$, the left side is zero or negative and the inequality holds for no k. If the body is as short as a CALL, the macro is always smaller. In that case a procedure does not reduce code size; it only adds RET and call time.
- If $n > 3$, the break-even point is $k > (n + 1) / (n - 3)$. For $n = 8$, $k > 9/5 = 1.8$, so from two uses on the procedure is smaller: 15 bytes against 16. For $n = 5$, $k > 3$, that is, from four uses on. For $n = 4$, $k > 5$; at five uses both take 20 bytes, and at six uses the procedure pulls ahead with 23 bytes against 24.

Example 9.1.2

The PUTC macro is used 50 times. What would the code sizes be if the same job were done with a procedure?

Macro: 50 × 8 = 400 bytes. Procedure: the body and RET once (8 + 1 = 9 bytes), the CALLs 50 × 3 = 150 bytes; 159 bytes in all. The procedure is 241 bytes smaller. A frequent arithmetic mistake is to count only the CALLs (150 bytes) for the procedure, or to count the macro copy as 9 bytes with a RET. A macro has no RET.

Why does this calculation matter? The code, data and stack of a .COM program must all fit in a single 64 KB segment. In an embedded system the program is written to a small ROM and every byte is a cost. Using a macro with a long body in hundreds of places quickly strains these limits.

9.1.3.4 Speed: CALL, RET and the prefetch queue

On the speed side the table turns around. Every call of a procedure executes a CALL and a RET in addition to the job itself. According to Intel's 8086 data sheet, a near direct CALL takes 19 clock cycles and a near RET takes 8. Each call thus adds 27 cycles. On a 4.77 MHz 8086 one cycle is about 209.6 ns; 27 cycles come to about 5.66 µs. At 5 MHz one cycle is 200 ns and 27 cycles are 5.4 µs.

Behind this number lies what happens in the hardware. The CALL instruction writes the return address to the stack; this is a memory write bus cycle. A bus cycle lasts at least four clock cycles (T1–T4). RET reads a word from the stack, which is a read bus cycle. More importantly, both instructions change the flow of the program. In the 8086, while the execution unit (EU) executes one instruction, the bus interface unit (BIU) prefetches the following instruction bytes. These bytes wait in the six-byte prefetch queue. When the flow changes with a jump, the bytes in the queue become useless. The BIU flushes the queue and starts fetching again from the new address. The EU waits until the first instruction byte arrives. This is one reason why CALL and RET take so long.

In the expansion of a macro, by contrast, the flow is a straight line. The instructions sit one after another in memory, the BIU fetches them in order, and the queue is never flushed. No return address is written to or read from the stack. The body of PUTC takes about 78 cycles by itself: PUSH 11, two MOVs 4 each, INT 51, POP 8. To this are added the BIOS routine itself and its IRET. In the procedure version, 27 more cycles are added to the same job at every call.

Instruction	Bytes	Clock cycles	Memory/stack access
`PUSH AX`	1	11	1 write (stack)
`MOV AL, 'A'`	2	4	none
`MOV AH, 0Eh`	2	4	none
`INT 10h`	2	51	3 writes (FLAGS, CS, IP) + vector read
`POP AX`	1	8	1 read (stack)
macro copy, total	**8**	**78**	
`CALL` (near, direct)	3	19	1 write (return address)
`RET` (near)	1	8	1 read (return address)
procedure call, total	**3 (+ 9 once)**	**105**	

Table 9.1.7: Duration of the PUTC body and of the CALL/RET overhead in clock cycles (8086 data sheet; the BIOS routine's own time excluded).

Table 9.1.7 compares the two versions in bytes and cycles. The extra time per call is small but it accumulates. A procedure called 10,000 times in a loop adds 10,000 × 27 = 270,000 cycles, about 56.6 ms at 4.77 MHz. For a short job this overhead can be even longer than the job itself. For example, if a two-instruction job taking 6–8 cycles in all is turned into a procedure, every call makes it four or five times as slow.

9.1.3.5 Selection criteria

Weighing size and speed together gives the general rule. Short code that runs often and needs speed suits a macro; long code used in many places suits a procedure. Five typical cases make the rule concrete:

1. **Printing a single character, in 200 places.** The body is short (8 bytes), but 200 copies come to 1600 bytes. The procedure takes 9 + 600 = 609 bytes. If size matters, choose the procedure; if speed matters, the macro. Since printing already involves a BIOS call, the extra 27 cycles are usually negligible; here the procedure is the wiser choice.
2. **A one-line job that swaps two registers.** `XCHG` is already a single instruction. A macro may improve readability; a procedure would add a burden of 27 cycles and 4 bytes to a one-instruction job. Choose the macro.
3. **A 300-byte sorting algorithm, in 3 places.** The macro takes 900 bytes, the procedure about 300 + 9 bytes. Next to the running time of a sort, 27 cycles are negligible. Clearly a procedure.

4. **A two-instruction job done on every iteration of a tight loop.** The CALL/RET time exceeds the job itself. When the loop runs thousands of times, the difference becomes measurable. A macro.
5. **Putting parameters in registers and calling shared code.** The two methods are used together: the macro places the parameters in registers and calls the procedure. Use is as convenient as a macro and the size as small as a procedure. The emu8086.inc library also uses this pattern.

9.1.3.6 Code trace: the macro prepares, the procedure works

Let us trace the fifth case with a small program. The PUT_DIGIT macro puts a digit in AL and calls the print_digit procedure. The procedure converts the digit to its ASCII code and writes it to the screen:

```
PUT_DIGIT MACRO d
    mov  al, d
    call print_digit
ENDM

org 100h
    PUT_DIGIT 4
    PUT_DIGIT 7
    ret
print_digit:
    add  al, '0'     ; digit → ASCII
    mov  ah, 0Eh
    int  10h
    ret
```

Each macro copy is only 5 bytes: a 2-byte `MOV AL, imm8` and a 3-byte `CALL`. The first copy occupies 0100h–0104h and the second 0105h–0109h. The main program's RET is at 010Ah and the procedure starts at 010Bh. The body of the procedure is 2 + 2 + 2 + 1 = 7 bytes. If the macro were used in ten places the code would take 10 × 5 + 7 = 57 bytes. Had the whole body (`MOV AL`, `ADD AL`, `MOV AH`, `INT`) been in the macro, every copy would be 8 bytes and ten copies 80 bytes.

Step	Address	Instruction	AL	AH	SP	[SS:SP]	Next IP	Screen	Explanation
1	0100h	`mov al, 4` PUT_DIGIT expansion	04h	00h	FFFEh	0000h	0102h	–	1st expansion: AL ← 4
2	0102h	`call print_digit` PUT_DIGIT expansion	04h	00h	FFFCh	0105h	010Bh	–	return address 0105h to the stack; IP ← 010Bh
3	010Bh	`add al, '0'` print_digit procedure	34h	00h	FFFCh	0105h	010Dh	–	4 + 30h = 34h ('4')
4	010Dh	`mov ah, 0Eh` print_digit procedure	34h	0Eh	FFFCh	0105h	010Fh	–	AH = 0Eh: teletype output
5	010Fh	`int 10h` print_digit procedure	34h	0Eh	FFFCh	0105h	0111h	`4`	4 to the screen
6	0111h	`ret` print_digit procedure	34h	0Eh	FFFEh	0000h	0105h	`4`	0105h from the stack to IP

Step	Address	Instruction	AL	AH	SP	[SS:SP]	Next IP	Screen	Explanation
7	0105h	`mov al, 7` PUT_DIGIT expansion	07h	0Eh	FFFEh	0000h	0107h	`4`	2nd expansion: AL ← 7
8	0107h	`call print_digit` PUT_DIGIT expansion	07h	0Eh	FFFCh	010Ah	010Bh	`4`	this time the return address is 010Ah
9	010Bh	`add al, '0'` print_digit procedure	37h	0Eh	FFFCh	010Ah	010Dh	`4`	37h ('7')
10	010Dh	`mov ah, 0Eh` print_digit procedure	37h	0Eh	FFFCh	010Ah	010Fh	`4`	
11	010Fh	`int 10h` print_digit procedure	37h	0Eh	FFFCh	010Ah	0111h	`47`	47 on the screen
12	0111h	`ret` print_digit procedure	37h	0Eh	FFFEh	0000h	010Ah	`47`	return to 010Ah
13	010Ah	`ret` main program	37h	0Eh	0000h	–	0000h	`47`	the main program's RET

Table 9.1.8: Joint trace of the PUT_DIGIT macro and the print_digit procedure. The [SS:SP] column shows the word at the top of the stack.

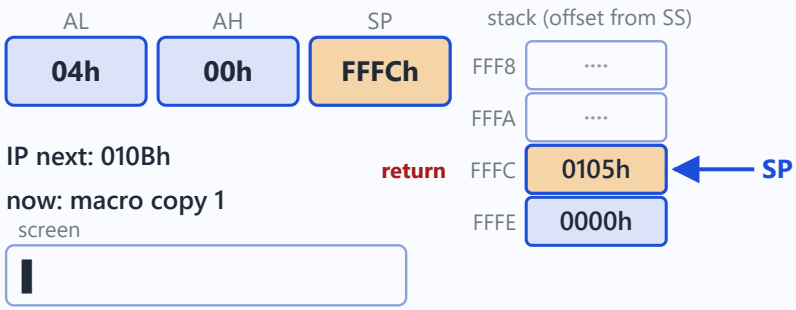


Figure 9.1.7: Right after the first CALL: the return address 0105h has been written to the stack, SP has become FFFCh, and IP has reached the start of the procedure, 010Bh.

Table 9.1.8 and Figure 9.1.7 should be read together. In the second step the CALL executes. The return address is 0105h, the address of the instruction after the CALL. This word is written to SS:FFFCh; since SS = 0700h, the physical address is 07000h + FFFCh = 16FFCh. SP becomes FFFCh and IP jumps to 010Bh. The procedure adds 30h to 4, obtaining the ASCII '4' (34h), and writes it with INT 10h. The RET in the sixth step takes 0105h from the top of the stack into IP and restores SP to FFFEh.

The second copy calls the same procedure, but this time the return address is 010Ah. The same single body has been called from two different places and has returned to the right place each time. 47 appears on the screen. The last RET belongs to the main program; it takes the 0000h at the top of the stack and ends the program. In this arrangement the macro acts as a thin wrapper that prepares the parameter; the work itself is in the procedure, in a single copy.

9.1.3.7 Code trace: RET inside a macro

Writing RET at the end of a macro out of procedure habit is one of the most instructive mistakes with macros. The EMIT macro below writes a character and ends with RET:

```
EMIT MACRO c
    MOV AL, c
    MOV AH, 0Eh
    INT 10h
    RET             ; wrong
ENDM

org 100h
    EMIT 'A'
    EMIT 'B'
    ret
```

The program assembles; to the assembler, RET is an ordinary instruction. The expanded code contains two copies of EMIT, each ending in a RET. The programmer expects AB on the screen.

Step	Address	Instruction	AX	SP	Next IP	Screen	Explanation
1	0100h	`MOV AL, 'A'` EMIT expansion	0041h	FFFEh	0102h	–	1st expansion: AL ← 'A'
2	0102h	`MOV AH, 0Eh` EMIT expansion	0E41h	FFFEh	0104h	–	AX = 0E41h
3	0104h	`INT 10h` EMIT expansion	0E41h	FFFEh	0106h	`A`	A to the screen
4	0106h	`RET` EMIT expansion	0E41h	0000h	0000h	`A`	the macro's RET takes the 0000h at the top of the stack
5	0000h	`INT 20h` (`PSP:0000h`) PSP	0E41h	0000h	–	`A`	the INT 20h in the PSP ends the program; nothing after 0107h ever runs

Table 9.1.9: Trace of a program that uses the EMIT macro, whose body contains a RET, twice.

Table 9.1.9 shows that what was expected does not happen. The first copy writes 'A'. The RET in the fourth step is the macro's RET. Since the macro was not entered with a CALL, there is no return address on the stack. RET nevertheless takes whatever it finds at the top of the stack into IP. In this .COM program the top holds the word 0000h that DOS placed there. IP becomes 0000h and the processor goes to the `INT 20h` instruction at the start of the PSP. The program ends. The copy of EMIT 'B' and the main program's RET never run; only A is on the screen.

This program ends luckily: the top of the stack happens to hold an address that terminates the program. Had the macro been used inside a procedure or after something was pushed on the stack, RET would have jumped to an arbitrary word on the stack and the program could have hung. RET makes sense only in a procedure entered with CALL.

Common mistake

Writing RET at the end of a macro body. A macro is not called with CALL; the RET in the body takes the return address of the code that uses the macro (or whatever is on the stack). A macro "ends" by itself on reaching ENDM: after the last instruction of the expansion, the instruction that follows the call line executes.

9.1.3.8 EQU, macros and procedures

Another tool of the assembler that works on text is the EQU directive. When `WIDTH EQU 80` is written, the assembler writes 80 wherever it sees the name WIDTH. This is like a one-line macro without parameters. The three tools are compared in Table 9.1.10.

Criterion	EQU	Macro	Procedure
What is substituted?	a single value or expression	one or more lines	nothing; control goes there with CALL
Parameters	none	yes (text)	register, stack or memory
When?	assembly	assembly	run time
In memory	takes no space	the body at every use	single copy + 3 bytes per call
Example	`WIDTH EQU 80`	`PUTC 'A'`	`CALL PRINT_NUM`

Table 9.1.10: EQU, macro and procedure.

What EQU and a macro have in common is that both are text operations the assembler performs on the source. The processor never sees either; it executes only the instructions they produce. An EQU constant by itself does not even produce a byte; only when it is used inside an instruction does it become that instruction's immediate value. The rule of choice is simple. Use EQU to name a single number, a macro for a short pattern of a few instructions, and a procedure for a long or widely shared job.

9.1.4 Labels and LOCAL

9.1.4.1 The same label twice

The body of every macro that contains a loop or a conditional jump contains a label. A label is a name that the assembler gives to an address. In a program each label may be defined only once; otherwise the same name would point to two different addresses and the target of a jump would become ambiguous. Because the body of a macro is rewritten at every use, this rule is easily broken in macros. A simple delay loop shows this:

```
DELAY MACRO
lp:  DEC CX
      JNZ lp
ENDM

MOV CX, 5
DELAY
MOV CX, 9
DELAY
```

With a single use of the macro there would be no problem. After two uses, the code the assembler sees is this:

```
MOV CX, 5
lp:  DEC CX    ; copy 1
      JNZ lp
MOV CX, 9
lp:  DEC CX    ; copy 2 !
      JNZ lp
```

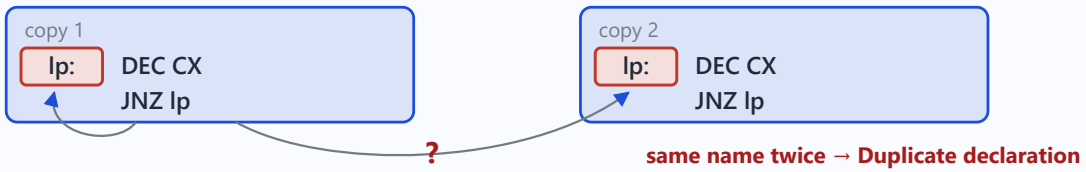


Figure 9.1.8: The same label `lp` in two expansions. `JNZ lp` cannot tell which of the two addresses to jump to; the assembler reports a *Duplicate declaration* error.

When the assembler reaches the second `lp:` line, it already finds this name in the symbol table. emu8086 reports a *Duplicate declaration* error and the program does not assemble. The error message points to the second DELAY line; when the programmer looks at that line, all that is visible is the macro's name. To understand the cause of the error one again has to think of the expansion.

9.1.4.2 The LOCAL directive

The solution is to give the label a different name in each expansion. The programmer cannot do this by hand, because the body is written in one place. The assembler takes over this job with the LOCAL directive.

Definition 9.1.3

The **LOCAL directive** is written on the first line of a macro body and makes the names that follow it local to the macro. At every expansion the assembler replaces these names with a unique name that has not been used so far. MASM and emu8086 generate these names as `??0000`, `??0001`, `??0002` ... with a counter that keeps increasing across all expansions in the program.

```
MyMacro2 MACRO
    LOCAL label1, label2
    CMP  AX, 2
    JE   label1
    CMP  AX, 3
    JE   label2
label1:
    INC  AX
label2:
    ADD  AX, 2
ENDM
```

MyMacro2 declares two local labels. At the first use label1 becomes `??0000` and label2 becomes `??0001`. At the second use the counter continues where it left off: label1 becomes `??0002` and label2 `??0003`. The counter belongs to the program, not to the macro; the local labels of another macro take numbers from the same sequence. Names starting with `??` are chosen deliberately: there is no chance of a clash with the programmer's own labels.

LOCAL must be the first line of the body; it must come right after the MACRO line. The assembler wants to know the local names before applying them to the rest of the body. LOCAL is used not only for labels but also for variable and procedure names defined inside the macro. If a macro defines a string inside itself (as in the PRINT macro), the name of the string must be local too.

The listing of a program with two uses shows the naming and the addresses together:

```
0100  3D 02 00   CMP AX, 2     ; 1st MyMacro2
0103  74 05      JE  ??0000
0105  3D 03 00   CMP AX, 3
0108  74 01      JE  ??0001
010A  40         ??0000: INC AX
010B  05 02 00   ??0001: ADD AX, 2
010E  3D 02 00   CMP AX, 2     ; 2nd MyMacro2
0111  74 05      JE  ??0002
0113  3D 03 00   CMP AX, 3
0116  74 01      JE  ??0003
0118  40         ??0002: INC AX
0119  05 02 00   ??0003: ADD AX, 2
011C  C3         RET
```

The machine code of the conditional jumps is instructive too. `JE ??0000` = `74 05`: 74h is the opcode and 05h is the relative displacement. The jump target is found by adding 5 to the address of the instruction after the JE (0105h): 0105h + 5 = 010Ah. The `JE ??0002` in the second copy is also `74 05`: 0113h + 5 = 0118h. The machine code of the two copies is identical byte for byte, because the 8086's short jumps store the displacement, not the address. What differs is the names of the labels and the absolute addresses they point to.

9.1.4.3 Code trace: MyMacro2

What MyMacro2 does depends on the value of AX. If AX = 2 it jumps to label1, INC and ADD run, and AX becomes 2 + 1 + 2 = 5. If AX = 3 it jumps to label2, only ADD runs, and again AX becomes 5. For any other value neither jump is taken and the flow falls through to label1; AX increases by 3. The trace of the program that starts with `MOV AX, 2` is in Table 9.1.11.

```
ORG 100h
MOV  AX, 2       ; before MyMacro2
CMP  AX, 2       ; MyMacro2 starts
JE   ??0000
CMP  AX, 3
JE   ??0001
??0000:
INC  AX
??0001:
ADD  AX, 2       ; MyMacro2 ends
RET
```

Step	Address	Instruction	AX	ZF	SF	CF	Next IP	Explanation
1	0100h	`MOV AX, 2` main program	0002h	0	0	0	0103h	AX = 2
2	0103h	`CMP AX, 2` MyMacro2 expansion	0002h	1	0	0	0106h	2 − 2 = 0: ZF = 1

Step	Address	Instruction	AX	ZF	SF	CF	Next IP	Explanation
3	0106h	`JE ??0000` MyMacro2 expansion	0002h	1	0	0	010Dh	ZF = 1: jump to ??0000 (label1)
4	010Dh	`INC AX` MyMacro2 expansion	0003h	0	0	0	010Eh	AX = 3
5	010Eh	`ADD AX, 2` MyMacro2 expansion	0005h	0	0	0	0111h	AX = 5
6	0111h	`RET` main program	0005h	0	0	0	0000h	the program ends

Table 9.1.11: Trace of MyMacro2 with AX = 2; the LOCAL labels became ??0000 (010Dh) and ??0001 (010Eh).

In the second step `CMP AX, 2` subtracts 2 from AX but writes the result nowhere; it only sets the flags. Since 2 − 2 = 0, ZF = 1. The `JE` in the third step looks at ZF and jumps to ??0000, that is, to 010Dh. The Next IP column shows this jump: 010Dh instead of 0108h. `CMP AX, 3` and the second `JE` are never executed. INC AX gives 3 and ADD AX, 2 gives 5. Had AX been 3 at the start of the program, the first JE would not jump, the second JE would jump to label2, and the result would again be 5. Had AX been 7, neither jump would be taken, INC and ADD would run and AX would be 10. A program that starts with AX = 0 would end with 3.

9.1.4.4 Code trace: the two copies of DELAY

When `LOCAL lp` is added to the DELAY macro, the error disappears:

```
DELAY MACRO
      LOCAL ilk
ilk:  DEC CX
      JNZ ilk
ENDM

org 100h
    mov cx, 3
    DELAY
    mov cx, 2
    DELAY
    ret
```

In the first copy the label lp gets the name `??0000` and is at 0103h. In the second copy it becomes `??0001` and is at 0109h. Each copy loops back to its own label.

Step	Address	Instruction	CX	ZF	Next IP	Explanation
1	0100h	`mov cx, 3` main program	0003h	0	0103h	counter of copy 1
2	0103h	`??0000: DEC CX` DELAY expansion	0002h	0	0104h	1st pass: CX = 2
3	0104h	`JNZ ??0000` DELAY expansion	0002h	0	0103h	ZF = 0: loop back to ??0000 (0103h)

Step	Address	Instruction	CX	ZF	Next IP	Explanation
4	0103h	`??0000: DEC CX` DELAY expansion	0001h	0	0104h	2nd pass
5	0104h	`JNZ ??0000` DELAY expansion	0001h	0	0103h	0103h again
6	0103h	`??0000: DEC CX` DELAY expansion	0000h	1	0104h	3rd pass: CX = 0, ZF = 1
7	0104h	`JNZ ??0000` DELAY expansion	0000h	1	0106h	no jump; copy 1 is done
8	0106h	`mov cx, 2` main program	0002h	1	0109h	counter of copy 2
9	0109h	`??0001: DEC CX` DELAY expansion	0001h	0	010Ah	1st pass: CX = 1
10	010Ah	`JNZ ??0001` DELAY expansion	0001h	0	0109h	loop back to ??0001 (0109h), not to 0103h
11	0109h	`??0001: DEC CX` DELAY expansion	0000h	1	010Ah	CX = 0, ZF = 1
12	010Ah	`JNZ ??0001` DELAY expansion	0000h	1	010Ch	no jump
13	010Ch	`ret` main program	0000h	1	0000h	the program ends; 5 DECs in all

Table 9.1.12: Trace of two copies of the DELAY macro with the label declared LOCAL lp.

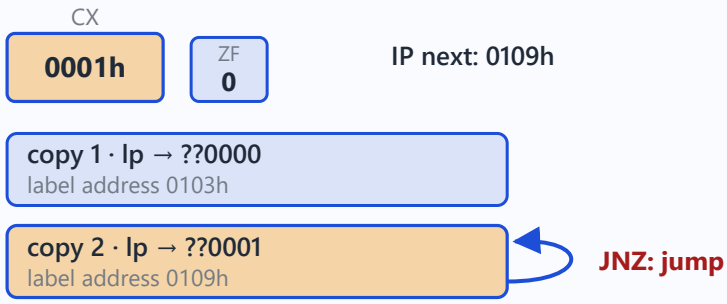


Figure 9.1.9: The first pass of the second copy: `JNZ ??0001` loops back to 0109h, not to the label at 0103h of the first copy.

In Table 9.1.12 the first copy loops three times. DEC CX decreases CX by one on every pass and sets ZF according to the result. As long as CX stays nonzero, ZF = 0 and JNZ jumps back to 0103h. On the third pass CX becomes 0, ZF becomes 1 and JNZ does not jump. The second copy runs two passes with CX = 2 and each time loops back to 0109h (Figure 9.1.9). Five DECs are executed in all. Recall also that DEC does not touch CF; it changes only ZF, SF, OF, AF and PF.

The delay time can be computed in clock cycles. `DEC r16` takes 2 cycles. `JNZ` takes 16 cycles when it jumps and 4 when it does not; because the prefetch queue is flushed on a jump, the difference is large. With CX = 3 the first copy takes 2 × (2 + 16) + (2 + 4) = 42 cycles. Such a loop produces a crude delay that depends on the processor clock.

9.1.4.5 Code trace: MAX2 and signed comparison

The last example combines LOCAL with a signed comparison. The MAX2 macro writes the larger of p and q to r:

```
MAX2 MACRO p, q, r
    LOCAL ok
    MOV AX, p
    CMP AX, q
    JGE ok
    MOV AX, q
ok: MOV r, AX
ENDM

org 100h
    MAX2 x, y, m1
    MAX2 y, 4, m2
    ret
x  dw 7
y  dw -3
m1 dw ?
m2 dw ?
```

In the first call all three parameters are variables. In the second q is a constant, so `CMP AX, 4` and `MOV AX, 4` use the immediate form. y = −3 is stored as FFFDh in two's complement.

Step	Address	Instruction	AX	SF	OF	CF	ZF	m1	m2	Explanation
1	0100h	`MOV AX, x` MAX2 expansion	0007h	0	0	0	0	0000h	0000h	p = x: AX = 7
2	0103h	`CMP AX, y` MAX2 expansion	0007h	0	0	1	0	0000h	0000h	7 − (−3) = 10: SF = 0, OF = 0
3	0107h	`JGE ??0000` MAX2 expansion	0007h	0	0	1	0	0000h	0000h	SF = OF: jump to ??0000
4	010Ch	`??0000: MOV m1, AX` MAX2 expansion	0007h	0	0	1	0	0007h	0000h	m1 = 7
5	010Fh	`MOV AX, y` MAX2 expansion	FFFDh	0	0	1	0	0007h	0000h	p = y: AX = FFFDh (−3)
6	0112h	`CMP AX, 4` MAX2 expansion	FFFDh	1	0	0	0	0007h	0000h	−3 − 4 = −7: SF = 1, OF = 0
7	0115h	`JGE ??0001` MAX2 expansion	FFFDh	1	0	0	0	0007h	0000h	SF ≠ OF: no jump
8	0117h	`MOV AX, 4` MAX2 expansion	0004h	1	0	0	0	0007h	0000h	q = 4 is the larger one
9	011Ah	`??0001: MOV m2, AX` MAX2 expansion	0004h	1	0	0	0	0007h	0004h	m2 = 4
10	011Dh	`ret` main program	0004h	1	0	0	0	0007h	0004h	the program ends

Table 9.1.13: Trace of the expansions of `MAX2 x, y, m1` and `MAX2 y, 4, m2` (x = 7, y = −3).

The `CMP AX, y` of the second step performs 7 − (−3). Under the signed interpretation the result is 10: positive (SF = 0) with no overflow (OF = 0). `JGE` (jump if greater or equal) looks at the result of the signed comparison with the condition SF = OF. Since the condition holds, control jumps to ??0000 and m1 = 7. Under the unsigned interpretation the same subtraction is 0007h − FFFDh; because a larger number is subtracted from a smaller one, a borrow occurs and CF = 1. Since JGE does not look at CF, this flag does not matter here.

In the second expansion AX = FFFDh (−3) and `CMP AX, 4` gives −3 − 4 = −7: SF = 1, OF = 0. Since SF ≠ OF, JGE does not jump. The flow continues with `MOV AX, 4` and m2 = 4. The label of the second copy is ??0001; it does not clash with the first one's.

The choice of conditional jump matters in this example. Had `JAE`, which compares as unsigned, been used instead of JGE, the second comparison would have treated FFFDh and 4 as unsigned numbers. FFFDh = 65,533 is greater than 4; there is no borrow (CF = 0) and JAE jumps. m2 would receive FFFDh, that is −3, instead of 4. Even though the macro stays the same, a single conditional jump instruction decides whether the result is right. With signed numbers use JG, JGE, JL, JLE; with unsigned numbers use JA, JAE, JB, JBE.

Common mistake

Forgetting LOCAL because a macro with a label worked correctly when tried once. The error appears only when the macro is used a second time, and then the error message points not to the body but to the line of use. Every macro that has a label gets a LOCAL, no matter how many times it will be used.

9.1.5 PUTC and preserving registers

9.1.5.1 The PUTC macro

PUTC is a short macro that writes a single character to the screen. It is identical to the PUTC macro of the emu8086.inc library:

```
; prints a character, advances the cursor
PUTC  MACRO  char
      push   ax
      mov    al, char
      mov    ah, 0eh
      int    10h
      pop    ax
ENDM

ORG  100h
START:
      PUTC  'A'
      PUTC  'B'
      RET
```

The real work of the macro is done by `INT 10h`. AH = 0Eh is the BIOS teletype output service. The character in AL is written where the cursor is and the cursor advances one position to the right. At the end of a line the cursor moves to the next line; at the end of the screen the screen scrolls by one line. Control characters such as 0Dh (carriage return), 0Ah (line feed), 07h (bell) and 08h (backspace) are not printed but interpreted.

Because the service uses AH and AL, the macro has to modify AX. The `push ax` at the start of the macro and the `pop ax` at the end hide this change from the caller. On leaving the macro, AX returns to the value it had before the macro was entered. The macro has no visible side effect. This principle holds for all library macros: a macro may use registers for its own work but restores them on exit.

Definition 9.1.4

Register preservation means that a macro or procedure saves the registers it modifies with PUSH at the start and restores them with POP in reverse order at the end. The values in the caller's registers are thus not corrupted by the use; only the registers explicitly designated for returning a result change.

The RET at the end of the program matters too. After the two PUTC copies the program's code ends in memory. Without the RET the processor would take the bytes after the last POP AX for instructions and keep executing. The processor does not distinguish code from data; it decodes every byte that IP points to as an opcode. Whatever is left in memory would run as random instructions and the system could hang. Macro expansion does not add this RET by itself.

9.1.5.2 Code trace: PUTC 'A' and PUTC 'B'

The expanded code is two 8-byte copies and a RET, 17 bytes in all. Because emu8086 writes the program size into CX at load time, CX = 0011h at the start of the trace. This value does not change during the program, because PUTC does not touch CX. AX is initially 0000h. The machine code of each

copy is the same except for the character byte: `50 B0 41 B4 0E CD 10 58` and `50 B0 42 B4 0E CD 10 58`.

Step	Address	Instruction	AX	CX	SP	[SS:SP]	Screen	Explanation
1	0100h	`push ax` PUTC expansion	0000h	0011h	FFFCh	0000h	–	the caller's AX (0000h) to the stack; SP = FFFCh
2	0101h	`mov al, 'A'` PUTC expansion	0041h	0011h	FFFCh	0000h	–	AL ← 41h
3	0103h	`mov ah, 0eh` PUTC expansion	0E41h	0011h	FFFCh	0000h	–	AX = 0E41h
4	0105h	`int 10h` PUTC expansion	0E41h	0011h	FFFCh	0000h	`A`	A to the screen; cursor one position right
5	0107h	`pop ax` PUTC expansion	0000h	0011h	FFFEh	0000h	`A`	AX has its old value back; SP = FFFEh
6	0108h	`push ax` PUTC expansion	0000h	0011h	FFFCh	0000h	`A`	2nd copy: AX is saved again
7	0109h	`mov al, 'B'` PUTC expansion	0042h	0011h	FFFCh	0000h	`A`	AL ← 42h
8	010Bh	`mov ah, 0eh` PUTC expansion	0E42h	0011h	FFFCh	0000h	`A`	AX = 0E42h
9	010Dh	`int 10h` PUTC expansion	0E42h	0011h	FFFCh	0000h	`AB`	AB on the screen
10	010Fh	`pop ax` PUTC expansion	0000h	0011h	FFFEh	0000h	`AB`	AX = 0000h: the caller's AX was not corrupted
11	0110h	`RET` main program	0000h	0011h	0000h	–	`AB`	the program ends

Table 9.1.14: Trace of the expansions of `PUTC 'A'` and `PUTC 'B'`; at load time CX = 0011h (17 bytes).

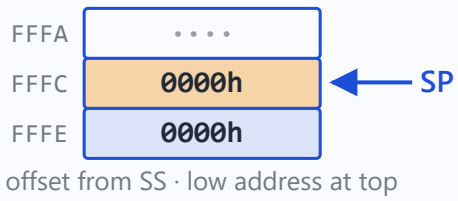


Figure 9.1.10: The stack after `push ax`: the caller's AX (0000h) has been saved at address FFFCh and SP = FFFCh.

In Table 9.1.14 the first step pushes AX on the stack. SP decreases by 2 to FFFCh and 0000h is written to the word at SS:FFFCh (Figure 9.1.10). The second and third steps load 'A' (41h) into AL and 0Eh into AH; AX becomes 0E41h. In the fourth step INT 10h writes A to the screen and advances the cursor. The INT instruction itself pushes FLAGS, CS and IP on the stack, goes to the BIOS routine and returns with IRET; this internal detail appears as a single step in the trace. The POP AX of the fifth step takes the 0000h from the stack into AX and restores SP to FFFEh. The second copy follows the same path with 'B', and AB appears on the screen. In the final state AX = 0000h: the macro has not corrupted the caller's AX.

In this trace the stack grows and shrinks by one word in each copy. As long as PUSH and POP are paired within the same macro, the stack is balanced: SP is the same on leaving the macro as on entering it. If this balance is broken, for example if a conditional jump in the body skips the POP, an extra word stays on the stack. The main program's RET takes this word for the return address and the program goes to a wrong place.

9.1.5.3 Register-saving macros

The `push ax` / `pop ax` pattern of PUTC grows to four registers in larger macros. This pattern can itself be a macro:

```
SAVE MACRO
  PUSH AX
  PUSH BX
  PUSH CX
  PUSH DX
ENDM

RESTORE MACRO
  POP DX
  POP CX
  POP BX
  POP AX
ENDM
```

RESTORE pops the registers in the reverse of the order in which SAVE pushed them. The stack works on the last-in, first-out (LIFO) principle: DX, the last one pushed, must be the first one popped. Suppose SAVE is used while AX = 1111h, BX = 2222h, CX = 3333h, DX = 4444h and the registers are then cleared. After RESTORE all four registers return to their old values. If the pops are written in the same order as the pushes (`POP AX`, `POP BX`, `POP CX`, `POP DX`), the result is AX = 4444h, BX = 3333h, CX = 2222h, DX = 1111h: AX and DX swap, and so do BX and CX. This error is not caught at assembly time, because every line is valid by itself.

Each SAVE and RESTORE pair adds 8 bytes; every PUSH and POP is a single byte. In time, on the 8086 `PUSH r16` takes 11 cycles and `POP r16` takes 8: 4 × 11 + 4 × 8 = 76 cycles. Within this time eight stack accesses, that is, eight bus cycles, are made. In a short macro this time can exceed that of the body itself; a macro should therefore save only the registers it really modifies.

The two macros must be used together in the same body. If there is a conditional exit in between (for instance a jump to the end of the body in an error case), this exit must also pass through RESTORE; otherwise the stack is left unbalanced. On the 80186 and later processors the single instructions `PUSHA` and `POPA` save and restore all the general-purpose registers. The 8086 does not have these instructions, which is why the same job is written as a macro.

9.1.5.4 Nested macros

The body of a macro may contain a call to another macro. PRINT2 below uses PUTC twice to print two characters:

```
PRINT2 MACRO c1, c2
       PUTC c1
       PUTC c2
ENDM

PRINT2 'O', 'K'
```

The assembler expands in two levels. It first expands the outer macro: the line `PRINT2 'O', 'K'` turns into the lines `PUTC 'O'` and `PUTC 'K'`. These two lines are macro calls as well, and the assembler expands them too. In the end the one-line call becomes ten instructions and 16 bytes (Figure 9.1.11). OK appears on the screen.

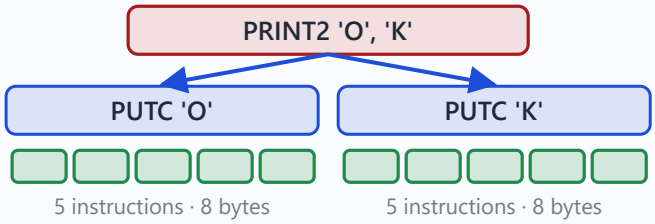


Figure 9.1.11: Nested expansion: PRINT2 expands into two PUTC calls, and each PUTC into five instructions.

With nesting, the order of definition matters again. PUTC must be defined before the line where PRINT2 is used. A macro calling itself (recursion) is a boundary case. In a procedure, recursion is limited by the stack and stopped by a condition. In a macro the stopping condition must be an assembly-time one, not a run-time one; otherwise the expansion never ends. Conditional assembly directives make such a stop possible.

9.1.6 The POWER macro

9.1.6.1 Exponentiation: MUL and LOOP

The POWER macro computes the power of a base: $r = b^e$. The method is the simplest one. The running result is initialized to 1 and multiplied by the base e times:

```
POWER macro b, e, r
    mov ax, 1        ; result so far = 1
    mov cx, e        ; exponent → CX
    mov bx, b        ; base → BX
powerloop:
    mul bx           ; DX:AX = AX * BX
    loop powerloop   ; repeat e times
    mov r, ax        ; store result
endm

ORG 100h
start:
    POWER base, exponent, result
    ret
base     dw 2     ; base
exponent dw 3     ; exponent
result   dw ?     ; result
```

The macro uses two important instructions. `MUL BX` is unsigned multiplication: it multiplies AX by BX and writes the 32-bit result to the register pair DX:AX. The high 16 bits go to DX and the low 16 bits to AX. MUL therefore always modifies DX, even though DX is not named as an operand. If the high half of the result is nonzero, CF and OF become 1; if it is zero, both are 0. SF, ZF, AF and PF are undefined after MUL.

`LOOP label` decrements CX by 1 and jumps to the label if CX is nonzero. It does not change the flags. CX is the loop counter here; the macro loads it with the exponent. On the 8086, LOOP takes 17 cycles when it jumps and 5 when it does not. A 16-bit MUL takes between 118 and 133 cycles depending on the operand values; it is one of the slowest instructions in the instruction set.

The size of the expansion can also be computed. `mov ax, 1` is 3 bytes, `mov cx, exponent` and `mov bx, base` are 4 bytes each with the general memory-operand form, `mul bx` is 2 bytes, `loop` is 2 bytes, and `mov result, ax` is 3 bytes with the short AX-specific form. The total is 18 bytes; with the RET, 19 bytes (13h). The variables start right after the code: base is at 0113h, exponent at 0115h and result at 0117h.

9.1.6.2 Code trace: POWER 2, 3

The expanded code is:

```
ORG 100h
mov ax, 1        ; POWER expansion
mov cx, exponent
mov bx, base
powerloop:
mul bx
loop powerloop
mov result, ax
ret
base     dw 2    ; 0113h
exponent dw 3    ; 0115h
result   dw ?    ; 0117h
```

Step	Address	Instruction	AX	BX	CX	DX	CF	result	Explanation
1	0100h	`mov ax, 1` POWER expansion	0001h	0000h	0000h	0000h	0	0000h	result so far is 1
2	0103h	`mov cx, exponent` POWER expansion	0001h	0000h	0003h	0000h	0	0000h	CX ← [0115h] = 3
3	0107h	`mov bx, base` POWER expansion	0001h	0002h	0003h	0000h	0	0000h	BX ← [0113h] = 2
4	010Bh	`mul bx` POWER expansion	0002h	0002h	0003h	0000h	0	0000h	1st pass: DX:AX = 1 × 2
5	010Dh	`loop powerloop` POWER expansion	0002h	0002h	0002h	0000h	0	0000h	CX = 2 ≠ 0: powerloop
6	010Bh	`mul bx` POWER expansion	0004h	0002h	0002h	0000h	0	0000h	2nd pass: 2 × 2
7	010Dh	`loop powerloop` POWER expansion	0004h	0002h	0001h	0000h	0	0000h	CX = 1
8	010Bh	`mul bx` POWER expansion	0008h	0002h	0001h	0000h	0	0000h	3rd pass: 4 × 2
9	010Dh	`loop powerloop` POWER expansion	0008h	0002h	0000h	0000h	0	0000h	CX = 0: leave the loop
10	010Fh	`mov result, ax` POWER expansion	0008h	0002h	0000h	0000h	0	0008h	[0117h] ← 0008h
11	0112h	`ret` main program	0008h	0002h	0000h	0000h	0	0008h	the program ends

Table 9.1.15: Trace of the expansion of `POWER base, exponent, result` (base = 0113h, exponent = 0115h, result = 0117h).

Table 9.1.15 shows the three passes of the loop. In the first pass MUL computes 1 × 2 = 2: DX:AX = 0000:0002h. Since the high half is zero, CF = 0. LOOP decreases CX from 3 to 2 and returns to powerloop. In the second pass AX = 4, in the third AX = 8. The third LOOP makes CX 0 and does not jump. `mov result, ax` writes 0008h to the word at 0117h; $2^3 = 8$.

The table shows that DX is 0000h after every MUL. This holds as long as the result fits in 16 bits. Had the result exceeded 16 bits, for example if 2^{16} were computed, after the last MUL DX = 0001h, AX = 0000h and CF = OF = 1. Because the macro writes only AX to result, result would be 0; the high part of the correct result stays in DX and is lost.

9.1.6.3 Four pitfalls

The POWER macro is small but hides four separate bugs. Each represents a kind of problem that is common when writing macros.

1. Exponent zero. In mathematics $b^0 = 1$; the macro starting with AX = 1 looks as if it would give this result. But MUL runs at least once, because the loop condition is checked at the end. Worse, LOOP first decrements CX and then compares with zero. With CX = 0 the decrement makes CX FFFFh; this is not zero and the loop continues. The loop ends only when CX becomes 0 again, that is, after 65,536 passes. For base = 2, after 2^{16} the low 16 bits of AX are zero and stay so; result = 0000h. For base = 1, result = 1. This is correct by accident; 65,536 needless MULs have still been executed. Since each MUL takes about 120 cycles, this wasted time exceeds one and a half seconds at 4.77 MHz.

2. The label. `powerloop` is not declared with LOCAL. When the macro is used a second time the label is defined twice and a *Duplicate declaration* error occurs.

3. Register corruption. The macro puts the result in AX; that is an expected change. But it loads BX with the base and CX with the counter, and because of MUL it also modifies DX. The caller's values in these three registers are lost.

4. Overflow. If the result exceeds 65,535, the high part stays in DX and is not written. The macro also does not report this to the caller through CF; the CF of the last MUL remains after the macro, but any later instruction can erase it.

The corrected version fixes the first three problems:

```
POWER MACRO b, e, r
      LOCAL p_loop, p_end
      PUSH BX
      PUSH CX
      PUSH DX
      MOV  AX, 1
      MOV  CX, e
      MOV  BX, b
      JCXZ p_end   ; e = 0 → 1
p_loop: MUL BX
      LOOP p_loop
p_end: MOV r, AX
      POP  DX
      POP  CX
      POP  BX
ENDM
```

`JCXZ` (jump if CX is zero) checks the counter before LOOP. If CX = 0 it skips the whole loop and the 1 in AX is written directly to the result. LOCAL makes the two labels unique in every expansion. PUSH and POP preserve BX, CX and DX; the macro now modifies only AX. The overflow problem remains in

this version too. To solve it, either the result must be written as 32 bits (DX:AX) or the overflow must be reported to the caller through CF.

> **Common mistake**
>
> Starting a LOOP-controlled loop when the counter may be zero. LOOP decrements first and checks afterwards; with CX = 0 it runs 65,536 times. Every loop whose counter comes from a variable or a parameter gets a `JCXZ` in front of it.

9.1.6.4 Code trace: the corrected POWER

The expansion of the corrected macro, `POWER base, expn, res`, is traced for base = 7 and expn = 2. To see the effect of register preservation, let the caller's registers hold BX = 0AAAh, CX = 0BBBh, DX = 0CCCh:

```
org 100h
push bx          ; POWER expansion
push cx
push dx
mov  ax, 1
mov  cx, expn
mov  bx, base
jcxz ??0001      ; expn = 0 → 1
??0000: mul bx
loop ??0000
??0001: mov res, ax
pop  dx
pop  cx
pop  bx
ret
base dw 7
expndw 2
res dw ?
```

The expansion is 26 bytes (up to 011Ah); the RET is at 011Ah, base at 011Bh, expn at 011Dh and res at 011Fh. The LOCAL names are ??0000 for p_loop and ??0001 for p_end.

Step	Address	Instruction	AX	BX	CX	DX	SP	res	Explanation
1	0100h	`PUSH BX` POWER expansion	0000h	0AAAh	0BBBh	0CCCh	FFFCh	0000h	BX saved
2	0101h	`PUSH CX` POWER expansion	0000h	0AAAh	0BBBh	0CCCh	FFFAh	0000h	CX saved
3	0102h	`PUSH DX` POWER expansion	0000h	0AAAh	0BBBh	0CCCh	FFF8h	0000h	DX saved; SP = FFF8h
4	0103h	`MOV AX, 1` POWER expansion	0001h	0AAAh	0BBBh	0CCCh	FFF8h	0000h	result so far is 1

Step	Address	Instruction	AX	BX	CX	DX	SP	res	Explanation
5	0106h	`MOV CX, expn` POWER expansion	0001h	0AAAh	0002h	0CCCh	FFF8h	0000h	CX ← expn = 2
6	010Ah	`MOV BX, base` POWER expansion	0001h	0007h	0002h	0CCCh	FFF8h	0000h	BX ← base = 7
7	010Eh	`JCXZ ??0001` POWER expansion	0001h	0007h	0002h	0CCCh	FFF8h	0000h	CX ≠ 0: no jump
8	0110h	`??0000: MUL BX` POWER expansion	0007h	0007h	0002h	0000h	FFF8h	0000h	1 × 7; MUL also writes DX
9	0112h	`LOOP ??0000` POWER expansion	0007h	0007h	0001h	0000h	FFF8h	0000h	CX = 1
10	0110h	`??0000: MUL BX` POWER expansion	0031h	0007h	0001h	0000h	FFF8h	0000h	7 × 7 = 49 = 0031h
11	0112h	`LOOP ??0000` POWER expansion	0031h	0007h	0000h	0000h	FFF8h	0000h	CX = 0: the loop ends
12	0114h	`??0001: MOV res, AX` POWER expansion	0031h	0007h	0000h	0000h	FFF8h	0031h	[011Fh] ← 0031h
13	0117h	`POP DX` POWER expansion	0031h	0007h	0000h	0CCCh	FFFAh	0031h	DX = 0CCCh
14	0118h	`POP CX` POWER expansion	0031h	0007h	0BBBh	0CCCh	FFFCh	0031h	CX = 0BBBh
15	0119h	`POP BX` POWER expansion	0031h	0AAAh	0BBBh	0CCCh	FFFEh	0031h	BX = 0AAAh; SP = FFFEh
16	011Ah	`ret` main program	0031h	0AAAh	0BBBh	0CCCh	0000h	0031h	the program ends

Table 9.1.16: Expansion of `POWER base, expn, res` of the corrected POWER macro (base = 7, expn = 2); the caller's registers are BX = 0AAAh, CX = 0BBBh, DX = 0CCCh.

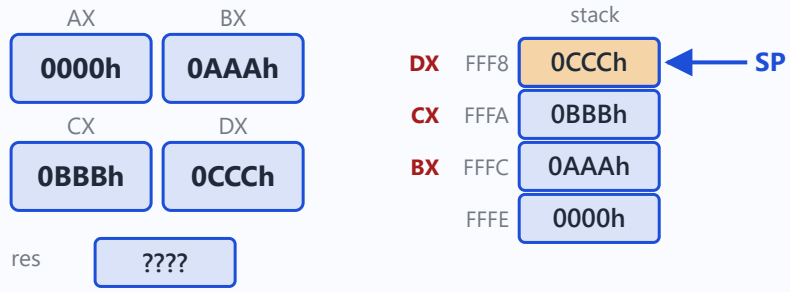


Figure 9.1.12: After three PUSHes: the caller's BX, CX and DX values are on the stack, SP = FFF8h.

In Table 9.1.16 the first three steps save the caller's registers on the stack (Figure 9.1.12). The stack grows downward: 0AAAh is written to FFFCh, 0BBBh to FFFAh, 0CCCh to FFF8h. In the next three steps the registers pass into the macro's use: AX = 1, CX = 2, BX = 7. JCXZ does not jump, since CX ≠ 0. The first MUL gives 1 × 7 = 7 and makes DX 0000h. The second MUL gives 7 × 7 = 49 = 0031h. When LOOP brings CX down to 0 the loop ends and 0031h is written to res.

The last three steps unwind the stack in reverse order. POP DX restores 0CCCh, POP CX restores 0BBBh, POP BX restores 0AAAh; SP is FFFEh again. When the macro is left, the values of BX, CX and DX are the same as on entry. Only AX has changed, and AX carries the result anyway. Had it been run with expn = 0, JCXZ would have jumped straight to ??0001, res = 1, and not a single MUL would have run.

9.1.7 Macro libraries

9.1.7.1 INCLUDE and emu8086.inc

Frequently used macros are not rewritten in every program; they are collected in a separate file. Such a file is called a macro library. The `INCLUDE` directive inserts the text of the library file into the source at assembly time, at the place of the directive. To the assembler this is no different from typing the macros at the start of the program by hand. Because unused macros generate no code, including a large library does not enlarge the program; only the macros that are used are expanded.

```
include 'emu8086.inc'
ORG 100h
    GOTOXY 10, 5
    PRINT 'Hello'
    PUTC  '!'
    RET
```

The emu8086.inc file that comes with emu8086 is a ready-made example of this arrangement. It contains macros that write characters and text to the screen (PUTC, PRINT, PRINTN), move the cursor (GOTOXY) and hide it (CURSOROFF). For longer jobs such as reading and printing numbers, the library also offers procedures. These procedures are added to the program once with special macros such as `DEFINE_SCAN_NUM` and called with CALL. The library thus follows the pattern in which macros and procedures work together: short jobs are macros, long jobs are single-copy procedures. The library's macros and procedures are treated in detail in the notes on the emu8086.inc library.

9.1.7.2 Code trace: expansion of PRINT 'OK'

The PRINT macro of the library uses all the techniques so far together: register preservation, LOCAL labels and data defined inside the macro. The call `PRINT 'OK'` expands to the following code (the LOCAL names are as the assembler generates them):

```
org 100h
push ax          ; PRINT 'OK'
push si
jmp  ??0003      ; skip the data
??0001 db 'OK', 0
??0003: lea si, ??0001
??0000: mov al, cs:[si]
cmp  al, 0
jz   ??0002
inc  si
mov  ah, 0Eh
int  10h
jmp  ??0000
??0002: pop si
pop  ax
ret
```

The string 'OK' and the 0 byte after it are embedded in the code, at the place where the macro was expanded: 4Fh, 4Bh, 00h starting at 0104h. Because the processor does not distinguish code from data, it might take these three bytes for instructions. To prevent this, the macro first jumps over the string with `JMP`. The name of the string is LOCAL too (`??0001`); a second PRINT in the same program defines its own string under another name.

Because the string sits in the code segment, the instruction that reads it uses the CS: segment prefix: `MOV AL, CS:[SI]`. Without the prefix the read would use the default segment, DS. In a .COM program CS = DS, so the result would be the same, but the macro must also work in .EXE programs where DS points to another segment. In machine code the prefix is the byte 2Eh and it makes the instruction one byte longer.

In emu8086 the instruction `LEA SI, ??0001` assembles to the same 3-byte code as `MOV SI, OFFSET ??0001`: `BE 04 01`. In real MASM, a LEA with a direct address takes 4 bytes (`8D 36` + address) and the expansion would be one byte longer. The total size of the expansion is 26 bytes (0100h–0119h), 3 bytes of which are data. In general, for a text of n characters the PRINT expansion takes 24 + n bytes.

Step	Address	Instruction	AL	SI	SP	ZF	Screen	Explanation
1	0100h	`push ax` PRINT 'OK' expansion	00h	0000h	FFFCh	0	–	AX saved
2	0101h	`push si` PRINT 'OK' expansion	00h	0000h	FFFAh	0	–	SI saved; SP = FFFAh
3	0102h	`jmp ??0003` PRINT 'OK' expansion	00h	0000h	FFFAh	0	–	jump over the string's bytes
4	0107h	`??0003: lea si, ??0001` PRINT 'OK' expansion	00h	0104h	FFFAh	0	–	SI ← 0104h
5	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' expansion	4Fh	0104h	FFFAh	0	–	AL ← CS:[0104h] = 'O'
6	010Dh	`cmp al, 0` PRINT 'OK' expansion	4Fh	0104h	FFFAh	0	–	4Fh ≠ 0: ZF = 0

Step	Address	Instruction	AL	SI	SP	ZF	Screen	Explanation
7	010Fh	`jz ??0002` PRINT 'OK' expansion	4Fh	0104h	FFFAh	0	–	no jump
8	0111h	`inc si` PRINT 'OK' expansion	4Fh	0105h	FFFAh	0	–	SI = 0105h
9	0112h	`mov ah, 0Eh` PRINT 'OK' expansion	4Fh	0105h	FFFAh	0	–	
10	0114h	`int 10h` PRINT 'OK' expansion	4Fh	0105h	FFFAh	0	`O`	O to the screen
11	0116h	`jmp ??0000` PRINT 'OK' expansion	4Fh	0105h	FFFAh	0	`O`	back to the loop start
12	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' expansion	4Bh	0105h	FFFAh	0	`O`	AL = 'K'
15	0111h	`inc si` PRINT 'OK' expansion	4Bh	0106h	FFFAh	0	`O`	SI = 0106h
17	0114h	`int 10h` PRINT 'OK' expansion	4Bh	0106h	FFFAh	0	`OK`	OK on the screen
19	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' expansion	00h	0106h	FFFAh	0	`OK`	AL = 00h
20	010Dh	`cmp al, 0` PRINT 'OK' expansion	00h	0106h	FFFAh	1	`OK`	terminator: ZF = 1
21	010Fh	`jz ??0002` PRINT 'OK' expansion	00h	0106h	FFFAh	1	`OK`	jump to the exit
22	0118h	`??0002: pop si` PRINT 'OK' expansion	00h	0000h	FFFCh	1	`OK`	SI = 0000h
23	0119h	`pop ax` PRINT 'OK' expansion	00h	0000h	FFFEh	1	`OK`	AX restored; SP = FFFEh
24	011Ah	`ret` main program	00h	0000h	0000h	1	`OK`	the program ends

Table 9.1.17: Trace of the expansion of `PRINT 'OK'`; the string is at 0104h, inside the code.

Figure 9.1.13: The string embedded in the code: 'O', 'K' and the terminating 0 starting at 0104h. The JMP jumps over these bytes.

Table 9.1.17 shows the three passes of the loop; the repeated MOV AH and INT steps are abbreviated. In the first pass AL = 4Fh ('O') is read. `CMP AL, 0` gives ZF = 0 and JZ does not jump. SI advances by one, INT 10h writes 'O' and JMP returns to the loop start. The second pass writes 'K'. In the third pass

AL = 00h is read; CMP gives ZF = 1 and JZ jumps to the exit (??0002). The 0 byte at the end is the terminator that marks the end of the string. The two POPs return SI and AX to their old values; SP is FFFEh again. OK has been printed and the caller's registers are intact.

This expansion shows the power and the cost of the macro together. The single source line `PRINT 'OK'` has produced 26 bytes. If twenty different texts are printed with PRINT in a program, each carries its own loop. The same job takes far less space with an arrangement that keeps the strings in a separate place and calls a single print procedure. The library's procedures such as PRINT_STRING exist for exactly this purpose.

9.1.8 Macro operators

Since macro expansion is a text operation, special marks are needed to split and join text. MASM and TASM define five macro operators for this purpose. These operators have meaning only inside macro definitions and calls.

Operator	Role	Example and expansion
`&`	joins a parameter to adjacent text	`CLEAR MACRO r` · `XOR r&X, r&X` → `CLEAR B` : `XOR BX, BX`
`< >`	makes text containing commas or spaces a single parameter	`EMIT <'A', 'B'>` → one parameter: `'A', 'B'`
`!`	makes the next character an ordinary character	`<5 !> 3>` → text: `5 > 3`
`%`	evaluates an expression and substitutes the result as text	`SHOW %(4*8)` → parameter `32`
`;;`	macro comment; not copied into the expansion	`;; visible only in the definition`

Table 9.1.18: Macro operators.

The `&` (concatenation) operator is the most frequently used. The assembler looks for parameter names in the body as whole words. Had `XOR rX, rX` been written, "rX" would count as a single word and would not match the parameter r; the expansion would remain `XOR rX, rX` and give an unknown-name error. Writing `r&X` states that r is a parameter name and that the X that follows is to be glued to it. In the call `CLEAR B`, r = B and the expansion is `XOR BX, BX`. XORing a register with itself clears it and is a 2-byte, 3-cycle instruction; `MOV BX, 0` takes 3 bytes and 4 cycles.

The angle brackets `< >` make it possible to pass text containing commas as a single parameter. Without the brackets, `EMIT 'A', 'B'` would count as two separate parameters. The `!` operator removes the special meaning of the character after it; the first `>` in `<5 !> 3>` does not close the bracket but becomes part of the text. The `%` operator passes the parameter not as text but as a value: `%(4*8)` is first evaluated to 32, and the text "32" goes to the macro. This is an alternative to parentheses as protection against the operator-precedence trap.

The last operator concerns comments. A comment written with a single `;` is part of the body and is written to the listing file again at every expansion. A comment written with a double `;;` stays in the definition and is not copied into the expansion. For a macro used a hundred times, this saves the listing file hundreds of needless lines.

9.1.9 Repeat blocks

9.1.9.1 REPT, IRP and IRPC

Sometimes one wants a piece of code written several times in place, without having to define a macro. Repeat blocks do this job.

> **Definition 9.1.5**
>
> A **repeat block** is a macro that has no name and is expanded immediately where it is defined. `REPT n` writes the body n times, `IRP p, <list>` writes it once for each item of the list (with p replaced by that item), and `IRPC p, text` writes it once for each character of the text. The block ends with `ENDM`. In MASM 6, IRP and IRPC are also written as FOR and FORC.

```
REPT 3
  SHL AX, 1
ENDM                ; → SHL AX, 1 · SHL AX, 1 · SHL AX, 1

IRP r, <AX,BX,CX>
  PUSH r
ENDM                ; → PUSH AX · PUSH BX · PUSH CX

IRPC c, OK
  PUTC '&c'
ENDM                ; → PUTC 'O' · PUTC 'K'
```

The REPT example shifts AX left by one bit three times, that is, multiplies it by 8: with AX = 5 the result is 40 (0028h). The IRP example pushes three registers on the stack; it is a short way of writing the SAVE macro. In the IRPC example c becomes the characters O and K in turn. For c to be recognized as a parameter inside the quotes, `&c` is written; the text inside quotes would otherwise stay as it is. Since both PUTCs in the expansion are macros, the expansion is nested, and OK is written on the screen.

A repeat block is not a loop. It needs no loop counter, but the code grows: `REPT 100` copies the body a hundred times. The three SHLs of the REPT example take 6 bytes and run in 3 × 2 = 6 cycles. The same shift can be written in 4 bytes with `MOV CL, 3` and `SHL AX, CL`. But on the 8086 a shift by CL

adds 4 cycles per bit, and the job takes 4 + (8 + 4 × 3) = 24 cycles. For short repetitions REPT is both fast and of reasonable size; `REPT 100`, however, takes 200 bytes, and the same job is done with a counted LOOP in a single copy.

The most common use of repeat blocks is to generate tables at assembly time. This needs an assembler symbol whose value can be changed. A constant defined with `EQU` cannot be redefined. A symbol defined with `=`, however, can take a new value at every assignment. The block below generates the squares of the numbers from 0 to 7:

```
n = 0
squares LABEL BYTE
REPT 8
    DB n*n
    n = n + 1
ENDM
```

`squares LABEL BYTE` gives the current address a name of type byte but reserves no space. The body of REPT is expanded eight times. In each expansion `DB n*n` writes the square of the current n as one byte, and `n = n + 1` increments the symbol. As a result the bytes 0, 1, 4, 9, 16, 25, 36, 49 sit one after another in memory. n is not a variable; it has no place in memory. It is a number that lives only at assembly time, in the assembler's table. The same method generates tables such as masks, powers of 2 or precomputed sine values.

9.1.9.2 Code trace: a table of squares with REPT

The generated table can be used to find the square of a number without multiplying, with the XLAT instruction:

```
org 100h
lea  bx, squares
mov  al, 5
xlat           ; AL = squares[5]
mov  dl, al
mov  al, 7
xlat           ; AL = squares[7]
add  al, dl
ret
squares db 0, 1, 4, 9, 16, 25, 36, 49   ; the bytes generated by the REPT block
```

`XLAT` (translate by table) uses AL as an index into the table that BX points to: AL ← DS:[BX + AL]. The address of the table must first be loaded into BX. The size of the code is 3 + 2 + 1 + 2 + 2 + 1 + 2 + 1 = 14 bytes; since LEA takes 3 bytes in emu8086, the table starts at 0100h + 0Eh = 010Eh.

Step	Address	Instruction	BX	AL	DL	Explanation
1	0100h	`lea bx, squares` main program	010Eh	00h	00h	BX ← 010Eh (offset of squares)
2	0103h	`mov al, 5` main program	010Eh	05h	00h	the number to look up, and index 5
3	0105h	`xlat` main program	010Eh	19h	00h	AL ← [BX + AL] = [0113h] = 25
4	0106h	`mov dl, al` main program	010Eh	19h	19h	first result in DL
5	0108h	`mov al, 7` main program	010Eh	07h	19h	index 7
6	010Ah	`xlat` main program	010Eh	31h	19h	AL ← [0115h] = 49
7	010Bh	`add al, dl` main program	010Eh	4Ah	19h	25 + 49 = 74 = 4Ah
8	010Dh	`ret` main program	010Eh	4Ah	19h	the program ends

Table 9.1.19: Reading with XLAT from the table of squares generated by REPT; the table starts at 010Eh.

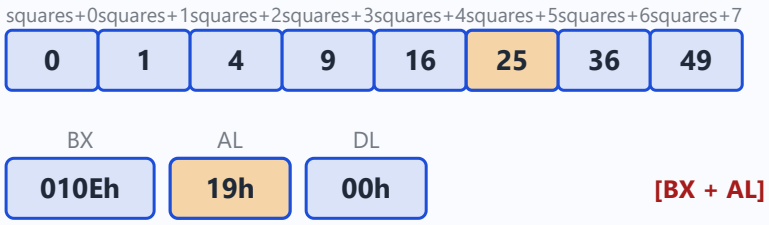


Figure 9.1.14: The first XLAT: with index AL = 5, the 25 (19h) at address squares + 5 = 0113h is read.

In Table 9.1.19 the first XLAT reads the byte at address 010Eh + 5 = 0113h: 25 = 19h (Figure 9.1.14). This value is saved in DL. The second XLAT reads 49 = 31h from address 010Eh + 7 = 0115h. ADD AL, DL adds the two: 25 + 49 = 74 = 4Ah, that is, $5^2 + 7^2$. No multiply instruction has run anywhere in the program.

The speed difference is clear. XLAT finishes in 11 cycles. An 8-bit MUL on the 8086 takes 70–77 cycles. The price of the table is memory: eight bytes. This is the classic trade-off between memory and time, and REPT keeps it as short as a single line in the source. Because the table is generated at assembly time, no time is spent computing it at run time either.

9.1.10 Conditional assembly

9.1.10.1 The IF family

Conditional assembly chooses whether a section of source code is included in the program at assembly time. It is the assembly counterpart of the `#if` and `#ifdef` preprocessor directives of C.

> **Definition 9.1.6**
>
> **Conditional assembly** is the selection, with directives such as `IF`, `IFB`, `ELSE` and `ENDIF`, of whether a source section is included in the code at assembly time. The condition is evaluated at assembly time; a branch that is not selected never enters the machine code.

Directive	When is the branch taken?	Typical use
`IF expression`	expression ≠ 0	test builds, options
`IFE expression`	expression = 0	the opposite of IF
`IFB <p>`	the parameter is blank	optional parameter
`IFNB <p>`	the parameter is not blank	optional parameter
`IFIDN <a>,<b>`	the two texts are identical	checking a register name
`IFDIF <a>,<b>`	the two texts differ	checking a register name

Table 9.1.20: Conditional assembly directives.

Every conditional block ends with `ENDIF` and may carry an optional `ELSE` branch. The condition does not look at a flag or register at run time. It looks at what the assembler knows at assembly time: the values of symbols and the text of macro parameters. Conditional assembly therefore produces no instruction, comparison or jump at run time.

The first area of use is producing different versions of a program from the same source:

```
DEBUG = 1
IF DEBUG
  PUTC '*'          ; test build only
ENDIF
```

With DEBUG = 1, PUTC is expanded and the program grows by 8 bytes; a star is printed on the screen first. When the symbol is set to 0, these lines are not assembled at all and leave no trace in the program. Debug output or checks are thus removed from the final version without being deleted from the source.

The second area of use is inside macros. `IFB` and `IFNB` (blank, not blank) make optional parameters possible. The SHIFTN macro below shifts a register left; if the shift count is not given, 1 is assumed:

```
SHIFTN MACRO r, n
  IFB <n>          ;; was n omitted?
    SHL r, 1
  ELSE
    MOV CL, n
    SHL r, CL
  ENDIF
ENDM
```

On the 8086 the shift count is either 1 or given in CL. An immediate count such as `SHL AX, 3` came with the 80186. The macro hides this rule from the user: if n is blank it produces a one-bit shift, otherwise a shift by CL. Writing the parameter inside angle brackets (`<n>`) lets even an empty text be compared as a text.

`IFIDN` and `IFDIF` set up a safety net against parameter pitfalls. SHIFTN clobbers CL; the call `SHIFTN CX, 2` therefore gives a wrong result. `MOV CL, 2` overwrites the low byte of CX, and the value being shifted is no longer the programmer's CX. A block starting with `IFIDN <r>, <CX>` can be added to the body to produce an error message in this case. The `EXITM` directive ends the expansion at that point; the remaining body lines are not written. Used together with conditional assembly, it also provides the stopping condition of a recursive macro.

9.1.10.2 Code trace: SHIFTN

```
org 100h
mov ax, 5
mov bx, 3
SHIFTN AX          ; → shl ax, 1
SHIFTN BX, 3       ; → mov cl, 3 · shl bx, cl
ret
```

Step	Address	Instruction	AX	BX	CL	CF	Explanation
1	0100h	`mov ax, 5` main program	0005h	0000h	00h	0	AX = 5
2	0103h	`mov bx, 3` main program	0005h	0003h	00h	0	BX = 3
3	0106h	`SHL AX, 1` SHIFTN expansion	000Ah	0003h	00h	0	n is blank, IFB branch: AX = 5 × 2
4	0108h	`MOV CL, 3` SHIFTN expansion	000Ah	0003h	03h	0	n = 3, ELSE branch: the count goes to CL
5	010Ah	`SHL BX, CL` SHIFTN expansion	000Ah	0018h	03h	0	$3 \times 2^3 = 24 = 0018$h

Step	Address	Instruction	AX	BX	CL	CF	Explanation
6	010Ch	`ret` main program	000Ah	0018h	03h	0	the program ends

Table 9.1.21: The expanded code of the calls `SHIFTN AX` and `SHIFTN BX, 3`.

In the first call n is blank; the assembler selects the IFB branch and writes a single `SHL AX, 1` (2 bytes, `D1 E0`). AX = 5 × 2 = 000Ah. In the second call n = 3; the ELSE branch is selected and two instructions are written: `MOV CL, 3` (`B1 03`) and `SHL BX, CL` (`D3 E3`), 4 bytes in all. BX = 3 × 2^3 = 24 = 0018h. The same macro has produced code of two different lengths. The first expansion does not touch CL; only the second expansion changes CL.

The trace contains no comparison or jump. The assembler decided which branch to write, and the processor merely executed the result. Had the same choice been made at run time, every call would have needed a CMP and a conditional jump; the code would have grown and the prefetch queue would have been flushed.

9.1.11 Debugging and common mistakes

Macros shorten the code in the source but make debugging harder, because the code that runs is not written in the source. Four habits reduce this difficulty.

Look at the expansion. The instruction listing in emu8086's emulator window shows not the macro names but the expanded instructions; single-stepping also advances over these instructions. Because macro labels get names such as `??0000` through LOCAL, jump targets do not appear under their source names. These numbers show which copy is running.

The listing file. In MASM, `.LISTMACROALL` writes all lines of the expansion to the listing file. `.LISTMACRO` writes only the lines that generate code, and `.NOLISTMACRO` hides the expansion completely. Because the listing file gives the address and machine code of every line side by side, it is the most reliable source for byte counting and address checking.

Read the error line correctly. An error in the body is reported separately at every line where the macro is used. If the same error message is seen several times, the cause is usually in the body.

Expand by hand first. A new macro is developed by first writing its body as plain code and trying it out. Once the code works correctly with one parameter value, it is turned into a parameterized macro. Then it is tried separately with parameters of different kinds (register, variable, constant, expression).

Mistake	Result	Fix
defining after use	the call is treated as an unknown instruction	put definitions at the top of the file or in an INCLUDE file
forgetting ENDM	the rest of the file counts as the body; the error shows up far below	check the ENDM of every MACRO
not making a label LOCAL	*Duplicate declaration* at the second use	`LOCAL` as the first line of the body

Mistake	Result	Fix
RET in the body	RET takes the caller's return address or an arbitrary word	do not write RET in a macro
passing a register the body clobbers as a parameter	in calls such as `ADD3 AX, BX, CX` the parameter changes before it is read	document the registers used; check with IFIDN
expression without parentheses	operator precedence changes	`(wd)*(ht)`
mistaking a macro for a loop	`REPT 1000` produces 1000 copies	LOOP for repetition at run time
using a long body many times	code size grows with the number of uses	move the job into a procedure, make the macro a wrapper

Table 9.1.22: Common macro mistakes and their consequences.

Table 9.1.22 collects the eight common mistakes. They all share the same root: a macro is a text pattern, not a unit that is called. The most reliable way to understand what a macro does is to expand it and read the resulting instructions like an ordinary program.

9.1.12 Exercises and solutions

Exercise 1. What is the expanded form of the call `MyMacro AX, BX, CX`? Which lines are unnecessary?

Solution. p1 = AX, p2 = BX, p3 = CX. The expansion is the lines `MOV AX, AX`, `MOV BX, BX`, `MOV CX, CX`. All three are valid instructions and assemble to 2 bytes each (`8B C0`, `8B DB`, `8B C9`). But none of them changes a register; MOV does not affect the flags either. All three lines are unnecessary: 6 bytes and 6 cycles are wasted. A program that starts with AX = 1, BX = 2, CX = 3 carries the same values after the macro. The macro assembles without error but does nothing; the assembler gives no warning for such ineffective expansions.

Exercise 2. Write an `ADD3 a, b, c` macro that adds two word variables and writes the sum to a third.

Solution.

```
ADD3 MACRO a, b, c
    MOV AX, a
    ADD AX, b
    MOV c, AX
ENDM
```

Because the 8086 has no memory-to-memory addition, AX is the intermediate register. The macro modifies AX; this must be stated in the macro's documentation. With s1 = 1234h and s2 = F00Fh, the call `ADD3 s1, s2, s3` gives s3 = 0243h and CF = 1; the 17th bit of the unsigned sum is in the carry.

Exercise 3. What are the code sizes if the PUTC macro is used 50 times, and if a procedure doing the same job is called 50 times?

Solution. Macro: 50 × 8 = 400 bytes. Procedure: body 8 bytes + RET 1 byte = 9 bytes once, plus 50 × 3 = 150 bytes of CALLs; 159 bytes in all. In time, the procedure adds 50 × 27 = 1350 cycles.

Exercise 4. What error occurs when the first POWER macro, `POWER base, exponent, result`, is used twice, with a second call? How is it fixed?

Solution. The second expansion redefines the label `powerloop:`; the assembler reports a *Duplicate declaration* (powerloop) error and the program does not assemble. The fix is to add `LOCAL powerloop` as the first line of the macro. The label is then ??0000 in the first expansion and ??0001 in the second, and each LOOP returns to the label of its own copy.

Exercise 5. How many times does the first POWER macro execute MUL when exponent = 0?

Solution. 65,536 times. Since MUL is at the start of the loop, it runs once first. LOOP decrements CX from 0 to FFFFh and jumps. The loop continues until CX is 0 again: CX = 0 → FFFFh → … → 1 → 0. With base = 2, result = 0000h; with base = 1, result = 0001h. If the counter is checked beforehand with JCXZ, no MUL runs and result = 1.

Exercise 6. Which instructions must be added so that a macro corrupts no register except AX?

Solution. At the start of the body, one PUSH for every register the macro modifies (except AX); at the end, POPs for the same registers in reverse order. In the corrected POWER these are `PUSH BX`, `PUSH CX`, `PUSH DX` and `POP DX`, `POP CX`, `POP BX`. Instructions that modify registers implicitly deserve particular attention (MUL changes DX, LOOP changes CX, INT services change their return registers). If the flags must also be preserved, PUSHF and POPF are used.

9.1.13 Summary

- A macro is a code template defined with `name MACRO p1, p2 … ENDM`. The assembler writes the body in place of every line that uses the macro; this is called expansion. The processor never sees the macro.
- A macro must be defined before the line that uses it. A macro that is never used generates no code.
- Parameters are text, not values. Their values are read when the expanded line executes. A register that the body modifies must not be passed as a parameter; expression parameters must be parenthesized in the body.
- Which call is valid is decided by the instruction produced in the expansion; the error is reported at the line of use, but its cause is in the body.
- Size: a macro takes k · n bytes, a procedure (n + 1) + 3k bytes. If the body is longer than 3 bytes, the procedure is smaller beyond the break-even point.

- Speed: every procedure call adds CALL + RET, 19 + 8 = 27 cycles, two stack accesses and two queue flushes. A macro has no such cost.
- RET is not written in a macro; RET takes the caller's return address or whatever is on the stack.
- Macros with labels are written with LOCAL; the assembler generates unique names such as ??0000, ??0001 … at every expansion.
- The registers a macro modifies are saved with PUSH and restored with POP in reverse order (PUTC, the corrected POWER).
- LOOP with CX = 0 runs 65,536 times; a JCXZ goes in front of loops whose counter is a variable. MUL also changes DX.
- INCLUDE inserts the library text into the source; emu8086.inc offers short jobs as macros and long jobs as procedures.
- The macro operators &, <>, !, % and ;; join, group and evaluate text.
- REPT, IRP and IRPC are unnamed repeat blocks expanded in place; with the `=` symbol they generate tables at assembly time.
- IF, IFE, IFB, IFNB, IFIDN and IFDIF evaluate the condition at assembly time; the branch not selected never enters the machine code.

9.1.14 Key terms

macro macro expansion LOCAL directive conditional assembly repeat block (REPT / IRP / IRPC) directive assembler procedure procedure call return address register preservation label library clock cycle prefetch queue bus cycle listing file emulator

---

9.1.15 Test Yourself

1. The ADD3 macro (`MOV AX, x` · `ADD AX, y` · `MOV z, AX`) is called as `ADD3 BX, AX, CX` with AX = 5 and BX = 7. What does CX become?

   Answer

   The expansion is `MOV AX, BX` · `ADD AX, AX` · `MOV CX, AX` . The first line makes AX 7; the text y = AX now refers to 7. ADD gives 7 + 7 = 14: CX = **000Eh**, not the expected 12.

2. What does the call `AREA 10-2, 3` of the macro `AREA MACRO wd, ht` / `MOV AX, wd*ht` / `ENDM`

load into AX? What does the parenthesized version load?

Answer

The expansion is `MOV AX, 10-2*3`; multiplication comes first: 10 − 6 = **4**. The parenthesized `(10-2)*(3)` gives **24** (0018h). In both cases the multiplication is done at assembly time and the machine code contains only one MOV.

3. From how many uses on does a job with a 4-byte body take less space as a procedure?

Answer

Macro 4k, procedure 5 + 3k bytes. 4k > 5 + 3k gives k > 5. At five uses both take 20 bytes; from **six** uses on the procedure is smaller (23 bytes against 24).

4. If a macro that contains a label but does not use LOCAL is used only once in a program, is there an assembly error?

Answer

No. With a single expansion the label is defined once and the program assembles. The error appears as *Duplicate declaration* the moment the macro is used a second time. This is why LOCAL is written even if the macro will be used only once.

5. MyMacro2 (jumps to label1 if AX = 2 and to label2 if AX = 3; label1: INC AX, label2: ADD AX, 2) is run with AX = 0. What does AX become?

Answer

Neither JE jumps and the flow falls into label1. INC AX gives 1 and ADD AX, 2 then gives **3**.

6. The SHIFTN macro (when n is given: `MOV CL, n` · `SHL r, CL`) is called as `SHIFTN CX, 2` with CX = 5. The programmer expects 5 × 4 = 20. What does CX become?

Answer

`MOV CL, 2` overwrites the low byte of CX: CX = 0002h. `SHL CX, CL` shifts 2 left by 2 bits: CX = **0008h**. A register that the body clobbers (CL, and therefore CX) has been passed as a parameter; IFIDN can catch this call.

7. Which bytes does the block `n = 1` · `table LABEL BYTE` · `REPT 4` · `DB n` · `n = n * 2` · `ENDM`

write to memory? How many instructions does this block execute at run time?

> **Answer**
>
> 01h, 02h, 04h, 08h. The block produces only data; n is a symbol that lives at assembly time. **No** instruction is executed at run time.

8. A procedure is called 1000 times on a 5 MHz 8086. How much extra time do CALL and RET add?

> **Answer**
>
> 1000 × (19 + 8) = 27,000 cycles. Since one cycle is 200 ns, 27,000 × 200 ns = **5.4 ms**. With a macro this time would not exist, but the code would grow by 1000 copies.

9. How many bytes does the PRINT macro of emu8086.inc produce for `PRINT 'Hello'` ? Although the string is inside the code, why does the processor not execute it as instructions?

> **Answer**
>
> 26 bytes for 'OK'; every extra character adds one byte. For a 5-character text, 24 + 5 = **29 bytes**. The macro puts a JMP in front of the string and the processor jumps over the string's bytes; the string is accessed only as data, with `MOV AL, CS:[SI]`.

9.1.16 Further reading

- Microsoft, *MASM Programmer's Guide*: the reference description of macro definition, LOCAL, macro operators, repeat blocks and conditional assembly directives.
- K. R. Irvine, *Assembly Language for x86 Processors*, Chapter 10: treats macros, conditional assembly and repeat blocks with many examples.
- emu8086 help documents, *8086 Assembler Tutorial, Part 10 (Macros)*: writing macros in emu8086 and the emu8086.inc library.
- B. B. Brey, *The Intel Microprocessors*: treats the choice between a macro and a procedure together with 8086 instruction timings and the hardware structure.

9.1.17 References

1. emu8086 help documents: *8086 Assembler Tutorial, Part 10 (Macros)* and `emu8086.inc`.
2. Microsoft, *MASM Programmer's Guide* — the MACRO, ENDM and LOCAL directives.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — Chapter 10 (Macros).
4. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.

5. Intel, *8086 16-Bit HMOS Microprocessor* data sheet (instruction timings).

PART III · CHAPTER 9.2

The emu8086.inc Library

Almost every program written in assembly prints something to the screen, reads something from the keyboard or shows a number in decimal. Each of these jobs takes several BIOS and DOS services called in the right order. emu8086 collects these frequently needed jobs in a ready-made library called emu8086.inc. These notes begin by showing how the library is brought into a program with the INCLUDE directive and what determines the size of the program. They then treat the screen macros (PUTC, GOTOXY, PRINT, PRINTN, CURSOROFF, CURSORON) together with their expansions. Next come the procedures used with DEFINE_ and CALL: PRINT_STRING and PTHIS, which print strings, GET_STRING, which reads a string from the keyboard, CLEAR_SCREEN, which clears the screen, SCAN_NUM, which reads a number, and PRINT_NUM and PRINT_NUM_UNS, which print one. The logic of each procedure is traced register by register with a plain equivalent code. The last sections are given to example programs that put all the parts together and to the mistakes most often made when using the library.

LEARNING OUTCOMES

Explain the INCLUDE directive

Explain that a library file is added to the source as text during assembly, which folders the assembler searches for the file, and that program size is determined only by the macros that are used and the procedures that are defined.

Use the screen macros

Show which BIOS service and which instruction sequence the macros PUTC, GOTOXY, PRINT, PRINTN, CURSOROFF and CURSORON expand to; work out the cursor position, the end of line and screen scrolling in advance.

Define and call procedures correctly

Add procedures to the program with the DEFINE_ macros, call them with CALL, know the input and output registers of each procedure, and say which registers are preserved.

Trace the string and number procedures
Trace the logic of PRINT_STRING, PTHIS, GET_STRING, SCAN_NUM, PRINT_NUM and PRINT_NUM_UNS step by step at the level of registers, stack and buffer; explain the difference between the signed and unsigned interpretations.

Write example programs
Write programs that read numbers, compute, print the result and arrange the screen using the library's macros and procedures; recognize the common mistakes.

9.2.1 The library and INCLUDE

9.2.1.1 A library of common functions

Even printing a single character on the 8086 takes several instructions. The character goes in AL, the service number in AH, and the BIOS screen interrupt `INT 10h` is called. In the process the old value of AX is destroyed; if the value is needed it must be pushed on the stack first and popped afterwards. Printing a number in decimal is longer still: the number is divided by 10, the remainders become digits, the digits are converted to ASCII codes and printed in the right order. Reading a multi-digit number from the keyboard is the same job in reverse. These patterns are needed again in almost every program. Rewriting them every time is both a waste of time and a source of bugs.

emu8086 solves this with a library. The `emu8086.inc` file that comes with the installation contains ready-made macros and procedures for screen, keyboard and number handling. The programmer adds this file at the top of the source and uses the functions in it by name. The library brings no new capability: everything in it is built from BIOS and DOS interrupts, that is, from instructions the programmer could have written too. The gain is the reuse of tested code.

Definition 9.2.1

A **library** is a source file that contains ready-made macros and procedures and is used by being added to other programs. The library of emu8086 is the file `emu8086.inc`; the extension *inc* shows that it is an include file.

One limit of the library should be known from the start. emu8086.inc is specific to the emu8086 environment. Another assembler such as MASM or TASM does not recognize this file, and the library's macros may not be fully compatible with the macro language of those assemblers. The code inside, however, consists of ordinary 8086 instructions. Understanding how the library works therefore also teaches how to write code that does the same job in other environments. For this purpose the trace sections of these notes show the logic of each procedure with a plain equivalent code.

9.2.1.2 The INCLUDE directive

Definitions in another file are brought into the source with the `INCLUDE` directive. For emu8086.inc the line is:

```
include 'emu8086.inc'
```

Definition 9.2.2

INCLUDE is the directive that tells the assembler to put the text of the named file in place of this line. The directive produces no processor instruction; its job is finished during assembly. A running program contains no instruction called "INCLUDE".

The file name may be written with or without quotes: `include 'emu8086.inc'` and `include emu8086.inc` do the same job. Letter case does not matter either. The assembler searches for the file in two places (Figure 9.2.1). First it looks in the folder of the source file being assembled. If the file is not there, it looks in the `Inc` folder of the emu8086 installation. Because emu8086.inc is placed in this folder by the installation, in most programs it is found in the second step. If the programmer writes a library of their own and puts it in the same folder as the source, it is found in the first step. If a file with the same name also exists in the source folder, the one in the source folder wins.

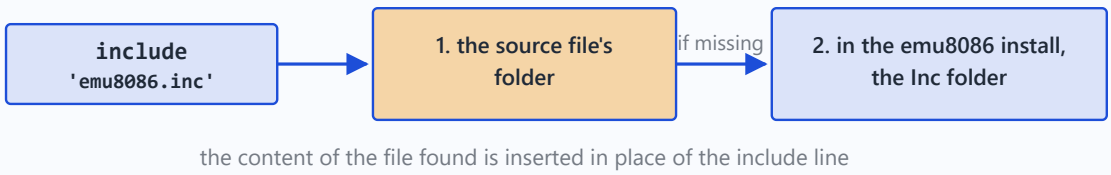


Figure 9.2.1: The INCLUDE line is searched for first in the source file's folder and, if not found, in the Inc folder of the emu8086 installation. The content of the file found is inserted in place of the line.

The position of the directive matters too. A macro can be used only after its definition; the assembler reads the source from top to bottom and, if it meets a macro name whose definition it has not seen, takes it for an instruction and reports an error. The `include` line is therefore written at the very beginning of the source, before `ORG 100h`. Since the line itself produces no bytes, putting it first does not change the program's addresses.

Common mistake

Writing the `include` line at the end of the program, right before `END`. When the assembler reaches a line such as `PUTC 'A'` it has not yet seen the definition of PUTC and reports an

unknown-instruction error. The definitions of macros must come before their uses.

9.2.1.3 The source after INCLUDE

The easiest way to see what INCLUDE does is to think about the source the assembler sees. The assembler deletes the include line and puts the whole text of emu8086.inc in its place. It then assembles the resulting single piece of source. A simplified picture looks like this:

```
; in place of the include line:
PUTC MACRO char     ; template: 0 bytes
  ...
ENDM
GOTOXY MACRO col, row
  ...
ENDM
DEFINE_SCAN_NUM MACRO
  ...               ; procedure template
ENDM
; the program itself:
ORG  100h
PUTC 'A'            ; 8 bytes here
RET
```

Three conclusions follow from this picture. First, most of the library consists of `MACRO … ENDM` blocks. A macro definition is only a template; the assembler notes it down somewhere but produces no bytes for it. A macro that is never used adds not a single byte to the program. Second, code is generated where the macro is used. In the program above, the line `PUTC 'A'`, once expanded, produces a sequence of five instructions and 8 bytes. The whole program consists of these 8 bytes plus the 1 byte of `RET`: 9 bytes.

The third conclusion concerns procedures. The procedures in the library are also stored not directly as code but inside macros. For example, the whole code of the SCAN_NUM procedure is the body of a macro called `DEFINE_SCAN_NUM`. If the program does not write the `DEFINE_SCAN_NUM` line, the macro is never expanded and the code of SCAN_NUM does not enter the program. Thus including the whole library does not enlarge the program. Program size is determined by only two things: every use of the macros that are used, and the procedures defined with DEFINE_.

In the source	Code produced	When?
`include 'emu8086.inc'`	none	definitions are only read
macro use (`PUTC 'A'`)	the macro's body, with the parameters substituted	again at every use
`DEFINE_SCAN_NUM`	the whole code of the procedure	once, at the place of the line
`CALL scan_num`	a 3-byte CALL instruction	at every call

Table 9.2.1: When do bytes from emu8086.inc enter the program?

Nothing called "emu8086.inc" is left inside the running program. In memory there are only expanded macro instructions and defined procedures. In emu8086's memory window or disassembler, PUSH, MOV, INT and POP instructions are seen where PUTC was. This matters for debugging: when single-stepping, one macro line corresponds to several instruction steps.

9.2.1.4 What is in the library?

The contents of emu8086.inc fall into two groups (Table 9.2.2, Table 9.2.3). Macros are written directly by name. Procedures are first defined with a DEFINE_ macro and then called with CALL.

Macro	Task	In the background
`PUTC char`	writes a single character where the cursor is	INT 10h, AH = 0Eh
`GOTOXY col, row`	moves the cursor to the given column and row	INT 10h, AH = 02h
`PRINT string`	prints a string	character by character with 0Eh
`PRINTN string`	prints a string, then moves to a new line	character by character with 0Eh; 13, 10 at the end
`CURSOROFF`	hides the cursor	INT 10h, AH = 01h
`CURSORON`	shows the cursor	INT 10h, AH = 01h

Table 9.2.2: The macros in the library (used directly).

Procedure	Task	Definition line
`PRINT_STRING`	prints the zero-terminated string at DS:SI	`DEFINE_PRINT_STRING`
`PTHIS`	prints the string right after the CALL	`DEFINE_PTHIS`
`GET_STRING`	reads a string from the keyboard	`DEFINE_GET_STRING`
`CLEAR_SCREEN`	clears the screen	`DEFINE_CLEAR_SCREEN`
`SCAN_NUM`	reads a signed number from the keyboard into CX	`DEFINE_SCAN_NUM`
`PRINT_NUM`	prints AX as a signed number	`DEFINE_PRINT_NUM` + `DEFINE_PRINT_NUM_UNS`
`PRINT_NUM_UNS`	prints AX as an unsigned number	`DEFINE_PRINT_NUM_UNS`

Table 9.2.3: The procedures in the library (defined with DEFINE_, called with CALL).

The difference between the two groups is the basic difference between a macro and a procedure. A macro is copied to every place where it is used; it suits short, frequent jobs. The code of a procedure exists once in memory and every use goes there with CALL; it suits long jobs. The library makes this distinction sensibly. PUTC is 8 bytes, and the cost of a CALL and RET would be close to the job itself; that is why it is a macro. SCAN_NUM, on the other hand, is hundreds of bytes of code that checks the minus sign, backspace and overflow. Copying it at every use would swell the program quickly; that is why it is a procedure.

The time of a procedure call can be computed too. On the 8086 a direct CALL within the same segment takes 19 clock cycles and a near RET 8. On a 5 MHz 8086 one cycle is 200 ns, so the cost of calling and returning is 27 × 200 ns = 5.4 µs. An `INT 10h` that writes a character, however, runs hundreds of instructions inside the BIOS. For screen and keyboard jobs the time of CALL/RET is therefore negligible; what really decides is the space taken in memory.

9.2.2 Screen macros

9.2.2.1 The text screen, the cursor and BIOS services

The output window of emu8086 imitates the standard text screen of the PC. The screen consists of 80 columns and 25 rows. Each cell holds one character. The position of a cell is given as a (column, row) pair: the top-left corner is (0, 0) and the bottom-right corner is (79, 24). Counting starts at zero; "column 10" is therefore the eleventh cell from the left.

Definition 9.2.3

The **cursor** is the position on the screen where the next character will be written. The BIOS keeps the cursor's column and row in its own data area. The services that write characters put the character in the cell where the cursor is and advance the cursor one position to the right.

Behind the screen there is a region of memory. In colour text mode the memory of the video card starts at segment B800h in the processor's address space. Each cell holds two bytes: the ASCII code of the character and an attribute byte that determines its colour. The video card reads this memory tens of times a second and draws it on the screen. The most direct way to put a character on the screen is to write to this memory with `MOV`. For the processor this is an ordinary memory write bus cycle; a physical address such as B8xxxh appears on the address bus and the character code on the data bus.

The library's macros do not write to this memory directly; they use the BIOS screen services. The `INT 10h` instruction takes the processor to a BIOS routine in the ROM on the motherboard. The routine decides what to do by looking at the service number in AH. The BIOS takes care of details such as advancing the cursor, moving to the next line at the end of a line, and scrolling the screen on the last row. The three services the library uses are in Table 9.2.4.

AH	Service	Inputs	Used by
01h	set the cursor shape	CH = start, CL = end scan line	CURSOROFF, CURSORON
02h	set the cursor position	BH = page, DH = row, DL = column	GOTOXY
0Eh	write a character and advance the cursor (teletype)	AL = character	PUTC, PRINT, PRINTN

Table 9.2.4: The INT 10h services used by the screen macros.

The name of the 0Eh service comes from old typewriter-telegraph machines. Like those machines, the service writes the character and advances the paper, that is, the cursor. It also interprets control characters as such a machine would: 13 takes the cursor to the start of the line, 10 to the next line, 8 moves it one position left and 7 makes a sound. This feature explains most of the behavior of the screen macros.

9.2.2.2 What do PUTC and GOTOXY expand to?

PUTC takes a single parameter: the character to print. The parameter may be a constant (`PUTC 65`), a character constant (`PUTC 'B'`) or an 8-bit register or variable (`PUTC AL`). When expanding the macro, the assembler puts this text where the parameter is written. `PUTC 'A'` expands to these five instructions:

```
PUSH AX        ; save AX
MOV  AL, 'A'   ; the parameter
MOV  AH, 0Eh   ; teletype
INT  10h
POP  AX        ; restore AX
```

The instructions do the following in order. `PUSH AX` decreases SP by 2 and writes the current value of AX to the stack. `MOV AL, 'A'` puts 41h in AL and does not touch AH. `MOV AH, 0Eh` puts the service number in AH; AX = 0E41h now. `INT 10h` pushes the flags, CS and IP on the stack and passes to the BIOS. The BIOS writes 'A' to the cell where the cursor is, moves the cursor one position to the right and returns with `IRET` . `POP AX` restores the old value from the stack into AX and increases SP by 2. On leaving the macro, AX, SP and the flags are back to their values at entry. The calling code does not notice that AX changed temporarily.

The size of the expansion is computed from the machine codes: `PUSH AX` 1 byte (50h), `MOV AL, 41h` 2 bytes (B0 41), `MOV AH, 0Eh` 2 bytes (B4 0E), `INT 10h` 2 bytes (CD 10), `POP AX` 1 byte (58h). The total is 8 bytes.

GOTOXY takes two parameters: the column first, then the row. `GOTOXY 10, 5` expands like this:

```
PUSH AX
PUSH BX
PUSH DX
MOV  AH, 02h   ; set cursor position
MOV  DH, 5     ; row
MOV  DL, 10    ; column
MOV  BH, 0     ; page 0
INT  10h
POP  DX
POP  BX
POP  AX
```

The macro uses AH, BH, DH and DL; that is why it saves AX, BX and DX. The POPs are done in the reverse order of the PUSHes: DX, the last one pushed, is restored first. The expansion is 16 bytes: three PUSHes and three POPs take 6 bytes, four `MOV r8, constant` take 8 bytes, and `INT 10h` takes 2 bytes. BH = 0 selects the first display page of the text mode. The memory of a colour text card has room for several screen pages; emu8086 programs always use page 0.

The order of the parameters is the point most often confused. The macro uses the mathematical (x, y) order: the horizontal position (column) first, then the vertical position (row). The BIOS, however, expects the row in DH and the column in DL. The macro does this conversion itself: it puts the second parameter in DH and the first in DL.

Common mistake

Writing `GOTOXY 5, 10` and expecting the cursor to go to row 5. The macro takes 5 as the column and 10 as the row; the cursor goes to column 5 of row 10. A similar trap is to keep the position in AH: in the expansion of `GOTOXY AL, AH`, `MOV AH, 02h` runs first and then `MOV DH, AH` writes 2 as the row. For example with AL = 20 and AH = 7 the cursor goes not to row 7 but to row 2, to position (20, 2). The position must first be moved into registers the macro does not use.

9.2.2.3 PRINT and PRINTN

PRINT prints a string. Its parameter is a constant text given in quotes in the source: `PRINT 'Hello World!'`. The macro embeds this text in the code and prints it character by character with service 0Eh. The expansion of PRINT in emu8086.inc has the following structure (the local label names are simplified):

```
PUSH AX
PUSH SI
JMP  skip           ; jump over the string
str DB 'Hello World!', 0
skip:
LEA  SI, str
next:
MOV  AL, CS:[SI]    ; read the character from the code segment
CMP  AL, 0
JZ   done
INC  SI
MOV  AH, 0Eh
INT  10h
JMP  next
done:
POP  SI
POP  AX
```

The most interesting aspect of the expansion is the position of the string. The string sits in the middle of the code, and a `JMP` jumps over it so that the processor does not take it for instructions. The labels specific to the macro get unique names at every expansion through the `LOCAL` directive; when PRINT is used a second time there is no label clash. Because the string is inside the code, it is in the code segment, and the characters are read with `CS:[SI]`. In a .COM program CS = DS, so the prefix makes no difference; but thanks to the prefix the macro also works correctly when DS points to another segment.

The expansion saves SI and AX. Its size depends on the length of the string: for a string of n characters, n + 1 bytes of data and 23 bytes of code, 24 + n bytes in all. `PRINT 'Hello World!'` takes 36 bytes for its 12 characters. The loop code is copied again at every use of PRINT.

PRINTN does the same job and merely appends the bytes 13 and 10 to the end of the string. The macro defines the string as `DB 'text', 13, 10, 0`. 13 (CR, carriage return) takes the cursor to column 0 of the line and 10 (LF, line feed) takes it one line down. Together they move to a new line (Figure 9.2.2). `PRINT 'a'` followed by `PRINT 'b'` leaves *ab* on the screen. `PRINTN 'a'` followed by `PRINT 'b'`, however, writes *a* on one line and *b* at the start of the next line. The expansion of PRINTN is 2 bytes longer than that of PRINT.

Figure 9.2.2: PRINTN 'a' followed by PRINT 'b': CR takes the cursor to the start of the line and LF one line down; b is written at the start of the next line.

The parameter of PRINT must be a constant text known at assembly time. Because the macro embeds the text in the code, it cannot print a string that is created at run time, for example a name read from the keyboard. When `PRINT name` is written, the assembler produces the line `DB name, 0`. This line tries to define the address of the variable name as a byte; the result is either an error or a meaningless byte. Strings created at run time are printed with PRINT_STRING.

9.2.2.4 Example: a screen with macros

The first example in emu8086's library documentation uses three macros together:

```
include emu8086.inc
ORG    100h
PRINT 'Hello World!'
GOTOXY 10, 5
PUTC 65           ; 65 - is an ASCII code for 'A'
PUTC 'B'
RET               ; return to operating system.
END               ; directive to stop the compiler.
```

The first two lines produce no code. `include` adds the macro definitions, and `ORG 100h` tells the assembler that the program starts at offset 0100h. In .COM programs the first 256 bytes (0000h–00FFh) are reserved for DOS's program segment prefix (PSP), so the code starts at 0100h. When the program starts the screen is empty and the cursor is at (0, 0).

`PRINT 'Hello World!'` writes 12 characters to columns 0–11 of row 0. Because the BIOS moves the cursor one position right after each character, the cursor ends at (12, 0). `GOTOXY 10, 5` moves the cursor to column 10, row 5; it writes nothing to the screen. `PUTC 65` uses the number as a character code: 65 = 41h, which is 'A' in ASCII. 'A' is written to cell (10, 5) and the cursor moves on to (11, 5). `PUTC 'B'` does the same job with a character constant; the assembler converts `'B'` to 42h. 'B' is written to (11, 5) and the cursor moves on to (12, 5). `RET` ends the program. The `END` directive only tells the assembler that the source is over; it produces no code.

Line	Cursor (column, row)	Written to the screen	Explanation
`include emu8086.inc`	(0, 0)	—	the macro definitions are added, no code
`ORG 100h`	(0, 0)	—	the .COM program starts at 0100h
`PRINT 'Hello World!'`	**(12, 0)**	Hello World! (columns 0–11, row 0)	12 characters
`GOTOXY 10, 5`	**(10, 5)**	—	only the cursor moves
`PUTC 65`	**(11, 5)**	A into cell (10, 5)	65 = 41h = 'A'
`PUTC 'B'`	**(12, 5)**	B into cell (11, 5)	character constant
`RET`	(12, 5)	—	return to DOS

Table 9.2.5: Trace of the screen program with macros (the state after each line has run).

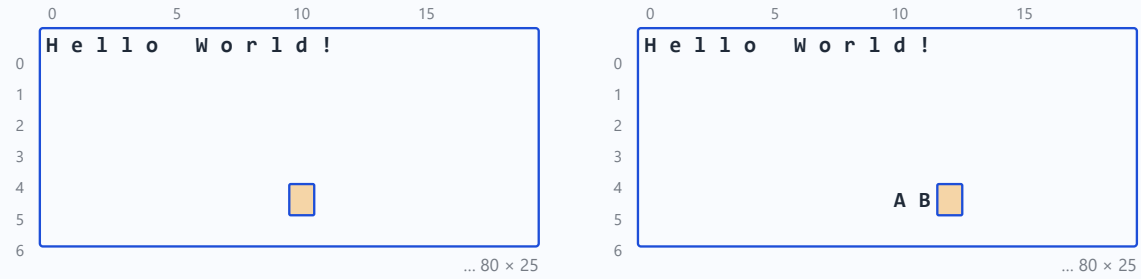


Figure 9.2.3: After GOTOXY 10, 5 (left) the cursor is at (10, 5); after the two PUTCs (right) AB is written on row 5 of the screen.

The machine code of the program runs from 0100h to 0144h. The 36-byte PRINT expansion occupies 0100h–0123h, the 16-byte GOTOXY expansion 0124h–0133h and the two PUTC expansions 0134h–0143h; `RET` is at 0144h. Although eight lines show in the source, the processor runs 35 different instructions. Because the PRINT loop runs 12 times, the number of instructions executed exceeds a hundred.

9.2.2.5 CURSOROFF and CURSORON

The cursor on the text screen is a small line that the video card makes blink. The height of the line is given in scan lines of the character cell. In colour text mode a character cell is 8 or 16 scan lines high; the cursor normally covers a line or two at the bottom of the cell. Service 01h of INT 10h sets the shape of the cursor: CH is the start and CL the end scan line.

Bit 5 of CH has a special meaning. If this bit is set to 1, the cursor is not drawn on the screen at all. CURSOROFF in the library writes 28h to CH: 28h = 0010 1000b, and its bit 5 is 1. CURSORON loads a normal shape with bit 5 equal to 0 through the same service. Both macros save and restore AX and CX.

```
PRINTN 'Full name:'   ; print, then go to the next line
CURSOROFF
GOTOXY 30, 12
PRINT 'Welcome'
CURSORON
```

In this fragment "Full name:" is printed first and the cursor moves to the start of the next line. CURSOROFF hides the cursor. GOTOXY moves the cursor to (30, 12) and "Welcome" is written to columns 30–36. During this the blinking cursor does not wander over the screen. CURSORON makes the cursor visible again; the cursor is at (37, 12). Hiding the cursor changes only the display; the cursor's position is still kept in the BIOS and the writing services keep advancing it. On game and menu screens the blinking cursor is hidden because it distracts.

9.2.2.6 Control characters

The first 32 codes of the ASCII table do not correspond to a printable shape. These codes are called **control characters**. Service 0Eh interprets four of them like a typewriter (Table 9.2.6). When PUTC is used with these codes it writes no shape on the screen but steers the cursor.

Code	Name	Effect
07h	BEL (bell)	a short beep from the speaker; the cursor does not move
08h	BS (backspace)	the cursor moves one position left; it does not erase the character; no effect at column 0
0Ah	LF (line feed)	the cursor moves down one line, the column is unchanged; on the last row the screen scrolls up one line
0Dh	CR (carriage return)	the cursor returns to column 0 of the same line; the text on the line stays in place

Table 9.2.6: Control characters given to PUTC.

A new line is therefore written with two characters: CR first, then LF. The 13, 10 pair that PRINTN appends is the combination of these two movements. Even with the order reversed (10, 13) the cursor ends up in the same place: it first moves down one line and then returns to the start of that line. This is why the line ending in DOS and Windows text files is the CR LF pair.

To erase a character from the screen, BS alone is not enough; it moves the cursor left but the character stays. Erasing is done in three steps. `PUTC 8` moves the cursor left. `PUTC ' '` writes a space over the character and moves the cursor right again. The second `PUTC 8` brings the cursor back to the emptied cell. The library's procedures that handle the backspace key fix the screen with this triple.

9.2.2.7 CR and LF separately: a code trace

To see what CR and LF do separately, the program below uses INT 10h without macros. AH = 0Eh is loaded once; since the BIOS does not change AH in service 0Eh, it stays valid for the later calls too.

```
ORG  100h
MOV  AH, 0Eh     ; teletype
MOV  AL, 'a'
INT  10h
MOV  AL, 'b'
INT  10h
MOV  AL, 10      ; LF only
INT  10h
MOV  AL, 'c'
INT  10h
MOV  AL, 13      ; CR only
INT  10h
MOV  AL, 'd'
INT  10h
RET
```

Instructions	AL	Cursor (column, row)	Screen
`MOV AH, 0Eh` / `MOV AL, 'a'` / `INT 10h`	61h	(1, 0)	a
`MOV AL, 'b'` / `INT 10h`	62h	(2, 0)	ab
`MOV AL, 10` / `INT 10h`	0Ah	**(2, 1)**	ab
`MOV AL, 'c'` / `INT 10h`	63h	(3, 1)	ab / ··c
`MOV AL, 13` / `INT 10h`	0Dh	**(0, 1)**	ab / ··c
`MOV AL, 'd'` / `INT 10h`	64h	(1, 1)	ab / d·c

Table 9.2.7: The separate effects of CR and LF.

In the table "/" marks a line break and "·" an empty cell. When only LF is sent, the cursor moves down a line but stays in column 2; 'c' is written to column 2 of the next line, and the lines drift like a staircase. When only CR is sent, the cursor returns to the start of the same line; the 'c' on the line is not erased. 'd' is written to column 0, and the screen shows "d c" under "ab" (Figure 9.2.4). A proper new line needs both codes together.

Figure 9.2.4: After LF alone the cursor is at (2, 1) (left); after CR alone and 'd', the screen shows ab and, below it, d c (right).

INT 10h/0Eh changes no register except AX; in this program AH also stays 0Eh throughout the calls. This behavior is a documented feature of the BIOS, but it must not be generalized to other services. Some INT 10h services change AX, others BP. That the library macros save AX at every use is

therefore a safe habit.

9.2.2.8 End of line and screen scrolling

After service 0Eh writes a character to column 79, it moves the cursor to the start of the next line by itself. This is how PRINT continues a long string without cutting it off. If the cursor is on the last row (24) and has to move to the next line, the BIOS scrolls the whole screen up one line. The top row disappears, an empty row appears at the bottom and the cursor moves to this new row. Scrolling is the copying of the lower 24 rows of video memory up by one row.

Example 9.2.1

`GOTOXY 75, 24` is followed by `PRINT 'Welcome'`. Let us find where the text ends up on the screen and the final position of the cursor.

1. The cursor is at (75, 24). "W", "e", "l", "c", "o" are written to columns 75, 76, 77, 78 and 79 in turn.
2. After writing to column 79 the cursor must move to the next row. Since the cursor is on the last row, the screen scrolls up one line. "Welco" moves up to row 23; row 24 is emptied and the cursor comes to (0, 24).
3. "m" and "e" are written to columns 0 and 1 of row 24. The cursor stays at (2, 24).

In the end "Welco" is in columns 75–79 of row 23, "me" at the start of row 24, and the cursor is at (2, 24) (Figure 9.2.5).

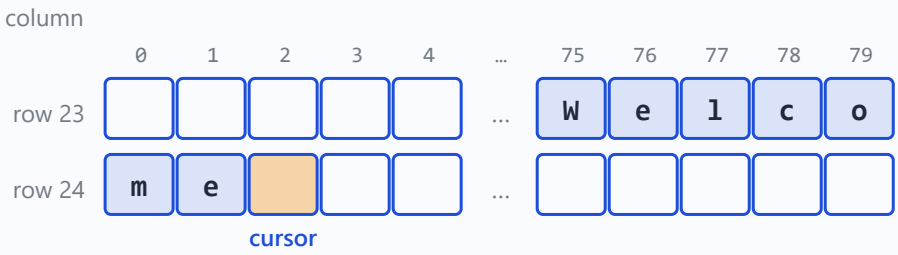


Figure 9.2.5: The last two rows of the screen: "Welco" in columns 75–79 of row 23, "me" at the start of row 24; the cursor at (2, 24).

GOTOXY and PUTC do not check positions. For a call such as `GOTOXY 85, 3` the BIOS's behavior is not defined; the cursor may go somewhere invisible and later writes appear in unexpected places. The programmer must keep positions within 0–79 (column) and 0–24 (row).

9.2.2.9 GOTOXY and video memory

A chosen cell on the screen can be described in three different ways: with the macro call (`GOTOXY 10, 5`), with the register values that go to the BIOS (AH = 02h, DH = 5, DL = 10, BH = 0), and with the cell's address in video memory. The last is useful for understanding what happens inside the BIOS.

In video memory the cells are laid out row by row, from left to right. Since each cell takes two bytes, the offset of the (column, row) cell is computed as follows:

`offset = (row × 80 + column) × 2`

The byte at an even address is the ASCII code of the character, and the byte at an odd address is the attribute byte. For `GOTOXY 10, 5` followed by `PUTC 'A'` the computation is: (5 × 80 + 10) × 2 = 410 × 2 = 820 = 0334h. 'A' (41h) is written to B800:0334h and the attribute byte (07h, light gray on black) to B800:0335h. The physical address is B8000h + 0334h = B8334h. For the bottom-right corner of the screen, (79, 24), the offset is (24 × 80 + 79) × 2 = 3998 = 0F9Eh. A page of 80 × 25 cells takes 4000 bytes.

Knowing these addresses also makes it possible to do what the macros do by writing to memory directly. For example, the sequence `MOV AX, 0B800h` , `MOV ES, AX` , `MOV BYTE PTR ES:[0334h], 'A'` writes 'A' to the same cell. The difference is that the cursor does not advance and the screen does not scroll: direct writing leaves all the details the BIOS takes care of to the programmer. The library macros are slower but do not require thinking about these details.

9.2.2.10 Printing two digits: a code trace

That PUTC preserves AX may look like a small detail. The example below shows why this detail is useful. The program prints a number from 0 to 99 in AL as two decimal digits. The `AAM` instruction is used to separate the digits. In the code the expansions of two PUTCs are written out explicitly instead of as macros:

```
ORG  100h
MOV  AL, 58
AAM              ; AH = 5, AL = 8
ADD  AX, 3030h   ; '5', '8'
PUSH AX          ; PUTC AH expansion
MOV  AL, AH
MOV  AH, 0Eh
INT  10h
POP  AX          ; AX restored: 3538h
PUSH AX          ; PUTC AL expansion
MOV  AL, AL
MOV  AH, 0Eh
INT  10h
POP  AX
RET
```

Definition 9.2.4

AAM (ASCII adjust after multiply) divides AL by 10: it writes the quotient to AH and the remainder to AL. For a number from 0 to 99, the tens digit is left in AH and the units digit in AL. SF, ZF and PF are set according to AL. On the 8086, AAM takes 83 clock cycles.

`MOV AL, 58` puts 58 = 3Ah in AL. `AAM` divides 58 by 10: AH = 5, AL = 8, that is, AX = 0508h. Since only AL is divided, the number must be in the range 0–99; from 100 on, the quotient is no longer a single digit. `ADD AX, 3030h` adds 30h to both bytes at once. Adding 30h to a digit converts it to an ASCII digit: 5 + 30h = 35h = '5', 8 + 30h = 38h = '8'. No carry arises between the two bytes in the addition, because 08h + 30h = 38h fits in 8 bits.

The first expansion prints the tens digit. `PUSH AX` puts 3538h on the stack, SP = FFFCh. `MOV AL, AH` copies '5' to AL and `MOV AH, 0Eh` loads the service number; AX = 0E35h. The units digit has been erased from the register. `INT 10h` prints 5 on the screen. `POP AX` brings back 3538h; '8' is in AL again. The second expansion prints the units digit with the same pattern. `MOV AL, AL` changes nothing but is harmless; the macro produces this instruction when the parameter is AL.

Instruction(s)	AH	AL	SP	[FFFCh]	Screen
`MOV AL, 58`	00h	**3Ah**	FFFEh	—	
`AAM`	**05h**	**08h**	FFFEh	—	
`ADD AX, 3030h`	**35h**	**38h**	FFFEh	—	
`PUSH AX`	35h	38h	**FFFCh**	**3538h**	
`MOV AL, AH` / `MOV AH, 0Eh` / `INT 10h`	**0Eh**	**35h**	FFFCh	3538h	5
`POP AX`	**35h**	**38h**	**FFFEh**	—	5
second expansion (5 instructions)	35h	38h	FFFEh	—	58

Table 9.2.8: Trace of printing two digits with AAM and PUTC.

At the end of the program 58 is on the screen and AX is again 3538h (Table 9.2.8). Because PUTC preserves AX, the two digits could be printed one after the other without being copied to a separate register. Had the macro not preserved AX, the first PUTC would have erased '8' and the programmer would have had to move the units digit into another register beforehand. Written with macros in the source, the same program contains only the lines `PUTC AH` and `PUTC AL`.

9.2.3 Using the procedures

9.2.3.1 DEFINE_ and CALL

Using a procedure from the library takes two steps. First the procedure is defined once: for this, the macro whose name starts with `DEFINE_` is written. Then the procedure is called with `CALL` wherever the program needs it. A typical skeleton looks like this:

```
include 'emu8086.inc'
ORG 100h
    ...
    CALL print_num   ; use
    ...
RET
DEFINE_PRINT_NUM    ; define
DEFINE_PRINT_NUM_UNS
END
```

Definition 9.2.5

A **DEFINE_ macro** is a macro whose body carries the entire code of a library procedure. Where the macro is expanded, it places the procedure as a `PROC … ENDP` block. Each procedure is defined at most once in a program; if it is defined twice, the same label is created twice and the assembler reports an error.

The place where the macro is expanded is the procedure's place in memory. In the source the DEFINE_ lines are usually written between the main program's final `RET` and the `END` directive (Figure 9.2.6). In memory the main code comes first, then the RET, then the code of the procedures. The processor runs the main code from top to bottom, returns to DOS with the RET, and enters the code of the procedures only through CALL.

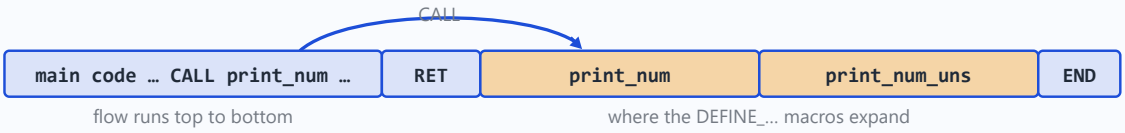


Figure 9.2.6: In memory the main code and RET come first, followed by the procedures expanded by the DEFINE_ macros; CALL jumps to the procedure.

This layout is the usual one but not the only safe place. Every DEFINE_ macro in emu8086.inc contains a guard: at the start of the body there is a `JMP` that jumps over the procedure. The structure of DEFINE_SCAN_NUM is roughly this:

```
JMP skip_proc_scan_num   ; keep the flow out of the body
SCAN_NUM PROC NEAR
    ...                  ; the procedure's code
    RET
SCAN_NUM ENDP
skip_proc_scan_num:
```

Even if a DEFINE_ line is written by mistake right below `ORG 100h`, the program works correctly. The first instruction the processor runs is this JMP, which jumps past the body to the start of the main program. The price is an extra JMP instruction in the program. Still, collecting the DEFINE_ lines between RET and END keeps the program readable and avoids needless jumps.

Procedure names are written in lowercase in the source: `CALL print_num`, `CALL scan_num`. emu8086 does not distinguish case in labels; `CALL PRINT_NUM` goes to the same label. Lowercase is the notation used in the documentation, and it visually separates the procedure name from the macro name (`DEFINE_PRINT_NUM`).

If a procedure calls another procedure, the two must be defined together. After printing the sign of the number, PRINT_NUM has PRINT_NUM_UNS print the absolute value. If only `DEFINE_PRINT_NUM` is written, the assembler cannot find the label at the line `CALL PRINT_NUM_UNS` inside PRINT_NUM and reports an undefined-label error. The reverse causes no error: PRINT_NUM_UNS can be defined alone.

Common mistake

Writing `CALL scan_num` and forgetting `DEFINE_SCAN_NUM`. That the library has been included does not mean the procedure is defined; unless the macro is expanded, the label scan_num does not exist. emu8086 reports an "undefined" error for this label during assembly.

The memory cost of a macro and a procedure can be compared with a concrete example. Suppose a program prints ten different messages. With PRINT every message carries its own loop code: every expansion holds 23 bytes of code plus the bytes of the message. With PRINT_STRING the loop exists once in memory; for each message only a 3-byte `LEA SI, msg` and a 3-byte `CALL print_string` are added. With ten seven-character messages the program takes about 310 bytes using PRINT and about 160 bytes using PRINT_STRING. The price of the saving is time: every call spends an extra 27 clock cycles on CALL and RET.

9.2.3.2 Input and output registers

A procedure receives its parameters in registers and returns its result in a register. What each register means is written in the procedure's documentation, and the programmer must follow it (Table 9.2.9).

Procedure	Input	Output	Details
PRINT_STRING	DS:SI = address of the string	—	the string must end with 0
PTHIS	the string right after the CALL	—	advances the return address itself
GET_STRING	DS:DI = buffer, DX = size	a 0-terminated string in the buffer	at most DX − 1 characters
CLEAR_SCREEN	—	—	the cursor goes to (0, 0)
SCAN_NUM	keyboard	CX (signed)	the old value in CX is lost
PRINT_NUM	AX (signed)	—	PRINT_NUM_UNS must be defined too
PRINT_NUM_UNS	AX (unsigned)	—	0 … 65,535

Table 9.2.9: Input and output registers of the library procedures.

The library's procedures push the other registers they use on the stack on entry and pop them on exit. For example GET_STRING saves AX, CX, DI and DX, and SCAN_NUM saves DX, AX and SI. The only register that changes for the calling program is the output register. The flags, however, are not preserved: the CMP and arithmetic instructions inside the procedure change them. A flag set before a CALL must not be relied on after the CALL.

The output register has a consequence too. Because SCAN_NUM writes its result to CX, the old value in CX is lost. A program that keeps a loop counter in CX must save the counter itself before calling SCAN_NUM:

```
MOV  CX, 3          ; three numbers to read
again: PUSH CX        ; save the counter
CALL scan_num       ; CX = the number read
MOV  AX, CX
ADD  BX, AX         ; add to the sum
POP  CX             ; restore the counter
LOOP again
```

Without the PUSH CX, LOOP would use the number the user entered as the counter. If the user enters 0, LOOP decrements CX to FFFFh and the loop runs another 65,535 times.

9.2.3.3 PRINT_STRING and PTHIS

The library has two procedures that print strings. Both print a string marked at its end with a 0 byte; they differ in where they get the string's address.

Definition 9.2.6

A **null-terminated string** is a string that ends with a byte of value 0 after its characters. The length is not stored separately; the code that reads the string stops when it sees the 0 byte. Strings in the C language have this form too.

PRINT_STRING takes the string's address from DS:SI. Its use takes two instructions:

```
LEA  SI, msg
CALL print_string
...
msg DB 'Hello', 0
```

`LEA SI, msg` loads the offset of msg into SI. emu8086 assembles this LEA with a direct address to the same 3-byte code as `MOV SI, OFFSET msg` (BE ll hh); in real MASM the same instruction is 4 bytes. The procedure saves AX and SI, then reads the bytes that SI points to in turn. If a byte is not 0 it is printed with service 0Eh and SI is incremented. When 0 is read, SI and AX are restored and the procedure returns. The string sits in the program's data area; the same string can be printed again in different places of the program.

PTHIS takes its name from the English phrase *print this*. It takes the string's address not from a register but from the stack. The string is written right after the CALL instruction:

```
CALL pthis
DB 'Hello', 0
; execution continues here
```

The PRINT macro copies its loop code at every use; PRINT_STRING and PTHIS print any number of strings with a single copy of code. PTHIS's advantage over PRINT_STRING is convenience: there is no need to name the string and define it in a separate data area.

9.2.3.4 Ways to print a string

With the library there are four ways to print a string on the screen (Table 9.2.10). The fourth is a DOS service outside the library but is included in the table for comparison.

Method	Where is the string?	Terminator	Address	Suitable when
`PRINT 'text'`	inside the code (macro expansion)	0 (added by the macro)	the macro takes it itself (CS:SI)	a constant message printed once
PRINT_STRING	in the data area	0	DS:SI	a string printed in several places or created at run time

Method	Where is the string?	Terminator	Address	Suitable when
PTHIS	right after the CALL	0	the return address on the stack	many short constant messages
INT 21h, AH = 09h	in the data area	'$'	DS:DX	a DOS program without the library

Table 9.2.10: Comparison of the ways to print a string.

The terminator itself cannot be printed. In a string ending in '$' the '$' character, and in a string ending in 0 the 0 byte, counts as the end of the string. In a program that prints prices, the message "100 $" cannot be printed with INT 21h/09h; the message ends before the '$'. The same message prints without trouble with PRINT_STRING.

The buffer must match the method too. The buffer that GET_STRING fills ends with 0 and is printed directly with PRINT_STRING. If the same buffer is printed with INT 21h/09h, DOS looks for '$'. Because there is no '$' in the buffer, DOS dumps the bytes beyond the buffer onto the screen as well and continues until it meets the first '$'. Meaningless characters appear on the screen.

9.2.3.5 How does PTHIS work?

PTHIS relies on knowing what CALL pushes on the stack. CALL pushes the address of the instruction after it, that is, the **return address**, on the stack and jumps to the procedure. The processor does not check whether there is an instruction at the "next instruction"; it merely adds the length of the CALL to IP. If a string sits right after the CALL, the pushed address is the address of the string's first byte. PTHIS uses this address as the address of the string.

Figure 9.2.7: After CALL pthis the stack holds the address of the string's first byte (0Dh) (left). After PTHIS has printed the string, it replaces the address on the stack with one past the 0 byte, the address of CALL print_num (right).

The procedure does four things (Figure 9.2.7). First it takes the return address from the stack into SI with `POP SI`. Then it reads and prints the bytes SI points to until a 0; SI increases after every read. When 0 is read, SI points one past the 0 byte: this address is the first instruction after the string. The procedure puts this address back on the stack with `PUSH SI`. Finally `RET` takes the new address on the stack into IP. The processor skips the string and continues from the instruction after it; the string's bytes are never mistaken for instructions and executed.

The PTHIS in the library also preserves the caller's SI and AX. AX is pushed on the stack. SI cannot be pushed, because the return address sits on top of the stack and has to be taken first. The procedure therefore stores the old value of SI in a 2-byte variable in the code segment (`CS:temp1`) and restores

it from there at the end. Since the string's bytes are also in the code segment, they are read with `CS:[SI]`.

Tracing with addresses makes the mechanism concrete. In the example program that reads and prints a number, `CALL pthis` is at 010Bh and is 3 bytes long. After it comes a 21-byte string starting at 010Eh: 13, 10, "You have entered: " and 0. The string occupies 010Eh–0122h; `CALL print_num` is at 0123h.

Moment	SP	Top of stack	SI	Explanation
before `CALL pthis`	FFFEh	0000h	msg1	IP = 010Bh
after `CALL pthis`	**FFFCh**	**010Eh**	msg1	return address = the string's first byte
after `POP SI`	**FFFEh**	0000h	**010Eh**	the old SI is in CS:temp1
when 0 is read	FFFEh	0000h	**0123h**	21 bytes read, 20 characters printed
after `PUSH SI`	**FFFCh**	**0123h**	0123h	the new return address
after `RET`	**FFFEh**	0000h	msg1	IP = 0123h: CALL print_num

Table 9.2.11: The stack and SI during the PTHIS call (the read-a-number-and-print-it program).

The "0000h" in the table is the word DOS puts on the stack when a .COM program starts (Table 9.2.11). In the library version AX is also pushed after the POP SI, so the stack's inner state is one word deeper than shown in the table; the table shows only the steps concerning the return address.

The same idea exists in the PRINT macro: PRINT also embeds the data in the code. The difference is that PRINT jumps over the data with a JMP, whereas PTHIS changes the address RET will return to. Changing the return address is a powerful but dangerous technique. If the string is not terminated with 0, PTHIS also prints the instruction bytes after the string and returns to a random place where it finds a 0; the program may hang.

9.2.3.6 GET_STRING and CLEAR_SCREEN

GET_STRING reads a line of text from the user. The characters read are written to the memory area that DS:DI points to. This area is called a **buffer**; the programmer reserves it beforehand with a definition such as `DB n DUP(0)`. DX is the size of the buffer in bytes. Because the procedure puts a 0 at the end of the string, it accepts at most DX − 1 characters. Reading ends when the user presses Enter.

CLEAR_SCREEN clears the screen and takes the cursor to the top-left corner, (0, 0). It has no input or output register. The procedure first fills the whole screen with blank lines using service 06h of INT 10h (scroll window up), then moves the cursor to (0, 0) with service 02h. It reads the number of rows and columns of the screen from the BIOS data area: the byte at 0040:0084h holds the number of rows minus one, and the word at 0040:004Ah the number of columns. For this it sets DS temporarily to 0040h; it saves DS and the other registers it uses on the stack.

```
LEA  DI, buf          ; buffer
MOV  DX, 20          ; size
CALL get_string
CALL clear_screen
LEA  SI, buf
CALL print_string
RET
buf DB 20 DUP(0)
```

This program reads a name of at most 19 characters, clears the screen and prints the name in the top-left corner. Because GET_STRING ends the buffer with 0, the same buffer is given to PRINT_STRING without any conversion. For the three procedures the program uses, the lines `DEFINE_GET_STRING`, `DEFINE_CLEAR_SCREEN` and `DEFINE_PRINT_STRING` must be added after the RET.

9.2.3.7 The logic of GET_STRING: a code trace

The GET_STRING in the library is long because it also handles backspace and a full buffer. The essence of the reading loop can be traced with the plain equivalent below. The buffer is 4 bytes and the user will type "Ece".

```
LEA  DI, buf      ; buffer address
MOV  DX, 4       ; buffer size
get_key: CMP DX, 1   ; keep 1 byte for the 0
JBE  done
MOV  AH, 01h
INT  21h         ; AL = key (echoed)
CMP  AL, 13      ; Enter?
JE   done
MOV  [DI], AL    ; store in the buffer
INC  DI
DEC  DX
JMP  get_key
done: MOV BYTE PTR [DI], 0
RET
buf  DB 4 DUP(?)
```

The program starts at 0100h. `LEA DI, buf` is 3 bytes, `MOV DX, 4` 3 and `CMP DX, 1` 3; the rest of the loop, `MOV BYTE PTR [DI], 0` and `RET` end at 011Ch. The buffer starts at 011Dh: buf+0 = 011Dh, buf+3 = 0120h.

The `CMP DX, 1` / `JBE done` pair at the start of the loop checks the remaining room. DX is the number of bytes left. If DX ≤ 1, only the final 0 has room and reading ends. JBE makes an unsigned comparison; it also jumps when DX is 0. Service 01h of INT 21h waits for a key, writes the key's ASCII code to AL and echoes the character to the screen. If it is not Enter (13), the character is written to the buffer, DI advances and the remaining room DX decreases by one.

Step	AL	DI	DX	ZF	Buffer (buf+0…3)	Screen
`LEA DI, ad` / `MOV DX, 4`	00h	**011Dh**	**0004h**	0	? ? ? ?	
`CMP DX, 1` / `JBE` (4 > 1, no jump)	00h	011Dh	0004h	0	? ? ? ?	
`INT 21h` ('E'), `CMP AL, 13`	**45h**	011Dh	0004h	0	? ? ? ?	E
`MOV [DI], AL` / `INC DI` / `DEC DX`	45h	**011Eh**	**0003h**	0	E ? ? ?	E
2nd pass ('c')	**63h**	**011Fh**	**0002h**	0	E c ? ?	Ec
3rd pass ('e')	**65h**	**0120h**	**0001h**	0	E c e ?	Ece
`CMP DX, 1` / `JBE done` (jumps)	65h	0120h	0001h	**1**	E c e ?	Ece
`MOV BYTE PTR [DI], 0`	65h	0120h	0001h	1	**E c e 0**	Ece

Table 9.2.12: Trace of the GET_STRING reading loop (DX = 4, input "Ece").

At the start of the fourth pass DX = 1: only the final 0 has room (Table 9.2.12). `CMP DX, 1` finds equality and ZF = 1; JBE jumps. Even if the user typed more, it would not be read; at most DX − 1 = 3 characters have been accepted. The buffer becomes 'E', 'c', 'e', 0 and can be printed with PRINT_STRING. The plain version ends by itself when the buffer is full; the version in the library, however, waits for Enter.

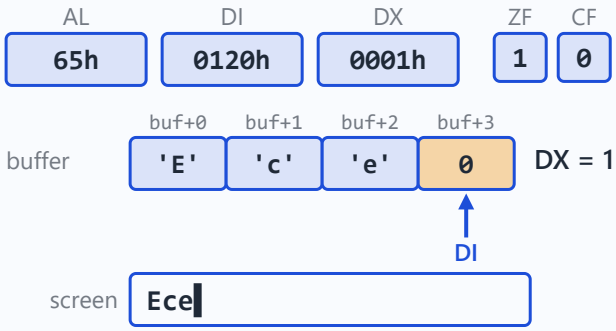


Figure 9.2.8: The buffer at the end of the loop: 'E', 'c', 'e' and the final 0; DI points to buf+3 (0120h).

9.2.3.8 GET_STRING in detail and backspace handling

The GET_STRING in the library differs from the plain version in four respects.

- **Echo control.** The key is read without echo: service 00h of INT 16h writes the key to AL (ASCII) and AH (scan code) but prints nothing on the screen. The procedure prints on the screen, with service 0Eh, only the character it has taken into the buffer. Keys that are not accepted thus leave no trace on the screen.
- **Full buffer.** After DX − 1 characters have been taken, new letters are ignored. The procedure does not finish by itself; it waits for Enter or backspace. The user can erase the last wrongly typed letter and correct it.

- **Backspace (8).** If there are letters in the buffer, DI is moved back by one and 8, ' ', 8 are printed on the screen. If the buffer is empty the key is ignored; the cursor does not go back past the start of the input and the prompt on the screen is not erased.
- **Enter (13).** Enter is not written to the buffer. A 0 is put in its place and the procedure returns. If DX ≤ 1 is given, no reading is done and the buffer is not touched.

The hardware side of keyboard reading should be known too. When a key is pressed, the keyboard controller activates the IRQ1 line of the 8259 interrupt controller. The processor drops what it is doing and runs the BIOS routine INT 09h. The routine reads the key's scan code from port 60h, finds its ASCII equivalent and writes both to the keyboard buffer in segment 0040h. INT 16h/00h reads from this buffer; if the buffer is empty it waits until a key arrives. The "waiting" of GET_STRING lies inside this loop.

When GET_STRING returns, the cursor is right after the text entered. The procedure does not print Enter on the screen; the next output starts on the same line. To move to the next line, `PUTC 13` and `PUTC 10` are added after the call.

The logic of backspace is again traced with a plain equivalent. The code below reads keys without echo, appends letters to the buffer, on backspace erases the last letter from the buffer and the screen, and on Enter ends the buffer with 0. The user will press A, b, backspace, c and Enter in turn.

```
ORG  100h
LEA  DI, buf
get_key: MOV AH, 0
INT  16h         ; AL = key (no echo)
CMP  AL, 13      ; Enter?
JE   done
MOV  AH, 0Eh
CMP  AL, 8       ; backspace?
JNE  store
CMP  DI, OFFSET buf
JE   get_key         ; no letter to erase
DEC  DI
INT  10h         ; 8: cursor left
MOV  AL, ' '
INT  10h         ; a space over the letter
MOV  AL, 8
INT  10h         ; left again
JMP  get_key
store: MOV [DI], AL
INC  DI
INT  10h         ; echo
JMP  get_key
done: MOV BYTE PTR [DI], 0
RET
buf  DB 6 DUP(?)
```

Two details of the code need attention. `MOV AH, 0Eh` is done before the comparisons; AH is thus ready for both the echo and the erase triple. Because MOV does not change the flags, the result of the CMP that follows is not affected. The `INT 10h` in the backspace branch moves the cursor left because AL still holds 8; the key's code is used directly as a control character.

Event	AL	DI	Cursor column	Buffer	Screen
`LEA DI, ad`	00h	buf+0	0	? ? ? ? ? ?	
INT 16h: 'A'; not Enter or backspace	**41h**	buf+0	0	? ? ? ? ? ?	
`store` : write, advance, echo	41h	**buf+1**	**1**	**A ? ? ? ? ?**	A
same path for 'b'	**62h**	**buf+2**	**2**	**A b ? ? ? ?**	Ab
INT 16h: backspace; `CMP AL, 8` → ZF = 1	**08h**	buf+2	2	A b ? ? ? ?	Ab
`CMP DI, OFFSET ad` (not equal) / `DEC DI`	08h	**buf+1**	2	A b ? ? ? ?	Ab
the 8, ' ', 8 triple	08h	buf+1	**1**	A b ? ? ? ?	A
'c': written over the byte of 'b'	**63h**	**buf+2**	**2**	**A c ? ? ? ?**	Ac
Enter: `JE done` , 0 is written	**0Dh**	buf+2	2	**A c 0 ? ? ?**	Ac

Table 9.2.13: Trace of reading that handles backspace (A, b, backspace, c, Enter).

`DEC DI` does not erase the byte of 'b' from memory; that byte merely falls outside the string and the next letter is written over it (Table 9.2.13). The erasing is done in two places at once: moving DI back in the buffer, and printing the triple on the screen. If one of the two is forgotten, the screen and the buffer no longer agree. For example, if the triple is forgotten, "Abc" stays on the screen and "Ac" in the buffer. With the same input the library version makes the buffer 'A', 'c', 0; it sends 'A', 'b', 8, ' ', 8 and 'c' to the screen, and "Ac" appears on the screen.

9.2.3.9 Buffer size and overflow

GET_STRING does not know the real size of the buffer; it trusts only the number the programmer gives in DX. If DX is larger than the buffer's real size, the procedure overwrites the variables after the buffer.

Example 9.2.2

In memory `buf DB 4 DUP(0)` is defined, followed by `value DW 0` and `last DB 5` . The program calls GET_STRING with `MOV DX, 8` and the user types "Istanbul" and presses Enter.

1. Since DX = 8, the procedure accepts at most 7 characters: "Istanbu". The eighth letter, 'l', is ignored.
2. The first four characters, "Ista", are written to the four bytes of the buffer.
3. 'n' (6Eh) and 'b' (62h) are written to the two bytes of value. Since the word is read little-endian, value = 626Eh.
4. 'u' (75h = 117) is written over last; last = 117.

5. The final 0 is written to the byte after last. That byte may belong to another variable or to code.

No error message appears in the program. value and last have been silently corrupted.

This error is called a **buffer overflow**. In assembly there is no mechanism that checks array bounds; every memory write looks only at the address. The safe rule is simple: DX is always given equal to the size reserved with DUP. One way to guarantee this automatically is to put a label right after the buffer and write `MOV DX, buf_end - buf`; then when the buffer is enlarged, DX changes by itself.

Common mistake

Taking DX to be "the number of characters to read". DX is the size of the buffer and includes the final 0. To read a name of 10 characters the buffer must be `DB 11 DUP(0)` and DX = 11. With `DB 10 DUP(0)` and DX = 10 only 9 characters are accepted.

9.2.4 Number procedures

9.2.4.1 SCAN_NUM, PRINT_NUM and PRINT_NUM_UNS

The processor keeps numbers in binary; the user, however, writes and reads numbers with decimal digits. The "307" that comes from the keyboard is three separate ASCII codes: 33h, 30h and 37h. For the program to compute, these codes must be converted into the binary number 0133h. When printing a number on the screen the job runs the other way: the digits of 0133h are found and each is converted to an ASCII code. The library's three number procedures perform these two conversions (Table 9.2.14).

Procedure	What does it do?	Input / output	Definition
SCAN_NUM	reads a multi-digit signed number from the keyboard; ends with Enter	output: CX	`DEFINE_SCAN_NUM`
PRINT_NUM	prints the signed number in AX in decimal	input: AX	`DEFINE_PRINT_NUM` + `DEFINE_PRINT_NUM_UNS`
PRINT_NUM_UNS	prints the unsigned number in AX in decimal	input: AX	`DEFINE_PRINT_NUM_UNS`

Table 9.2.14: The number procedures.

The difference between the two print procedures is how the 16 bits in AX are interpreted. In the unsigned interpretation the 16 bits denote the range 0 … 65,535. In the signed interpretation two's complement is used and the range is −32,768 … 32,767. For numbers whose most significant bit is 0,

the two interpretations agree. For numbers whose most significant bit is 1, the signed value is found by subtracting 65,536 from the unsigned value.

Definition 9.2.7

In **two's complement** representation, −x is written by inverting all the bits of x and adding 1. In 16 bits this equals 65,536 − x. The 8086 instruction `NEG` takes the two's complement of a number.

For example, the unsigned value of FFE7h is 65,511. Because its most significant bit is 1, its signed value is 65,511 − 65,536 = −25. Seen the other way round, 25 = 0019h; inverting the bits gives FFE6h and adding 1 gives FFE7h. The same FFE7h is printed as "-25" by PRINT_NUM and as "65511" by PRINT_NUM_UNS. The bits do not change; only the interpretation does.

9.2.4.2 The logic of SCAN_NUM: a code trace

When a decimal number is read from left to right, every new digit shifts the previous result one place to the left. For "307", 3 is read; when 0 arrives, the result becomes 3 × 10 + 0 = 30; when 7 arrives, 30 × 10 + 7 = 307. The rule in short is *result ← result × 10 + digit*. This method is also known as Horner's method and does not require knowing the length of the number in advance. The essence of SCAN_NUM is the plain equivalent below; the minus sign and the overflow check have been left out.

```
ORG  100h
XOR  CX, CX      ; result = 0
get_key: MOV AH, 01h
INT  21h         ; AL = key (echoed)
CMP  AL, 13      ; Enter?
JE   done
SUB  AL, '0'     ; ASCII → digit
CBW              ; AX = digit
XCHG AX, CX      ; AX=result, CX=digit
MOV  BX, 10
MUL  BX          ; AX = result × 10
ADD  CX, AX      ; CX = result×10 + digit
JMP  get_key
done: RET
```

Every line of the code serves a purpose. `XOR CX, CX` clears CX; XORing a register with itself is shorter than `MOV CX, 0` (2 bytes against 3) and sets ZF to 1. `SUB AL, '0'` finds the value of the digit by subtracting 30h from the ASCII code. `CBW` extends AL to AX as a signed value; for digits 0–9, AH = 0. `XCHG AX, CX` swaps the two registers: the result moves to AX for the multiplication, and the digit waits in CX. The swap is necessary because `MUL` always expects the multiplicand in AX. `MUL BX` performs DX:AX ← AX × BX; as long as the result fits in 16 bits, DX stays 0. `ADD CX, AX` adds the product to the digit and writes the new result into CX again.

Step	AX	CX	BX	ZF	CX (decimal)	Screen
`XOR CX, CX`	0000h	0000h	0000h	1	0	
`MOV AH, 01h` / `INT 21h` : '3'	**0133h**	0000h	0000h	1	0	3
`CMP` / `JE` / `SUB AL, '0'`	**0103h**	0000h	0000h	**0**	0	3
`CBW` / `XCHG AX, CX`	**0000h**	**0003h**	0000h	0	3	3
`MOV BX, 10` / `MUL BX` / `ADD CX, AX`	0000h	0003h	**000Ah**	0	3	3
'0' is read, `SUB` , `CBW` , `XCHG`	**0003h**	**0000h**	000Ah	1	0	30
`MUL BX` / `ADD CX, AX`	**001Eh**	**001Eh**	000Ah	0	30	30
'7' is read, `SUB` , `CBW`	**0007h**	001Eh	000Ah	0	30	307
`XCHG` / `MUL BX` / `ADD CX, AX`	**012Ch**	**0133h**	000Ah	0	307	307
Enter: AL = 0Dh, `CMP AL, 13`	**010Dh**	0133h	000Ah	**1**	307	307
`JE done` / `RET`	010Dh	0133h	000Ah	1	307	307

Table 9.2.15: Trace of the SCAN_NUM logic (input 307, then Enter).

A few points of the table need explanation (Table 9.2.15). INT 21h/01h writes the key to AL but does not touch AH; AH stays at 01h, the service number, and AX appears as 0133h. When '0' is read, the result of SUB is 0 and ZF = 1. 30 × 10 = 300 = 012Ch; adding 7 gives CX = 307 = 0133h. When Enter is read, CMP finds equality, ZF = 1 and JE leaves the loop. The result is in CX, as in SCAN_NUM.

Multiplication is the most expensive step of this loop. On the 8086 `MUL r16` takes 118–133 clock cycles; at 5 MHz about 25 µs. For a three-digit number three multiplications are made. This time is negligible next to the speed at which the user presses keys. Still, multiplying by 10 can also be done with shifts. Since x × 10 = x × 8 + x × 2, the product can be computed with `SHL` and `ADD` instructions in far fewer cycles.

9.2.4.3 The checks in SCAN_NUM

The SCAN_NUM in the library adds five checks on top of the plain equivalent:

- **The minus sign.** The '-' key sets a sign flag. When Enter is pressed and the flag is set, `NEG CX` is applied.
- **A key that is not a digit.** The procedure first prints every key on the screen and then checks it. If the key is not in the range '0'–'9', it is erased from the screen (8, ' ', 8) and does not join the number.
- **Backspace.** It discards the last digit: CX ← CX / 10. The last character is erased from the screen as well.
- **Overflow.** If the number would not fit in 16 bits, the last digit is not accepted. If DX ≠ 0 after the multiplication or CF = 1 in the addition, the procedure restores the previous value and erases the last digit from the screen.
- **The echo of Enter.** Enter is also printed on the screen. Code 13 returns the cursor to the start of the line but does not move it down a line.

The last item is the most surprising behavior in practice. When SCAN_NUM returns, the cursor is at the start of the line that holds the number entered. Anything printed right afterwards overwrites the number entered and the prompt. This is why example programs add `PUTC 13`, `PUTC 10` after SCAN_NUM, or start the next message with 13, 10.

Input	CX	PRINT_NUM	Explanation
307	0133h	307	the usual case
-25	FFE7h	-25	NEG at Enter: 0019h → FFE7h
12a3	007Bh	123	'a' is erased from the screen and does not join the number
40000	9C40h	-25536	fits in 16 bits but exceeds the signed range
70000	1B58h	7000	the last 0 is not accepted: 7000 × 10 = 70,000 > 65,535
-40000	63C0h	25536	the two's complement of 9C40h is 63C0h

Table 9.2.16: Inputs given to SCAN_NUM, the result in CX and the value PRINT_NUM prints.

The last three rows of the table show a limit of the procedure (Table 9.2.16). Digits are accumulated as an unsigned 16-bit number and the sign is applied at the end. The overflow check looks only at the 65,535 limit. Inputs between 32,768 and 65,535 are therefore accepted without error but appear negative in the signed interpretation. For example 40,000 = 9C40h; its signed value is 40,000 − 65,536 = −25,536. When −40,000 is entered, the two's complement of 9C40h, 63C0h = 25,536, results, and an input with a minus sign gives a positive result. For a correct result the user's input must stay in the range −32,768 … 32,767.

9.2.4.4 The logic of PRINT_NUM_UNS: a code trace

The decimal digits of a number are found by dividing it repeatedly by 10. The remainder of each division is a digit, and the quotient is the number for the next step. The problem is that the digits come out from right to left, that is, in reverse order: the first remainder of 1205 is 5 and the last is 1. On the screen, however, 1 must be printed first. The stack is a natural tool for this reversal: the remainders are pushed in order and then popped in reverse order.

```
ORG  100h
MOV  AX, 1205    ; the number to print
MOV  BX, 10
XOR  CX, CX      ; digit counter
divide: XOR DX, DX
DIV  BX          ; AX=quotient, DX=remainder
PUSH DX          ; digit onto the stack
INC  CX
OR   AX, AX      ; quotient 0?
JNZ  divide
emit: POP AX      ; most significant first
ADD  AL, '0'
MOV  AH, 0Eh
INT  10h
LOOP emit
RET
```

`DIV BX` divides the 32-bit DX:AX by BX: it writes the quotient to AX and the remainder to DX. `XOR DX, DX` before the division is mandatory. If the previous remainder is left in DX, the dividend DX:AX becomes too large; if the quotient does not fit in 16 bits, the processor generates interrupt type 0 (divide error). CX counts how many digits have been pushed; in the second loop, LOOP makes sure the same number of POPs is done. `OR AX, AX` does not change AX but sets the flags: if the quotient is 0, ZF = 1 and the loop ends.

Step	AX	DX	CX	SP	Stack (from the top)	Screen
start (`MOV` , `MOV` , `XOR`)	04B5h	0000h	0000h	FFFEh	—	
1st division: 1205 / 10	**0078h**	**0005h**	**0001h**	**FFFCh**	5	
2nd division: 120 / 10	**000Ch**	**0000h**	**0002h**	**FFFAh**	0, 5	
3rd division: 12 / 10	**0001h**	**0002h**	**0003h**	**FFF8h**	2, 0, 5	
4th division: 1 / 10	**0000h**	**0001h**	**0004h**	**FFF6h**	1, 2, 0, 5	
`OR AX, AX` → ZF = 1, JNZ does not jump	0000h	0001h	0004h	FFF6h	1, 2, 0, 5	
1st print: POP, add '0', INT 10h, LOOP	**0E31h**	0001h	**0003h**	**FFF8h**	2, 0, 5	1
2nd print	**0E32h**	0001h	**0002h**	**FFFAh**	0, 5	12
3rd print	**0E30h**	0001h	**0001h**	**FFFCh**	5	120
4th print, LOOP ends	**0E35h**	0001h	**0000h**	**FFFEh**	—	1205

Table 9.2.17: Trace of digit extraction with the stack (AX = 1205 = 04B5h).

After four divisions the most significant digit, 1, is on top of the stack (Table 9.2.17). The print loop pops the digits in the order 1, 2, 0, 5 and prints them. After each POP, `ADD AL, '0'` converts the digit to ASCII and `MOV AH, 0Eh` loads the service number; this is where AX values such as 0E31h and 0E32h come from. When the loop ends, SP is FFFEh again: four POPs have matched four PUSHes. The stack column of the table does not show DOS's 0000h word at FFFEh.

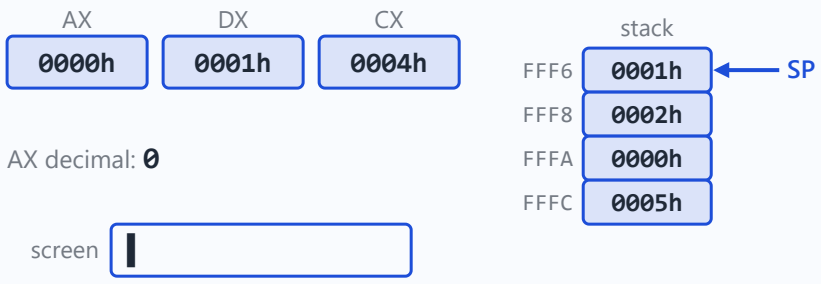


Figure 9.2.9: When the divisions are finished, the digits 1, 2, 0 and 5 sit on the stack from FFF6h toward FFFCh; SP = FFF6h.

9.2.4.5 The library's method: a decreasing divisor

PRINT_NUM_UNS in the library produces the same output with another method. Instead of always dividing the number by 10, it divides by the place values: first by 10,000, then by 1000, 100, 10 and 1. The quotient of each division is a digit and is printed the moment it is computed. The remainder becomes the number for the next pass. Because the digits come out from left to right, no stack or reversal is needed.

- The divisor BX starts at 10,000 and is divided by 10 after every pass. When BX becomes 0, printing ends.
- The quotient (0–9) is a digit; it is converted to ASCII with `ADD AL, 30h` and printed with PUTC. Since 65,535 ÷ 10,000 = 6, the quotient always fits in AL.
- CX is used as a flag. While CX = 1, a pass is skipped if the number is smaller than the divisor; this way leading zeros are not printed. Once the first digit has been printed CX becomes 0, and the zeros in between are printed from then on.
- There is a separate branch for AX = 0 that prints '0' directly. Without this branch every pass would be skipped and nothing would be printed on the screen.

Divisor (BX)	Flag (CX)	Number (AX)	Action	Screen
10.000	1	3007	number < divisor and no digit yet: skip	—
1000	1	3007	3007 ÷ 1000 = 3, remainder 7; CX = 0	3
100	0	7	7 ÷ 100 = 0, remainder 7	30
10	0	7	7 ÷ 10 = 0, remainder 7	300
1	0	7	7 ÷ 1 = 7, remainder 0	3007
0	0	0	divisor 0: done	3007

Table 9.2.18: Printing 3007 with a decreasing divisor (the state at the start of each pass).

The 3007 example shows clearly why the flag is needed (Table 9.2.18). In the first pass, since 3007 < 10,000, the quotient would be 0; this zero is a leading zero and must not be printed. In the passes that divide by 100 and 10 the quotient is also 0, but these zeros are inside the number and must be

printed. The difference is told apart only by the question "has the first digit been printed?"; CX holds this information. For 40,000, in the first pass 40,000 ÷ 10,000 = 4 is printed and CX becomes 0. In the next four passes four 0s are printed; 40000 appears on the screen.

Reducing the divisor is also a division: the library does `BX ← BX / 10` with DIV. During this the number in AX is saved on the stack. The procedure pushes AX, BX, CX and DX on entry and pops them on exit; no register changes for the caller.

9.2.4.6 PRINT_NUM and the minus sign: a code trace

PRINT_NUM uses PRINT_NUM_UNS to print a signed number. If the number is negative, '-' is printed first, and then the absolute value of the number is printed as unsigned. The code below is the equivalent of PRINT_NUM in the library:

```
ORG  100h
MOV  AX, -25       ; AX = FFE7h
CALL print_num
RET
print_num PROC
PUSH AX
OR   AX, AX        ; SF = sign bit
JNS  positive
NEG  AX            ; AX = 0019h
PUSH AX            ; PUTC '-' expansion
MOV  AL, '-'
MOV  AH, 0Eh
INT  10h
POP  AX
positive: CALL print_num_uns
POP  AX            ; AX = FFE7h restored
RET
print_num ENDP
```

At the line `MOV AX, -25` the assembler computes the two's complement of −25 and produces the instruction that loads FFE7h into AX. `CALL print_num` is at 0103h and is 3 bytes long; the return address 0106h is pushed on the stack. The procedure first saves AX, because NEG will change it. `OR AX, AX` sets the flags without changing the number: since the most significant bit is 1, SF = 1. `JNS` (jump if no sign) does not jump. `NEG AX` computes 0 − FFE7h = 0019h = 25; since the result is positive, SF = 0. The expansion of PUTC '-' prints the minus sign. `CALL print_num_uns` prints the digits of 25. Finally `POP AX` restores FFE7h into AX.

Instruction(s)	AX	SP	SF	Stack (FFFAh, FFFCh)	Screen
`MOV AX, -25`	**FFE7h**	FFFEh	0	—, —	
`CALL print_num`	FFE7h	**FFFCh**	0	—, 0106h	
`PUSH AX` / `OR AX, AX`	FFE7h	**FFFAh**	**1**	FFE7h, 0106h	

Instruction(s)	AX	SP	SF	Stack (FFFAh, FFFCh)	Screen
`JNS` (no jump) / `NEG AX`	**0019h**	FFFAh	**0**	FFE7h, 0106h	
PUTC '-' expansion (5 instructions)	0019h	FFFAh	0	FFE7h, 0106h	-
`CALL print_num_uns` (after it returns)	0019h	FFFAh	0	FFE7h, 0106h	-25
`POP AX`	**FFE7h**	**FFFCh**	0	—, 0106h	-25
`RET` (from print_num)	FFE7h	**FFFEh**	0	—, —	-25

Table 9.2.19: Trace of the PRINT_NUM equivalent (AX = −25).

After the number has been printed it is still in AX (Table 9.2.19). The calling program can go on using the same number for another job. Because PRINT_NUM_UNS saves all the registers it uses, the steps inside it are reduced to a single row of the table. The PRINT_NUM in the library also contains a branch that prints '0' directly for AX = 0. It also tests the sign with `CMP AX, 0` instead of `OR`; the two instructions set SF in the same way.

One consequence of this design is remarkable: for −32,768 (8000h), NEG again produces 8000h, because +32,768 does not exist in the 16-bit signed range. PRINT_NUM first prints '-' and then gives 8000h to PRINT_NUM_UNS. PRINT_NUM_UNS prints 8000h as the unsigned 32,768, and the correct result, −32768, appears on the screen. Printing the absolute value as unsigned solves this edge case by itself.

9.2.4.7 Extracting digits: comparing the two methods

Criterion	Stack method (divide by 10, push, pop in reverse order)	Decreasing divisor (10,000, 1000, ...)
Number of divisions	as many as there are digits	up to five; plus one division each to reduce the divisor
Stack use	2 bytes per digit	only the saved registers
Leading zeros	do not arise by themselves	skipped with the flag
Fixed width (00042)	extra code needed	automatic when the flag is cleared
Other bases	the divisor becomes 2, 8 or 16; in base 16 digits above 9 become 'A'–'F'	the divisor sequence changes with the base (4096, 256, 16, 1 in base 16)
Output order	first onto the stack, then to the screen in reverse order	left to right, the moment it is computed

Table 9.2.20: Comparison of the stack method and the decreasing-divisor method.

The two methods give the same output: for 1205 both print the digits 1, 2, 0, 5 (Table 9.2.20). The choice is decided by stack use, fixed width and the need for other bases. In time, the cost of DIV is decisive. On the 8086 `DIV r16` takes 144–162 clock cycles. For 1205 the stack method does 4 divisions. The decreasing-divisor method does four divisions by 1000, 100, 10 and 1, and five more

divisions to reduce the divisor; the division is skipped in the first pass. The stack method is therefore about twice as fast. The library's choice was made for not using the stack and for easy adaptation to fixed width.

9.2.4.8 The same bits, two interpretations

When the outputs of the two procedures for a value given in AX are put side by side, the difference of interpretation is clearly visible (Table 9.2.21).

AX	Most significant bit	PRINT_NUM	PRINT_NUM_UNS
0000h	0	0	0
7FFFh	0	32767	32767
8000h	1	-32768	32768
9C40h	1	-25536	40000
FFE7h	1	-25	65511
FFFFh	1	-1	65535

Table 9.2.21: The same 16 bits, two procedures.

The rule is single: if the most significant bit is 0, the two procedures print the same thing; if it is 1, PRINT_NUM prints the unsigned value minus 65,536. The program must know which interpretation is correct; the processor does not. The addition and subtraction instructions produce the same bits for both interpretations; only the reading of the flags changes. CF shows unsigned overflow and OF signed overflow. Values that cannot be negative, such as a counter, an address or a length, are printed with PRINT_NUM_UNS; values that can be negative, such as a temperature or a difference, with PRINT_NUM.

> **Common mistake**
>
> Printing an unsigned value that can exceed 32,767 with PRINT_NUM. AX = 40,000 (9C40h) appears as −25536 with PRINT_NUM. If the value's meaning is unsigned, PRINT_NUM_UNS must be used.

9.2.5 Example programs

9.2.5.1 Read a number and print it

The second example in emu8086's library documentation uses four procedures together. The program asks the user for a number, reads it and prints it back:

```
include 'emu8086.inc'
ORG    100h
LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.
MOV    AX, CX         ; copy the number to AX.
CALL   pthis          ; print the following string:
DB  13, 10, 'You have entered: ', 0
CALL   print_num      ; print number in AX.
RET                   ; return to operating system.
msg1   DB  'Enter the number: ', 0
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for print_num.
DEFINE_PTHIS
END                   ; directive to stop the compiler.
```

The flow of the program is short. `LEA SI, msg1` loads the address of the prompt string into SI and PRINT_STRING prints it. SCAN_NUM reads the number and puts it in CX. `MOV AX, CX` moves the number to the register PRINT_NUM expects. PTHIS prints the string after the CALL: since the string starts with 13, 10, it first moves to the next line. PRINT_NUM prints the number as signed and RET ends the program. The lines after msg1 are data and procedure definitions; the flow never reaches them after the RET.

The layout of the main program in memory is in Table 9.2.22. All the CALLs are 3-byte near calls (E8 and a 16-bit displacement). Note that the string sits in the middle of the code area: 010Eh–0122h is data, not instructions.

Address	Bytes	Line
0100h	3	`LEA SI, msg1` (BE 27 01)
0103h	3	`CALL print_string`
0106h	3	`CALL scan_num`
0109h	2	`MOV AX, CX` (8B C1)
010Bh	3	`CALL pthis`
010Eh	21	`DB 13, 10, 'You have entered: ', 0`
0123h	3	`CALL print_num`
0126h	1	`RET` (C3)
0127h	19	`msg1 DB 'Enter the number: ', 0`
013Ah	…	procedures expanded by the DEFINE_ macros

Table 9.2.22: Layout of the main part of the read-a-number-and-print-it program.

Suppose the user types -25 and presses Enter. In the trace table, two moments are shown for the CALL lines: inside the procedure (while the return address is on the stack) and the return (Table 9.2.23).

Line	SI	CX	AX	SP	Screen
`LEA SI, msg1`	**0127h**	(size)	0000h	FFFEh	
`CALL print_string` (inside)	0127h	(size)	0000h	**FFFCh**	Enter the number:␣
`CALL scan_num` (inside)	0127h	...	0000h	FFFCh	Enter the number: -25
`CALL scan_num` (return)	0127h	**FFE7h**	0000h	**FFFEh**	Enter the number: -25
`MOV AX, CX`	0127h	FFE7h	**FFE7h**	FFFEh	Enter the number: -25
`CALL pthis`	0127h	FFE7h	FFE7h	**FFFCh**	...
the string is printed	0127h	FFE7h	FFE7h	FFFCh	... / You have entered:␣
`CALL print_num` (inside)	0127h	FFE7h	FFE7h	FFFCh	... / You have entered: -25
`RET`	0127h	FFE7h	FFE7h	**0000h**	... / You have entered: -25

Table 9.2.23: Trace of the read-a-number-and-print-it program (input −25).

A few details of the table need explanation. SI stays at 0127h throughout the program, because PRINT_STRING and PTHIS save and restore SI. AX is also 0000h after SCAN_NUM; SCAN_NUM saves AX. The only register that changes is the output register CX: the two's complement of −25, FFE7h. The initial value of CX is shown as "(size)". When emu8086 loads a .COM program it puts the program's size in bytes into CX. This value is lost when SCAN_NUM clears CX and does not affect the result. The last RET takes the 0000h that DOS put on the stack into IP; SP = FFFEh + 2 = 0000h. The processor goes to the `INT 20h` at the start of the PSP and the program ends.

The final state of the screen is two lines. The first line reads "Enter the number: -25" and the second "You have entered: -25". Because SCAN_NUM echoed Enter only as CR, the cursor had returned to the start of the first line. The 13 and 10 at the start of the PTHIS string take the cursor to the start of the second line. Had the string not started with 13, 10, "You have entered: " would have been written over the first line and the prompt would have been erased.

9.2.5.2 The sum of two numbers

The second example reads two numbers and prints their sum. Signed overflow is checked too.

```
include 'emu8086.inc'
ORG  100h
CALL scan_num     ; 1st number → CX
MOV  BX, CX       ; save
PUTC 13
PUTC 10           ; next line
CALL scan_num     ; 2nd number → CX
PUTC 13
PUTC 10
MOV  AX, BX
ADD  AX, CX       ; signed sum
JO   overflow
CALL print_num
RET
overflow: PUTC '?'
RET
DEFINE_SCAN_NUM
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
END
```

Three details of the program are important. First, the second SCAN_NUM also writes its result to CX; so that the first number is not lost, it is moved to BX first. The choice of BX is not random: SCAN_NUM does not use BX, and PUTC touches only AX. Second, because Enter only returns to the start of the line, `PUTC 13` and `PUTC 10` follow every SCAN_NUM; the second number and the result are printed on separate lines. Third, `JO` catches signed overflow.

Definition 9.2.8

The **overflow flag** (OF) becomes 1 when the result of a signed operation does not fit in the 16-bit signed range (−32,768 … 32,767). In an addition OF = 1 if the two operands have the same sign and the result has the opposite sign. The `JO` instruction jumps if OF = 1.

Inputs	BX	CX	AX = BX + CX	OF	Last line
120, −45	0078h	FFD3h	004Bh	0	75
−7, −8	FFF9h	FFF8h	FFF1h	0	-15
30000, 5000	7530h	1388h	88B8h	**1**	?

Table 9.2.24: The sum-of-two-numbers program with different inputs.

In the first two rows the result is in the signed range (Table 9.2.24). In the addition 120 + (−45), CF = 1, because as unsigned numbers 0078h + FFD3h exceeds 16 bits. This carry is meaningless in signed arithmetic and OF stays 0. In the third row the sum of the two positive numbers is 35,000 and

exceeds 32,767. The most significant bit of the result 88B8h is 1; the sum of two positive numbers appears negative and OF = 1. JO jumps and '?' is printed on the screen. Without the JO, PRINT_NUM would have printed 88B8h as −30536.

9.2.5.3 A framed title

The third example clears the screen and draws a framed heading starting at column 10. The library macros are used together with the programmer's own procedure.

```
include 'emu8086.inc'
ORG  100h
CALL clear_screen
GOTOXY 10, 2
CALL rule
GOTOXY 10, 3
PRINT '|  REPORT  |'
GOTOXY 10, 4
CALL rule
GOTOXY 0, 6       ; cursor below
RET
rule PROC        ; +----------+
PUTC '+'
MOV  CX, 10
dash: PUTC '-'
LOOP dash
PUTC '+'
RET
rule ENDP
DEFINE_CLEAR_SCREEN
END
```

The rule procedure prints a '+', prints '-' ten times and ends with a '+': the 12-character "+----------+". The string in the heading line is also 12 characters: two edge bars, two spaces, "REPORT" and two spaces. The three lines are therefore aligned one above the other in columns 10–21 (Figure 9.2.10). The final `GOTOXY 0, 6` takes the cursor below the frame; when the program ends, whatever emu8086 prints does not land on top of the heading.

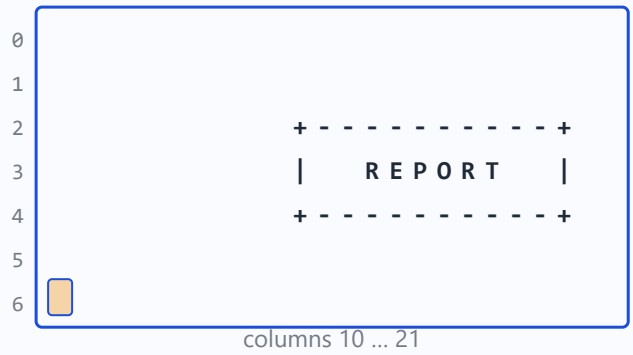


Figure 9.2.10: The top-left part of the screen: the framed REPORT heading in rows 2–4 starting at column 10; the cursor is at (0, 6).

Two points of the program show the relationship between a macro and a loop. The `PUTC '-'` in the loop is written once in the source, so it is expanded once; the 8-byte expansion runs 10 times but is not copied 10 times. The copying of the macro happens at assembly time, the repetition of the loop at run time. The second point concerns registers. The LOOP counter is in CX. Because PUTC uses only AX and restores it, it does not touch CX and the counter is safe. Had a procedure that clobbers CX, such as SCAN_NUM, been called in the loop, the counter would have had to be protected with PUSH CX / POP CX. The definition of rule is also after the RET, like the library procedures.

9.2.6 Common mistakes

The mistakes made when using the library are mostly not noticed during assembly; the program runs but gives a wrong result. The mistakes below are the ones students working with emu8086 meet most often.

Common mistake

Forgetting the DEFINE_ line. If there is a `CALL scan_num` but no `DEFINE_SCAN_NUM`, the assembler cannot find the label and reports an "undefined" error. For PRINT_NUM, `DEFINE_PRINT_NUM_UNS` is also needed.

Common mistake

Not terminating the string with 0. PRINT_STRING and PTHIS print until they see a 0. With a definition without 0 such as `msg DB 'Number: '`, the bytes after the string are also dumped to the screen. These are often procedure code, and meaningless characters appear on the screen. With PTHIS, in addition, the return address slips to a wrong place and the program may hang.

Common mistake

Looking for the result in AX. SCAN_NUM writes its result to CX. If `MOV AX, CX` is forgotten before PRINT_NUM, the old value in AX is printed. General rule: if the documentation of a procedure does not say that a register is preserved, the register must not be assumed to be preserved.

Common mistake

Expecting a new line after Enter. After SCAN_NUM the cursor is at the start of the line where the input is. The next output is written over the number entered and the prompt. If 42 is read with the prompt "Num: " and PRINT_NUM is called right away, "42m: 42" is left on the screen. `PUTC 13`, `PUTC 10` must be inserted in between.

Common mistake

Giving too large a buffer size. If DX is larger than the buffer's real size, GET_STRING overwrites the variables after the buffer. The error appears only when those variables are used.

The position of the DEFINE_ lines, however, is not a source of errors. Because every DEFINE_ macro begins with a JMP that jumps over the procedure body, the processor does not enter the body even if the line is put at the start of the program (Figure 9.2.11). The price is an extra jump instruction. The usual place is between RET and END.

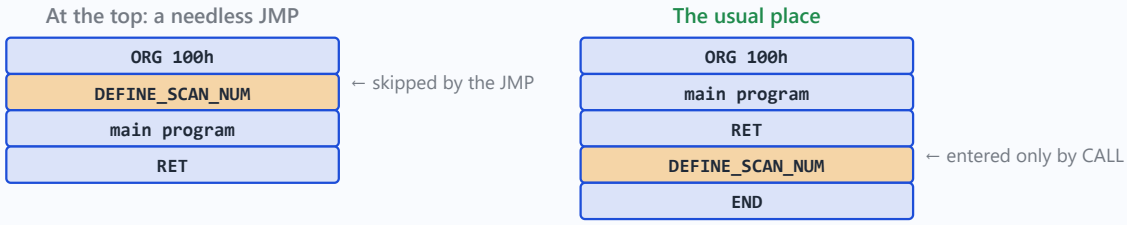


Figure 9.2.11: Even if DEFINE_SCAN_NUM is put below ORG 100h, the macro's first instruction, a JMP, skips the procedure body (left). The usual place is between RET and END; the procedure is entered only through CALL (right).

9.2.7 Exercises and solutions

Exercise 1. Write, with macros, the program that prints "CENTER" in the middle of the screen (column 35, row 12).

```
include 'emu8086.inc'
ORG 100h
GOTOXY 35, 12
PRINT 'CENTER'
RET
```

"CENTER" is written to columns 35–40 of row 12 and the cursor stays at (41, 12). For a six-character text to sit exactly in the middle, the start column must be (80 − 6) / 2 = 37; 35 is an approximate middle. General rule for centering: start column = (80 − length) / 2.

Exercise 2. Write the program that reads two numbers from the user and prints their sum.

The solution is the sum-of-two-numbers example. The parts needed: `CALL scan_num` → `MOV BX, CX` → `PUTC 13`, `PUTC 10` → `CALL scan_num` → `MOV AX, BX` → `ADD AX, CX` → `CALL print_num`; the definitions `DEFINE_SCAN_NUM`, `DEFINE_PRINT_NUM` and `DEFINE_PRINT_NUM_UNS`. If PUTC 13 and PUTC 10 are forgotten, the second number is written over the first and only the last things written can be read on the screen.

Exercise 3. If the PRINT macro is used 10 times, how many times do the strings and the loop code exist? Compare with PRINT_STRING.

With PRINT there are 10 copies: every expansion carries its own string and its own 23-byte loop code. With PRINT_STRING the loop code exists once; memory holds 10 strings and, for each string, one `LEA SI` and one `CALL` (3 + 3 bytes). With ten seven-character messages the program with PRINT is about 310 bytes and the one with PRINT_STRING about 160 bytes. The gap grows as the number of messages grows; on the other hand every CALL/RET pair adds 27 clock cycles.

Exercise 4. What do PRINT_NUM and PRINT_NUM_UNS print when AX = 8000h?

The unsigned value of 8000h is 32,768; PRINT_NUM_UNS prints "32768". Because its most significant bit is 1, its signed value is 32,768 − 65,536 = −32,768; PRINT_NUM prints "-32768". This is the smallest number of the 16-bit signed range, and NEG does not change it; PRINT_NUM still prints correctly, because it prints the absolute value as unsigned.

Exercise 5. What happens if `DEFINE_PTHIS` is written right below `ORG 100h`?

The program still works correctly. The first instruction of the macro is `JMP skip_proc_pthis`; the processor jumps over the body of PTHIS and reaches the first instruction of the main program. The price is an extra JMP, and the addresses of the main program shift forward by the size of the body. Without the protective JMP the processor would have entered the code of PTHIS directly: `POP SI` would take DOS's 0000h word from the stack, print the bytes of the PSP as a string and return to an invalid address.

Exercise 6. Get the user's name with GET_STRING and print "Hello, name!".

```
include 'emu8086.inc'
ORG 100h
LEA  DI, buf
MOV  DX, 20
CALL get_string
PUTC 13
PUTC 10
PRINT 'Hello, '
LEA  SI, buf
CALL print_string
PUTC '!'
RET
buf DB 20 DUP(0)
DEFINE_GET_STRING
DEFINE_PRINT_STRING
END
```

If the user types "Ece" and presses Enter, "Ece" appears on the first line of the screen and "Hello, Ece!" on the second. Without PUTC 13 and PUTC 10, after GET_STRING the cursor would stay right after "Ece" and "EceHello, Ece!" would appear on a single line. The buffer is 20 bytes and DX = 20; a name of at most 19 characters is accepted.

9.2.8 Summary

- **emu8086.inc** is the library that contains ready-made macros and procedures for screen, keyboard and number jobs. Everything in it is ordinary 8086 code built on BIOS and DOS interrupts.
- **INCLUDE** is a directive: it puts the text of the file in place of the line during assembly and produces no code. The file is searched for first in the source folder, then in emu8086's Inc folder. The line is written at the start of the source.
- A macro definition and a procedure whose DEFINE_ has not been written add no bytes to the program. Program size is determined by every use of the macros that are used and the procedures defined with DEFINE_.
- **The screen macros** are built on INT 10h: PUTC and PRINT use service 0Eh, GOTOXY 02h, CURSOROFF and CURSORON 01h. The macros save the registers they use; PUTC is 8 bytes, GOTOXY 16 bytes.
- **GOTOXY** uses the column, row order; the BIOS expects the row in DH and the column in DL. The top-left is (0, 0), the bottom-right (79, 24). The address of a cell in video memory is B800h:(row × 80 + column) × 2.
- **PRINT** embeds the string in the code and jumps over it with a JMP; the loop code is copied at every use. **PRINTN** appends 13, 10. A new line needs CR (13) and LF (10) together. After column 79 the cursor moves to the next line, and on the last row the screen scrolls.

- **Procedures** are defined once with DEFINE_ and called with CALL. Every DEFINE_ macro begins with a JMP that jumps over the body; the usual place is between RET and END. PRINT_NUM also requires PRINT_NUM_UNS.
- Input and output registers: PRINT_STRING DS:SI, GET_STRING DS:DI and DX, SCAN_NUM → CX, PRINT_NUM and PRINT_NUM_UNS ← AX. The other registers are preserved, the flags are not.
- **PTHIS** uses the return address that CALL pushed on the stack as the address of the string; after printing the string it moves the address past the 0, and RET returns to the instruction after the string.
- **GET_STRING** reads at most DX − 1 characters, ends the buffer with 0, handles backspace and does not print Enter. If DX is given larger than the real size of the buffer, the variables after the buffer are overwritten.
- **SCAN_NUM** accumulates digits with the rule result ← result × 10 + digit; it applies the minus sign at the end with NEG, rejects keys that are not digits and digits that push the number past 65,535. Enter returns only to the start of the line.
- **PRINT_NUM_UNS** prints the digits left to right with a decreasing divisor (10,000, 1000, …); it skips leading zeros with a flag. The stack method gives the same result by dividing by 10 and popping the digits from the stack in reverse order. **PRINT_NUM** prints '-' for a negative number and then the absolute value as unsigned.
- The same 16 bits are read in two ways: if the most significant bit is 1, the signed value = the unsigned value − 65,536 (FFE7h = −25 = 65,511).

9.2.9 Key terms

library directive macro macro expansion procedure procedure call return address

parameter passing register preservation cursor string control character ASCII BIOS service

video memory (text mode) keyboard buffer signed / unsigned number two's complement label

---

9.2.10 Test Yourself

1. In the expansion of `GOTOXY 3, 7`, what are AH, BH, DH and DL just before INT 10h is called? How many bytes is the expansion and which registers does it save?

   Answer

   AH = 02h, BH = 0, DH = 7 (row), DL = 3 (column). The expansion is 16 bytes: three PUSHes and three POPs take 6 bytes, four `MOV r8, constant` take 8 bytes, and `INT 10h` takes 2 bytes. AX, BX and DX are saved on the stack and restored in reverse order.

2. In colour text mode, at which offset in video memory does the character in cell (35, 12) sit? What is its physical address?

   Answer

   (12 × 80 + 35) × 2 = 995 × 2 = 1990 = 07C6h. The character is at B800:07C6h and the attribute byte at B800:07C7h. The physical address is B8000h + 07C6h = B87C6h.

3. On an empty screen with the cursor at (0, 0), `PRINTN 'Hello'` and then `PUTC 'X'` run. Which cell is X written to? Where is the cursor at the end?

   Answer

   "Hello" is written to columns 0–4 of row 0. The 13 that PRINTN appends takes the cursor to (0, 0) and the 10 to (0, 1). X is written to (0, 1) and the cursor moves to (1, 1).

4. While SCAN_NUM runs, the user presses the keys 5, b, 0 and Enter in turn. What is CX when the procedure returns? What is left on the screen?

   Answer

   Because 'b' is not a digit, it is printed on the screen, then erased with 8, ' ', 8 and does not join the number. The number becomes 5 × 10 + 0 = 50: CX = 0032h. "50" is left on the screen; Enter returns the cursor to the start of the line.

5. What does the stack digit-extraction code (divide by 10, push, pop with LOOP) print for AX = 0? Why does the library's decreasing-divisor method need a separate branch for AX = 0?

   Answer

   The stack code enters the loop at least once: 0 / 10 = 0, the remainder 0 is pushed and CX = 1. OR AX, AX makes ZF = 1, the print loop runs once and "0" is printed. In the decreasing-divisor method, while the flag CX = 1 a pass is skipped if the number is smaller than the divisor. Since AX = 0 is smaller than every divisor, all the passes are skipped and nothing is printed; the separate branch therefore prints '0'.

6. GET_STRING is called with `MOV DX, 1` and the user types "abc" and presses Enter. What is written to the buffer?

   Answer

   Nothing. With DX ≤ 1 the buffer has room only for the final 0, and the procedure returns without reading anything; not even a 0 is written to the buffer, and the old contents stay as they were. The keys pressed wait in the keyboard buffer and the next read service picks them up.

7. In a .COM program `CALL pthis` is at 0100h, followed right after by `DB 'Hello', 13, 10, 0`. Which address does CALL push on the stack? Which address does the RET of PTHIS return to?

   Answer

   CALL is 3 bytes; 0103h is pushed on the stack, the address of the string's first byte, 'H'. The string is 5 + 2 + 1 = 8 bytes and occupies 0103h–010Ah. When PTHIS reads the 0 at 010Ah, SI = 010Bh; this value is put on the stack and RET returns to 010Bh.

8. What do PRINT_NUM and PRINT_NUM_UNS print when AX = C350h?

   Answer

   C350h = 50,000; PRINT_NUM_UNS prints "50000". Because the most significant bit is 1, the signed value is 50,000 − 65,536 = −15,536; PRINT_NUM prints "-15536".

9.2.11 Further reading

- The `Inc/emu8086.inc` file in the emu8086 installation: the source code of all the macros and procedures in the library. The overflow check of SCAN_NUM and the decreasing-divisor loop of PRINT_NUM_UNS can be read there line by line.
- The emu8086 help documents, the pages "emu8086.inc — library of common functions" and "Macros": usage examples of the library and the general rules of macros.
- B. B. Brey, *The Intel Microprocessors* (8th ed.): procedures, macros and BIOS video services; converting decimal numbers to binary and back.
- R. Brown, "Ralf Brown's Interrupt List": a detailed list of the INT 10h, INT 16h and INT 21h services with their input and output registers.

9.2.12 References

1. emu8086 help documents: the pages "emu8086.inc — library of common functions" and "Macros".
2. The Inc/emu8086.inc file in the emu8086 installation (source code of the macros and procedures).
3. B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — procedures and macros.
4. A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.
5. Intel, *The 8086 Family User's Manual*, 1979 — definitions of the CALL, RET, MUL, DIV and AAM instructions and their clock-cycle counts.

PART IV

Hardware and Instruction Encoding

Chapters 10–11

PART IV · CHAPTER 10

8086 Hardware and Device Control

The 8086 carries its address, data, control, status and interrupt signals on the 40 pins of a DIP package. This chapter explains the multiplexing of address and data lines, ALE and address latches, memory banks with BHE', read and write bus cycles made of the T1–T4 states, minimum and maximum mode, and the 8284A and 8288 chips. The device-control part covers the IN and OUT instructions, polling, waiting with INT 15h and software delays. The chapter ends by programming emu8086 devices such as traffic lights, a stepper motor, an LED display, a thermometer, a printer and a robot.

NOTES IN THIS CHAPTER

10.1 8086 Pins

10.2 Device Control

PART IV · CHAPTER 10.1

8086 Pins

The effect of an assembly instruction on registers and flags is the face the programmer sees. The same instruction also leaves a trace outside the chip: address, data and control signals appear on the pins in a definite order. These notes treat the 8086 as a circuit component. They divide the pins of the 40-pin package into groups and explain how address and data pass through the same pins. They examine, at the level of clock cycles, the address latches and data transceivers that undo the multiplexing, the even and odd memory banks, and the read and write bus cycles. They go on to cover the wait states that slow memory requires, the handing over of the bus to another master, the clock and reset circuit, the 8288 bus controller in maximum mode, and the interrupt pins. In every topic, code traces show which value an instruction produces on which pin.

LEARNING OUTCOMES

Recognize the pins
Divide the 40 pins of the 8086 into groups; state each pin's direction, its active level and what it carries in which T state; match the eight pins that change between minimum and maximum mode.

Undo the multiplexing
Explain why address and data pass through the same pins; draw how separate address and data buses are built with ALE, an address latch and a data transceiver.

Select the banks
Explain how BHE' and A0 select the even and odd banks; find how many data bus cycles an instruction needs and over which lines the data passes.

Trace the bus cycle
Trace the read and write cycles signal by signal through the states T1–T4; explain how READY adds wait states and compute the number of wait states a memory needs.

Read the status and control pins
Decode the S2'–S0', S3–S6 and QS1–QS0 codes; give the values that RD', WR', M/IO', DT/R', DEN', HOLD/HLDA, RQ'/GT and LOCK' take during an instruction.

Build the system
Explain the role of the 8284A clock generator, the start-up after RESET, the 8288 bus controller and the INTR, NMI and INTA' pins in an 8086 system.

10.1.1 The 8086 package and its pins

A microprocessor talks to the outside world only through its pins. Reading a byte from memory, writing a value to a port, receiving an interrupt request or taking in the clock signal each needs a separate electrical connection. The room set aside for these connections on the 8086 is very tight: the whole processor is squeezed into a 40-pin package. This section introduces the package, the naming of the pins and the pin groups. It then shows why 40 pins are not enough. Intel overcame this shortage in two ways: it used the same pin for different purposes at different times, and it tied the meaning of one group of pins to the type of system.

10.1.1.1 The 40-pin DIP package

The Intel 8086 is a 16-bit microprocessor made in HMOS (high-performance n-channel MOS) technology. The chip sits inside a rectangular package called a **dual in-line package** (DIP). Each of the two long sides of the package has twenty pins. The pins are numbered starting from the top-left corner and going counterclockwise: 1–20 from top to bottom on the left side, 21–40 from bottom to top on the right side. A semicircular notch on the top edge of the package marks the position of pin 1.

The 8086 works from a single +5 V supply. The supply is connected to pin 40 (V_{CC}) and ground to pins 1 and 20 (GND). The data sheet allows a tolerance of ±10% on the supply: between 4.5 V and 5.5 V the chip works correctly. A 5 MHz 8086 draws at most 340 mA; that is 5 V × 0.34 A = 1.7 W of power. This power, which looks small by today's standards, was a serious figure for heating the chip in 1978.

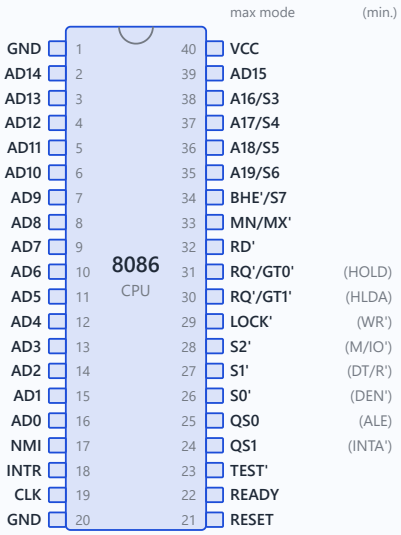


Figure 10.1.1: The pin diagram of the 8086. Pins 1–20 are on the left side and pins 21–40 on the right. Pins 24–31 have two names: the maximum-mode name outside, the minimum-mode name in parentheses.

Figure 10.1.1 shows the position of the pins on the package. What stands out at first glance is that many pins have two names separated by a slash: A17/S4, BHE'/S7, RQ'/GT0' and so on. These double names tell that the pin does two different jobs, at different times or depending on the type of system.

Definition 10.1.1

A signal that does its job when at logic level 1 (about +5 V) is an **active-high** signal. A signal that does its job when at logic level 0 (about 0 V) is **active low**. The data sheet draws an overbar above the names of active-low signals. In this course the same meaning is shown with an apostrophe at the end of the name: RD' = 0 means "a read is in progress".

Active-low signals are a habit of the TTL era. In TTL gates, pulling a line to 0 can be made stronger than pulling it to 1. Moreover, a TTL input that has come loose behaves like a 1; a broken "active-low" line therefore stays passive by itself. This is why most of the read, write, interrupt-acknowledge and select signals are active low.

Some names describe two states at once. When a name like M/IO' is read, the slash can be thought of as "or": if the pin is 1 the first meaning (M, memory) holds, if it is 0 the second (IO', input/output). MN/MX', DT/R' and BHE', which is read together with S7, follow the same pattern. The pins also have three kinds of direction. An **input** (I) pin only brings information to the processor, an **output** (O) pin only carries information out. A **bidirectional** (I/O) pin works in both directions.

Common mistake

Reading the DT/R' signal as "DT/R entirely active low" is wrong. The apostrophe belongs only to R: with the pin at 1 the meaning DT (data transmit) holds, at 0 the meaning R (data receive). Likewise in the M/IO' signal, M is active high and IO is active low. In the data sheet the overbar is only over the relevant part of the name.

10.1.1.2 Inside the chip

In the middle of the plastic or ceramic package there is a piece of silicon a few millimetres square, the **die**. **Bonding pads** are lined up along the edges of the die. Each pad is connected to one pin of the package with a gold or aluminium wire thinner than a hair. The real job of the package is to protect this small die, to dissipate its heat and to carry its wires out to pins spaced to suit the printed circuit board.

Figure 10.1.2: Die photograph of the 8086 (Ken Shirriff, righto.com). The address/data pads are on the left edge, the 16-bit registers, the ALU and the queue registers on the left, and the microcode ROM at the bottom

right.

Figure 10.1.2 shows an 8086 die with its functional blocks marked. The die has about 29,000 transistors; the process is 3 µm. The pads on the left edge are the lines going to the AD pins. The registers, the ALU and the registers of the prefetch queue are close to these pads, because these blocks are the parts that exchange data with the bus most often. The large regular area at the bottom right is the microcode ROM: the internal memory that holds the steps of the complex instructions. The bus control logic generates the timing of the signals on the pins.

The die photograph shows one thing clearly: there is no memory inside the processor. Apart from the registers, all programs and data are in chips outside. Every memory access of the processor has to pass through the pins. The number and timing of the pins therefore directly limit the processor's speed.

10.1.1.3 Why complicated pins?

A microprocessor is connected to memory and I/O devices through an address bus, a data bus and a control bus. How many pins would the 8086 need if every signal had its own pin? A rough count goes like this:

- The address bus is 20 lines: A0–A19. 2^{20} = 1 MB of memory is addressed.
- The data bus is 16 lines: D0–D15.
- Control, status and interrupt signals are about 20 lines: read, write, memory/I/O select, ready, interrupt request, bus request, status information.
- Power and clock are 4 lines: V_{CC}, two GNDs and CLK.

The total exceeds 60. In the late 1970s 64-pin packages were expensive, took up a lot of room on the board and caused trouble in production. Intel chose to stay with the 40-pin package familiar from the 8080 and 8085. The only way to close the gap was to make some pins do more than one job.

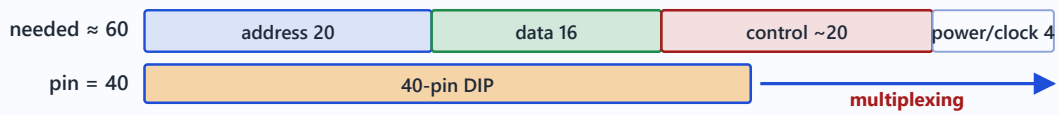


Figure 10.1.3: The number of signals needed with separate pins is about 60; the package has 40 pins. The gap is closed by multiplexing.

Definition 10.1.2

Multiplexing is the carrying of different information by the same pins at different times. The 8086 uses time-division multiplexing: pins AD0–AD15 carry the low 16 bits of the address in the first clock cycle of a bus cycle and data in the following cycles. Pins A16–A19 carry the top 4 bits of the address in the first cycle and status information afterwards.

The pin shortage of the 8086 has a second source. The physical address produced with the segment registers is 20 bits, while the data bus is 16 bits. The low 16 bits of the address can share the data lines, but the extra 4 address bits (A16–A19) have no data lines to pair with. So that these four pins are not left idle, they carry status information (S3–S6) once the address time has passed.

Multiplexing has two costs. First, the rest of the system wants a separate, stable address bus; latch circuits that capture and hold the address are therefore added outside the processor. Second, because address and data pass in turn through the same pins, every memory access is spread over at least four clock cycles. A processor with 60 pins and separate buses could have been built with less external circuitry using the same technology. But the 40-pin package was cheap, and designers coming from 8085 systems already knew the multiplexed bus.

10.1.1.4 Pin groups

The 40 pins can be divided into seven groups by function. Table 10.1.1 gives these groups with their pin numbers.

Group	Pins	Description
Power	V_{CC} (40), GND (1, 20)	+5 V ± 10%
Clock and reset	CLK (19), RESET (21), READY (22)	all three come from the 8284A clock generator
Address/data	AD0–AD15 (2–16, 39)	address in T1, then data
Address/status	A16/S3–A19/S6 (38–35), BHE'/S7 (34)	address in T1, then status
Interrupt	NMI (17), INTR (18), INTA' (24, minimum mode)	hardware interrupts
Common control	RD' (32), TEST' (23), MN/MX' (33)	the same in both modes
Mode-dependent	24–31	minimum: INTA', ALE, DEN', DT/R', M/IO', WR', HLDA, HOLD; maximum: QS1, QS0, S0', S1', S2', LOCK', RQ'/GT1', RQ'/GT0'

Table 10.1.1: The pin groups of the 8086.

The numbers of the address/data pins look irregular at a glance. AD15 is at pin 39, while the pins from AD14 down to AD0 run from 2 to 16. The four pins between A19/S6 and A16/S3 sit on the right edge, numbers 35–38, in reverse order. This layout comes from the order of the pads on the die. The

AD pads in Figure 10.1.2 are lined up one after another on the left edge of the die. They are connected with the shortest wires to the pins on the left edge of the package.

The direction of a signal also depends on its group. AD0–AD15 are bidirectional: outputs during a write, inputs during a read. A16/S3–A19/S6 and BHE'/S7 are outputs only, because no external device drives these pins. CLK, RESET, READY, NMI, INTR, TEST' and MN/MX' are inputs only. RD', ALE, DEN', DT/R', M/IO', WR', INTA' and HLDA are outputs; HOLD is an input. RQ'/GT0' and RQ'/GT1' in maximum mode, however, carry out a two-way handshake on a single pin.

Pin	Name	Dir.	Function
1, 20	GND	—	ground
2–16	AD14–AD0	I/O	address (T1) / data (T2–T4)
17	NMI	I	non-maskable interrupt, rising edge
18	INTR	I	interrupt request, active-high level
19	CLK	I	clock, 33% duty cycle
21	RESET	I	reset, active high
22	READY	I	memory/I/O ready, active high
23	TEST'	I	input tested by the WAIT instruction
24	INTA' / QS1	O	interrupt acknowledge / queue status
25	ALE / QS0	O	address latch enable / queue status
26	DEN' / S0'	O	data enable / status
27	DT/R' / S1'	O	data transmit/receive / status
28	M/IO' / S2'	O	memory or I/O / status
29	WR' / LOCK'	O	write / bus lock
30	HLDA / RQ'/GT1'	O / I/O	hold acknowledge / bus request-grant
31	HOLD / RQ'/GT0'	I / I/O	hold request / bus request-grant
32	RD'	O	read
33	MN/MX'	I	mode select
34	BHE'/S7	O	bus high enable / spare status bit
35–38	A19/S6–A16/S3	O	address (T1) / status (T2–T4)
39	AD15	I/O	address (T1) / data (T2–T4)
40	V_{CC}	—	+5 V

Table 10.1.2: The 40 pins of the 8086: number, name, direction and short function (minimum / maximum mode for 24–31).

Table 10.1.2 collects all 40 pins in one list and can be used as a reference table when reading circuit diagrams. There is no need to memorize pin numbers. What matters is knowing which group each pin is in and what it carries at what moment.

10.1.1.5 Minimum and maximum mode

The second remedy for the pin shortage is to change the meaning of the eight pins numbered 24–31 according to the type of system. In a small system there is a single processor, and it can generate control signals such as read, write and address latch enable by itself. In a large system, however, other chips such as the 8087 floating-point coprocessor or the 8089 I/O processor share the same bus. In that case the control signals are generated by a separate chip, the 8288 bus controller. The pins freed on the 8086 are given to the new signals needed for the multiprocessor arrangement.

Definition 10.1.3

If the MN/MX' pin (pin 33) is connected to V_{CC}, the 8086 works in **minimum mode**. This is a small single-processor system; pins 24–31 generate the control signals (INTA', ALE, DEN', DT/R', M/IO', WR', HLDA, HOLD) directly. If MN/MX' is connected to ground, it works in **maximum mode**. The same pins turn into the status code, the queue status and the bus-sharing signals: QS1, QS0, S0'–S2', LOCK', RQ'/GT1', RQ'/GT0'. The control signals are generated by the 8288.

The mode is chosen by hardware. The MN/MX' pin is permanently wired to the supply or to ground on the board and does not change while the processor runs. No instruction can change the mode; from the software's point of view there is no difference between the two modes. The same program gives the same result in both systems.

Pin	Minimum mode (MN/MX' = 1)	Maximum mode (MN/MX' = 0)
24	INTA'	QS1
25	ALE	QS0
26	DEN'	S0'
27	DT/R'	S1'
28	M/IO'	S2'
29	WR'	LOCK'
30	HLDA	RQ'/GT1'
31	HOLD	RQ'/GT0'

Table 10.1.3: The eight pins whose meaning changes with the mode.

The pairing in Table 10.1.3 is not random. Pin 28 carries, in minimum mode, the M/IO' signal that separates memory from I/O; in maximum mode it carries the status bit S2', which makes the same distinction (1 memory, 0 I/O). DT/R' on pin 27 gives the data direction, and S1' helps separate reads from writes. The 8288 in maximum mode thus receives the information of the minimum-mode signals in coded form and regenerates them. The maximum-mode counterpart of the HOLD/HLDA pair on pins 30–31 is again the RQ'/GT pins, which also hand over the bus.

Example 10.1.1

A designer is preparing a board to which an 8087 coprocessor will be added. Where should MN/MX' be connected, and which signals that it used in minimum mode can it no longer take from the 8086?

1. The 8087 watches the 8086's queue on the QS1 and QS0 pins and requests the bus with the RQ'/GT pins. These pins exist only in maximum mode; MN/MX' is connected to ground.
2. Because pins 24–31 change in maximum mode, the signals INTA', ALE, DEN', DT/R', M/IO', WR', HLDA and HOLD cannot be taken from the 8086.
3. The 8288 does their job: from the S2'–S0' code it generates the signals MRDC', MWTC', IORC', IOWC', INTA', ALE, DEN and DT/R'. RQ'/GT0' and RQ'/GT1' are used in place of HOLD/HLDA.

The RD' pin (32) is on the 8086 in both modes; but in maximum mode memory and I/O reads are usually done with MRDC' and IORC', which the 8288 generates.

10.1.2 Address and data pins

In a memory access the processor has two things to say: which address and which data. The 8086 sends these two pieces of information in turn on the same pins. This section first describes the job of the address/data and address/status pins over time, then how the multiplexing is undone outside. It then shows, with a code trace, how the 16-bit data bus is divided into two byte banks and why a word at an odd address takes two bus cycles.

10.1.2.1 AD0–AD15 and A16/S3–A19/S6

The **AD0–AD15** pins carry the low 16 bits of the address and the 16-bit data. In the first clock cycle of a bus cycle (T1) the pins hold the address; at this moment they are called A0–A15. In the following cycles (T2, T3, T4 and the wait cycles, if any) the same pins carry data; then they are called D0–D15. In a byte access only one half of the data lines is used: D0–D7 or D8–D15. The pins are bidirectional. In a write the processor drives the data out; in a read the external device drives the data in. When needed the processor can release the drivers of these pins entirely; this is called the **float**, or high-impedance (tri-state), state.

Definition 10.1.4

A **tri-state output** is an output that can take a third state in addition to the levels 0 and 1: **high impedance**. In this state the output drives the line neither to 0 nor to 1; electrically it is as if disconnected from the line, and another driver can drive the line. Only one driver may be active on a line at a time.

The **A16/S3–A19/S6** pins carry the top 4 bits of the address. In a memory access these pins hold A16–A19 during T1. From T2 on, the same pins switch to the status bits S3–S6. In an I/O access the address is only 16 bits; the lines A16–A19 are therefore 0 in T1. S3 and S4 show which segment register is used in the access, and S5 the current value of the interrupt flag IF; S6 is always 0. The status bits are treated in detail in their own subsection.

Figure 10.1.4: The AD15–AD0 pins carry the address in T1 and data in T2–T4. The A19/S6–A16/S3 pins carry the top four bits of the address in T1 and the status bits afterwards.

BHE'/S7 (pin 34) is close to this group. During T1 it works as BHE' (bus high enable) and tells that the upper half of the data bus (D8–D15) will be used in this cycle. During T2–T4 it turns back into the status bit S7; on the 8086, S7 is an unused spare bit. Figure 10.1.4 summarizes this time sharing: the meaning of a pin depends on the clock cycle in which it is looked at.

Common mistake

The idea that "the memory reads the address from the AD pins throughout the cycle" is wrong. The address stays on the AD pins only during T1. In T2 the pins switch to data or are floated. The memory chip, however, has to see the address during T2–T4 too. For this reason the address is taken at the end of T1 into an external latch circuit. The memory reads the address from the output of that latch.

10.1.2.2 Demultiplexing: ALE and the address latch

The rest of the system, that is, memory chips, address decoders and I/O devices, wants a separate, stable address bus and a separate data bus. Separating these two buses from the multiplexed bus of the 8086 is called **demultiplexing**. Two kinds of circuit are used for the job: the **address latch**, which captures and holds the address, and the **transceiver**, which strengthens the data lines and sets their direction. The processor tells the latch when to capture the address with the **ALE** (address latch enable, pin 25) signal.

Definition 10.1.5

A **latch** is a memory circuit that passes the value at its input to its output while an enable input is 1 and holds the last value when the enable falls to 0. While the enable is 1 the latch is "transparent"; on the falling edge it captures the value. In 8086 systems the 8-bit 8282 or 74LS373 latches are used; ALE is connected to the enable input (STB or LE).

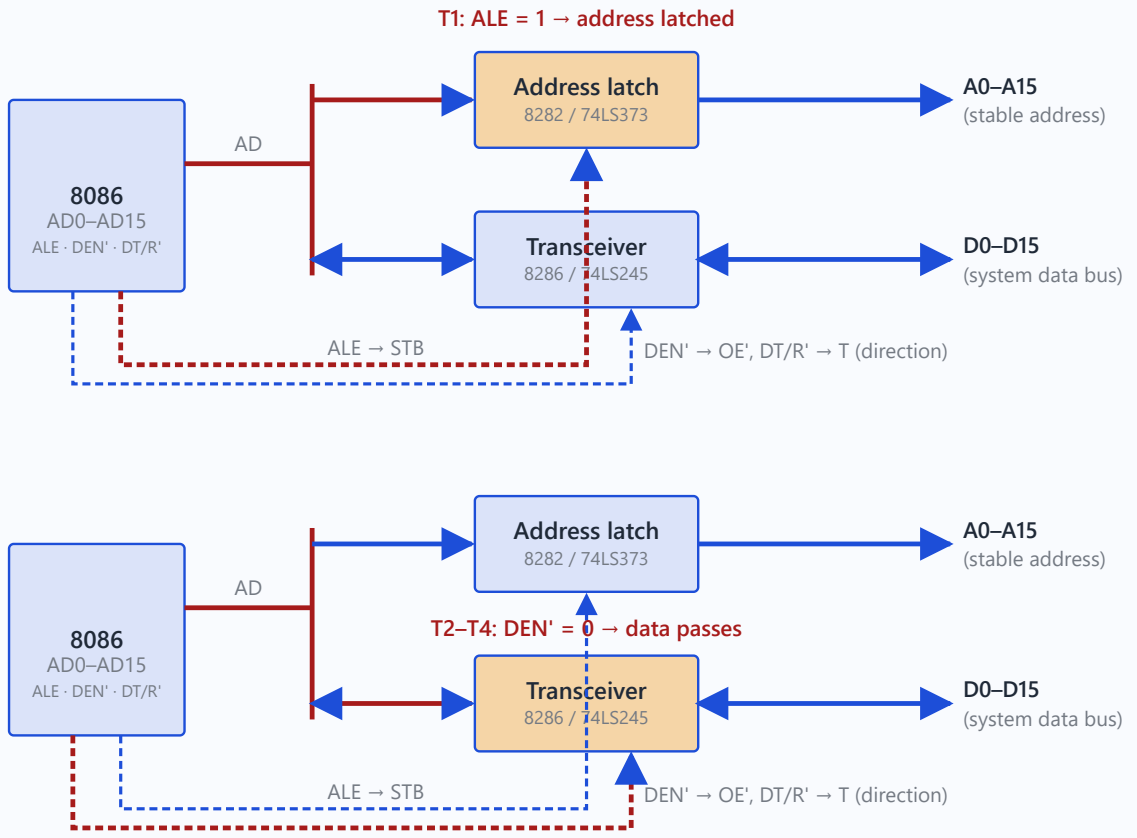


Figure 10.1.5: Demultiplexing. Top, T1: the 8086 puts the address on the AD pins and makes ALE 1; as ALE falls, the latch captures the address. Bottom, T2–T4: the same pins carry data; DEN' = 0 turns on the transceiver and DT/R' selects the direction.

Figure 10.1.5 shows the two phases of a cycle side by side. In T1 the 8086 puts the address on the AD pins and makes ALE 1 for a short pulse. While ALE is 1 the latch is transparent; the address on the pins passes to the latch's output. When ALE falls to 0, the latch captures the current value and holds it at its output until the end of the cycle. After T2 the AD pins carry data, but the address at the latch's output does not change. Memory and I/O devices see the address at the latch's output and the data at the far end of the transceiver.

Why ALE uses the falling edge can be understood from the timing values in the data sheet. On a 5 MHz 8086, ALE becomes 1 at most 80 ns after the clock edge at the start of T1 (T_{CLLH}). The address becomes valid at most 110 ns after (T_{CLAV}). So when ALE rises, the address may not yet be stable. The

data sheet, however, guarantees two things. The address is valid at least $T_{AVAL} = T_{CLCH} - 60$ ns before ALE falls. After ALE falls it also stays on the pins for at least $T_{LLAX} = T_{CHCL} - 10$ ns. A transparent latch is therefore the right choice: while ALE is 1 the output follows the address, and as ALE falls the settled address is captured.

Example 10.1.2

On a 5 MHz 8086 (T_{CLCL} = 200 ns) the low time of the clock is at least T_{CLCH} = 118 ns and the high time at least T_{CHCL} = 69 ns. How long before ALE falls is the address valid at the least, and how long does it stay on the pins after ALE falls at the least?

1. $T_{AVAL} = T_{CLCH} - 60 = 118 - 60 = 58$ ns: the address at the latch input is stable at least 58 ns before the moment of capture.
2. $T_{LLAX} = T_{CHCL} - 10 = 69 - 10 = 59$ ns: the address stays on the pins at least 59 ns after the moment of capture.
3. According to the 74LS373 data sheet, the data at the input must be set up at least 5 ns before LE falls and held at least 20 ns after it falls. Since 58 ns ≥ 5 ns and 59 ns ≥ 20 ns, both conditions are met with a wide margin.

As long as the latch's setup and hold times are smaller than the guarantees given by the 8086, the captured address is correct.

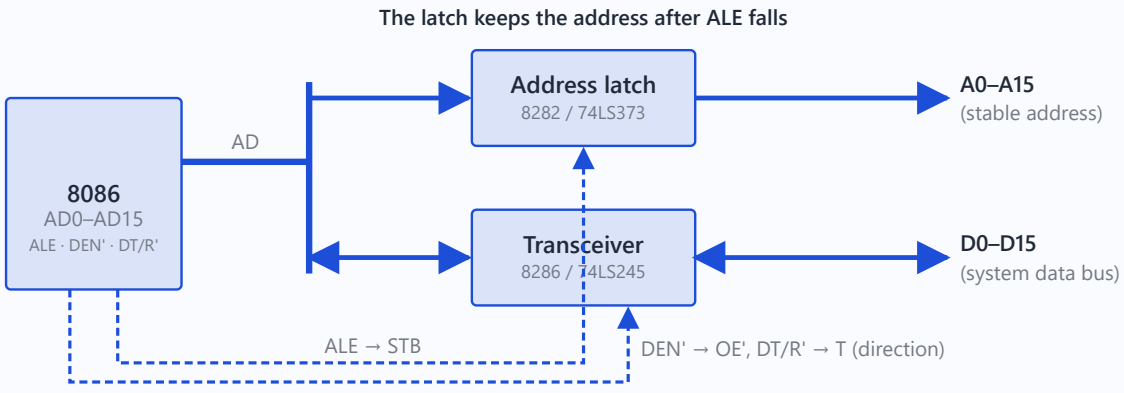


Figure 10.1.6: Top: the multiplexed bus and the two demultiplexing circuits, the address latch (8282 / 74LS373) and the transceiver (8286 / 74LS245). Bottom: the rest of the cycle: the latch keeps holding the address after ALE falls.

For a 20-bit address three 8-bit latches are needed. One captures A0–A7, one A8–A15. The third holds A16–A19 and the BHE' signal: these five signals also switch to status information after T1, so they must be captured too. The unused inputs of the third latch are left idle. Figure 10.1.6 gives a simplified view of this arrangement.

The 8282 and the 74LS373 do the same job: both are 8-bit transparent latches with tri-state outputs. The 74LS373 has two control inputs: LE (latch enable) and OE' (output enable). ALE is connected to LE, and OE' is kept constantly at 0 so that the latch's output always drives the address bus. When the bus is handed to another master, the latch outputs can also be floated by making OE' 1. The 8283 is the version of the 8282 whose outputs are inverted.

Common mistake

Using, instead of an address latch, a D flip-flop chip that captures on the rising edge (for example a 74LS374) and connecting ALE directly to its clock input is a mistake. The flip-flop captures the address on the rising edge of ALE. At that moment the address may not yet be valid: ALE rises at 80 ns at the latest, the address settles at 110 ns at the latest. If a flip-flop is to be used, the inverse of ALE is given as the clock; the capture then moves to the falling edge of ALE.

10.1.2.3 A buffered and latched system

The latches produce the address bus. For the data bus a **transceiver** is used. A transceiver is a bidirectional buffer: a combination of a transmitter and a receiver in one package. The 8286, and the 74LS245 often used in its place, are 8-bit transceivers; two are needed for the 16-bit data bus. A transceiver has two control inputs. The direction input (T on the 8286, DIR on the 74LS245) selects which side the data flows from and to. When the enable input (OE' or G') is 1, it floats both sides and isolates the lines from each other.

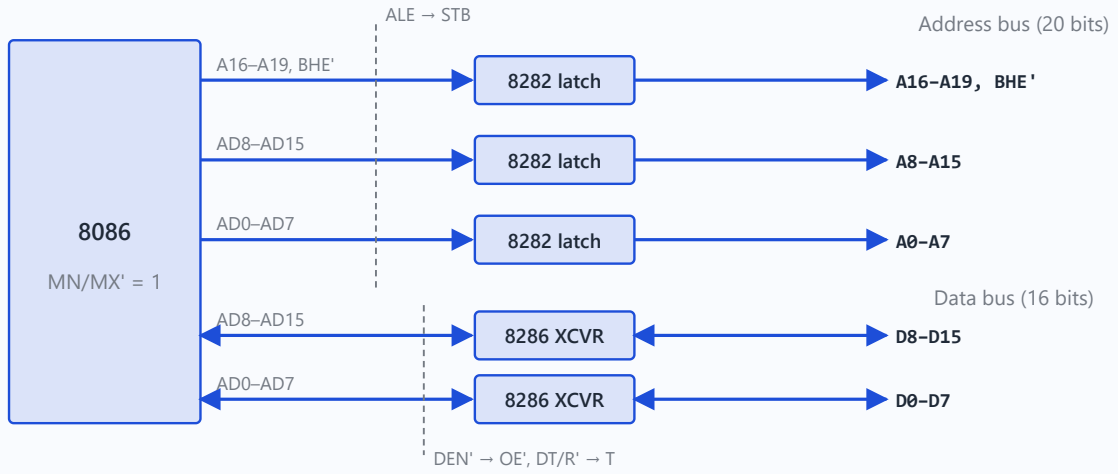


Figure 10.1.7: A buffered and latched minimum-mode system: three 8282 latches produce the 20-bit address bus and the BHE' signal, and two 8286 transceivers produce the 16-bit data bus. ALE controls the latches, DEN' and DT/R' the transceivers.

Figure 10.1.7 shows a complete minimum-mode bus interface. The 8086's DT/R' output is connected to the transceiver's direction input and its DEN' output to the enable input. With DT/R' = 1 data flows from the processor to memory, with 0 from memory to the processor. DEN' is 0 only while data is being transferred; it stays 1 in T1 so that the address on the AD pins does not leak onto the data bus. These two signals are treated, with their timing, in the subsection on the control pins.

The second job of the buffers is to provide **drive strength**. An output pin of the 8086 can sink at most 2.5 mA at the low level, and the data sheet limits the output load to 20–100 pF. The current that a 74LS input supplies at the low level is about 0.4 mA. A single pin of the 8086 can therefore safely drive only a few LS inputs. On a large board with dozens of memory chips every bus line goes to many inputs, and the capacitance of long traces is added. Latches and transceivers carry this load in place of the processor. This is why in large systems the address, data and control buses are all buffered.

Example 10.1.3

One AD pin of the 8086 can sink 2.5 mA at the low level. At most how many 74LS inputs, each requiring 0.4 mA at the low level, can it drive directly? How many inputs can a 74LS245 output drive if it can sink 24 mA at the low level?

1. The 8086 pin: 2.5 / 0.4 = 6.25 → at most 6 inputs. In practice fewer are used, for line capacitance and noise margin.
2. The 74LS245 output: 24 / 0.4 = 60 inputs.

The transceiver raises the current-driving capacity about tenfold. As the memory board grows, this difference becomes decisive.

10.1.2.4 BHE' and A0: the memory banks

The data bus of the 8086 is 16 bits wide, but memory is addressed at byte level. Every address points to one byte. So that the processor can transfer both a single byte and a word made of two consecutive bytes in one cycle, the 1 MB memory is physically divided into two halves. Each half is called a **memory bank**.

Definition 10.1.6

The **low bank** (even bank) holds the bytes at even addresses and is connected to the data lines D0–D7. The **high bank** (odd bank) holds the bytes at odd addresses and is connected to D8–D15. Each bank is 512 KB and is addressed in parallel by the address lines A1–A19. A0 = 0 selects the low bank and BHE' = 0 the high bank.

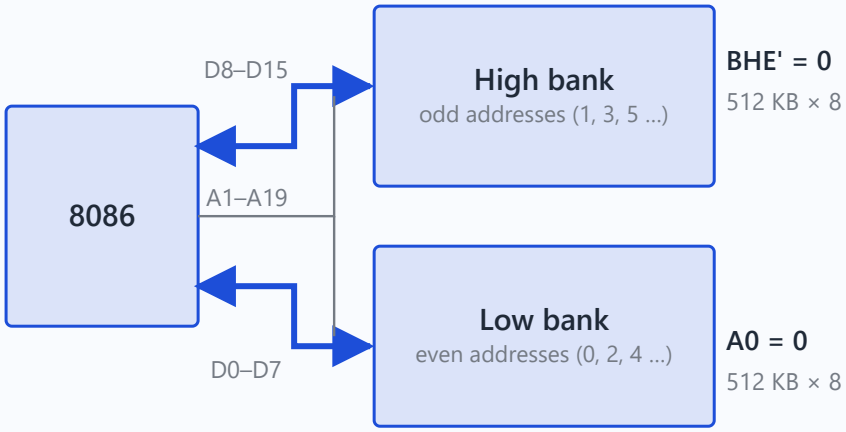


Figure 10.1.8: The even and odd memory banks. The low bank is connected to D7–D0 and selected with A0 = 0; the high bank is connected to D15–D8 and selected with BHE' = 0. Both banks are addressed with A19–A1.

In Figure 10.1.8 the two banks share the same address lines. A1–A19 go to the banks, not A0; because consecutive bytes within one bank are at addresses that increase by 2. For example, bytes 0, 1, 2 of the low bank correspond to addresses 00000h, 00002h, 00004h, and those of the high bank to 00001h, 00003h, 00005h. Two addresses with the same A19–A1, that is 2k and 2k + 1, are in the same row of the two banks and can be read together in one cycle.

BHE'	A0	Transfer	Lines used
0	0	whole word (even address)	D15–D0
0	1	high byte (odd address)	D15–D8
1	0	low byte (even address)	D7–D0
1	1	no transfer	—

Table 10.1.4: Bank selection with BHE' and A0.

Table 10.1.4 gives the four cases. When a byte at an even address is written, only the write enable of the low bank must be opened; the neighbouring byte in the high bank must not be corrupted. The bank selection is therefore combined with the memory write signal: the low bank's write input is derived from WR' and A0, the high bank's from WR' and BHE'. In a read, there is no harm in reading both banks together; the processor takes only the half it needs. Even so, in most designs bank selection is applied in reads too.

For a word starting at an odd address, the low byte is in the high bank and the high byte is in the low bank of the next row. Because the two bytes are in different rows, they cannot be read together in one cycle. The 8086 splits this into two bus cycles by itself: first it transfers the byte at the odd address over D8–D15, then the byte at the next even address over D0–D7. This splitting is invisible to the

programmer; the instruction gives the right result, it merely takes longer. The data sheet notes that this penalty applies only to word operands, not to instruction fetch: instructions are always fetched into the queue as words starting at an even address.

Common mistake

The idea that "a byte at an odd address also takes two cycles" is wrong. A byte access is a single cycle at whatever address: a byte at an even address passes over D0–D7, a byte at an odd address over D8–D15. Two cycles are needed only for a word access that starts at an odd address. To align data on even addresses the assembler's `EVEN` directive is used.

10.1.2.5 Code trace: aligned and misaligned access

The program below writes the same word once to an even and once to an odd address; then it reads a byte and a word. Tracing how many bus cycles are made in every data access, the values of BHE' and A0, and over which lines the data passes makes the bank layout concrete. The contents of addresses 20100h–20102h are assumed unknown (??) at the start.

```
mov ax, 2000h
mov ds, ax
mov ax, 1234h
mov [0100h], ax    ; even address
mov ax, 0ABCDh
mov [0101h], ax    ; odd address
mov al, [0100h]    ; byte
mov bx, [0101h]    ; odd address
```

The first two instructions load 2000h into DS; the data segment starts at physical address 20000h. These two instructions change only registers. On the bus only the cycles that fetch the instruction bytes are seen; there are no data cycles. `mov ax, 1234h` puts 1234h in AX: AH = 12h, AL = 34h.

`mov [0100h], ax` computes the physical address 20000h + 0100h = 20100h. The address is even. The 8086 puts BHE' = 0 and A0 = 0 in T1; both banks are selected at once. The word is written in a single cycle: the 34h in AL goes over D7–D0 to the even address (20100h), the 12h in AH over D15–D8 to the odd address (20101h). The little-endian layout fits the bank layout naturally: the low byte is written to the low address and the low bank.

`mov ax, 0ABCDh` makes AX ABCDh. The 0 at the start of the number keeps the assembler from taking ABCDh for a label name. `mov [0101h], ax` writes to the odd address 20101h. The 8086 splits this access into two cycles. In the first cycle the address is 20101h: with BHE' = 0, A0 = 1 only the high bank is selected, and the CDh in AL is written to 20101h over D15–D8. In the second cycle the address is 20102h: with BHE' = 1, A0 = 0 only the low bank is selected, and the ABh in AH is written to 20102h over D7–D0. This write overwrites the old 12h at 20101h with CDh.

`mov al, [0100h]` reads a byte from the even address: BHE' = 1, A0 = 0, the data comes over D7–D0. AL = 34h; AH does not change, so AX = AB34h. The last instruction, `mov bx, [0101h]`, is again a misaligned word read. The first cycle fetches the CDh at 20101h over D15–D8. The processor puts this byte in the low half of the word, BL. The second cycle fetches the ABh at 20102h over D7–D0; this byte goes to BH. The result is BX = ABCDh.

Instruction	Cycle	PA	BHE'	A0	D15–D8	D7–D0	RD'/WR'	Result
`mov [0100h],ax`	1/1 write	20100h	0	0	12h	34h	1/0	[20100h] = 34h, [20101h] = 12h
`mov [0101h],ax`	1/2 write	20101h	0	1	CDh	—	1/0	[20101h] = CDh
	2/2 write	20102h	1	0	—	ABh	1/0	[20102h] = ABh
`mov al, [0100h]`	1/1 read	20100h	1	0	—	34h	0/1	AX = AB34h
`mov bx, [0101h]`	1/2 read	20101h	0	1	CDh	—	0/1	BL = CDh
	2/2 read	20102h	1	0	—	ABh	0/1	BX = ABCDh

Table 10.1.5: The data bus cycles of the aligned and misaligned access program.

Table 10.1.5 shows that the four data instructions make a total of 6 data bus cycles: 1 + 2 + 1 + 2. At the end of the program AX = AB34h and BX = ABCDh; in memory 20100h = 34h, 20101h = CDh, 20102h = ABh. When the program is run in emu8086, the register and memory values come out the same. The emulator does not show banks and cycles; the number of cycles and the BHE'/A0 values follow only from whether the physical address is odd or even.

The lengths of the instruction codes also give information. `mov [0100h], ax` is 3 bytes, A3 00 01; the short accumulator form is used for AX. `mov bx, [0101h]` is 4 bytes, 8B 1E 01 01. These bytes are brought into the queue by instruction-fetch cycles, in addition to the data cycles.

Example 10.1.4

A 5 MHz 8086 reads an array of 1000 words from memory with no wait states. How long do the data cycles take if the array starts at an even address and if it starts at an odd address?

1. One bus cycle is 4 × 200 ns = 800 ns.
2. In an array that starts at an even address every word is a single cycle: 1000 × 800 ns = 800 µs.
3. In an array that starts at an odd address every word takes two cycles: 2000 × 800 ns = 1600 µs.

The difference in alignment alone doubles the data access time. Keeping the stack at an even address is important for the same reason: every PUSH and POP in interrupts and procedure calls is a word access.

10.1.3 Bus cycles

To understand in which order the address, data and control signals appear on the pins, time must be divided into clock cycles. This section defines the T states of a bus cycle and then traces the write and read cycles cycle by cycle. It gives all the control signals of the read cycle together with the timing values from the data sheet. Finally it describes how a slow memory requests a wait state through the READY signal, and how to compute the number of wait states a memory needs.

10.1.3.1 The bus cycle and the T states

Definition 10.1.7

Every exchange of data between the processor and memory or I/O is a **bus cycle**. On the 8086 a bus cycle lasts at least four clock cycles. These cycles are called the **T states**: T1, T2, T3, T4. For a slow device, **wait states** (Tw) can be inserted between T3 and T4. The cycles in which the processor does not use the bus at all are **idle states** (Ti).

Every T state begins on one falling edge of the clock and ends on the next. The meaning of the pins changes with the T state. In T1 the AD pins hold the address, A16–A19 hold the upper bits of the address, and the ALE pulse is given. T2 is mostly preparation: in a read the direction of the bus is changed and the control signals (RD', WR') become active. The data transfer takes place in T3 and T4. T4 is the closing of the cycle: the control signals return to passive and the bus is prepared for the next cycle.

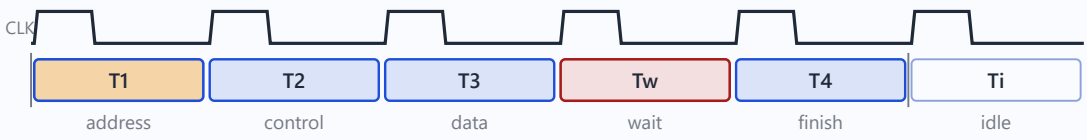


Figure 10.1.9: Timeline of a bus cycle: T1 address, T2 control, T3 data, Tw wait if needed, T4 finish. When the bus is not in use the processor is in idle (Ti) cycles.

Once the clock frequency is known, the duration of a bus cycle is computed directly. At a 5 MHz clock one cycle is T = 1 / 5 MHz = 200 ns; a bus cycle without wait states takes 4 × 200 = 800 ns. At most 1 / 800 ns = 1.25 million bus cycles can be made in a second. Since every cycle carries at most 2 bytes, the highest bandwidth of the bus is about 2.5 MB/s.

Example 10.1.5

Find the duration of a bus cycle without wait states and the highest bus bandwidth for the 8086 (5 MHz), the 8086-2 (8 MHz) and the 8086-1 (10 MHz).

1. 5 MHz: T = 200 ns, cycle 800 ns, 1.25×10^6 cycles/s × 2 bytes = 2.5 MB/s.
2. 8 MHz: T = 125 ns, cycle 500 ns, 2×10^6 cycles/s × 2 bytes = 4 MB/s.
3. 10 MHz: T = 100 ns, cycle 400 ns, 2.5×10^6 cycles/s × 2 bytes = 5 MB/s.

These values are upper limits. In reality part of the bus goes to instruction fetch, misaligned words take two cycles and slow memories add wait states.

Idle states arise at moments when the processor does not use the bus. For example, when the prefetch queue is full and the instruction being executed asks for no memory, the BIU has nothing to do; the bus waits in Ti cycles. The data sheet notes that these cycles are used for internal work. In Ti cycles the control signals stay passive, and the bus can easily be handed to another master by a HOLD request.

10.1.3.2 The write bus cycle

A write cycle consists of four T states. Figure 10.1.10 shows a simplified timeline of a memory write.

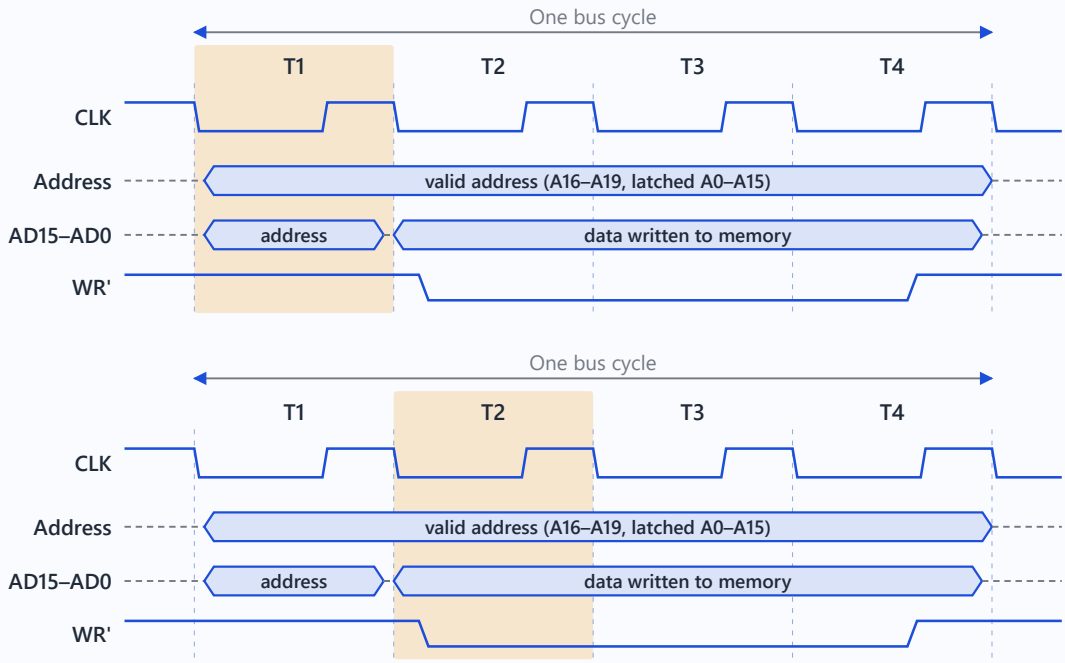


Figure 10.1.10: The write bus cycle. Top, T1: the target address is on the AD and A pins and is captured by the latches with ALE. Bottom, T2: the AD pins carry the data to be written and WR' = 0.

1. **T1:** The processor puts the target address on the AD0–AD15 and A16–A19 pins. The ALE pulse captures the address in the external latches. M/IO' became valid in T4 of the previous cycle; it is 1 for a memory write and 0 for an I/O write.
2. **T2:** The AD pins switch from address to data: the processor drives the data to be written at once. The WR' signal becomes 0 at the start of T2. The A16–A19 pins switch to the status bits (S3–S6).
3. **T3:** The data and WR' stay stable. The memory prepares to capture the data. If the device is not ready it keeps READY at 0; in that case a Tw is added after T3.
4. **T4:** WR' returns to 1. Most memories write the data on the rising edge of WR'. The data stays on the pins until the middle of T4; thus on the rising edge the memory sees stable data.

In a write cycle the address is in T1 and the data is on the same pins from T2 to T4, with no gap in between. The reason is that the source of the data is the processor itself: the drivers that drove the address go on to drive the data in the next cycle. The direction of the bus does not change.

The data sheet gives the width of the WR' pulse as $T_{WLWH} \geq 2T_{CLCL} - 60$. At 5 MHz that is $2 \times 200 - 60 = 340$ ns. The shortest write pulse a memory chip asks for must be shorter than this; if not, the pulse is lengthened by adding wait states. Every Tw extends the time WR' stays 0 by one more cycle.

10.1.3.3 The read bus cycle

A read cycle also consists of four T states. The important difference is that there is a gap between the address and the data on the AD pins. Figure 10.1.11 shows this gap.

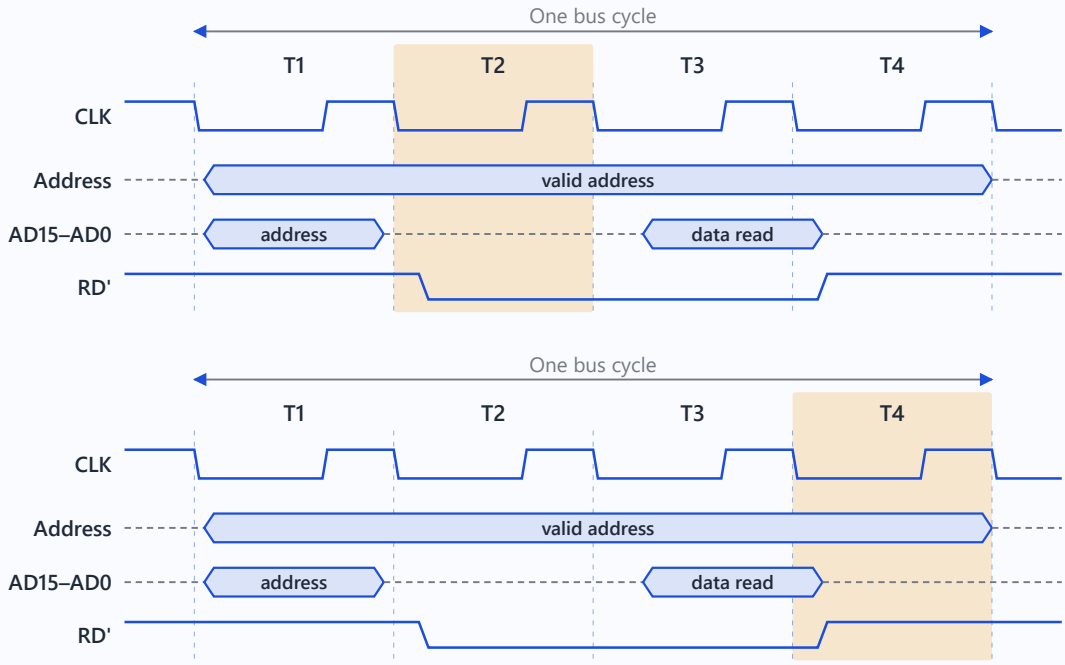


Figure 10.1.11: The read bus cycle. Top, T3: the memory puts the requested data on the data bus. Bottom, T4: the 8086 samples the data at the start of T4, RD' returns to 1 and the memory releases the bus. Between the address and the data the AD pins float in T2.

1. **T1:** As in a write, the address to be read is put on the pins and captured with ALE.
2. **T2:** The 8086 makes the output drivers of the AD pins tri-state; the pins float. From this moment on the memory may drive the bus. The RD' signal becomes 0.
3. **T3:** When the memory or I/O device sees RD' = 0, it turns on its outputs and puts the requested data on the data bus.
4. **T4:** At the start of T4, that is, on the falling clock edge that ends T3, the 8086 reads the data from the pins. RD' returns to 1 and the memory releases the bus by turning off its outputs.

Definition 10.1.8

Bus turnaround is the moment at which the driver of a bidirectional line changes. In a read cycle the AD lines are driven by the processor in T1 and by the memory in T3–T4. The float state in T2 is for this turnaround: if the memory started driving the data before the processor stopped driving the address, the two outputs would collide on the same line.

A collision does not mean just wrong data. If one output drives a line to 1 and the other to 0, a current close to a short circuit flows between the two drivers; this is called **bus contention**. Continuous contention heats the chips and damages them over time. This is why the data sheet gives two separate guarantees. The AD pins float at most T_{CLAZ} = 80 ns after the clock edge at the start of T2. The RD' signal also stays 1 in T2 until the local bus floats. The memory turns on its output only after RD' = 0. The memory thus safely takes over the line that the processor has released.

Common mistake

The assumption that "in a read the data is ready in T2 as in a write" is wrong. In a read the data comes from an external device, and the device turns on its output only after it sees RD' = 0. RD' falls within T2, and the device's access delay is added; the data goes onto the bus within T3 at the earliest and is sampled at the start of T4. In a write, since the processor drives the data, it is on the pins from the start of T2.

10.1.3.4 Bus timing for a read

The simplified timelines show only the address, the data and the read signal. In a memory read in minimum mode six more control signals play a part: M/IO', ALE, DT/R', RD', DEN' and READY. Figure 10.1.12 gives them all on the same clock axis.

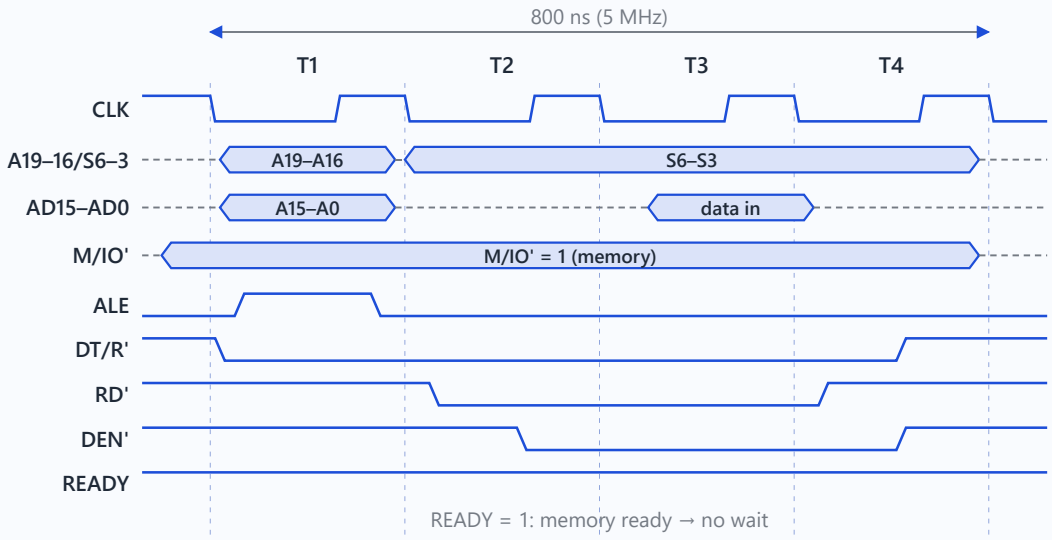


Figure 10.1.12: All the signals in a memory read cycle in minimum mode (5 MHz, 800 ns). M/IO' is 1 throughout the cycle, ALE pulses in T1, DT/R' is 0 throughout the cycle, RD' and DEN' are 0 from T2 to T4; because READY = 1 there is no wait state.

Let us read the signals cycle by cycle:

- **T1:** M/IO' = 1 (memory) has been valid since T4 of the previous cycle and stays through the cycle. The address is on the AD and A pins. The ALE pulse captures the address. DT/R' = 0: the transceiver is set to the "receive" direction. DEN' = 1: the transceiver is still off.
- **T2:** The AD pins float; the status bits S6–S3 appear on the A19–A16 pins. RD' becomes 0. Then DEN' becomes 0 and the transceiver turns on. According to the data sheet, in a read DEN' is active from the middle of T2 to the middle of T4; in a write, from the start of T2 to the middle of T4.
- **T3:** The memory puts the data on the data bus; the data passes through the transceiver and reaches the AD pins. The processor samples READY during T3. If READY = 1 the cycle goes on to T4.
- **T4:** The data is sampled at the start of T4. RD', DEN' and DT/R' return to passive. M/IO' may change within T4 depending on the type of the next cycle.

Symbol	Meaning	8086 (5 MHz)	8086-2 (8 MHz)	8086-1 (10 MHz)
T_{CLCL}	clock period	200–500 ns	125–500 ns	100–500 ns
T_{CLCH}	clock low time (minimum)	118 ns	68 ns	53 ns
T_{CHCL}	clock high time (minimum)	69 ns	44 ns	39 ns
T_{CLAV}	clock to address valid (maximum)	110 ns	60 ns	50 ns
T_{CLAZ}	clock to address float (maximum)	80 ns	50 ns	40 ns
T_{CLRL}	clock to RD' falling (maximum)	165 ns	100 ns	70 ns
T_{DVCL}	data setup before the clock edge (minimum)	30 ns	20 ns	5 ns
T_{RYHCH}	READY setup before the clock edge (minimum)	118 ns	68 ns	53 ns

Symbol	Meaning	8086 (5 MHz)	8086-2 (8 MHz)	8086-1 (10 MHz)
T_{RLRH}	RD' pulse width (minimum)	$2T_{CLCL} - 75$	$2T_{CLCL} - 50$	$2T_{CLCL} - 40$
T_{WLWH}	WR' pulse width (minimum)	$2T_{CLCL} - 60$	$2T_{CLCL} - 40$	$2T_{CLCL} - 35$

Table 10.1.6: Some timing values of the 8086 (data sheet, minimum mode). T_{CLCL} is the clock period.

Table 10.1.6 gives values selected from the data sheet's AC characteristics table. The naming rule of the symbols makes them easier to read: the first two letters describe the starting event, the last two the ending event. CL means "clock falls", CH "clock rises", AV "address valid", RL "RD' falls", DV "data valid". T_{CLAV} is the longest time from the falling edge of the clock until the address becomes valid.

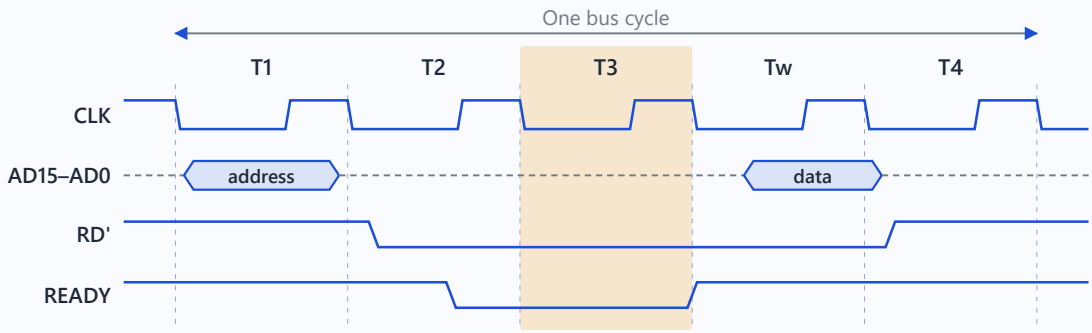
The upper limit of the clock period also draws attention: T_{CLCL} can be at most 500 ns, that is, the clock cannot fall below 2 MHz. The 8086 is a dynamic design; the charges on its internal nodes are lost if they wait longer than a certain time. The clock therefore cannot be stopped or slowed down very much.

10.1.3.5 READY and the wait state

The **access time** of a memory chip is the time from the moment it receives the address to the moment it puts the data on its output. A slow memory with a long access time cannot prepare its data by the end of T3. The 8086 solves this with the READY input.

Definition 10.1.9

READY (pin 22) is an active-high input that tells that the addressed memory or I/O device will complete the data transfer. The processor samples READY during T3. If READY = 0 it does not go on to T4; it adds a **wait state** (Tw) and samples READY again in Tw. It keeps adding Tw until READY becomes 1.



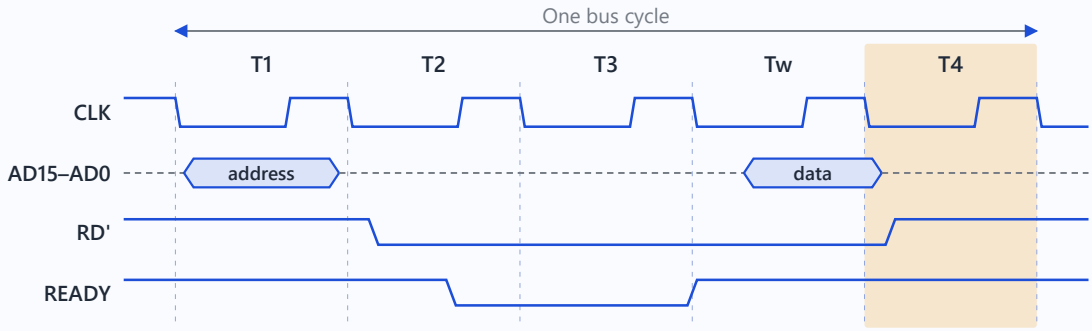


Figure 10.1.13: A read cycle with a wait state. Top, T3: READY = 0 is seen, the processor adds a Tw instead of T4 and all the signals stay as they are. Bottom, T4: READY = 1 has been sampled in Tw; the data is read. The cycle takes 5 × 200 = 1000 ns.

Figure 10.1.13 shows a read with a single wait state. When a slow memory is selected, the address decoder immediately pulls READY to 0. In T3 the processor sees READY = 0 and does not go to T4; it adds a Tw. During Tw, RD', DEN', DT/R' and the address stay as they are. The memory prepares its data in this extra cycle and raises READY to 1. When READY = 1 is sampled in Tw, the next cycle is T4 and the data is read. The cycle takes five cycles instead of four, that is, 5 × 200 = 1000 ns; each wait state adds 200 ns to the cycle.

The timing of READY is tight. At 5 MHz, READY must be stable at least T_{RYHCH} = 118 ns before the rising clock edge in T3. Since the low time of the clock is also about 133 ns, READY must in practice be ready at the start of T3. Moreover, the READY input of the 8086 is not synchronized internally: if the setup and hold times are not met, the processor may behave erratically. This is why the ready signal of the memory and I/O devices is connected not directly to the 8086 but to the RDY input of the 8284A clock generator. The 8284A synchronizes RDY with the clock and gives the 8086 a READY that obeys the rules. The data sheet notes that RDY is sampled toward the ends of T2, T3 and Tw.

Common mistake

Thinking of READY as an active-low "wait" signal is confusing. READY is active high: 1 means "I am ready, continue", 0 means "I am not ready, wait". In a system with no slow device attached READY is held at 1 constantly. A READY whose connection was forgotten and which stays at 0 puts the processor into an endless wait state at the first memory access.

10.1.3.6 How many wait states are needed?

The number of wait states needed for a memory is found by comparing the time the processor allows the memory with the memory's access time. In a read the address becomes valid at most T_{CLAV} after the clock edge at the start of T1. The data is sampled at the end of T3, on the clock edge that goes to T4, and must be ready at least T_{DVCL} before this edge. There are three cycles between these two moments. The delays of the address latch, the address decoder and the data transceiver eat part of the time in between; let us call their sum t_{buf}.

The time left to the memory is written as:

$$t_{allowed} = 3 \cdot T - T_{CLAV} - T_{DVCL} - t_{buf}$$

Every Tw extends this window by one T. With n wait states the allowed time becomes $t_{allowed} + n \cdot T$. The memory's access time t_{access} must not exceed this time:

$$t_{access} \le t_{allowed} + n \cdot T \Rightarrow n = \lceil (t_{access} - t_{allowed}) / T \rceil \quad (n = 0 \text{ if the result is negative})$$

Example 10.1.6

In a 5 MHz 8086 system, T = 200 ns, $T_{CLAV} \le 110$ ns and $T_{DVCL} \ge 30$ ns are taken from the data sheet; the sum of the latch, decoder and transceiver delays is assumed to be $t_{buf} = 40$ ns. How many wait states are needed for a memory with an access time of 700 ns?

1. $t_{allowed} = 3 \times 200 - 110 - 30 - 40 = 600 - 180 = 420$ ns.
2. $n = \lceil (700 - 420) / 200 \rceil = \lceil 280 / 200 \rceil = \lceil 1.4 \rceil = 2$.
3. Check: with 1 Tw the window is 420 + 200 = 620 ns; 620 < 700, not enough. With 2 Tw it is 420 + 400 = 820 ns; 820 ≥ 700, enough.
4. The cycle time is $(4 + 2) \times 200 = 1200$ ns.

For a memory with an access time of 400 ns, since 400 ≤ 420 no wait state is needed; the cycle stays at 800 ns.

The computation uses the ceiling function ($\lceil\ \rceil$), because wait states are added as whole cycles; adding 1.4 cycles is not possible. If a memory misses the window by even 1 ns, one more full cycle is added. A memory whose access time slightly exceeds the window therefore slows the system down by a whole cycle.

This computation examines only the "address to data" condition. In a read there is a second condition. The memory turns on its output only after RD' = 0. The **output-enable time**, from the fall of RD' until the data appears at the output, must also fit in the window. RD' falls at most $T_{CLRL} = 165$ ns after the start of T2. The data must be ready T_{DVCL} before the edge 2T = 400 ns after the start of T2. This second window is 400 − 165 − 30 = 205 ns; the transceiver delay is also subtracted from it. The memory's output-enable time must be shorter than this value. Every Tw lengthens this window by one T as well.

Example 10.1.7

An 8 MHz 8086-2 system uses the same 700 ns memory. The data sheet gives $T_{CLAV} \le 60$ ns and $T_{DVCL} \ge 20$ ns; $t_{buf} = 40$ ns. How many Tw are needed, and how much of the gain of the faster clock do the wait states take back?

1. T = 125 ns; $t_{allowed}$ = 3 × 125 − 60 − 20 − 40 = 375 − 120 = 255 ns.
2. n = ⌈(700 − 255) / 125⌉ = ⌈445 / 125⌉ = ⌈3.56⌉ = 4.
3. Check: with 3 Tw, 255 + 375 = 630 ns, not enough; with 4 Tw, 255 + 500 = 755 ns, enough.
4. The cycle is (4 + 4) × 125 = 1000 ns.

In the 5 MHz system the same memory gave a 1200 ns cycle. Although the clock is 1.6 times faster, the memory cycle shortens only 1.2 times (from 1200 ns to 1000 ns). A slow memory makes the fast processor spend most of its gain in wait states.

10.1.4 Status pins

In every bus cycle the 8086 also reports to the outside, in coded form, what it is doing. The pins that carry this information are called the **status pins**. In maximum mode the pins S2', S1', S0' give the type of the cycle; the 8288 bus controller decodes this code and generates the read and write commands. In both modes the A16–A19 and BHE' pins carry the status bits S3–S7 after the address time. In maximum mode QS1 and QS0 describe what is going on in the prefetch queue. This section treats the three groups in turn and traces which codes appear on each instruction of a program.

10.1.4.1 S2', S1', S0'

S2', S1' and S0' are pins 28, 27 and 26 in maximum mode. Three bits give eight different codes; each code describes a type of bus cycle. The codes are valid in T4 of the previous cycle, in T1 and in T2. In T3, or in a Tw when READY = 1, the pins return to the passive code (1, 1, 1).

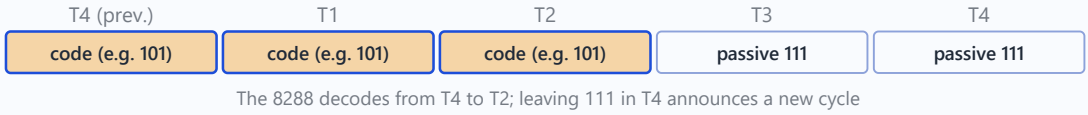


Figure 10.1.14: The S2' S1' S0' status code is valid in T4 of the previous cycle, in T1 and in T2; in T3 it returns to the passive code (111). The change from the passive code to another code announces that a new cycle is starting.

The timing of the status pins makes the 8288's job easier. Between cycles the pins are in the passive code. In T4 a change from the passive code to any code tells the 8288 "a new bus cycle is starting". The 8288 decodes this code in the next T1, generates ALE and prepares the appropriate command signal. That the pins return to passive in T3 also announces that the end of the cycle is near.

S2'	S1'	S0'	Cycle type	8288 command
0	0	0	interrupt acknowledge	INTA'
0	0	1	read I/O port	IORC'

S2'	S1'	S0'	Cycle type	8288 command
0	1	0	write to I/O port	IOWC', AIOWC'
0	1	1	halt (HLT)	none
1	0	0	code access (instruction fetch)	MRDC'
1	0	1	memory read	MRDC'
1	1	0	memory write	MWTC', AMWC'
1	1	1	passive (no cycle)	none

Table 10.1.7: The S2' S1' S0' codes and the command signals the 8288 generates.

Table 10.1.7 can be read by a rule. If S2' = 1 the cycle concerns memory; if S2' = 0 it is I/O or a special cycle. The pair S1' S0' means read when it is 01 and write when it is 10. This rule coincides with the minimum-mode signals: S2' shares a pin (28) with M/IO' of minimum mode, and S1' shares a pin (27) with DT/R'. The pairs 00 and 11 are special cases: for memory 00 describes instruction fetch, on the I/O side 00 describes interrupt acknowledge, and 11 describes halt.

The status pins are active low. The code 111, in which all bits are 1, therefore means "no cycle". If a line breaks or floats it is pulled to 1 and the 8288 does not generate commands for nothing.

Common mistake

It is wrong to think that the codes 100 (code access) and 101 (memory read) produce different commands. The 8288 generates MRDC' for both; for memory they are the same read. The distinction is for other chips that watch the bus. For example, debugging hardware may record only instruction fetches. In maximum mode a system can also build separate memories for code and data.

10.1.4.2 S3–S6 and BHE'/S7

The pins A16/S3, A17/S4, A18/S5 and A19/S6 carry the top four bits of the address in T1 and switch to status bits during T2–T4. These bits exist in both modes.

S4 (A17)	S3 (A16)	Segment used
0	0	ES (extra segment)
0	1	SS (stack)
1	0	CS or no segment (I/O, interrupt vector)
1	1	DS (data)

Table 10.1.8: The segment shown by the bits S4 and S3.

Table 10.1.8 gives how the bits S4 and S3 show which segment register was used to compute the physical address in that cycle. Instruction fetch is always done with CS and gives the code 10. In an I/O access and in reading an interrupt vector the address does not come from a segment register; they also use the code 10. A segment override changes the code: the read of the instruction `MOV AL, ES:[BX]` gives 00 (ES).

The other status bits are:

- **S5 (A18):** the current value of the interrupt flag IF. It is updated at the start of every clock cycle. An external monitoring circuit sees from this bit whether the processor is open to interrupts.
- **S6 (A19):** always 0 on the 8086; it shows that the bus is being used by the 8086.
- **S7 (BHE'):** the spare status bit seen on the BHE' pin in T2–T4; it has no meaning on the 8086.

The bits S3 and S4 offer an interesting system possibility. These bits are not latched with ALE, but they stay on the pins from T2 on and an external circuit can use them as additional address lines. A separate 1 MB memory could thus be attached to every segment: code, data, stack and extra segment go to separate memory banks. The total address space would in theory rise to 4 MB. This arrangement did not become widespread in practice, but it shows well the function of the status bits.

Example 10.1.8

What are S4 S3 in the data cycles of the following accesses?

1. `MOV AX, [SI+2]` : no base register, index SI; default segment DS → 11.
2. `MOV AX, [BP+SI]` : BP is in the address; default segment SS → 01.
3. `STOSB` : the string destination is always ES:DI → 00.
4. Writing the return address to the stack with `CALL` : SS → 01.
5. `OUT DX, AL` : I/O, no segment → 10.

The segment rule follows from how the instruction is written. The status bits are the reflection of this rule in the hardware.

10.1.4.3 Code trace: instructions and status codes

The program below consists of seven instructions that produce various types of bus cycle. Let us trace what is seen on the S2' S1' S0' and S4 S3 pins in every data cycle of an 8086 running in maximum mode, and which command the 8288 generates. It is assumed that the word 1234h is at address DS:0200h in memory and the value 5Ah is at port number 60h. The program is in .COM form; CS = DS = SS = 0700h and initially SP = FFFEh, AX = 0000h.

```
org 100h
mov bx, 0200h
mov ax, [bx]      ; memory read
mov [bx+2], ax    ; memory write
in al, 60h        ; I/O read
out 61h, al       ; I/O write
push ax           ; write to the stack
hlt               ; halt
```

`mov bx, 0200h` works only with a register. On the bus only instruction-fetch cycles are seen for this instruction: S2' S1' S0' = 100, S4 S3 = 10 (CS). For this code the 8288 generates MRDC'; the memory supplies the instruction bytes that CS:IP points to.

`mov ax, [bx]` is a memory read: code 101. The address is DS:0200h; since DS = 0700h the physical address is 07000h + 0200h = 07200h. Because the address is even, the word comes in a single cycle. Since the data segment is used, S4 S3 = 11. The 8288 generates MRDC' and AX = 1234h.

Figure 10.1.15: Left, `mov ax, [bx]` : memory read, S2' S1' S0' = 101, S4 S3 = 11 (DS), the 8288 generates MRDC'. Right, `mov [bx+2], ax` : memory write, 110, the segment is again DS, the 8288 generates MWTC' and AMWC'.

`mov [bx+2], ax` writes to memory: code 110. The effective address is 0202h and the physical address 07202h; again an even address and a single cycle. The 8288 generates MWTC' (the memory write command). It also gives the AMWC' (advanced memory write) output, which starts one clock earlier. Since the segment is DS, S4 S3 = 11 stays. 34h is written to 07202h and 12h to 07203h.

`in al, 60h` is an I/O read: code 001, and the 8288 generates IORC'. Because the I/O address does not come from a segment register, S4 S3 = 10; the lines A19–A16 are 0 in T1. The port supplies 5Ah and AL = 5Ah, AX = 125Ah. Because the port number 60h is even, the byte comes over D7–D0.

Figure 10.1.16: Left, `out 61h, al` : I/O write, 010, the 8288 generates IOWC'; since S2' = 0 memory takes no part in this cycle. Right, `push ax` : again a memory write (110), but the stack segment is used: S4 S3 = 01 (SS).

`out 61h, al` is an I/O write: code 010, and the 8288 generates IOWC' and AIOWC'. Because S2' = 0, the memory control logic takes no part in this cycle at all. Since port 61h is an odd address, the 5Ah in AL goes out over D15–D8 and BHE' = 0.

`push ax` is also a memory write (110), but the address comes from the stack segment. PUSH first decreases SP by 2: SP = FFFCh. The word is written to SS:FFFCh, that is, to physical address 07000h + FFFCh = 16FFCh. S4 S3 = 01 (SS). Thanks to the status bits a system can put the stack in a separate memory.

`hlt` stops the processor. In maximum mode the 8086 gives the code S2' S1' S0' = 011; the 8288 generates no command for this code and only gives an ALE pulse. In minimum mode the processor announces that it has entered the halt state with a single ALE pulse and no control signal. The processor leaves this state only through an interrupt or RESET.

Instruction	S2' S1' S0'	Cycle	S4 S3	8288 command	PA	AX	SP
`mov bx,0200h`	100	code fetch	10 (CS)	MRDC'	CS:0100h	0000h	FFFEh
`mov ax,[bx]`	101	memory read	11 (DS)	MRDC'	07200h	1234h	FFFEh
`mov [bx+2],ax`	110	memory write	11 (DS)	MWTC', AMWC'	07202h	1234h	FFFEh
`in al,60h`	001	I/O read	10	IORC'	port 0060h	125Ah	FFFEh
`out 61h,al`	010	I/O write	10	IOWC', AIOWC'	port 0061h	125Ah	FFFEh
`push ax`	110	memory write	01 (SS)	MWTC', AMWC'	16FFCh	125Ah	FFFCh
`hlt`	011	halt	—	none	—	125Ah	FFFCh

Table 10.1.9: Trace of the status-code program (data cycles only; each instruction also has fetch cycles with code 100).

The register and memory values in Table 10.1.9 come out the same when the program is run in emu8086. In the final state AX = 125Ah and SP = FFFCh; the word at 07202h is 1234h and the word pushed on the stack is 125Ah. The status codes and the 8288 commands, however, are not visible in the emulator; they follow from the rules in Table 10.1.7 and Table 10.1.8. Between cycles the pins return to the code 111 (passive).

10.1.4.4 Decoding the status code

From the values of the five status pins, the type of the cycle and the segment used are found in three steps. First S2' is looked at: if 1 it is memory, if 0 I/O or a special cycle. Then the pair S1' S0' is read: 01 read, 10 write, 00 and 11 special cases. Finally S4 S3 gives the segment.

Example 10.1.9

A logic analyzer shows S2' S1' S0' = 1 0 1 and S4 S3 = 1 1 during T2. What is the cycle and which instruction might have produced it?

1. S2' = 1 → a memory cycle.
2. S1' S0' = 01 → a read. The code 101 is "memory read"; the 8288 generates MRDC'.
3. S4 S3 = 11 → the segment is DS.

Any instruction that reads memory through DS produces this cycle: `MOV AL, [BX]`, `ADD AX, [SI]`, `LODSB` and the like. Had the code been 100, the cycle would be an instruction fetch and S4 S3 = 10 (CS) would be seen.

10.1.4.5 QS1, QS0: the queue status

The prefetch queue of the 8086 is inside the chip. The bytes entering the queue are visible on the bus, but from outside it cannot be known when a given byte starts being executed. In maximum mode pins 24 and 25 give this information to the outside.

QS1	QS0	Meaning
0	0	no operation: the queue did not change in this cycle
0	1	the first byte of an opcode was taken from the queue
1	0	the queue was flushed (a branch)
1	1	a subsequent byte was taken from the queue

Table 10.1.10: The QS1 and QS0 queue status codes.

The queue status is valid in the clock cycle that follows the cycle in which the queue operation was done. The main user of the codes in Table 10.1.10 is the 8087 coprocessor. The 8087 listens to the same bus as the 8086 and copies the fetched instruction bytes into its own queue. With the QS codes it follows which byte the 8086 has taken for execution. When it sees that the byte leaving the queue is an ESC (escape) instruction, it executes that instruction itself. The code 01 reports that a new instruction has started, the code 11 the later bytes of the same instruction. The code 10 shows that the queue was flushed after a branch; the 8087 then clears its own copy.

10.1.5 Control pins

The address and data pins answer the questions "where" and "what". The control pins answer "when" and "how". Is the transfer a read or a write, with memory or with I/O? In which direction will the transceiver turn on, will the bus be given to another master? This section begins with the read/write and transceiver signals of minimum mode and traces the bus cycles of two instructions signal by signal. It then completes the picture with the HOLD/HLDA, RQ'/GT and LOCK' pins that make bus sharing possible, with a semaphore program that uses the LOCK prefix, and with the TEST' input.

10.1.5.1 RD', WR' and M/IO'

RD' (pin 32) is an active-low output and tells that the processor is reading from memory or I/O. It is 0 in the T2, T3 and Tw cycles of a read cycle. This pin exists in both modes. **WR'** (pin 29, minimum mode) is the write pulse; it is 0 in the T2, T3 and Tw cycles of a write cycle. The data sheet stresses a subtle difference between the two signals: WR' falls at the start of T2, while RD' falls slightly later within T2 to give the bus time to float.

M/IO' (pin 28, minimum mode) separates a memory access from an I/O access: 1 means memory, 0 means I/O. The signal becomes valid in T4 before the cycle and does not change until the last T4 of the cycle. Address decoders thus know which address space is selected before the address has even been latched. The condition M/IO' = 1 is added to the select logic of the memory chips and M/IO' = 0 to that of the I/O devices. On the 8088 this signal has the opposite polarity and is called IO/M': 1 means I/O, 0 means memory. It was chosen this way for compatibility with the 8085.

M/IO'	RD'	WR'	Operation	Example instruction
1	0	1	memory read	`MOV AL, [BX]`
1	1	0	memory write	`MOV [BX], AL`
0	0	1	I/O read	`IN AL, 60h`
0	1	0	I/O write	`OUT 43h, AL`

Table 10.1.11: The four basic operations with M/IO', RD' and WR' (minimum mode).

In Table 10.1.11 RD' and WR' are never 0 together in any row; the processor cannot read and write at the same time. Four separate command signals can be derived from these three signals. For example the memory read command is obtained with an OR gate: MEMR' = RD' + (M/IO')'. The output is 0 only when RD' = 0 and M/IO' = 1. The 8288 in maximum mode generates the same four signals (MRDC', MWTC', IORC', IOWC') from the status code.

Example 10.1.10

Let us generate the signals MEMR', MEMW', IOR' and IOW' with gates from the signals RD', WR' and M/IO' in minimum mode. All four signals are active low.

1. MEMR' must be 0 only when RD' = 0 and M/IO' = 1: MEMR' = RD' + (M/IO')'. M/IO' is passed through a NOT gate and ORed with RD'.
2. MEMW' = WR' + (M/IO')': 0 when WR' = 0 and M/IO' = 1.
3. IOR' = RD' + M/IO': 0 when RD' = 0 and M/IO' = 0.
4. IOW' = WR' + M/IO': 0 when WR' = 0 and M/IO' = 0.

Check: in the `OUT 43h, AL` cycle M/IO' = 0, RD' = 1, WR' = 0. MEMR' = 1 + 1 = 1, MEMW' = 0 + 1 = 1, IOR' = 1 + 0 = 1, IOW' = 0 + 0 = 0. Only IOW' is active. A 74LS32 and a 74LS04 are enough for this decoder.

10.1.5.2 DT/R' and DEN'

In a system that uses 8286 or 74LS245 transceivers on the data bus, the processor has to generate two more signals: the direction of the transceiver and when it is turned on.

DT/R' (data transmit / receive, pin 27) is connected to the direction input of the transceiver. When it is 1 the data flows from the processor out (write), when 0 from outside into the processor (read). Its timing is the same as that of M/IO': it becomes valid in T4 before the cycle and stays through the cycle. It is logically equivalent to S1' in maximum mode.

DEN' (data enable, pin 26) is connected to the output-enable (OE') input of the transceiver. It is active low in memory and I/O accesses and in interrupt-acknowledge cycles. It is 0 from the middle of T2 to the middle of T4 in a read and in an interrupt acknowledge, and from the start of T2 to the middle of T4 in a write. In T1 it must be 1: at that moment the AD pins carry the address, and if the transceiver were on, the address would leak onto the data bus.

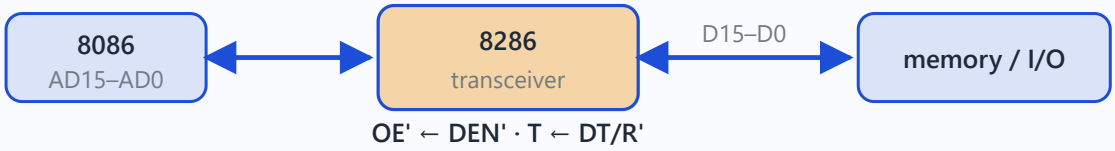


Figure 10.1.17: The 8286 transceiver: the DEN' output enables it (OE'), DT/R' selects the data direction (T).

Figure 10.1.17 summarizes the connection. While DEN' = 0 the transceiver is on and DT/R' selects its direction. In a read DT/R' = 0 and the data flows from memory to the processor; in a write DT/R' = 1 and the data flows from the processor to memory. While DEN' = 1 both sides float; the processor's AD pins and the system data bus are isolated from each other.

Common mistake

Turning the transceiver on in a read at the same time as RD', or even before it, creates contention. In a read DEN' falls in the middle of T2, that is, after the processor has released the AD pins. Had the transceiver turned on earlier, the address that the processor is still driving and the data that the transceiver starts driving would collide on the same lines. In a write there is no such danger; that is why DEN' falls at the start of T2 in a write.

10.1.5.3 From instruction to bus cycles: OUT and IN

The execution of one instruction consists of several bus cycles: first the instruction bytes are fetched, then the instruction's own data cycle is made. The two instructions below are suited to tracing this chain with minimum-mode signals. CS = 01A7h and AL = 12h are assumed; when port 9Ah is read it supplies the value 3Ch.

```
; CS = 01A7h, AL = 12h
; PA      bytes  instruction
01A88h  E6 59  out 59h, al
01A8Ah  E4 9A  in  al, 9Ah
```

The offsets of the instructions are IP = 0018h and 001Ah. The physical addresses are 01A70h + 0018h = 01A88h and 01A70h + 001Ah = 01A8Ah. The instruction `out 59h, al` is encoded as E6 59: E6 is the opcode "write AL to the port with a fixed number" and 59h is the port number. `in al, 9Ah` is E4 9A. To keep the trace simple, the cycles are assumed to be made in sequence.

1. **Cycle 1, T1 (code fetch):** The address 01A88h is even: BHE' = 0, A0 = 0 and a word is fetched in a single cycle. M/IO' = 1, ALE = 1, DT/R' = 0 (read), DEN' = 1.
2. **Cycle 1, T2–T4:** RD' = 0, DEN' = 0. The memory supplies the word: E6h (the opcode) at the even address comes over D7–D0, and 59h (the port) at the odd address over D15–D8. The two bytes enter the queue.
3. **Cycle 2, T1 (I/O write):** M/IO' = 0. The address is meaningful only on A7–A0 (59h); A15–A8 and A19–A16 are zero. The port number is odd: BHE' = 0, A0 = 1. Because it is a write, DT/R' = 1.
4. **Cycle 2, T2–T4:** WR' = 0, DEN' = 0. The 12h in AL goes out over D15–D8; an odd address corresponds to the high bank. An output port usually latches the data on the rising edge of WR'.
5. **Cycle 3, T1 (code fetch):** The next instruction is at 01A8Ah (even): again a word is fetched in a single cycle. M/IO' = 1, DT/R' = 0.
6. **Cycle 3, T2–T4:** E4h (IN AL, imm8) is read from D7–D0 and 9Ah (the port) from D15–D8.
7. **Cycle 4, T1 (I/O read):** The port address 009Ah is even: with BHE' = 1, A0 = 0 only the low bank is selected. M/IO' = 0, DT/R' = 0.
8. **Cycle 4, T2–T4:** RD' = 0, DEN' = 0. The port puts the value 3Ch on D7–D0; the value is taken into AL at the start of T4: AL = 3Ch.

Cycle	Type	A19–A0	BHE'	A0	D15–D8	D7–D0	M/IO'	RD'	WR'	DT/R'	AL
1	code fetch	01A88h	0	0	59h	E6h	1	0	1	0	12h
2	I/O write	00059h	0	1	12h	—	0	1	0	1	12h
3	code fetch	01A8Ah	0	0	9Ah	E4h	1	0	1	0	12h
4	I/O read	0009Ah	1	0	—	3Ch	0	0	1	0	3Ch

Table 10.1.12: The bus cycles of the OUT and IN instructions (minimum mode, no wait states).

Table 10.1.12 collects the T2–T4 values of the four cycles; in T1, RD' and WR' are 1, DEN' is 1 and ALE is 1. The bus activity is 4 cycles × 4 T = 16 clock cycles. At 5 MHz that is 3.2 µs. When the program is run in emu8086, the machine codes (E6 59, E4 9A) and the final value of AL, 3Ch, come out the same; the emulator does not show the pins.

This trace is a simplified sequential model. In reality the BIU prefetches code whenever at least two bytes of room open up in the queue. Cycle 3 may have been made before cycle 2; if the queue is empty, some fetches may also slip into idle cycles. The types of the cycles and the pin values do not change, only their order may.

Common mistake

Connecting an 8-bit peripheral only to D7–D0 and giving it an odd port address does not work. A byte at an odd port address is transferred over D15–D8; a device connected to D7–D0 never sees this data. The data sheet therefore asks that all the registers of an 8-bit device connected to the lower half be given even addresses. If the device is connected to the upper half, its registers are given odd addresses.

10.1.5.4 HOLD and HLDA

The processor is not the only device that uses the bus. A **direct memory access** (DMA) controller, for example the 8237, can write data coming from a disk controller straight to memory without passing it through the processor. For this it has to take the bus over from the processor for a while. In minimum mode this handover is done with the HOLD and HLDA pins.

Definition 10.1.10

HOLD (pin 31) is an active-high input that tells that another bus master wants the local bus. The processor finishes the bus cycle it is in. In the middle of a T4 or Ti cycle it gives its acknowledgement by making the **HLDA** (hold acknowledge, pin 30) output 1. With the acknowledgement it floats the address, data and control pins. When HOLD falls to 0 it drops HLDA and takes the bus back.

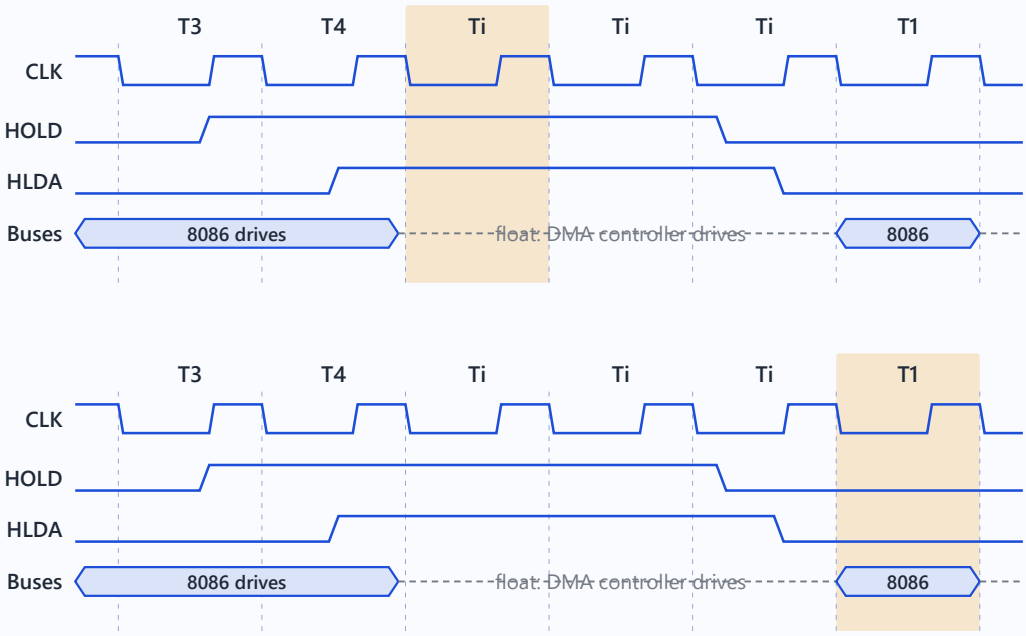


Figure 10.1.18: Handing over the bus with HOLD/HLDA. Top: the processor finishes its cycle and releases the bus with HLDA = 1 in the middle of T4; the pins float. Bottom: when HOLD = 0 the processor drops HLDA, takes the bus back and starts a new cycle.

Figure 10.1.18 shows the steps of the handover. The DMA controller makes HOLD 1. The processor does not cut the current bus cycle short; the cycle finishes and HLDA = 1 in the middle of the next T4 or Ti. At the same moment AD0–AD15, A16–A19, BHE', RD', WR', M/IO', DT/R' and DEN' go into the float state. ALE, however, never floats; it stays at 0 in the meantime. The DMA controller now drives the bus itself.

While the bus is handed over, the processor does not stop entirely. The EU can go on executing the instructions waiting in the queue. It waits only when it comes to a job that needs the bus, that is, a memory access or the queue running empty so that a new fetch is needed. Long-running register instructions such as multiplication can overlap with the DMA.

The data sheet also lists the cycles in which a HOLD request is not granted immediately. If the request comes after T2, it is granted at the first suitable moment after the cycle ends. If the first half of a word at an odd address is being transferred, the bus is not released before the second half is done too. No handover is made between the two cycles of the interrupt-acknowledge sequence, and the bus is not released while an instruction with the LOCK prefix is being executed. The HOLD input is not synchronized internally; if the setup time cannot be met, it must be synchronized to the clock outside with a flip-flop.

Example 10.1.11

In a 5 MHz minimum-mode system the DMA controller makes HOLD = 1 in T1 of a memory read cycle. When is the bus handed over at the earliest?

1. The processor finishes the cycle it has started: the cycles T2, T3 and T4 follow.
2. HLDA becomes 1 in the middle of T4. From the start of T1 to this moment 3.5 cycles pass: 3.5 × 200 = 700 ns.
3. If the cycle were the first half of a word at an odd address, the cycle of the second half would also be waited for and the delay would be 4 cycles longer.

The response time of DMA therefore varies between a few hundred nanoseconds and a few microseconds; long instructions with LOCK stretch this time.

10.1.5.5 RQ'/GT0', RQ'/GT1' and LOCK'

In maximum mode the bus handover is done not with HOLD/HLDA but with the **RQ'/GT0'** (pin 31) and **RQ'/GT1'** (pin 30) pins. Their names come from the words request and grant. These pins are used by other local bus masters such as the 8087 or the 8089. Each pin is bidirectional and carries out a three-pulse handshake on a single line:

1. The master that wants the bus gives a 0 pulse one clock cycle wide on the line: the request (RQ).
2. In a T4 or T1 cycle the 8086 gives a pulse one cycle wide on the same line: the grant (GT). It floats the local bus and enters the bus handover state on the next clock cycle.
3. When the master has finished its job it gives a third pulse on the line: the release. The 8086 takes the bus back on the next cycle.

After every handover between two masters there must be one idle clock cycle. The RQ'/GT0' pin has a higher priority than the RQ'/GT1' pin: if two requests arrive at the same time, the master attached to GT0 gets the bus first. The pins have pull-up resistors inside; if unused they can be left unconnected. Bidirectional communication on a single line saves pins: HOLD and HLDA in minimum mode take two pins, while here two separate masters are supported with two pins.

Definition 10.1.11

LOCK' (pin 29, maximum mode) is an active-low output that asks the other bus masters not to take the system bus. It is active while an instruction written with the `LOCK` prefix is executed: it becomes 0 in the clock cycle after the prefix is decoded and returns to 1 at the end of the last bus cycle of the prefixed instruction. In the interrupt-acknowledge sequence it is also active automatically, from T2 of the first cycle to T2 of the second.

While LOCK' is active, a request that arrives on an RQ'/GT pin is recorded but accepted only after the lock ends. All the bus cycles of the prefixed instruction are thus made back to back, with no other master slipping in between. In multiprocessor systems the LOCK' signal is connected to the arbiter that manages the shared bus (for example the 8289 bus arbiter) and makes sure that the system bus is not released for the duration of the lock either.

10.1.5.6 Code trace: a semaphore with LOCK XCHG

If two processors use a shared region of memory, only one of them may enter that region at a time. For this a flag, a **semaphore**, is kept in the shared memory: 0 means "free", 1 means "busy". A processor that wants to enter the region reads the flag; if it is 0 it makes it 1 and enters. The other processor must not slip in between the read and the write. A `XCHG` with the `LOCK` prefix provides this.

```
org 100h
mov bx, offset flag
retry: mov al, 1
lock xchg al, [bx]   ; read + write
cmp al, 0             ; was it free?
jne retry            ; busy: retry
; ... critical section ...
mov byte ptr [bx], 0  ; release
ret
flag db 0
```

The program is in .COM form and starts at 100h. The instruction lengths are as follows: `mov bx, offset flag` 3 bytes (BB 10 01), `mov al, 1` 2 bytes, `lock xchg al, [bx]` 3 bytes (F0 86 07), `cmp al, 0` 2 bytes, `jne retry` 2 bytes, `mov byte ptr [bx], 0` 3 bytes and `ret` 1 byte. The 16 bytes of code in total fill 100h–10Fh; the variable `flag` is at 0110h. BX = 0110h.

`mov al, 1` puts the "busy" value to be written to the flag in AL. `lock xchg al, [bx]` makes two bus cycles: first it reads the flag, then it writes the 1 in AL to the flag and puts the old value read into AL. The byte F0 is the LOCK prefix. Once the prefix is decoded LOCK' = 0, and for the two cycles the RQ'/GT request of the other master is not accepted. No other processor can slip in between the read and the write.

Figure 10.1.19: A semaphore with LOCK'. Left: `lock xchg` has started, LOCK' = 0; the first bus cycle reads the flag (0) and the other master's request is not accepted during this time. Right: the second cycle writes the 1 in AL to the flag; AL receives the old value (0).

When XCHG is done LOCK' returns to 1 and the bus can be shared again. `cmp al, 0` tests the old value: since the old value was 0, ZF = 1. `jne retry` does not jump because ZF = 1; the flag was free and is now this processor's. Had the other processor run the same code, XCHG would have returned 1

to it, ZF = 0, and it would have kept waiting in the loop. When the critical section is over, `mov byte ptr [bx], 0` releases the semaphore by writing 0 to the flag.

Instruction	Bus cycles	LOCK'	AL	flag [0110h]	ZF
`mov bx, offset flag`	—	1	00h	0	0
`mov al, 1`	—	1	01h	0	0
`lock xchg al, [bx]` (read)	memory read	0	01h	0	0
`lock xchg al, [bx]` (write)	memory write	0	00h	1	0
`cmp al, 0`	—	1	00h	1	1
`jne retry`	— (no jump)	1	00h	1	1
`mov byte ptr [bx], 0`	memory write	1	00h	0	1

Table 10.1.13: Trace of the semaphore program (the flag is initially 0).

The values in Table 10.1.13 come out the same when run in emu8086. After the XCHG, AL = 00h and the flag at 0110h is 1. After the CMP, ZF = 1; after the last instruction the flag is 0 again. There is no other processor in the emulator and the LOCK' pin is not visible; the effect of the lock has been traced according to the rule in the data sheet. CMP also sets PF = 1: the result is 00h and the number of one bits (0) is even.

Common mistake

Building the semaphore from separate instructions with `MOV AL, [BX]`, `CMP AL, 0` and `MOV [BX], 1` does not work in a two-processor system. Between the read and the write the other processor can take the bus and read the same 0. Both see the flag as free and both enter the critical section. Even XCHG is not safe without the LOCK prefix: the bus can be handed over with RQ'/GT between the read and write cycles of the XCHG.

10.1.5.7 TEST'

TEST' (pin 23) is an active-low input that the `WAIT` instruction tests. If TEST' = 0 while the processor executes `WAIT`, it goes on to the next instruction at once. If TEST' = 1 it waits idle and tests the pin again; it executes `WAIT` over and over until TEST' becomes 0. This wait uses no bus cycles. The input is synchronized internally on the rising edge of every clock cycle; according to the data sheet it must stay active for at least 5 clock cycles.

The typical user of the TEST' input is the 8087. The BUSY output of the 8087 is connected to the TEST' input of the 8086. While the 8087 is executing a floating-point operation, BUSY = 1. Before reading the result, the 8086 executes `FWAIT` (the same instruction as WAIT) and waits for the 8087 to finish. If interrupts are enabled, an interrupt may arrive during the wait; the processor services the interrupt and on return executes `WAIT` again.

10.1.6 Clock, reset and power

Three things are needed for a processor to work: a stable supply, a regular clock and a well-defined starting moment. In the 8086, the clock and reset signals are prepared by the 8284A clock generator. This section describes the CLK pin, the RESET pin and where the processor fetches its first instruction after reset, and then the internal functions of the 8284A.

10.1.6.1 CLK

CLK (pin 19) is the clock input that provides the basic timing of the processor and the bus control. The clock of the 8086 is not symmetric: the **duty cycle** is 33%. The clock is 1 for one third of a period and 0 for two thirds. According to the data sheet this asymmetry was chosen to optimize the internal timing. Some operations inside the processor are placed in the long low phase of the clock, some in the short high phase.

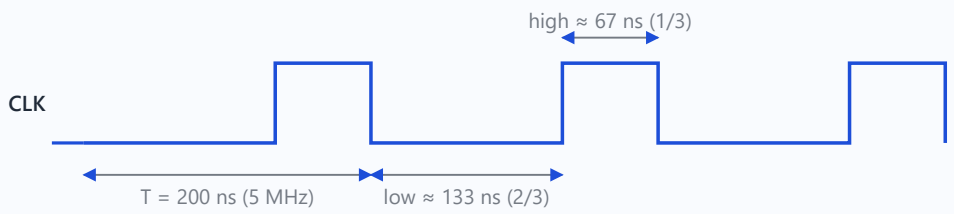


Figure 10.1.20: The 33% duty-cycle clock from the 8284A (5 MHz): period 200 ns, high phase about 67 ns, low phase about 133 ns.

The 8086 was produced in three speed grades: the 8086 runs at 5 MHz, the 8086-2 at 8 MHz and the 8086-1 at 10 MHz. The values T_{CLCH} and T_{CHCL} in Table 10.1.6 limit this asymmetry. At 5 MHz the low phase must be at least 118 ns and the high phase at least 69 ns. In a perfect 33% clock these values are 133 and 67 ns; the lower limit of 118 ns leaves a little margin. The lower limit of 69 ns for the high phase, however, is slightly above 67 ns. The duty cycle of the clock generator is therefore kept not exactly one third but very close to it. It is hard to produce such a clock with an ordinary oscillator; the 8284A is designed for this job.

10.1.6.2 RESET, V_{CC} and GND

Definition 10.1.12

RESET (pin 21) is an active-high input that ends the processor's current activity at once. The signal must stay 1 for at least four clock cycles; at power-up, RESET must fall to 0 at least 50 µs after the supply has settled. While RESET stays 1 the processor sleeps. When RESET falls to 0, an internal start-up sequence of about 10 clock cycles runs, and then the processor begins executing at the instruction at address FFFF0h.

On the rising edge of RESET the processor cuts off all bus operations. All tri-state outputs float; ALE and HLDA are driven to 0. Some of the registers are set to fixed values and the rest remain undefined. These fixed values determine where the system designer must put the first instruction.

Item	Value
CS	FFFFh
IP	0000h
DS, SS, ES	0000h
FLAGS	cleared (IF = 0: interrupts disabled, TF = 0)
Prefetch queue	empty
First instruction address	FFFFh × 10h + 0000h = FFFF0h

Table 10.1.14: The state of the processor after RESET.

The first instruction address in Table 10.1.14, FFFF0h, is the start of the last 16 bytes of the 1 MB address space. Permanent memory, that is, ROM (the BIOS), must be at this address; the contents of RAM are random at power-up. 16 bytes are far too few for a start-up program. So FFFF0h usually holds a far JMP to the real start-up code in ROM. Because IF = 0, the start-up code runs without being interrupted; the interrupt vector table has not been set up yet and an interrupt would mean a crash.

V_{CC} (pin 40) is the +5 V DC supply. There are two pins for GND: 1 and 20. The two ground pins share the large current pulses drawn when the outputs switch at the same time. If a sudden current passes through the thin bond wire of a single ground pin, the wire's inductance creates jumps in the ground level. This is called **ground bounce**. Two pins reduce this noise. On the board, a 0.1 µF decoupling capacitor is placed next to the supply pins.

Common mistake

It is wrong to think that after reset the processor starts from address 00000h. The range 00000h–003FFh is the interrupt vector table and is in RAM. The 8086 starts with CS = FFFFh and IP = 0000h; the first physical address is FFFF0h. The same address can also be written with another segment:offset pair such as F000h:FFF0h, but the values the 8086 loads are

FFFFh and 0000h.

10.1.6.3 The first instruction after RESET: step by step

What the processor does from RESET to the first BIOS code can be traced in eight steps. The example takes the first instruction used in the BIOS of the IBM PC: a five-byte far jump at FFFF0h, `EA 5B E0 00 F0`.

```
; RESET = 1 (≥ 4 clock cycles)
; CS ← FFFFh, IP ← 0000h
; DS, SS, ES ← 0000h
; FLAGS ← 0000h ; IF = 0
; the queue is flushed
; RESET 1 → 0
; fetch FFFF0h   ; EA 5B E0 00 F0
jmp far F000h:E05Bh
; fetch FE05Bh   ; BIOS
```

1. The RESET pin is held at 1 for at least four clock cycles. The processor drops what it is doing at once; the contents of the registers are undefined.
2. The hardware loads CS = FFFFh, IP = 0000h. The programmer does not choose these values; they are built into the processor.
3. The other segment registers DS, SS and ES become 0000h.
4. The flags are cleared: since IF = 0, interrupts are disabled.
5. The queue is flushed. When RESET falls to 0, the BIU starts the first fetch.
6. The first address is FFFFh × 10h + 0000h = FFFF0h. The five-byte far jump in ROM sits here: EA 5B E0 00 F0.
7. EA is the opcode of the far JMP. It is followed first by the offset (E05Bh), then by the segment (F000h); both are written little-endian: 5B E0 → E05Bh, 00 F0 → F000h. When the JMP executes, CS:IP = F000h:E05Bh and the queue is flushed again.
8. The new physical address is F0000h + E05Bh = FE05Bh. In the IBM PC the start-up code of the BIOS is at this address; the start-up program that does not fit in 16 bytes continues from here.

Figure 10.1.21: The first instruction after RESET. Left: the first address is FFFF0h, the last 16 bytes of the 1 MB; EA 5B E0 00 F0 is in ROM. Right: after the far JMP, CS:IP = F000h:E05Bh, the new address is FE05Bh

(BIOS code).

Step	CS	IP	DS/SS/ES	FLAGS	Address bus
RESET = 1	????	????	????	????	floats
CS, IP are loaded	FFFFh	0000h	????	????	—
the segments are zeroed	FFFFh	0000h	0000h	????	—
the flags are cleared	FFFFh	0000h	0000h	0000h	—
first fetch	FFFFh	0000h	0000h	0000h	FFFF0h
`jmp far F000h:E05Bh`	F000h	E05Bh	0000h	0000h	—
BIOS code is fetched	F000h	E05Bh	0000h	0000h	FE05Bh

Table 10.1.15: The registers from RESET to the BIOS code.

Table 10.1.15 summarizes the steps through the registers. The first fetch starts at FFFF0h; the address is even and the BIU fills the queue word by word: FFFF0h, FFFF2h, FFFF4h. After the far jump the queue is flushed and the fetch continues from FE05Bh. Because this address is odd, in the first fetch the BIU takes a single byte, and then continues to fetch words from even addresses.

10.1.6.4 The 8284A clock generator

The 8284A is an 18-pin clock generator designed for the 8086 and 8088. It has three jobs: to generate the CLK signal with a 33% duty cycle, to synchronize READY with the clock and to synchronize RESET with the clock.

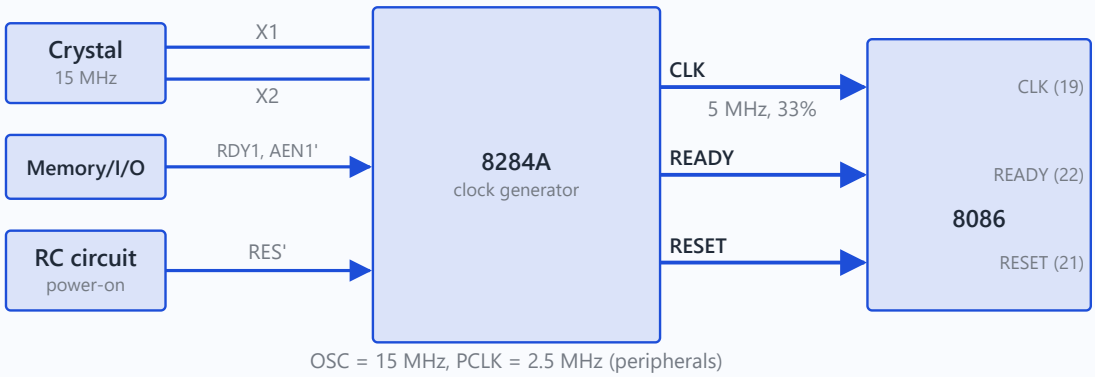


Figure 10.1.22: The wiring of the 8284A clock generator: a 15 MHz crystal on X1–X2, the memory/I/O ready signal on the RDY1 and AEN1' inputs, the power-on RC circuit on the RES' input. The outputs CLK, READY and RESET go to the 8086; OSC and PCLK are for peripherals.

Clock generation starts with a crystal. The crystal connected to the X1 and X2 pins drives the oscillator inside the 8284A. The 8284A divides this frequency by 3: a 15 MHz crystal gives a 5 MHz CLK. Dividing by three also produces the 33% duty cycle by itself: the output is 1 for one of three

crystal periods and 0 for two. The oscillator frequency is also given at the OSC output; the PCLK output is a peripheral clock at half the frequency of CLK with a 50% duty cycle. With a 15 MHz crystal, OSC = 15 MHz and PCLK = 2.5 MHz. If the F/C' input is made 1, the clock is taken from an external source on the EFI input instead of the crystal.

Example 10.1.12

In the IBM PC a 14.31818 MHz crystal is connected to the 8284A. What are CLK, the clock period and the duration of a bus cycle without wait states? How many MHz is PCLK?

1. CLK = 14.31818 / 3 = 4.77273 MHz.
2. T = 1 / 4.77273 MHz ≈ 209.5 ns.
3. The bus cycle is 4 × 209.5 ≈ 838 ns.
4. PCLK = CLK / 2 ≈ 2.386 MHz.

14.31818 MHz is four times the colour subcarrier of colour television (3.579545 MHz). IBM lowered the cost by using this crystal for both the processor clock and the video card. The same computation holds in 8086 systems.

The second job of the 8284A is READY. The ready signals of memory and I/O devices are connected to the RDY1 or RDY2 inputs; the AEN1' and AEN2' inputs select which RDY input is valid. This arrangement serves to handle the ready signals of two separate bus masters with a single 8284A. The ASYNC' input determines whether the synchronization will be one or two stages. The 8284A samples RDY on the clock edges and gives the 8086 a READY signal that meets the setup and hold conditions. Because the READY input of the 8086 is not synchronized internally, this step is mandatory.

The third job is RESET. At power-up the supply rises from 0 V to 5 V in a few milliseconds. An RC circuit connected to the RES' input creates a slowly rising voltage at this input while the capacitor charges. The RES' input has a Schmitt trigger: when the voltage passes a certain threshold the output changes with a clean edge, and the slowly rising voltage does not turn into a noisy signal. The 8284A synchronizes this signal with the clock and gives it to the 8086 as the active-high RESET. A "reset" button on the board triggers the same circuit by discharging the capacitor.

Example 10.1.13

An RC circuit with R = 10 kΩ and C = 10 µF is connected to the RES' input. Let the Schmitt threshold of the RES' input be about 1.7 V (V_{CC} = 5 V). How long does RESET stay active after power is applied?

1. The capacitor voltage rises as $v(t) = 5 \cdot (1 - e^{-t/RC})$; RC = 10,000 × 10 × 10^{-6} = 0.1 s.
2. For v(t) = 1.7 V: $1 - e^{-t/RC} = 0.34 \rightarrow e^{-t/RC} = 0.66 \rightarrow t = -0.1 \cdot \ln 0.66 \approx 0.0416$ s.
3. RESET stays active for about 42 ms. This is far longer than the 50 µs the data sheet asks for and also leaves time for the supply to settle.

> The threshold should be taken from the 8284A data sheet; an approximate value is used here to show the method of computation.

10.1.7 Maximum mode

Minimum mode is sufficient for small single-processor systems. When an 8087 coprocessor, an 8089 I/O processor or a second 8086 is added to the system, a separate chip must generate the control signals. The 8086 too needs pins that support bus sharing. This section once more collects the pins that change with the mode, and then describes the 8288 bus controller and a complete maximum-mode system.

10.1.7.1 MN/MX' and the pins by mode

In minimum mode (MN/MX' = 1) pins 24–31 carry these signals:

- **ALE** (25): address latch enable. It is an active-high pulse in T1 of every bus cycle and loads the 8282/8283 latches. The data sheet specifically notes that ALE never floats: even when the bus is handed over, ALE keeps being driven.
- **INTA'** (24): interrupt acknowledge.
- **DEN', DT/R', M/IO', WR'** (26–29): transceiver control and the memory/I/O distinction.
- **HOLD, HLDA** (31, 30): bus handover (DMA).

In maximum mode (MN/MX' = 0) the same pins turn into these signals:

- **S2', S1', S0'** (28–26): the cycle type; the 8288 generates the control signals from this code.
- **QS1, QS0** (24, 25): the queue status; for the 8087.
- **LOCK'** (29): locks the bus.
- **RQ'/GT0', RQ'/GT1'** (31, 30): bus request and grant.

There is a reason ALE does not float. When the bus is handed over, another master drives the address with its own latches or directly. Had ALE of the 8086 floated, the undefined level set by pull-up resistors could have made the latches transparent by mistake. With ALE driven at 0, the latches of the 8086 stay closed and the address is not corrupted during a bus handover.

10.1.7.2 The 8288 bus controller

Definition 10.1.13

The **8288 bus controller** is the 20-pin chip that, in maximum mode, decodes the S2'–S0' status code of the 8086. From this code it generates the memory and I/O command signals (MRDC', MWTC', AMWC', IORC', IOWC', AIOWC', INTA') and the bus control signals (ALE, DEN, DT/R', MCE/PDEN').

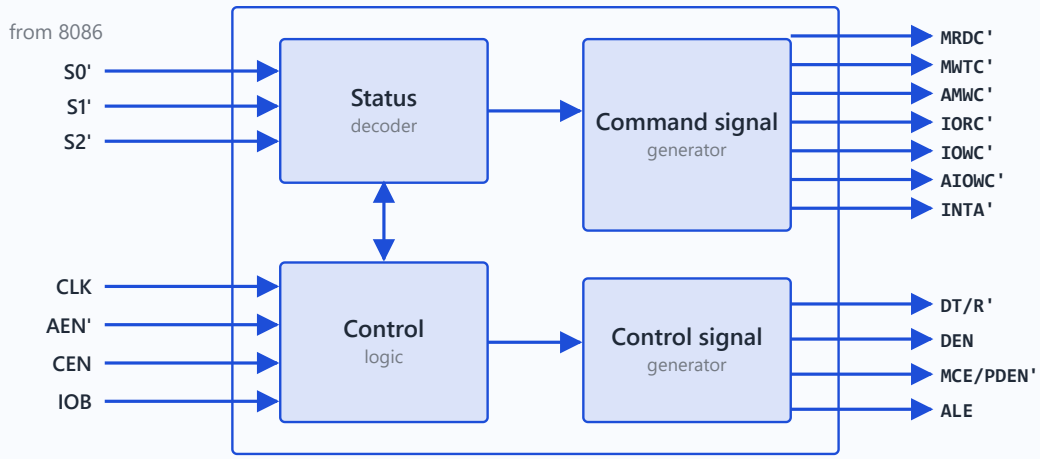


Figure 10.1.23: The internal structure of the 8288 bus controller: the status decoder decodes the S2'–S0' code, the control logic sets up the timing with the CLK, AEN', CEN and IOB inputs; the command and control signal generators give the outputs.

The inputs of the 8288 are the status pins of the 8086 and the clock of the 8284A. The status decoder inside decodes the code in Table 10.1.7. The control logic sets up the timing of the commands with the clock. The outputs are in two groups:

- **Command signals:** MRDC' (memory read), MWTC' (memory write), IORC' (I/O read), IOWC' (I/O write) and INTA' (interrupt acknowledge). AMWC' and AIOWC', which begin with "A", are "advanced" write signals: they start one clock cycle before the normal write signal. They exist so that a slow memory or I/O device can start the write early.
- **Control signals:** ALE, DEN and DT/R' now come from the 8288. DEN is active high here; an inverter may be needed when connecting it to the transceiver. The MCE/PDEN' output is used for cascading interrupt controllers or for buffering the I/O bus separately.

The 8288 also has control inputs. While AEN' (address enable) is 1, the command outputs float; in a multiprocessor system the bus arbiter uses this input when it gives the bus to another processor. While CEN (command enable) is 0, the command outputs stay passive. While IOB (I/O bus mode) is 1, the 8288 generates the I/O commands for a local I/O bus. The 8086, whose pins are freed, uses the QS1, QS0, LOCK' and RQ'/GT pins in place of these signals.

Example 10.1.14

Which signals does the 8288 generate in the data cycle of the instruction `MOV [BX], AX` in maximum mode? BX points to an even address.

1. In T4 of the previous cycle the 8086 gives S2' S1' S0' = 110 (memory write).
2. The 8288 generates the ALE pulse in T1; the address is captured by the latches. DT/R' = 1 (transmit).
3. AMWC' becomes 0 at the start of T2 and MWTC' one clock cycle later. DEN = 1 turns on the transceiver.
4. In T3 the status pins return to passive (111); the 8288 ends the commands in T4.

In minimum mode the same cycle would be made with WR' = 0, M/IO' = 1, DT/R' = 1 and DEN' = 0; the function of the signals is the same, the chip that generates them is different.

10.1.7.3 A maximum-mode 8086 system

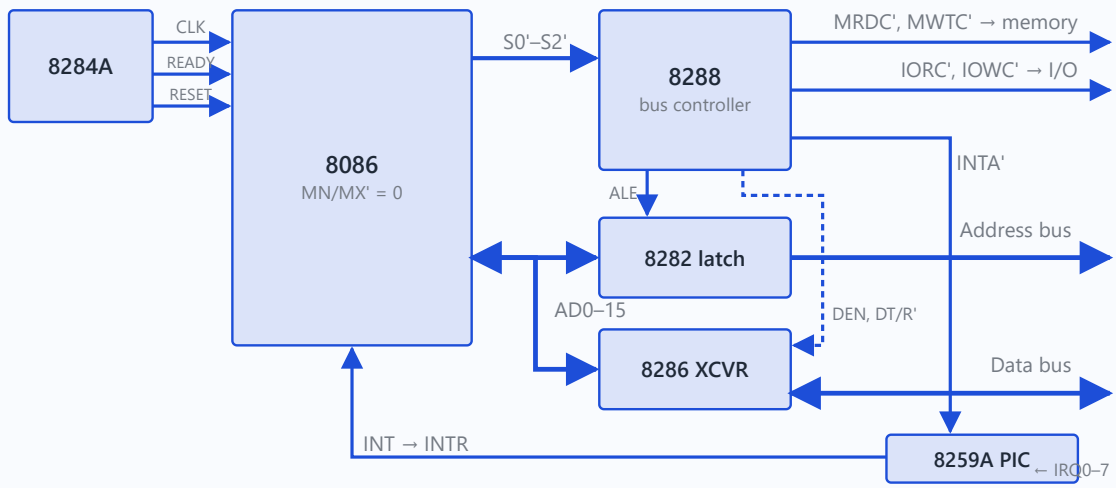


Figure 10.1.24: A maximum-mode 8086 system: the 8284A provides the clock, the 8288 the command and control signals, the 8282 latches the address bus, the 8286 transceivers the data bus; the 8259A interrupt controller connects eight hardware interrupts (IRQ0–IRQ7) to INTR.

Figure 10.1.24 shows a complete maximum-mode system. The 8284A gives CLK to both the 8086 and the 8288; the two must work from the same clock. The S2'–S0' pins of the 8086 go to the 8288. The 8288 controls the 8282 latches with ALE and the 8286 transceivers with DEN and DT/R'; with the command signals it also drives memory and I/O. The 8259A programmable interrupt controller puts eight hardware interrupt requests in priority order, connects them to the INTR input and answers the INTA' pulses with the type number. If an 8087 is to be added to this system, it is connected to the QS1, QS0 and RQ'/GT0' pins; its BUSY output goes to the TEST' input.

Minimum-mode and maximum-mode systems are completely the same from the software's point of view. A program cannot tell which mode it is running in. The difference is only in the hardware design: the number of chips, the ability to share the bus and support for additional processors.

10.1.8 Interrupt pins

The software side of hardware interrupts, that is, the vector table, IRET and interrupt service routines, is treated together with the interrupt mechanism. This section examines the same topic at the level of pins: through which pin and with what timing an interrupt request enters the processor, and through which bus cycles the processor reads the type number from the interrupt controller.

10.1.8.1 INTR, NMI and INTA'

INTR (pin 18) is the maskable interrupt request input. It is active high and **level triggered**: the request is valid as long as the line is held at 1. The processor samples INTR in the last clock cycle of every instruction; in string instructions it also samples at the end of every single transfer. If IF = 1 it accepts the request and enters the interrupt-acknowledge cycles. If IF = 0 the request waits; this is why the CLI instruction disables INTR. The input is synchronized internally on the rising edge of every clock cycle.

NMI (pin 17) is the non-maskable interrupt input. It is **edge triggered**: a transition from 0 to 1 starts the request. The rising edge is latched inside the chip and serviced at the end of the current instruction. NMI always leads to a type 2 interrupt; the type number is not read from outside. It cannot be masked by software, that is, by IF, and has higher priority than INTR. The data sheet asks that NMI stay at 1 for at least two clock cycles. Its typical uses are a power-failure warning and a memory parity error.

INTA' (pin 24, minimum mode) is the read pulse of the interrupt-acknowledge cycles. It is an active-low output and is 0 in the T2, T3 and Tw cycles of every interrupt-acknowledge cycle. In maximum mode the INTA' signal is generated by the 8288.

Property	INTR (18)	NMI (17)
Triggering	level, active high	rising edge
Masking	with IF = 0 (CLI)	cannot be masked
Sampling	in the last clock cycle of an instruction	the edge is latched internally, serviced at the end of the instruction
Type number	read from D7–D0 in the second INTA' cycle	always 2
Acknowledge cycle	two INTA' cycles	none
Use	peripherals through the 8259A	power failure, parity error

Table 10.1.16: Comparison of the INTR and NMI pins.

Common mistake

Thinking INTR is edge triggered leads to two kinds of error. A short pulse is lost if it ends before the processor samples it; INTR must be held at 1 until it is accepted. On the other hand, if the request is still 1 when the service routine ends, the processor enters the same interrupt again. The source of the request must therefore drop its request when it sees the acknowledgement; the 8259A does this with the INTA' pulses. The one that is edge triggered is NMI.

10.1.8.2 A hardware interrupt: the INTA cycles

When INTR = 1 and IF = 1, the processor runs two consecutive **interrupt-acknowledge cycles** after the current instruction ends. An interrupt-acknowledge cycle resembles a read cycle; the difference is that the INTA' signal becomes active instead of RD', and that the address bus floats.

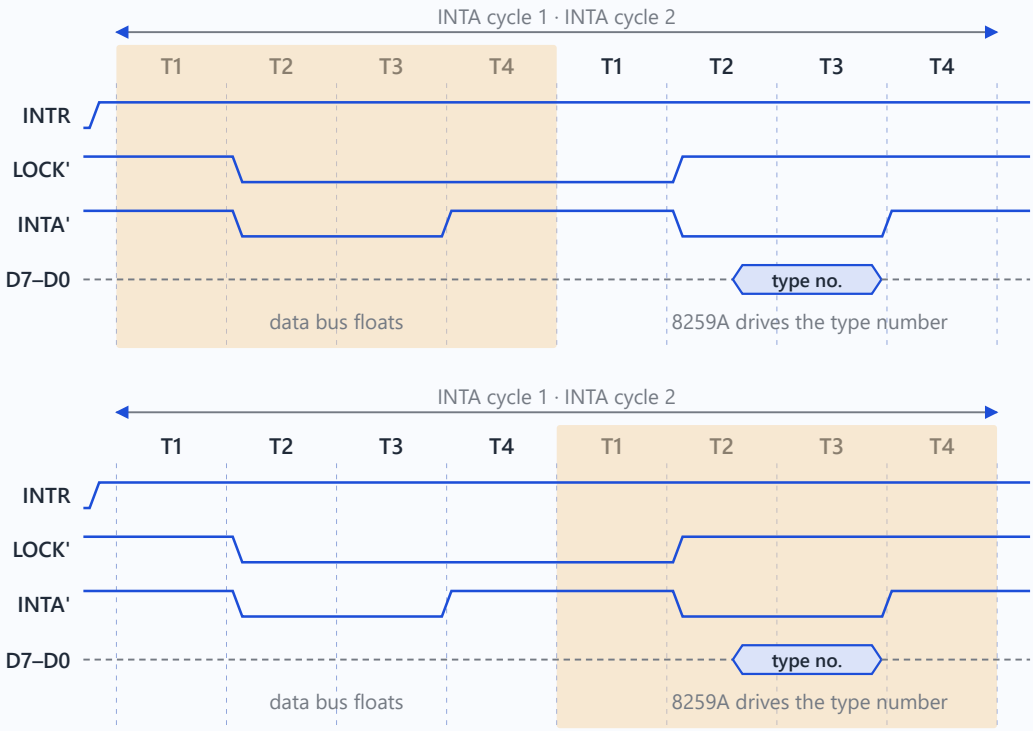


Figure 10.1.25: The two INTA' cycles. Top, the first cycle: the INTA' pulse tells the interrupt controller that the request has been accepted; the data bus stays floating. Bottom, the second cycle: the 8259A puts the type number on D7–D0 and the processor reads it.

1. **The first INTA' cycle:** The processor makes INTA' 0 throughout T2–T3. This pulse tells the interrupt controller (8259A) that the request has been accepted. The 8259A selects the highest-priority one among the pending requests and prepares the type number. In this cycle the data bus stays floating; no data is read. According to the data sheet BHE' is 0 in T1 of this cycle.

2. **The second INTA' cycle:** In the second INTA' pulse the 8259A puts the type number on the D7–D0 lines. For example, for IRQ0 in the PC this number is 08h. The processor reads the byte at the start of T4.
3. **The vector:** The processor multiplies the type number by 4 and finds the address in the interrupt vector table: 08h × 4 = 00020h. It reads the new IP from this address and the new CS from 00022h. It pushes FLAGS, CS and IP on the stack, clears IF and TF and jumps to the service routine.

In maximum mode LOCK' is automatically active so that no other bus master slips in between the two cycles. According to the data sheet LOCK' is 0 from T2 of the first cycle to T2 of the second. A bus handover request is not accepted until the end of the second cycle either. Had a DMA cycle come in between, the two-pulse handshake of the 8259A could have been disrupted.

Example 10.1.15

In a 5 MHz minimum-mode system the 8259A reports the IRQ1 (keyboard) request with type 09h. How many bus cycles and how much time do the two INTA' cycles and the vector read take? From which physical addresses is the service routine's address read?

1. Two interrupt-acknowledge cycles: 2 × 800 ns = 1.6 µs. In the second cycle 09h is read from D7–D0.
2. The vector address is 09h × 4 = 00024h. IP is read from 00024h and CS from 00026h; since both addresses are even, the two word reads take 2 cycles.
3. Pushing FLAGS, CS and IP on the stack is 3 word writes; if the stack is at an even address, 3 cycles.
4. In total at least 2 + 2 + 3 = 7 bus cycles: 7 × 800 ns = 5.6 µs. To this are added the internal operation cycles and the fetching of the first instructions of the service routine.

If the stack were at an odd address, every push would take two cycles and the time would be 3 cycles longer. This is why the data sheet recommends keeping the stack at an even address.

10.1.9 Exercises and solutions

The eight exercises below use knowledge about pins, bus cycles and the clock together. The solution of each is given with its intermediate steps.

Example 10.1.16

The 8086 runs at 8 MHz. How many ns does a bus cycle without wait states take? What if two Tw are added?

1. T = 1 / 8 MHz = 125 ns.
2. A cycle without wait states: 4 × 125 = 500 ns.

3. With two Tw: (4 + 2) × 125 = 6 × 125 = 750 ns.

Example 10.1.17

In minimum mode, when writing to a peripheral with `OUT`, what values do the pins M/IO', RD', WR' and DT/R' take in T2–T4 of the data cycle?

1. The target is I/O: M/IO' = 0.
2. The operation is a write: RD' = 1, WR' = 0.
3. The data flows from the processor outward: DT/R' = 1.

DEN' is also 0 throughout T2–T4; in T1 it is 1.

Example 10.1.18

How many data bus cycles does `MOV [2003h], AX` need? What are BHE' and A0 in each cycle? (DS = 0000h.)

1. The physical address 02003h is odd; the word is split into two cycles.
2. Cycle 1: address 02003h, BHE' = 0, A0 = 1; AL is written to the high bank over D15–D8.
3. Cycle 2: address 02004h, BHE' = 1, A0 = 0; AH is written to the low bank over D7–D0.

Run in emu8086 with AX = BEEFh, EFh is written to 02003h and BEh to 02004h.

Example 10.1.19

What is the difference between S2' S1' S0' = 1 0 0 and 1 0 1? Which command does the 8288 generate for each?

1. 100: code (instruction) fetch. 101: data read.
2. Both are reads from memory; the 8288 generates MRDC' for both.

The difference is not for the memory but for chips that watch the bus, such as the 8087 or debugging hardware; they have to tell instruction bytes from data reads.

Example 10.1.20

The 8259A reports the IRQ0 request with type 08h. What is seen on the data bus in the two INTA' pulses? In maximum mode, what is the state of LOCK' during this time?

1. 1st pulse: the data bus floats; only the acknowledgement is given.

2. 2nd pulse: the 8259A puts 08h on D7–D0; the processor reads it.
3. LOCK' stays 0 (active) from T2 of the 1st cycle to T2 of the 2nd cycle.

The processor then reads the vector from address 08h × 4 = 00020h.

Example 10.1.21

What CLK frequency does the 8284A generate if the crystal is 24 MHz? For which 8086 version is this suitable?

1. CLK = 24 / 3 = 8 MHz, T = 125 ns.
2. 8 MHz is the speed of the 8086-2. The 5 MHz 8086 cannot be run with this clock: T_{CLCL} must be at least 200 ns.

Example 10.1.22

In a system with T = 125 ns, $T_{CLAV} + T_{DVCL} + t_{buf}$ = 120 ns is assumed. How many Tw are needed for a memory with an access time of 450 ns?

1. $t_{allowed} = 3 \times 125 - 120 = 375 - 120 = 255$ ns.
2. $n = \lceil (450 - 255) / 125 \rceil = \lceil 195 / 125 \rceil = \lceil 1.56 \rceil = 2$.
3. Check: with 1 Tw, 380 ns, not enough; with 2 Tw, 505 ns, enough.
4. The cycle is 6 × 125 = 750 ns.

Example 10.1.23

In the I/O write cycle of the instruction `OUT 9Bh, AL`, what are A7–A0, BHE', A0 and the lines the data passes over?

1. The port number goes out on the address lines: A7–A0 = 9Bh, A19–A8 = 0. M/IO' = 0.
2. 9Bh is odd (1001 1011b, lowest bit 1): BHE' = 0, A0 = 1.
3. A byte at an odd address goes to the high bank: AL goes out over D15–D8.

If this port were on an 8-bit device connected to D7–D0, it would never see the data.

10.1.10 Summary

- **The package:** The 8086 is a 16-bit HMOS processor in a 40-pin DIP package, working from a single +5 V supply (pin 40; ground 1 and 20). The apostrophe at the end of a name marks an active-low signal. The pins are divided into seven groups: power, clock/reset, address/data, address/status, interrupt, common control and mode-dependent pins.
- **Multiplexing:** More than 60 signals fit on 40 pins through time-division multiplexing. AD0–AD15 carry the address in T1, then data; A16–A19 carry the address in T1, then S3–S6. On the falling edge of the ALE pulse the 8282/74LS373 latches capture the address; three latches are needed for 20 bits and BHE'. The 8286/74LS245 transceivers isolate the data bus and provide drive strength.
- **Banks:** The 1 MB memory is divided into two 512 KB banks: even addresses on D7–D0 (A0 = 0), odd addresses on D15–D8 (BHE' = 0). A word at an even address takes one cycle, a word at an odd address two cycles; a byte access is always a single cycle.
- **The bus cycle:** It consists of at least four T states: T1 address, T2 direction/control, T3 data, T4 finish. 800 ns at 5 MHz. In a write the data is on the pins from T2 on; in a read the AD pins float in T2 (bus turnaround) and the data is sampled at the start of T4.
- **READY:** It is sampled during T3; if 0, a Tw is inserted between T3 and T4. The number of wait states is found with $n = \lceil (t_{access} - t_{allowed}) / T \rceil$; $t_{allowed} = 3T - T_{CLAV} - T_{DVCL} - t_{buf}$. READY is given synchronized through the RDY input of the 8284A.
- **Status pins:** S2' S1' S0' (maximum mode) encode the cycle type: 000 interrupt acknowledge, 001 I/O read, 010 I/O write, 011 HLT, 100 code fetch, 101 memory read, 110 memory write, 111 passive. S4 S3 give the segment (00 ES, 01 SS, 10 CS/none, 11 DS); S5 = IF, S6 = 0. QS1 QS0 report the queue status to the 8087.
- **Control pins:** RD' and WR' are the read/write pulses, M/IO' separates memory from I/O. DT/R' controls the direction of the transceiver and DEN' its turning on; DEN' is 1 in T1. HOLD/HLDA (minimum) and RQ'/GT (maximum) hand the bus to another master. LOCK' locks the bus during a prefixed instruction; an indivisible semaphore is built with LOCK XCHG. TEST' is the input that the WAIT instruction waits on.
- **Clock and reset:** The 8284A divides the crystal by 3 (15 MHz → 5 MHz), generates the CLK with a 33% duty cycle and synchronizes READY and RESET. RESET is held at 1 for at least 4 clock cycles; afterwards CS = FFFFh, IP = 0000h, DS = SS = ES = 0000h, the flags are clear (IF = 0) and the first instruction is fetched from FFFF0h.
- **Maximum mode:** With MN/MX' = 0, pins 24–31 turn into QS1, QS0, S0'–S2', LOCK' and the RQ'/GT pins; the 8288 bus controller generates MRDC', MWTC', AMWC', IORC', IOWC', AIOWC', INTA', ALE, DEN and DT/R' from the status code.
- **Interrupt pins:** INTR is level triggered, is masked by IF and is acknowledged with two INTA' cycles; the type number is read from D7–D0 in the second cycle. NMI is triggered by the rising edge, cannot be masked and is always type 2.

10.1.11 Key terms

dual in-line package (DIP) active low / active high multiplexing latch buffer (bus transceiver) tri-state output, high impedance memory bank aligned / unaligned access bus cycle T state wait state (Tw) idle state (Ti) bus turnaround access time status code minimum / maximum mode bus controller (8288) clock generator (8284A) duty cycle reset (RESET) direct memory access (DMA) bus master semaphore interrupt acknowledge (INTA) non-maskable interrupt (NMI) programmable interrupt controller (8259A)

10.1.12 Test Yourself

1. For the 8086-2 (8 MHz) the data sheet gives $T_{CLAV} \le 60$ ns and $T_{DVCL} \ge 20$ ns; $t_{buf} = 40$ ns. How many wait states does a memory with an access time of 300 ns need, and how many ns does the cycle take?

   Answer

   T = 125 ns; $t_{allowed} = 375 - 60 - 20 - 40 = 255$ ns. $n = \lceil (300 - 255) / 125 \rceil = \lceil 0.36 \rceil = 1$. Check: with 0 Tw, 255 ns is not enough; with 1 Tw, 380 ns is enough. The cycle is $5 \times 125 = 625$ ns.

2. With DS = 1000h, `MOV [1FFFh], AX` runs. To which physical addresses and in how many cycles is it written? What are BHE', A0 and the lines used in each cycle?

   Answer

   PA = 10000h + 1FFFh = 11FFFh, an odd address; two cycles. Cycle 1: 11FFFh, BHE' = 0, A0 = 1, AL over D15–D8. Cycle 2: 12000h, BHE' = 1, A0 = 0, AH over D7–D0.

3. In maximum mode, what are S2' S1' S0' and S4 S3 in these cycles: the read of `POP BX`, the read of `IN AL, DX`, the write of `MOV ES:[DI], AL`?

   Answer

   `POP BX`: memory read 101, segment SS → 01. `IN AL, DX`: I/O read 001, no segment → 10. `MOV ES:[DI], AL`: memory write 110, segment ES → 00.

4. In minimum mode, what values do M/IO', RD', WR', DT/R' and DEN' take in T2–T4 of an

instruction fetch cycle? Are these values different from the data cycle of `MOV AL, [BX]` ?

Answer

M/IO' = 1, RD' = 0, WR' = 1, DT/R' = 0, DEN' = 0. The values are the same as for a memory read; an instruction fetch cannot be told from a data read by the control pins. The distinction is made by the S4 S3 bits in T2–T4 (10 = CS in a fetch, 11 = DS in a read of `[BX]`) and, in maximum mode, by the status code (100 / 101).

5. What happens if a D flip-flop chip that captures on the rising edge is used as the address latch and ALE is connected directly to its clock input? Explain with the timing values of the 5 MHz 8086.

Answer

The flip-flop captures the address on the rising edge of ALE. ALE rises at most T_{CLLH} = 80 ns after the start of T1; the address becomes valid at most T_{CLAV} = 110 ns after. At the moment of capture the address may not yet have settled and a wrong address is latched. The fix is to invert ALE and move the capture to the falling edge, or to use a transparent latch.

6. A 30 MHz crystal is connected to the 8284A. What are the CLK, PCLK and OSC frequencies? How many ns does a bus cycle without wait states take, and about how many ns is the high phase of the clock?

Answer

CLK = 30 / 3 = 10 MHz (8086-1), PCLK = 5 MHz, OSC = 30 MHz. T = 100 ns; the bus cycle is 400 ns. The high phase is about T / 3 ≈ 33 ns, the low phase about 67 ns.

7. After RESET the bytes `EA 00 E0 00 F0` are at address FFFF0h. What does this instruction do, and from which physical address does the processor fetch the next instruction? What is IF at that moment?

Answer

EA is a far JMP. The offset 00 E0 → E000h, the segment 00 F0 → F000h. CS:IP = F000h:E000h, physical address F0000h + E000h = FE000h. Because reset clears the flags, IF = 0; interrupts are disabled.

8. A semaphore is built with the sequence `MOV AL, [BX]` / `CMP AL, 0` / `JNE retry` / `MOV BYTE PTR [BX], 1` . Why is it not safe in a two-processor system? How many bytes and which prefix fix

it?

Answer

The read and the write are in separate instructions; in between, the other processor can take the bus and read the same 0, and both enter the critical section. The fix: `MOV AL, 1` and `LOCK XCHG AL, [BX]`. XCHG does the read and the write in one instruction; the LOCK prefix (F0h, 1 byte) prevents the bus from being handed over between the two cycles.

9. In minimum mode a DMA controller makes HOLD = 1 in T3 of a read cycle. Does the processor release the bus in T4 of this cycle?

Answer

No. According to the data sheet, for the bus to be released in T4 of that cycle the request must have arrived by T2 at the latest. A request that arrives in T3 is accepted in the next suitable T4 or idle cycle; HLDA becomes 1 in the middle of that cycle.

10.1.13 Further reading

- Intel, *8086 16-Bit HMOS Microprocessor* data sheet (231455): the numerical values in these notes are taken from the pin descriptions, the timing diagrams and the AC characteristics table.
- B. B. Brey, *The Intel Microprocessors* (8th ed.), Chapter 9: the pins of the 8086/8088, the clock generator, bus buffering, bus timing, wait states with READY and minimum/maximum mode, explained with diagrams.
- D. V. Hall, *Microprocessors and Interfacing* (2nd ed.): address decoding of an 8086 system, memory and I/O interfacing and maximum-mode design with the 8288, worked through on example boards.
- K. Shirriff's 8086 articles on righto.com: the bus interface, the queue and the microcode examined at the transistor level through die photographs.

10.1.14 References

1. Intel, *8086 16-Bit HMOS Microprocessor*, data sheet (231455).
2. Intel, data sheets for the *8284A Clock Generator and Driver*, the *8288 Bus Controller* and the *8282/8283 Octal Latch*.
3. Texas Instruments, data sheets for the *SN74LS373 Octal Transparent D-Type Latches* and the *SN74LS245 Octal Bus Transceivers*.
4. Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 (Chapter 9: 8086/8088 hardware specifications).

5. Douglas V. Hall, *Microprocessors and Interfacing: Programming and Hardware*, 2nd ed., McGraw-Hill.

6. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz* (in Turkish), EMO, 2014.

7. Ken Shirriff, 8086 die photographs and reverse-engineering articles, righto.com.

PART IV · CHAPTER 10.2

Device Control

A microprocessor reaches the outside world through the bytes it writes to peripherals and reads back from them. Lighting a traffic lamp, turning a motor by one step, and reading a temperature are the same job from the processor's point of view: write a number to a given port, or read a number from it. This note covers the two 8086 instructions that do this, `IN` and `OUT`, the I/O address space, and how the port number is written into the instruction. It then compares the three ways of waiting for a device to become ready (polling, interrupts, direct memory access). On the hardware side it treats the 8255 programmable peripheral interface, the unipolar stepper motor driver, and the 7-segment display. Programs that drive the seven virtual devices of emu8086 are traced line by line at register level: traffic lights, stepper motor, LED display, thermometer, printer, robot, and the I/O test device.

LEARNING OUTCOMES

Port-based I/O
Explain the four forms of `IN` and `OUT`, how the port number is given (constant or in DX), and the rule that data passes only through AL or AX; compare port-mapped and memory-mapped I/O.

Bit-level control
Decode how each bit of a port corresponds to a lamp, a magnet, or a status flag; compute the byte or word that produces a desired behavior.

Polling
Write loops that wait for a device's ready or busy bit; compare the cost of polling with interrupts and direct memory access.

Timing
Compute the CX:DX wait time for INT 15h / AH = 86h; find the number of clock cycles and the duration of a software delay built with LOOP.

Real interface chips
Build the control word and the bit set/reset word of the 8255 PPI; drive a unipolar stepper motor through a ULN2003 and a 7-segment display by multiplexed scanning.

emu8086 devices
Read, single-step, and modify the programs that drive the traffic lights, stepper motor, LED display, thermometer, printer, robot, and test device.

Embedded-system view
Explain how a program is loaded from ROM into RAM and runs in an endless loop.

10.2.1 I/O basics

10.2.1.1 Peripherals, ports, and embedded systems

On its own, a microprocessor deals only with the numbers in its registers and memory. A lamp lighting up, a motor turning, or a key being pressed is the business of a circuit outside the processor. The circuits that attach to the processor and exchange data with it are called **peripherals**. A keyboard, a video card, a printer, a timer chip, an analog-to-digital converter, and a motor driver are all peripherals.

What a peripheral shows to the processor is one or more registers. One register holds the command or data going to the device. Another holds data coming from the device, and a third reports the device's status: ready, busy, or in error. Each of these registers is selected by an address. The gateway through which the processor reads and writes at these addresses is called an **I/O port**.

Definition 10.2.1

An **I/O port** is the address, as seen by the processor, of a register in a peripheral. On the 8086 a port is read with `IN` and written with `OUT`. Only the hardware attached to the port decides what a written value means: the same `OUT` instruction lights a lamp on one device and turns a motor on another.

Port numbers belong to the system, not to the processor. The 8086 only produces an address; the **address decoder** on the motherboard decides which device answers at that address. An address decoder is a gate circuit that looks at the bits on the address bus and enables a device only within its own address range. For this reason base port numbers can vary from system to system, and the same device can be moved to another port by a different decoder. On the IBM PC the keyboard controller sits at 60h and the interrupt controller at 20h–21h; the virtual traffic lights of emu8086 are at port 4. These numbers are the result of a hardware design, not a rule of the 8086.

The most typical home of programs that drive peripherals is the **embedded system**. A washing machine, a printer, a modem, and a traffic-light controller are built around a microprocessor or microcontroller. These devices often have no operating system or disk. The program is compiled on a

development computer and converted to a plain binary file (.bin). The file is written to a **ROM** (read-only memory) chip. When the device powers up, the processor reads the program from ROM, copies it to RAM if necessary, and starts executing. Figure 10.2.1 shows these three steps.

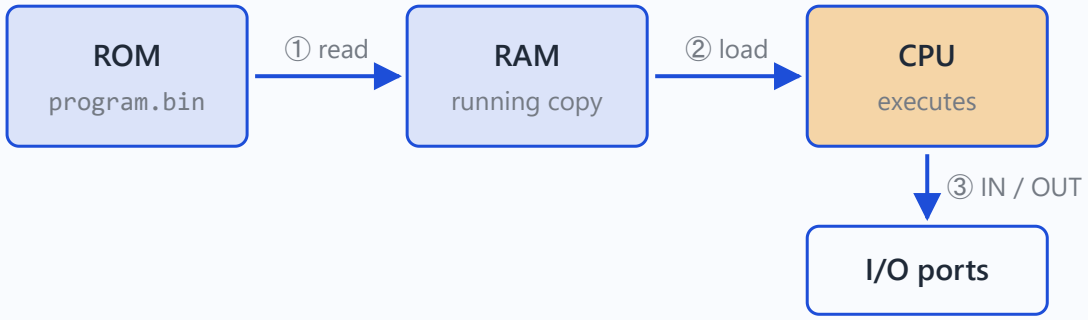


Figure 10.2.1: The path of a program in an embedded system. (1) At power-up the processor reads `program.bin` from ROM, (2) loads the running copy into RAM, (3) the program talks to the devices through I/O ports, with the `IN` and `OUT` instructions.

An embedded program has a shape of its own. A desktop program returns to the operating system when it finishes; an embedded program has nowhere to return to. It therefore begins with a setup part and then enters a loop that never ends. On every turn the loop reads the state of the devices, makes a decision, and writes new values to the devices. The traffic-light, stepper-motor, and thermostat programs have exactly this shape.

emu8086 imitates this world in software. Seven virtual devices attached to the 8086 emulator wait at specific port numbers (Table 10.2.1). The devices are opened from the *virtual devices* menu. If a directive such as `#start=Traffic_Lights.exe#` is written at the top of a program, the emulator opens the device window itself when the program starts. This line is not an 8086 instruction; it is only an instruction to the emulator and adds no bytes to the compiled code.

Device	Port(s)	Direction	Data
Traffic lights	4	OUT	word: 12 lamp bits
Stepper motor	7	OUT / IN	bits 0–2 magnets, bit 7 ready
Robot	9, 10, 11	OUT / IN	command, data, status
Test device (*simple io test*)	110, 112	IN / OUT	byte (110), word (112)
Thermometer and heater	125, 127	IN / OUT	temperature, heater on/off
Printer	130	OUT	character (via INT 21h/05h)
LED display	199	OUT	word: signed number

Table 10.2.1: Virtual devices attached to emu8086 and their ports.

All the port numbers in the table are less than 255. This detail looks small but it determines how the instruction is written: a port number in the range 0–255 can be written directly inside the instruction.

10.2.1.2 The IN and OUT instructions

The 8086 has two instructions that access ports. `IN` reads from a port and writes the value read into the accumulator (AL or AX). `OUT` writes the value in the accumulator to a port. The port number is given in one of two ways: as an 8-bit constant inside the instruction if it lies between 0 and 255, and in the DX register otherwise. Table 10.2.2 lists all the forms.

Form	Meaning	Port range	Machine code	Clock cycles
`IN AL, port`	read 1 byte from the port	0–255 (constant)	`E4 pp`	10
`IN AX, port`	read 1 word from the port	0–255 (constant)	`E5 pp`	10
`IN AL, DX` / `IN AX, DX`	port number in DX	0–65.535	`EC` / `ED`	8
`OUT port, AL` / `OUT port, AX`	write to the port	0–255 (constant)	`E6 pp` / `E7 pp`	10
`OUT DX, AL` / `OUT DX, AX`	port number in DX	0–65.535	`EE` / `EF`	8

Table 10.2.2: Forms of IN and OUT, their machine codes, and 8086 timings.

The machine codes make the logic of the forms clear. The constant-port form is two bytes: the first byte is the opcode and the second is the port number (pp). For example `IN AL, 7` assembles to `E4 07`, `OUT 4, AX` to `E7 04`, and `IN AX, 112` to `E5 70` (112 = 70h). Because only one byte is reserved for the port number, a number above 255 does not fit this form. The DX form is a single byte; the port number is in the register, not in the instruction. `OUT DX, AL` is just `EE`. The lowest opcode bit selects the data width: 0 means byte (AL), 1 means word (AX). That is why the pairs E4/E5, E6/E7, EC/ED, and EE/EF differ by 1.

Figure 10.2.2: The data path of IN and OUT. Data passes only through AL or AX; the port number is the constant n in the instruction, or DX.

Both instructions share one strict rule about the accumulator. Data passes only through AL (byte) or AX (word). `OUT 4, BX` and `IN CL, 7` cannot be written; the 8086 has no such opcode. To send a value held in BX to a port, write `MOV AX, BX` first. The order of the operands is fixed as well: in `OUT` the port comes first and AL/AX second; in `IN` it is the reverse. The order follows the "destination, source" convention of `MOV`: the destination of `OUT` is the port, the destination of `IN` is the accumulator.

These instructions do not touch the flags. `IN AL, 125` leaves SF unchanged even if the temperature read is negative; testing the value needs a `CMP` or `TEST` afterwards. The value read goes only into the accumulator, and no other register changes.

The word form uses two consecutive ports together. `OUT 4, AX` writes AL to port 4 and AH to port 5. `IN AX, 112` takes AL from port 112 and AH from port 113. As with words in memory, the low byte is at the low address (little-endian). The 16-bit data bus of the 8086 transfers an even-addressed word in a single bus cycle. An odd-addressed word needs two separate byte transfers and takes 4 more clock cycles. This is one reason device registers are placed at even addresses.

The clock counts hold an interesting detail. According to Intel's manual the DX form takes 8 cycles, faster than the constant-port form (10 cycles), and it is one byte shorter. If DX is loaded once for a frequently accessed port and reused throughout a loop, every access saves two cycles.

Common mistake

Writing `OUT 7, BL`, `IN BX, DX`, or `OUT AL, 7`. The first two break the accumulator rule and the last breaks the operand order. The assembler rejects all three. The correct forms are `MOV AL, BL` · `OUT 7, AL`, `IN AX, DX` · `MOV BX, AX`, and `OUT 7, AL`.

10.2.1.3 Port number: constant or in DX

The I/O address space of the 8086 is 16 bits wide: 65,536 byte ports from 0000h to FFFFh. The first 256 of them are reached with a constant inside the instruction. For the rest, the port number is first loaded into DX. For example, writing a byte to port 300 is done like this:

```
mov dx, 300     ; 300 = 012Ch > 255
mov al, 5Ah
out dx, al      ; port 012Ch ← 5Ah
```

The first instruction loads 012Ch into DX (`BA 2C 01`). The second puts the data to send into AL. The third is the single-byte `EE`; in the I/O bus cycle the processor puts DX on the address bus and AL on the data bus. Had you written `OUT 300, AL`, the assembler would reject it, because 300 does not fit in a byte.

The DX form is not only for large port numbers. It is also needed when the port number is computed while the program runs. For example, if the four registers of a board are at 310h, 311h, 312h, and 313h, the base address is loaded into DX and `INC DX` steps to the next register. With the constant-port form the port number is fixed at assembly time and cannot be changed.

All the virtual devices of emu8086 are below 255, so the programs that drive them use the constant-port form: `OUT 4, AX`, `IN AL, 7`, `OUT 199, AX`. Real boards such as the 8255 usually sit in the range 300h–31Fh on a PC and are driven through DX.

10.2.1.4 Port-mapped and memory-mapped I/O

There are two ways to connect a device register to the processor. In the first, devices sit in an address space separate from memory and are reached only with special I/O instructions. This is called **port-mapped I/O** (also *isolated I/O*). In the second, device registers are placed in part of the memory address space and are read and written with ordinary memory instructions. This is called **memory-mapped I/O**.

Definition 10.2.2

In **port-mapped I/O** the devices are in a separate 64 K I/O address space; access is only through `IN` and `OUT`, and no memory addresses are used up. In **memory-mapped I/O** the device registers are inside the 1 MB memory space; any instruction with a memory operand, such as `MOV`, `ADD`, `CMP`, or `AND`, can access the device.

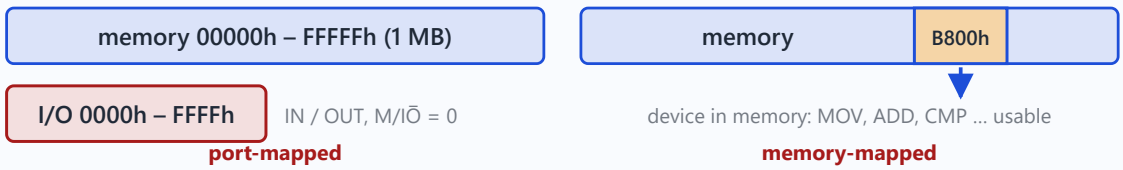


Figure 10.2.3: The two address spaces of the 8086. On the left, the 1 MB memory (00000h–FFFFFh); the text screen memory in segment B800h is a memory-mapped device. On the right, the 64 K I/O space (0000h–FFFFh); only IN and OUT reach it.

The processor uses both spaces with the same address and data buses. A control signal says which one is being accessed: on the 8086 the M/IO̅ pin is 1 in a memory cycle and 0 in an I/O cycle. `IN` and `OUT` are the only instructions that drive this pin to 0. The clock-cycle-level detail of this signal, of the read and write pins, and of the bus cycle belongs to the 8086 pins. For the programmer the consequence is simple: port 4 and memory address 0000:0004 are two completely independent places.

The most familiar example of memory-mapped I/O is the PC's text screen. In 80×25 text mode each cell of the screen holds two bytes in memory starting at B800:0000: the character code and the attribute (color) byte. The video card turns a byte written there into an image at once. The following program writes a yellow "A" on a blue background in the top-left corner of the screen, with a "B" next to it:

```
org 100h
mov ax, 0B800h
mov es, ax              ; ES = text screen segment
mov di, 0               ; row 0, column 0
mov al, 'A'
mov ah, 1Eh             ; blue background, yellow text
mov es:[di], ax         ; character + attribute
mov byte ptr es:[di+2], 'B'
ret
```

The program has not a single I/O instruction, yet the screen changes. Because a constant cannot be loaded directly into a segment register, B800h goes into AX first and from there into ES. `mov es:[di], ax` writes two bytes with one instruction: 41h in AL goes to B800:0000 and 1Eh in AH goes to B800:0001. The physical address is B800h × 16 + 0 = B8000h. The last instruction writes only the character byte; the attribute of the second cell keeps its old value (07h, light gray on black). Table 10.2.3 is the trace of the program.

Step	Instruction	AX	ES	DI	Written to memory	Explanation
1	`mov ax, 0B800h`	B800h	0700h	0000h		the segment value goes through a general register first
2	`mov es, ax`	B800h	B800h	0000h		ES = B800h: text screen memory
3	`mov di, 0`	B800h	B800h	0000h		first cell: row 0, column 0
4	`mov al, 'A'`	B841h	B800h	0000h		AL = 41h ('A')
5	`mov ah, 1Eh`	1E41h	B800h	0000h		AH = 1Eh: blue background, yellow text
6	`mov es:[di], ax`	1E41h	B800h	0000h	B800:0000 ← 41h, B800:0001 ← 1Eh	one MOV: character and attribute; a colored A appears on the screen
7	`mov byte ptr es:[di+2], 'B'`	1E41h	B800h	0000h	B800:0002 ← 42h	character byte of the second cell; the attribute (07h) is preserved
8	`ret`	1E41h	B800h	0000h		the program ends

Table 10.2.3: Memory-mapped I/O: writing to the text screen with MOV.

The pros and cons of the two methods mirror each other (Table 10.2.4). With a memory-mapped device every memory instruction and every addressing mode can be used. For example `INC BYTE PTR ES:[DI]` increments a device register with one instruction, and `TEST` tests a device's status bit directly. On the other hand the device uses up memory addresses, and no memory can be placed at those addresses. A port-mapped device does not touch the memory space but is reached with only two instructions, only through AL/AX. To operate on a value read from it, the value must first be brought into the accumulator.

Feature	Port-mapped	Memory-mapped
Address space	separate, 64 K (0000h–FFFFh)	shared with memory, part of the 1 MB
Instructions	IN, OUT only	MOV, ADD, CMP, TEST, AND ...
Data register	AL / AX only	any register or a constant
Addressing	constant (0–255) or DX	all memory addressing modes
Memory lost	none	the addresses the device occupies
8086 signal	M/IO̅ = 0	M/IO̅ = 1
Example	emu8086 devices, 8255, 8259A	B800h text screen, A000h graphics memory

Table 10.2.4: Comparison of port-mapped and memory-mapped I/O.

The 8086 family supports both methods, and the PC uses them together. Devices with few, small control registers (keyboard controller, timer, interrupt controller) are at ports. Devices that need a large data area (screen memory) are in memory. In processors with no separate I/O instructions (for example ARM) all devices are memory-mapped.

Common mistake

Confusing a port number with a memory address. `MOV AL, [4]` reads the memory byte at DS:0004, not port 4; nothing happens to the traffic lights. A port-mapped device is reached only by `IN` and `OUT`.

10.2.1.5 Polling, interrupts, and DMA

Devices are much slower than the processor. A stepper motor reaches its new position in milliseconds; in that time the 8086 executes thousands of instructions. While a printer prints a character or a robot takes a step, the processor has to wait before sending the next command. There are three ways to find out that a device is ready.

The simplest is **polling**. The processor reads the device's status register over and over in a loop. When the "ready" bit is 1 it leaves the loop and sends or receives the data. The stepper motor of emu8086 is a good example: bit 7 of the byte read from port 7 is 1 when the motor is ready.

```
wait: in   al, 7          ; read status
      test al, 10000000b  ; bit 7: ready?
      jz   wait           ; 0: read again
```

`TEST` computes the logical AND of AL with the mask but writes the result nowhere; it only sets the flags. If bit 7 is zero the result is zero and ZF = 1, so `JZ` jumps back to the top of the loop. If bit 7 is one, ZF = 0 and the loop ends. Three instructions build a wait loop that spins until the device is ready.

Definition 10.2.3

Polling is the processor waiting for a device to become ready by reading the device's status bit in a loop. Coordinating the data transfer between a device and the processor through ready and busy bits is called **handshaking**.

The advantage of polling is its simplicity: no extra hardware or interrupt routine is needed, and the flow of the program can be followed at a glance. Its drawback is that the processor can do nothing else while it waits. If the motor completes a step in 10 ms, a 4.77 MHz 8086 burns about 47,700 clock cycles in the empty loop. In an embedded system that drives a single device this is acceptable. It is not acceptable if a keyboard, a display, and a communication line must be watched at the same time.

The second way is the **interrupt**. When the device is ready it sends a request signal to the processor. The processor finishes the current instruction, pushes the flags and the return address onto the stack, runs the device's service routine, and returns to where it left off with `IRET`. On the PC the requests of the devices are collected in the 8259A interrupt controller and passed to the processor by priority. With interrupts the processor does not wait, but it still moves the data itself: the `IN` and `OUT` instructions in the service routine are executed by the processor.

The third way is **direct memory access** (DMA). If a block of thousands of bytes is to be moved from disk to memory, even one interrupt per byte is too expensive. In that case a DMA controller such as the 8237 asks the processor for the bus, and once it gets the bus it moves the data between device and memory by itself, then hands the bus back when done. During the transfer the processor merely cannot use the bus; it can carry on with work in its internal registers.

Method	Who waits?	Who moves the data?	Suited to
Polling	the processor, in a loop	the processor (IN/OUT)	a single device, simple embedded system
Interrupt	nobody; the device signals	the processor (in the service routine)	keyboard, serial line, timer
DMA	nobody; the controller takes the bus	DMA controller (8237)	block transfers such as disk and sound card

Table 10.2.5: Three ways of coordinating with a device.

None of the virtual devices of emu8086 generates interrupts. All the programs in this note therefore use polling. The common pattern of a polling loop repeats for every device: read the status, test the relevant bit, write or read the data if ready, otherwise read again.

Common mistake

Using `AND` instead of `TEST` in a polling loop and then trying to use the data in AL. `AND AL, 80h` tests the status but clears the other bits of AL. If the status byte carries other information from the device (an error bit, a data bit), it is lost. `TEST` performs the same test without

changing AL.

10.2.2 The 8255 programmable peripheral interface

10.2.2.1 Structure and address selection

The devices of emu8086 interpret the value written to a port themselves. On a real board, an interface chip stands between the processor and the lamp, switch, or motor driver. The most common parallel interface chip in 8086 systems is the Intel 8255. Its name is **programmable peripheral interface** (PPI). "Programmable" refers to the fact that software chooses whether its ports are inputs or outputs.

The 8255 has three 8-bit ports: A, B, and C. Port C can be split into two 4-bit halves: the upper half PC7–PC4 and the lower half PC3–PC0. Besides these there is a control register. The A1 and A0 pins of the chip select which of the four registers is accessed. Whether the chip is active at all is decided by the $\overline{CS}$ (chip select) pin, which comes from the address decoder. Reading and writing are done with the processor's I/O read and I/O write signals ($IO\overline{R}$, $IO\overline{W}$). Figure 10.2.4 shows the connection.

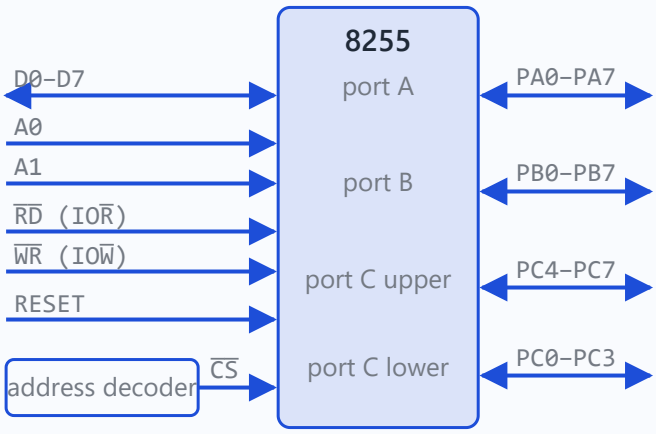


Figure 10.2.4: The processor side (D0–D7, A0, A1, $\overline{RD}$, $\overline{WR}$, RESET, $\overline{CS}$) and the device side (PA0–PA7, PB0–PB7, PC7–PC4, PC3–PC0) of the 8255 PPI.

If the base address of the board is 310h, the addresses of the four registers are as in Table 10.2.6. The address decoder recognizes the range 310h–313h and drives $\overline{CS}$ to 0; the lowest two address bits go to the A1 and A0 pins of the chip.

A1 A0	Register	Address
0 0	port A	310h
0 1	port B	311h

A1 A0	Register	Address
1 0	port C	312h
1 1	control register	313h

Table 10.2.6: Registers of the 8255 (base address 310h, A1 ← A1, A0 ← A0 wiring).

This consecutive address layout is correct for the 8088, which has an 8-bit data bus. The data bus of the 8086 is 16 bits wide and is divided into two banks: even-addressed bytes are transferred on D0–D7 (the low bank) and odd-addressed bytes on D8–D15 (the high bank). The 8255 has only eight data pins. If the chip is wired to D0–D7, only its registers at even addresses can be reached. The solution is to connect the chip's A0 pin to the processor's A1 and its A1 pin to A2. The four registers then fall at 310h, 312h, 314h, and 316h, always at even addresses. The programs in this note use the consecutive layout 310h–313h; on a board built for the 8086 the addresses are changed accordingly.

All the addresses are above 255. Every access to the 8255 is therefore made through DX: `MOV DX, 313h` · `OUT DX, AL`. After RESET all ports of the 8255 are inputs. There is a safety reason for this: at power-up a chip that has not been programmed yet must not drive the connected circuit at random. The program therefore writes a control word to the control register before using the chip.

10.2.2.2 Control word: mode selection

The highest bit (D7) of the byte written to the control register determines the kind of byte. If D7 = 1 the byte is a **mode set word**: it sets the mode and direction of the ports in one go. If D7 = 0 the byte is a **bit set/reset** word and changes only a single bit of port C.

D7	D6 D5	D4	D3	D2	D1	D0
1 = mode set	group A mode (00 = mode 0)	port A	PC7–PC4	group B mode (0 = mode 0)	port B	PC3–PC0

Table 10.2.7: The 8255 mode set word (D7 = 1). In the direction bits, 1 = input, 0 = output.

The ports of the 8255 are divided into two groups. Group A is port A plus the upper half of C; group B is port B plus the lower half of C. Mode 0 is simple I/O without handshaking: a value written to a port stays on the output pins, and a value read from an input is the current state of the pins. Modes 1 and 2 are for handshaking transfers. In these modes some pins of port C are used as "data ready" (*strobe*) and acknowledge signals; mode 2 makes port A bidirectional. For lamps, switches, and motor drivers mode 0 is enough.

Example 10.2.1

Required: port A output (drives LEDs or relays), port B input (switches are read), both halves of C output, all in mode 0.

1. D7 = 1: this is a mode set word.

2. D6 D5 = 00 and D2 = 0: both groups in mode 0.
3. D4 = 0 (A output), D3 = 0 (C upper output), D1 = 1 (B input), D0 = 0 (C lower output).
4. Writing the bits side by side: 1 00 0 0 0 1 0b = 1000 0010b = **82h**.

In the same way, A input, B output, and all of C input gives 1 00 1 1 0 0 1b = 99h; A output, B output, and all of C input gives 1 00 0 1 0 0 1b = 89h. If all ports are outputs the word is 1000 0000b = 80h.

The point that is most often confused when building the word is the meaning of the direction bits. In the 8255, 1 means "input" and 0 means "output". This is the opposite of the "1 = on" intuition and has to be memorized. The interactive control-word builder in the slides shows how the bits change as you click each port's direction; you can also type a value directly into the word box and read off the port directions.

10.2.2.3 Bit set/reset (BSR)

Port C is often used as individual control pins: one pin drives a relay coil, another an LED, a third a digit-select transistor. Writing the whole port just to change one pin would mean keeping the values of the other pins separately in the program. The 8255 offers the **bit set/reset** (BSR) word instead.

Definition 10.2.4

The **bit set/reset word** is a control byte with D7 = 0. D3–D1 give the number (0–7) of the C bit to change and D0 gives the new value of that bit; D6–D4 are unused (write 0). The word is written to the control register, not to port C. The other C bits and the port directions do not change.

To set PC7 to 1, the bit number is 7 = 111b and the value is 1: 0000 1111b = 0Fh. To clear PC7 to 0 only D0 changes: 0000 1110b = 0Eh. In the same way the word that sets PC3 to 1 is 0000 0111b = 07h, and the word that clears PC0 to 0 is 00h. The last one is interesting: although 00h looks like a mode word, D7 = 0 makes it mean "PC0 = 0".

Common mistake

Writing a BSR word to port C (312h). The 0Fh written to port C sets PC3–PC0 to 1 and PC7–PC4 to 0: all eight bits are rewritten. A bit set/reset word is interpreted as a single-bit word only when it is written to the control register (313h).

10.2.2.4 Code trace: setting up the 8255

The following program does four basic jobs in a row: it programs the chip with 82h, reads the switches on port B, writes the value read to the LEDs on port A, and sets PC7 to 1 with BSR.

```
org 100h
mov dx, 313h   ; control register
mov al, 82h    ; A,C out; B in
out dx, al
mov dx, 311h   ; port B
in  al, dx     ; read switches
mov dx, 310h   ; port A
out dx, al     ; drive the LEDs
mov dx, 313h
mov al, 0Fh    ; BSR: PC7 = 1
out dx, al
ret
```

Every I/O access of the program consists of the same two steps: first the address into DX, then `IN` or `OUT`. The switches wired to port B are assumed to be in the position 1001 0110b = 96h. Table 10.2.8 gives the trace, with each I/O operation in the last column.

Step	Instruction	DX	AL	I/O	Explanation
1	`mov dx, 313h`	0313h	00h		DX = 313h: the address of the control register; kept in DX because it is above 255
2	`mov al, 82h`	0313h	82h		AL = 82h = 1000 0010b: mode 0; A output, C upper output, B input, C lower output
3	`out dx, al`	0313h	82h	313h ← 82h	the control word has been written to the chip; the port directions now follow 82h
4	`mov dx, 311h`	0311h	82h		DX = 311h: port B
5	`in al, dx`	0311h	96h	311h → 96h	switches read: AL = 96h = 1001 0110b
6	`mov dx, 310h`	0310h	96h		DX = 310h: port A
7	`out dx, al`	0310h	96h	310h ← 96h	the LEDs copy the switches: bits 7, 4, 2, and 1 light up
8	`mov dx, 313h`	0313h	96h		DX points to the control register again
9	`mov al, 0Fh`	0313h	0Fh		AL = 0Fh = 0000 1111b: D7 = 0, so this is a bit set/reset word
10	`out dx, al`	0313h	0Fh	313h ← 0Fh	D3–D1 = 111 (PC7), D0 = 1: only PC7 becomes 1; the directions do not change
11	`ret`	0313h	0Fh		0000h popped from the stack → IP; returns to the INT 20h in the PSP

Step	Instruction	DX	AL	I/O	Explanation
12	`INT 20h` (`PSP:0000`)	0313h	0Fh		the program ends

Table 10.2.8: Trace of the 8255 setup program (switches on port B at 96h; each row shows the state after the instruction executes).

Three details stand out in the trace. First, the `OUT DX, AL` instructions change no register; their effect is only inside the chip. Second, `IN AL, DX` wipes out the old value of AL (82h) completely. Third, although the last `OUT` writes to the control register, it does not change the port directions: because D7 = 0 the chip interprets it as a bit set/reset word. After the setup, bits 7, 4, 2, and 1 of the port A LEDs are lit, and in port C only PC7 = 1.

The machine code of the program is 21 bytes. The four `MOV DX` instructions take three bytes each, the two `MOV AL` instructions two bytes each, and the four I/O instructions and `RET` one byte each. This brevity is another benefit of the DX form: the port number is given in a single `MOV`, and the I/O instruction itself shrinks to one byte.

10.2.3 Traffic lights

10.2.3.1 Port 4 and the lamp bits

The traffic-light device of emu8086 is a four-way intersection. There is a pole at each corner and three lamps on each pole: red, yellow, and green. Each of the twelve lamps is wired to one bit of the 16-bit word written to port 4. A bit of 1 lights the lamp and 0 turns it off. Bits 12–15 are unused. Writing one word to port 4 therefore draws a complete snapshot of the intersection in one go.

The bits are distributed over the lamps in a regular way. If the poles are numbered k = 0, 1, 2, 3, then pole k has its red lamp at bit 3k, yellow at bit 3k + 1, and green at bit 3k + 2. The lower-right pole uses bits 0–2, the upper-right 3–5, the upper-left 6–8, and the lower-left 9–11 (hex 9, A, B).

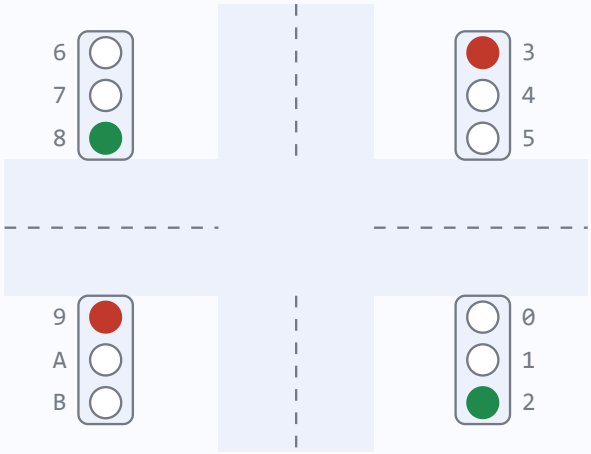


Figure 10.2.5: Port 4 = 030Ch = 0000 0011 0000 1100b. The bits that are on are 2 and 8 (green), 3 and 9 (red): lower right and upper left are green, upper right and lower left are red.

To decode the value in Figure 10.2.5, split the word into groups of four: 0000 0011 0000 1100b. The bits that are 1, counted from the right, are bits 2, 3, 8, and 9. Bit 2 = 3 · 0 + 2, so it is the green of the lower-right pole. Bit 3 = 3 · 1 + 0 is the red of the upper-right pole. Bit 8 = 3 · 2 + 2 is the green of the upper-left pole, and bit 9 = 3 · 3 + 0 is the red of the lower-left pole. The reverse is just as easy: find the bit numbers of the lamps you want and add these powers of 2.

Example 10.2.2

What word lights only the green lamps in all directions?

1. The green lamps are at bits 3k + 2: for k = 0, 1, 2, 3 that is bits 2, 5, 8, 11.
2. The word is $2^2 + 2^5 + 2^8 + 2^{11} = 4 + 32 + 256 + 2048 = 2340$.
3. In binary it is 0000 1001 0010 0100b, in hexadecimal **0924h**.

Such a state is of course dangerous at a real intersection; the example only shows the bit arithmetic. The word that lights all the reds comes out the same way from bits 0, 3, 6, 9: 0000 0010 0100 1001b = 0249h. In the program this value is named `all_red`.

10.2.3.2 State table

The operation of the intersection is a repetition of a few states in a fixed order. Each state is defined by a word to be written to port 4. The program keeps these words in memory as a consecutive table:

```
;                FEDC_BA98_7654_3210
situation  dw     0000_0011_0000_1100b
s1         dw     0000_0110_1001_1010b
s2         dw     0000_1000_0110_0001b
s3         dw     0000_1000_0110_0001b
s4         dw     0000_0100_1101_0011b
sit_end = $
all_red    equ    0000_0010_0100_1001b
```

The underscores inside the numbers are only for readability; emu8086 ignores them. The sequence FEDC_BA98_7654_3210 in the comment line shows which bit each column is. The line `sit_end = $` takes up no memory: `$` is the assembler's current location counter and gives the address where the table ends. The program uses this address to answer the question "have I reached the end of the table?". `all_red` is a constant defined with EQU; it lives inside the instruction, not in memory.

Ad	Hex	Lower right	Upper right	Upper left	Lower left
situation	030Ch	green	red	green	red
s1	069Ah	yellow	red + yellow	yellow	red + yellow
s2	0861h	red	green	red	green
s3	0861h	red	green	red	green
s4	04D3h	red + yellow	yellow	red + yellow	yellow
all_red	0249h	red	red	red	red

Table 10.2.9: Traffic states and the lamps that are lit.

Every row of the table can be verified by bit arithmetic. In s1 = 069Ah = 0000 0110 1001 1010b the bits that are 1 are 1, 3, 4, 7, 9, and 10. Bit 1 is the yellow of the lower-right pole, bits 3 and 4 are the red and yellow of the upper-right pole, bit 7 is the yellow of the upper-left pole, and bits 9 and 10 are the red and yellow of the lower-left pole. In s1 the directions that are about to go from green to red show yellow, and the directions that are about to go from red to green show the "red + yellow" warning. Red + yellow is the transition signal that means "get ready". s4 makes the same transition in the opposite direction.

s2 and s3 are the same word. Because the program waits the same time (5 seconds) in every state, the green of the side road lasts two intervals, that is 10 seconds. To change the duration there is no need to touch the program; writing the same word once more in the table is enough. That the behavior lives in the table and not in the code is the greatest strength of this design.

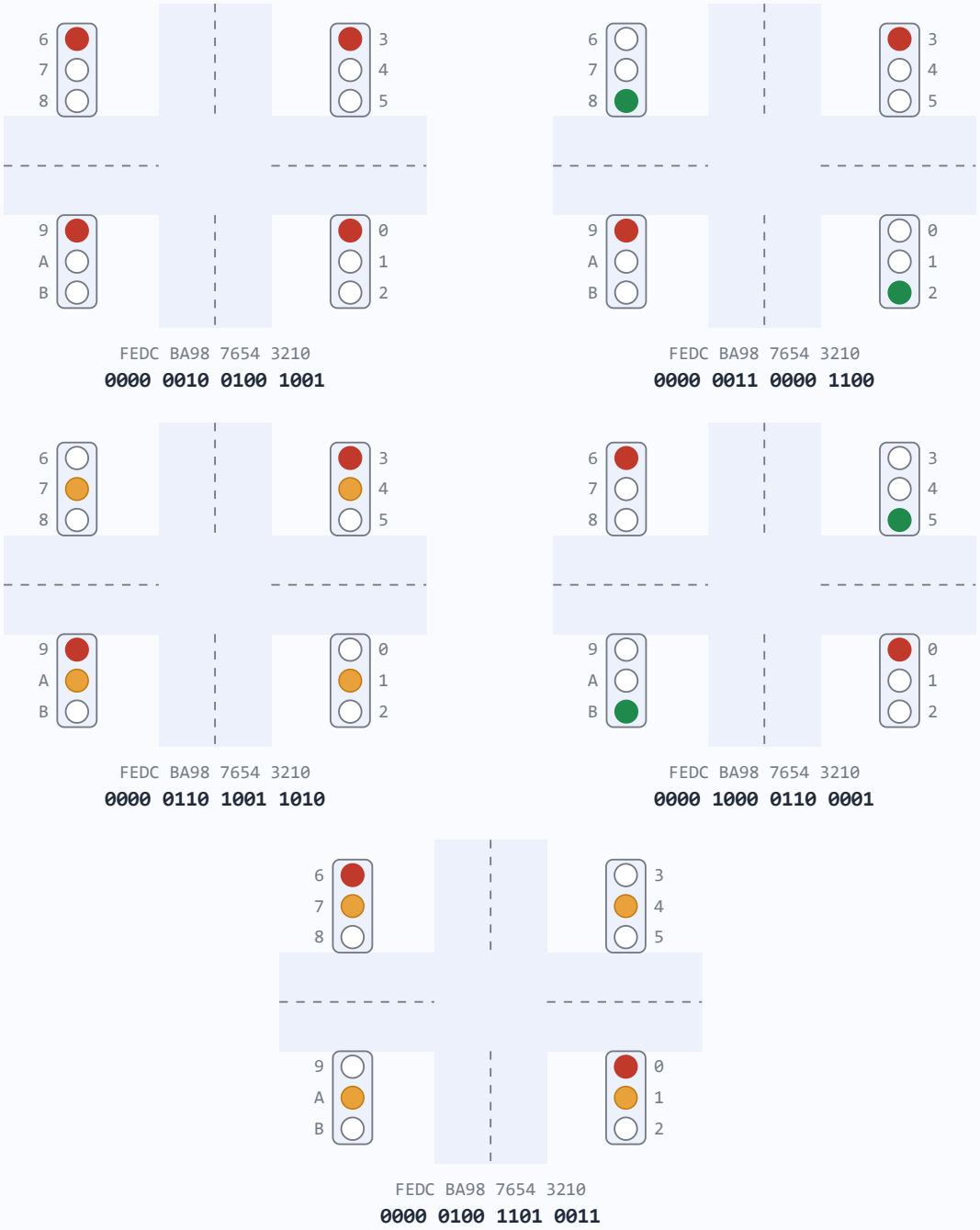


Figure 10.2.6: The states the program writes to the intersection in turn: all_red (0249h), situation (030Ch), s1 (069Ah), s2 = s3 (0861h), and s4 (04D3h). Below each is the binary form of the word written to port 4.

The interactive traffic bit editor in the slides does the same calculation in the other direction: clicking a bit turns that lamp on or off and updates the hexadecimal value of the word. Clicking a name in the table loads the word of that state.

10.2.3.3 Waiting with INT 15h

Every state must last a few seconds, so the program has to measure this time. The PC BIOS offers a ready-made service for it: **INT 15h, AH = 86h**. The service waits as many microseconds as the 32-bit number in the CX:DX pair. CX carries the high word and DX the low word.

Example 10.2.3

What must CX and DX be for a 5-second wait?

1. Convert the time to microseconds: 5 s = 5 × 1,000,000 = 5,000,000 µs.
2. Convert the decimal number to hexadecimal by repeated division by 16: 5,000,000 ÷ 16 = 312,500 remainder 0; 312,500 ÷ 16 = 19,531 remainder 4; 19,531 ÷ 16 = 1,220 remainder 11 (B); 1,220 ÷ 16 = 76 remainder 4; 76 ÷ 16 = 4 remainder 12 (C); 4 ÷ 16 = 0 remainder 4.
3. The remainders are read from last to first: 4C4B40h. Written in 32 bits it is 004C4B40h.
4. Split the number in two: CX = 004Ch, DX = 4B40h.

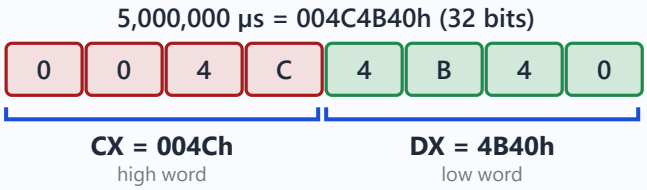


Figure 10.2.7: 5,000,000 µs = 004C4B40h. The high word goes into CX (004Ch) and the low word into DX (4B40h).

```
; wait 5 seconds (5 million µs)
mov cx, 4Ch     ; 004C4B40h = 5 000 000
mov dx, 4B40h
mov ah, 86h
int 15h
```

The same calculation works for other durations. 1 second is 1,000,000 = 000F4240h: CX = 000Fh, DX = 4240h. 0.2 second is 200,000 = 00030D40h: CX = 0003h, DX = 0D40h. In the second example the hexadecimal value of DX starts with a letter; in the source code it is written `0D40h`, because without the leading 0 the assembler would take it for a label name. The longest time the service can wait in one call is CX = DX = FFFFh, that is 4,294,967,295 µs: about 71.6 minutes.

After a successful wait CF = 0 is returned. In emu8086 the service leaves AX alone, whereas some real BIOSes write a status code into AH. The program therefore does not rely on the value in AX after the call. On a real PC this service is timed by the interrupt of the timer chip, and its duration does not

depend on the speed of the processor. In emu8086 too, timed waiting is done with this service; the emulator applies the wait according to its own clock.

> **Common mistake**
>
> Splitting the time between CX and DX in decimal: writing CX = 5, DX = 0 for 5,000,000. CX:DX is a single 32-bit binary number; the value CX = 5, DX = 0 is 5 × 65,536 = 327,680 µs, about 0.33 second. The number is first converted to hexadecimal as a whole and then split into two 16-bit halves.

10.2.3.4 Software delay: LOOP and clock cycles

In an embedded system that has no BIOS service, waiting is done by having the processor burn a known number of clock cycles. The simplest way is an empty `LOOP` loop. The duration of the loop is computed from the clock-cycle counts of the instructions. According to the 8086 manual the required timings are in Table 10.2.10.

Instruction	Clock cycles
`MOV CX, constant`	4
`LOOP` (CX ≠ 0, jumps)	17
`LOOP` (CX = 0, falls through)	5
`DEC r16`	2
`JNZ` (taken / not taken)	16 / 4
`CALL` near / `RET` near	19 / 8

Table 10.2.10: 8086 timings of the instructions in a delay loop.

```
; ≈ 1 ms delay (4.77 MHz 8086)
mov  cx, 281    ; 4 clocks
delay:
loop delay      ; 280 × 17 + 1 × 5 clocks
```

In a loop that starts with CX = N, `LOOP` executes N times. In the first N − 1 times CX has not reached zero and the instruction jumps (17 cycles). The last time CX becomes 0 and the instruction falls through (5 cycles). The total time is:

> T = 4 + 17(N − 1) + 5 = 17N − 8 clock cycles.

Converting clock cycles to time needs the frequency. The 8088 of the original IBM PC runs at 4.77 MHz: one clock cycle is 1 / 4.77 MHz ≈ 0.2096 µs. 1 ms is therefore 4,770 clock cycles. The equation 17N − 8 = 4,770 gives N ≈ 281.06. Since only an integer can be written into CX, the nearest value is

chosen: N = 281, 17 · 281 − 8 = 4,769 cycles, that is ≈ 0.9998 ms.

The same loop gives a different time on another processor. At 5 MHz, 1 ms is 5,000 cycles; N ≈ 294.6, and choosing N = 295 gives 5,007 cycles ≈ 1.0014 ms. The N = 281 found for 4.77 MHz takes only 4,769 / 5 ≈ 954 µs at 5 MHz. This is the greatest flaw of software delays: the time depends on the clock speed. When the program is moved to a faster processor, all the waits get shorter.

A single loop also has an upper limit. A `LOOP` that starts with CX = 0 first brings CX down to FFFFh and runs 65,536 times. That is 17 · 65,536 − 8 = 1,114,104 cycles, about 233.6 ms at 4.77 MHz. A longer wait needs nested loops:

```
       mov  dx, 100     ; 4 clocks
outer: mov  cx, 281     ; 4 clocks
inner: loop inner       ; 17 · 280 + 5 = 4,765 clocks
       dec  dx          ; 2 clocks
       jnz  outer       ; 16 clocks, 4 on the last pass
```

One pass of the outer loop is 4 + 4,765 + 2 = 4,771 cycles, followed by a `JNZ`. In 99 of the hundred passes `JNZ` jumps (16), and in the last one it does not (4). The total is 4 + 100 · 4,771 + 99 · 16 + 4 = 478,692 cycles, about 100.4 ms at 4.77 MHz. The inner counter is in CX and the outer counter in DX; since `LOOP` uses only CX, the two counters do not interfere. If the delay is written as a procedure and called, each call also adds the `CALL` (19) and `RET` (8) cycles.

These calculations are approximate. Interrupts can slip in between the loop's instructions and lengthen the time. The wait states added by slow memory and the fullness of the instruction queue also change the cycle count. emu8086 does not model instruction timings: the loop runs at the speed of the computer and according to the emulator's step delay setting. That is why timed waiting in emu8086 is done with INT 15h/86h; the software delay calculation is for real hardware.

10.2.3.5 Code trace: the delay loop

To see where the formula comes from, trace a short loop with CX = 4:

```
org 100h
mov cx, 4       ; 4 clocks
delay:
loop delay      ; 17 clocks, last one 5
ret
```

Step	Instruction	CX	IP	Explanation
1	`mov cx, 4`	0004h	0103h	4 cycles; total 4
2	`loop delay`	0003h	0103h	CX = 3 ≠ 0: jumps to the same instruction, 17 cycles; total 21
3	`loop delay`	0002h	0103h	CX = 2: 17 cycles; total 38
4	`loop delay`	0001h	0103h	CX = 1: 17 cycles; total 55

Step	Instruction	CX	IP	Explanation
5	`loop delay`	0000h	0105h	CX = 0: falls through, 5 cycles; total 60 = 17 · 4 − 8
6	`ret`	0000h	0000h	left the loop; 60 cycles ≈ 12.6 µs at 4.77 MHz

Table 10.2.11: Trace of the delay loop; the cycle totals in the explanation are computed from the 8086 instruction timings.

The `LOOP delay` instruction jumps to its own address (0103h); its machine code is `E2 FE`, and FEh = −2 means going back by the length of the instruction itself. In the first three executions IP is again 0103h. On the fourth execution CX drops to 0, there is no jump, and IP moves on to 0105h, the `RET`. The total is 4 + 17 + 17 + 17 + 5 = 60 = 17 · 4 − 8 cycles. 60 cycles take 60 × 0.2096 ≈ 12.6 µs at 4.77 MHz.

10.2.3.6 The traffic program

The program consists of four parts: the setup that turns all directions red, the loop body that writes the table word to the port, the 5-second wait, and the return to the start at the end of the table.

```
#start=Traffic_Lights.exe#
mov ax, all_red
out 4, ax
mov si, offset situation
next:
mov ax, [si]
out 4, ax
mov cx, 4Ch      ; wait 5 s
mov dx, 4B40h
mov ah, 86h
int 15h
add si, 2        ; next state
cmp si, sit_end
jb  next
mov si, offset situation
jmp next
; the state table follows (situation … all_red)
```

The first thing the program does is turn all directions red. This is a safety measure: when the controller powers up, it is not known what state the intersection was left in, and green must never be shown to one direction while another has green. `MOV AX, all_red` loads the constant directly into AX (`B8 49 02`); `OUT 4, AX` writes the two bytes of the word to ports 4 and 5.

SI holds the address of the next word in the table. `MOV AX, [SI]` reads the word at that address, and `OUT 4, AX` switches the intersection to that state. Then the program waits 5 seconds. `ADD SI, 2` moves to the next word; since the table consists of words, the step is 2 bytes. `CMP SI, sit_end` compares SI with the end address of the table. If SI is still smaller, CF = 1 and `JB next` goes back to

the loop. When they are equal, CF = 0; SI is set back to the start of the table and `JMP next` restarts the loop. The program never ends: in embedded systems the main program is usually an endless loop like this.

The location of the table in memory depends on the length of the code. If the program is assembled with ORG 100h, the code takes 36 bytes and the table starts at 0124h: situation is at 0124h, s1 at 0126h, s2 at 0128h, s3 at 012Ah, and s4 at 012Ch; sit_end = 012Eh. If the table's place in the source changes, these addresses change too, but the program keeps working correctly because it uses only labels. When emu8086 loads a .COM program it loads the program size into CX; the trace takes CX before the first `MOV CX` as 0000h. Since the program does not rely on this initial value, the result is unchanged.

Step	Instruction	AX	SI	CX	DX	CF	Port 4	Explanation
1	`mov ax, all_red`	0249h	0000h	0000h	0000h	0	—	AX = all_red = 0249h; an EQU constant, takes up no memory
2	`out 4, ax`	0249h	0000h	0000h	0000h	0	0249h	all directions red
3	`mov si, offset situation`	0249h	0124h	0000h	0000h	0	0249h	SI = 0124h: the first word of the table (situation)
4	`mov ax, [si]`	030Ch	0124h	0000h	0000h	0	0249h	AX ← [0124h] = 030Ch
5	`out 4, ax`	030Ch	0124h	0000h	0000h	0	030Ch	the lamps switch to the situation state
6	`mov cx, 4Ch`	030Ch	0124h	004Ch	0000h	0	030Ch	CX = 004Ch: the high word of 5,000,000 µs
7	`mov dx, 4B40h`	030Ch	0124h	004Ch	4B40h	0	030Ch	DX = 4B40h: the low word
8	`mov ah, 86h`	860Ch	0124h	004Ch	4B40h	0	030Ch	AH = 86h: the BIOS wait service; AL is unchanged
9	`int 15h`	860Ch	0124h	004Ch	4B40h	0	030Ch	waits 5 seconds; CF = 0 on success
10	`add si, 2`	860Ch	0126h	004Ch	4B40h	0	030Ch	SI = 0126h: next state
11	`cmp si, sit_end`	860Ch	0126h	004Ch	4B40h	1	030Ch	0126h − 012Eh: borrow, CF = 1
12	`jb next`	860Ch	0126h	004Ch	4B40h	1	030Ch	JB jumps: next
13	`mov ax, [si]`	069Ah	0126h	004Ch	4B40h	1	030Ch	AX ← [0126h] = 069Ah (s1)
14	`out 4, ax`	069Ah	0126h	004Ch	4B40h	1	069Ah	yellow transition state
…	wait, SI = 0128h, CMP and JB							
23	`out 4, ax`	0861h	0128h	004Ch	4B40h	1	0861h	s2 = 0861h; side road green
…	wait, SI = 012Ah							
32	`out 4, ax`	0861h	012Ah	004Ch	4B40h	1	0861h	s3 = 0861h; the same state lasts another 5 seconds
…	wait, SI = 012Ch							

Step	Instruction	AX	SI	CX	DX	CF	Port 4	Explanation
41	`out 4, ax`	04D3h	012Ch	004Ch	4B40h	1	04D3h	s4 = 04D3h; getting ready to return to the main road
…	wait							
46	`add si, 2`	86D3h	012Eh	004Ch	4B40h	0	04D3h	SI = 012Eh = sit_end
47	`cmp si, sit_end`	86D3h	012Eh	004Ch	4B40h	0	04D3h	equal: CF = 0, ZF = 1
48	`jb next`	86D3h	012Eh	004Ch	4B40h	0	04D3h	JB falls through: the table is finished
49	`mov si, offset situation`	86D3h	0124h	004Ch	4B40h	0	04D3h	SI goes back to the start of the table
50	`jmp next`	86D3h	0124h	004Ch	4B40h	0	04D3h	endless loop
51	`mov ax, [si]`	030Ch	0124h	004Ch	4B40h	0	04D3h	AX ← [0124h] = 030Ch
52	`out 4, ax`	030Ch	0124h	004Ch	4B40h	0	030Ch	situation again: the loop starts over

Table 10.2.12: Trace of the traffic-light program (table at 0124h; except for the INT 15h wait, each row shows the state after the instruction executes).

The trace also shows that the `INT 15h` call leaves a mark in AX. `MOV AH, 86h` changes only the upper byte; AL keeps the low byte of the last state read. That is why AX = 860Ch after step 8. Because `MOV AX, [SI]` at the top of the loop rewrites the whole of AX, this is no longer a problem.

> **Common mistake**
>
> Writing `INC SI` instead of `ADD SI, 2`. The table consists of words; if SI is incremented by one, the second read starts in the middle of a word. The word read from 0125h is 9A03h, the combination of the high byte of situation (03h) and the low byte of s1 (9Ah). This value lights a meaningless lamp pattern that also spills into bits 12–15.

10.2.4 Stepper motor

10.2.4.1 The emu8086 stepper motor

An ordinary DC motor turns for as long as voltage is applied; to know how far it has turned you need a separate sensor. A **stepper motor**, on the other hand, turns through a fixed angle with each new excitation pattern applied to its windings, and stops there. The shaft takes the position given by which windings carry current. If the excitation pattern is changed in sequence, the motor turns step by step. Its position is therefore known from the number of steps sent. Printer heads, floppy-disk drives, 3D printers, and CNC machines are driven by stepper motors.

The stepper motor of emu8086 is attached to port 7 and is a simplified model. Bits 0, 1, and 2 of the byte written to the port switch three electromagnets on or off. The motor can be turned both clockwise (CW) and counterclockwise (CCW). A half step is 11.25° and a full step is 22.5°; a full revolution is therefore 32 half steps or 16 full steps.

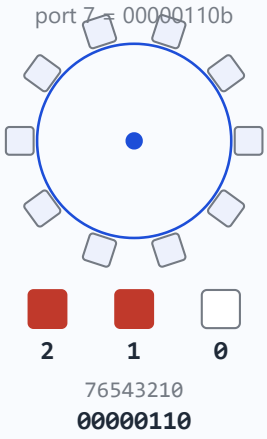


Figure 10.2.8: The emu8086 stepper motor. Port 7 = 0000 0110b: the magnets wired to bit 2 and bit 1 are on, and the rotor is pulled between the two.

The motor also has a status output. After an `OUT` instruction the motor is busy until it reaches its new position. When it is ready to accept a new command it makes bit 7 of the byte read from port 7 equal to 1. The program polls this bit before sending each byte. A byte sent to a motor that is not ready is lost, or the motor skips a step.

10.2.4.2 Data tables and magnet patterns

Turning the motor means switching the magnets on and off in a particular order. In the program this order is kept as four-byte tables. The sample program of emu8086 defines four tables (Table 10.2.13).

Table	Address	Meaning	1	2	3	4
`datcw`	0102h	CW half step	110	100	011	010
`datccw`	0106h	CCW half step	011	001	110	010
`datcw_fs`	010Ah	CW full step	001	011	110	000
`datccw_fs`	010Eh	CCW full step	100	110	011	000

Table 10.2.13: Stepper motor data tables (bits 2–0 = magnets 2–0; addresses for ORG 100h).

Each bit of the byte stands for one magnet: 110b means magnets 2 and 1 are on and 0 is off; 000b turns all the magnets off. The half-step table turns on two magnets and one magnet alternately: 110 (two), 100 (one), 011 (two), 010 (one). With two magnets on the rotor rests between them, and with one magnet on it rests facing that magnet. These intermediate positions halve the step. The full-step table sends the bytes single, double, double magnet, and "all off".

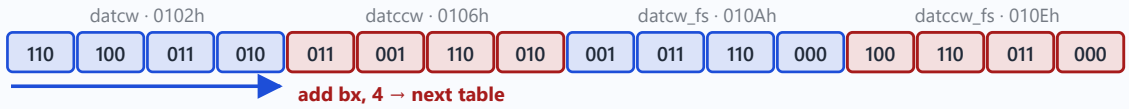


Figure 10.2.9: The four tables lie back to back in memory and each is 4 bytes. Adding 4 to BX moves to the next table.

That the tables are consecutive in memory is the most useful detail of the program. datcw starts at 0102h; datccw is 4 bytes later, at 0106h; datcw_fs is at 010Ah; datccw_fs is at 010Eh. If BX points to the start of one table, `ADD BX, 4` moves it to the next. After 32 (20h) rounds the program changes the table. The direction and step type thus cycle in the order CW half → CCW half → CW full → CCW full → back to the start.

The notation in the source code can vary as well. In the original emu8086 sample each byte is on a separate line and written with eight digits: `datcw db 0000_0110b`, on the next line `db 0000_0100b` … In this note's program the table is squeezed onto one line with the same values: `datcw db 110b,100b,011b,010b`. Both notations produce the same four bytes.

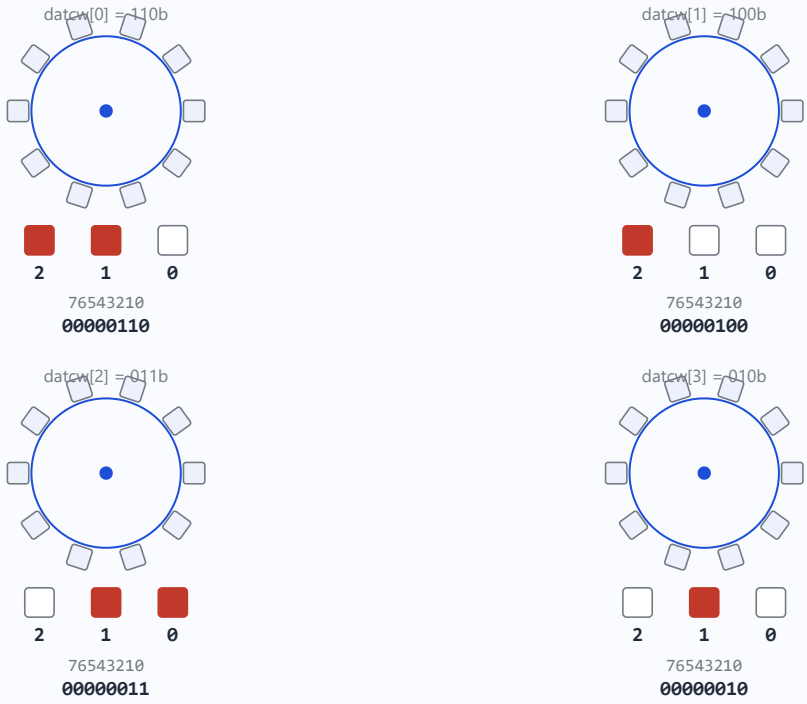


Figure 10.2.10: The four bytes of the datcw table in turn: 110b (magnets 2 and 1), 100b (2), 011b (1 and 0), 010b (1). Each byte turns the rotor by one half step, 11.25°.

10.2.4.3 The control program

```
ORG 100h
steps_before_direction_change = 20h
jmp start
datcw     db 110b,100b,011b,010b
datccw    db 011b,001b,110b,010b
datcw_fs  db 001b,011b,110b,000b
datccw_fs db 100b,110b,011b,000b
start:
mov bx, offset datcw
mov si, 0
mov cx, 0              ; round counter
next_step:
wait: in   al, 7
      test al, 10000000b
      jz   wait
mov al, [bx][si]
out 7, al
inc si
cmp si, 4
jb  next_step
mov si, 0
inc cx
cmp cx, steps_before_direction_change
jb  next_step
mov cx, 0
add bx, 4              ; next table
cmp bx, offset datccw_fs
jbe next_step
mov bx, offset datcw
jmp next_step
```

`steps_before_direction_change = 20h` is a constant. The `=` directive works like EQU: it takes up no memory and is replaced by its value wherever it is used. Because the tables sit at the start of the code, the first instruction `JMP start` jumps over them; otherwise the processor would take the table bytes for instructions and execute them. The code of the short jump is `EB 10`: 16 bytes forward from 0102h, to 0112h.

The program has three counters. BX holds the start of the table in use, SI the position within the table (0–3), and CX the number of completed rounds. The first three lines of the loop body are the polling loop: while bit 7 = 0 the motor is busy and the port is read again. When bit 7 = 1, `MOV AL, [BX][SI]` reads the next byte.

The notation `[BX][SI]` is the same as `[BX + SI]`: base plus index addressing. BX gives the start address of the table and SI the position in that table; the effective address is the sum of the two. For BX = 0102h and SI = 2 the byte read is [0104h] = 011b. This addressing mode can be thought of as the row and column of a two-dimensional array: BX says which table, SI which element of the table.

After the byte is written to the port, SI is incremented. If SI is less than 4, the next byte of the same table is used. When SI = 4 the table is finished: SI is reset and CX is incremented. If CX has not yet reached 20h, the same table starts again. If it has, CX is reset and BX is increased by 4. The last lines keep BX from running outside the tables: `CMP BX, offset datccw_fs` · `JBE next_step` continues if BX is equal to or below the address of the last table; otherwise BX returns to the first table.

The behavior of the program shows in a long run. When port 7 always returns "ready", the first 128 bytes written to the port are the four bytes of datcw repeated 32 times. From the 129th byte on comes datccw (011, 001, 110, 010), from the 257th byte datcw_fs, and from the 385th byte datccw_fs; at the 513th byte datcw starts again.

> **Common mistake**
>
> Writing `JB` instead of `JBE` after `CMP BX, offset datccw_fs`. When BX points to the last table (010Eh), `JB` does not jump and BX returns to datcw at once; the CCW full-step table is never used. Equality is included so that the last table is counted too.

10.2.4.4 Code trace: the first round

The trace assumes that port 7 returns 00h (busy) on the first read and 80h (ready) on all later reads. Table 10.2.14 shows the first four bytes and the end of the round; the repeated polling steps are abbreviated with "…".

Step	Instruction	AL	BX	SI	CX	ZF	CF	Port 7	Explanation
1	`jmp start`	00h	0000h	0000h	0000h	0	0	—	jumps over the data tables
2	`mov bx, offset datcw`	00h	0102h	0000h	0000h	0	0	—	BX = 0102h: datcw (clockwise half step)
3	`mov si, 0`	00h	0102h	0000h	0000h	0	0	—	SI = 0: position in the table
4	`mov cx, 0`	00h	0102h	0000h	0000h	0	0	—	CX = 0: round counter
5	`wait: in al, 7`	00h	0102h	0000h	0000h	0	0	—	AL = 00h: bit 7 = 0, motor busy
6	`test al, 10000000b`	00h	0102h	0000h	0000h	1	0	—	00h AND 80h = 0: ZF = 1
7	`jz wait`	00h	0102h	0000h	0000h	1	0	—	ZF = 1: read again
8	`wait: in al, 7`	80h	0102h	0000h	0000h	1	0	—	AL = 80h: motor ready
9	`test al, 10000000b`	80h	0102h	0000h	0000h	0	0	—	80h AND 80h ≠ 0: ZF = 0
10	`jz wait`	80h	0102h	0000h	0000h	0	0	—	does not jump: a byte will be sent
11	`mov al, [bx][si]`	06h	0102h	0000h	0000h	0	0	—	AL ← [0102h + 0] = 06h = 110b

Step	Instruction	AL	BX	SI	CX	ZF	CF	Port 7	Explanation
12	`out 7, al`	06h	0102h	0000h	0000h	0	0	06h	magnets 2 and 1 on: half step (11.25°)
13	`inc si`	06h	0102h	0001h	0000h	0	0	06h	SI = 1
14	`cmp si, 4`	06h	0102h	0001h	0000h	0	1	06h	1 − 4: CF = 1
15	`jb next_step`	06h	0102h	0001h	0000h	0	1	06h	JB jumps: next byte
…	the ready bit is read (80h)								
19	`mov al, [bx][si]`	04h	0102h	0001h	0000h	0	0	06h	AL ← [0103h] = 04h = 100b
20	`out 7, al`	04h	0102h	0001h	0000h	0	0	04h	only magnet 2 on
…	SI = 2, polling								
27	`mov al, [bx][si]`	03h	0102h	0002h	0000h	0	0	04h	AL ← [0104h] = 03h = 011b
28	`out 7, al`	03h	0102h	0002h	0000h	0	0	03h	magnets 1 and 0 on
…	SI = 3, polling								
35	`mov al, [bx][si]`	02h	0102h	0003h	0000h	0	0	03h	AL ← [0105h] = 02h = 010b
36	`out 7, al`	02h	0102h	0003h	0000h	0	0	02h	only magnet 1 on: the first round is finished
37	`inc si`	02h	0102h	0004h	0000h	0	0	02h	SI = 4
38	`cmp si, 4`	02h	0102h	0004h	0000h	1	0	02h	4 − 4 = 0: ZF = 1, CF = 0
39	`jb next_step`	02h	0102h	0004h	0000h	1	0	02h	JB does not jump
40	`mov si, 0`	02h	0102h	0000h	0000h	1	0	02h	SI = 0: back to the start of the table
41	`inc cx`	02h	0102h	0000h	0001h	0	0	02h	CX = 1: one round (4 bytes) complete
42	`cmp cx, steps_before_direction_change`	02h	0102h	0000h	0001h	0	1	02h	1 − 20h: CF = 1
43	`jb next_step`	02h	0102h	0000h	0001h	0	1	02h	jumps: 32 rounds not yet complete, continue with the same table

Table 10.2.14: The first round of the stepper motor program (port 7 returns 00h first, then always 80h).

The first part of the trace shows how polling works. With the 00h read at step 5 the `TEST` result is zero and ZF = 1, so `JZ` jumps back. With the 80h read at step 8, bit 7 is one; the `TEST` result is nonzero, ZF = 0, and the program moves on. `TEST AL, 80h` does not change AL; but the value of AL is wiped out anyway by the next `MOV AL, [BX][SI]`. The status byte is no longer needed.

The second part is the scan of the table. For SI = 0, 1, 2, 3 the values 06h, 04h, 03h, and 02h are written to port 7 in turn. When SI = 4, the result of `CMP SI, 4` is zero: ZF = 1, CF = 0. `JB` jumps only when CF = 1, so it falls through. SI is reset and CX becomes one. After `CMP CX, 20h` CF = 1 (1 < 32) and the program begins a second round with the same table. In the first round the motor has turned four half steps, that is 45°.

10.2.4.5 Real hardware: unipolar motor and ULN2003

The three-magnet motor of emu8086 is a model simplified for teaching. The type of stepper motor most often met in the laboratory is the **unipolar** motor. This motor has two center-tapped windings. Each winding is split in two at its center tap, giving four winding halves: A, B, $\bar{A}$, and $\bar{B}$. The center taps are connected to the motor's supply voltage (+V). That is why the motor has five or six wires.

To pass current through a winding half, its other end is pulled to ground. The winding current is hundreds of milliamperes, while one output pin of the 8255 can drive only a few milliamperes. A driver chip is placed in between: the **ULN2003**. The ULN2003 has seven Darlington transistor channels, and each channel can sink up to 500 mA. When the input of a channel is 1, its output pulls to ground and current flows through that winding half.

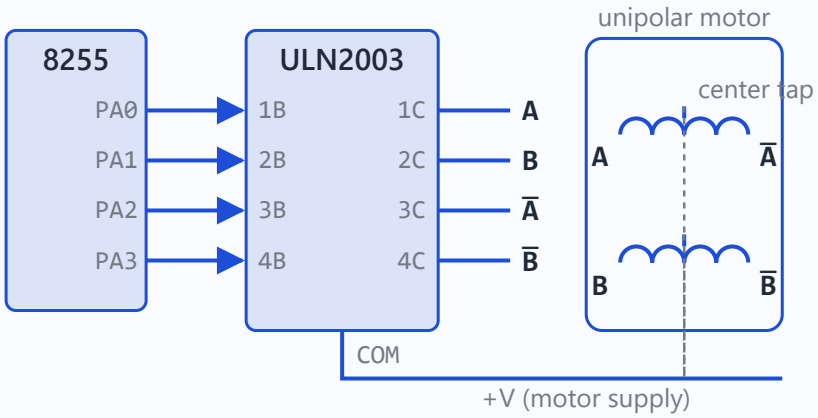


Figure 10.2.11: Pins PA0–PA3 of the 8255 drive the winding halves A, B, $\bar{A}$, and $\bar{B}$ of a unipolar motor through four channels of the ULN2003. The center taps and the COM pin of the ULN2003 are connected to the motor supply (+V).

A winding is a coil, and the moment its current is cut a high voltage of opposite polarity appears across it. This voltage can destroy the transistor. Each channel of the ULN2003 has a diode for this purpose; when the COM pin is connected to +V, the diodes discharge the reverse voltage into the supply. Leaving the COM pin unconnected is the most common cause of a driver failing quickly.

A real motor is driven **open loop**: there is no "ready" bit as in the emu8086 motor. The program sets the speed of the motor by how long it waits between steps. If the wait is too short, the rotor cannot keep up with the next excitation and **skips a step**. The program cannot notice the lost step; the shaft

is no longer at the position the program believes. For this reason the step interval is chosen according to the maximum step rate in the motor's datasheet.

Drive	Port A sequence	Property
Single phase (wave), full step	01h 02h 04h 08h	one winding half at a time; least current
Two phase, full step	03h 06h 0Ch 09h	two winding halves at a time; higher torque
Half step	01h 03h 02h 06h 04h 0Ch 08h 09h	mixes the two sequences; the step angle is halved

Table 10.2.15: Excitation sequences of a unipolar motor (bits 3…0 of port A = $\bar{B}$ $\bar{A}$ B A).

The logic of the sequences is visible at bit level. In the single-phase sequence the 1 bit moves from A to B, $\bar{A}$, and $\bar{B}$: 0001, 0010, 0100, 1000. In the two-phase sequence two adjacent bits move together: 0011, 0110, 1100, 1001. The last byte, 1001b, says that $\bar{B}$ and A are on together; the sequence closes into a ring. The half-step sequence interleaves the elements of these two: single, double, single, double. The step angle is thus halved; for example a 1.8° motor (200 steps per revolution) turns in 0.9° steps. If the sequence is applied in reverse, the motor turns in the opposite direction.

The tables of the emu8086 motor (110b, 100b …) cannot be applied to this four-phase motor. The bit layout of the three-magnet model differs from that of the motor with four winding halves. For a real motor the table is rewritten according to the order in which the windings are wired to the port.

10.2.4.6 Code trace: half-step drive

The following program sends the half-step sequence to port A of the 8255. Port A is assumed to have been made an output earlier with 82h. The position in the sequence is held in the memory word `pos`.

```
org 100h
fwd:
inc  pos            ; reverse: dec pos
mov  bx, pos
and  bx, 7          ; index 0–7
mov  al, half[bx]
mov  dx, 310h       ; port A
out  dx, al
call delay          ; step gap = speed
jmp  fwd
delay: ret          ; (delay)
half db 01h,03h,02h,06h
     db 04h,0Ch,08h,09h
pos  dw 6
```

The key trick of the program is `AND BX, 7`. `pos` keeps increasing: 6, 7, 8, 9 … The sequence, however, has only eight elements. ANDing with 7 = 0000 0111b leaves only the last three bits of the number, so the result is always between 0 and 7. For 8 = 1000b the result is 0, for 9 it is 1, and the

sequence wraps around by itself. This is a way to compute the remainder of division by 8 without a division instruction, and it works only for sequences whose length is a power of 2.

To turn backward it is enough to write `DEC pos` instead of `INC pos`. When `pos` drops by 1 from 0 it becomes FFFFh; `AND BX, 7` turns this into 7 and the sequence is read from the end to the start. Because the counter lives in memory, `INC pos` is an instruction with a memory operand (`FF 06 21 01`): the processor reads the word, increments it, and writes it back. The delay routine in this trace is just `RET` ; in a real program it sets the step interval, that is the speed of the motor.

Step	Instruction	pos	BX	AL	SP	Port A	Explanation
1	`inc pos`	0007h	0000h	00h	FFFEh	—	the word in memory 6 → 7
2	`mov bx, pos`	0007h	0007h	00h	FFFEh	—	BX ← pos = 7
3	`and bx, 7`	0007h	0007h	00h	FFFEh	—	the index stays in the range 0–7: 7
4	`mov al, half[bx]`	0007h	0007h	09h	FFFEh	—	AL ← half[7] = 09h = 1001b: $\bar{B}$ and A
5	`mov dx, 310h`	0007h	0007h	09h	FFFEh	—	DX = 310h: 8255 port A
6	`out dx, al`	0007h	0007h	09h	FFFEh	09h	the windings are driven: the field lies between A and $\bar{B}$
7	`call delay`	0007h	0007h	09h	FFFCh	09h	return address (0116h) onto the stack: SP = FFFCh
8	`delay: ret`	0007h	0007h	09h	FFFEh	09h	the delay routine is just RET here
9	`jmp fwd`	0007h	0007h	09h	FFFEh	09h	next step
10	`inc pos`	0008h	0007h	09h	FFFEh	09h	pos = 8
11	`mov bx, pos`	0008h	0008h	09h	FFFEh	09h	BX = 8
12	`and bx, 7`	0008h	0000h	09h	FFFEh	09h	8 AND 7 = 0: the sequence wraps around
13	`mov al, half[bx]`	0008h	0000h	01h	FFFEh	09h	AL ← half[0] = 01h: only A
…	DX = 310h						
15	`out dx, al`	0008h	0000h	01h	FFFEh	01h	the field returns to winding A: one half step
…	CALL, RET, JMP						
19	`inc pos`	0009h	0000h	01h	FFFEh	01h	pos = 9
20	`mov bx, pos`	0009h	0009h	01h	FFFEh	01h	BX = 9
21	`and bx, 7`	0009h	0001h	01h	FFFEh	01h	9 AND 7 = 1
22	`mov al, half[bx]`	0009h	0001h	03h	FFFEh	01h	AL ← half[1] = 03h: A and B
…	DX = 310h						
24	`out dx, al`	0009h	0001h	03h	FFFEh	03h	the field moves between A and B: one more half step

Table 10.2.16: Trace of the program that drives the half-step sequence for a real motor (pos starts at 6).

The angles in the trace show the direction of the field between the windings. 09h = 1001b says that $\bar{B}$ and A are on together; the field points between the two, at about 315°. At 01h only A is on (0°), at 03h A and B (45°). Every `OUT` advances the field by a winding angle of 45°, that is the rotor by one half step.

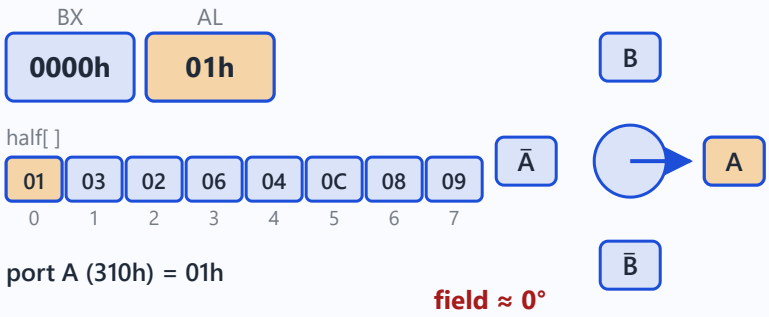


Figure 10.2.12: When pos = 8, `AND BX, 7` wraps the index to 0; with half[0] = 01h only winding A stays on and the field comes to 0°.

10.2.5 LED display and 7-segment display

10.2.5.1 The emu8086 LED display

The LED display of emu8086 is attached to port 199 and shows a five-digit decimal number. The 16-bit word written to the port is interpreted as a **signed** number in two's complement. The range that can be shown is therefore −32,768 … 32,767. Since 199 ≤ 255 the port is written as a constant: `OUT 199, AX`. AX is used because the display expects a word; `OUT 199, AL` would send only the low byte.

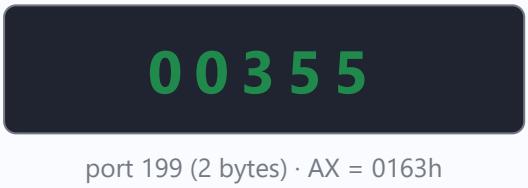


Figure 10.2.13: LED display: AX = 0163h written to port 199 appears as decimal 355.

The job of the display is to convert a binary number to decimal. The programmer thinks of the number in decimal, the assembler converts it to binary, and the device turns it back into decimal. `MOV AX, 1234` writes 04D2h into AX (1234 = 4 · 256 + 210 = 04D2h). It is this binary word that goes to the display.

Negative numbers are written in two's complement. For −5678, first find the hexadecimal value of 5678: 162Eh. The two's complement is 10000h − 162Eh = E9D2h. When `MOV AX, -5678` is written, the assembler does this computation and produces the code `B8 D2 E9`. The display sees that the highest bit of E9D2h is 1 and shows the number as −5678. Read as unsigned, the same word would be 59,858; the interpretation by the display determines the meaning of the bits.

The LED display example of emu8086 begins with the `#make_bin#` directive. This directive produces a plain binary file (.bin) that does not depend on DOS: no header, no PSP, only machine code. This is exactly the kind of file that would be written into the ROM of an embedded system.

10.2.5.2 Code trace: the LED display program

```
#start=led_display.exe#
mov ax, 1234
out 199, ax
mov ax, -5678
out 199, ax
; write the port in an endless loop:
mov ax, 0
x1:
out 199, ax
inc ax
jmp x1
hlt              ; never runs
```

The program first shows two constant numbers, then sets up a counter that starts at 0 and increases without stopping. The loop contains `INC AX` and `JMP x1` but no exit condition. The `HLT` after the loop is therefore never reached. `HLT` is written only to mark the end of the program.

Step	Instruction	AX	Display	Explanation
1	`mov ax, 1234`	04D2h	—	1234 = 04D2h
2	`out 199, ax`	04D2h	1234	1234 on the display
3	`mov ax, -5678`	E9D2h	1234	−5678 = 10000h − 162Eh = E9D2h
4	`out 199, ax`	E9D2h	−5678	the display reads the word as signed: −5678
5	`mov ax, 0`	0000h	−5678	counter 0
6	`out 199, ax`	0000h	0	0 on the display
7	`inc ax`	0001h	0	AX = 1
8	`jmp x1`	0001h	0	back to x1; HLT is never reached
9	`out 199, ax`	0001h	1	1 on the display
10	`inc ax`	0002h	1	AX = 2
11	`jmp x1`	0002h	1	loop
12	`out 199, ax`	0002h	2	2 on the display

Table 10.2.17: Trace of the LED display program; the display column is the signed value of the word last written to port 199.

After the counter reaches 32,767 (7FFFh), `INC AX` makes it 8000h. The display reads 8000h as signed and shows −32,768. From there the counter rises to −1 (FFFFh) and then wraps to 0. `INC` sets OF to 1 at this crossing, but since the program does not look at the flags nothing happens.

In emu8086 with the *step delay* set to 0, the numbers stream by too fast to read. If a wait with INT 15h/86h is added to the loop, the counter can be followed by eye. For example, to wait 0.2 second after each number, add `MOV CX, 3`, `MOV DX, 0D40h`, `MOV AH, 86h`, `INT 15h` after the `OUT`. Because the wait changes AH, the counter must be kept in a register other than AX; for example count in `BX` and copy it to AX with `MOV AX, BX` before the `OUT`.

10.2.5.3 The 7-segment display

The LED display of emu8086 decodes the number itself. In real hardware the program does this job. The most common numeric display is the **seven-segment display**. It has seven bar-shaped LEDs (a–g); the decimal point (dp) is the eighth LED. Each LED is called a **segment** (not to be confused with a memory segment).

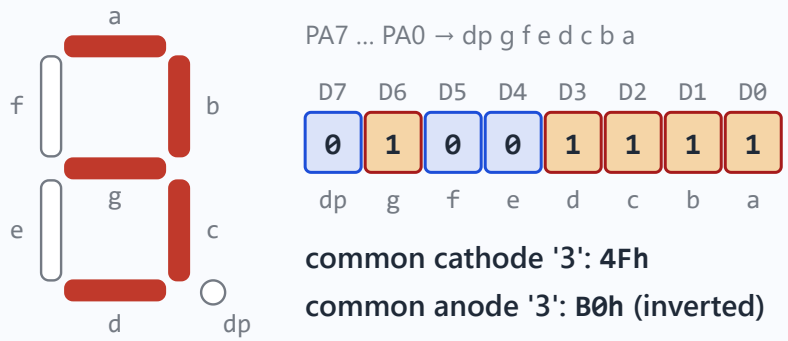


Figure 10.2.14: The 7-segment display and its connection to port A (PA7 ... PA0 → dp g f e d c b a). For the digit '3', 4Fh is sent for a common cathode and its inverse B0h for a common anode.

One terminal of each of the display's eight LEDs is joined at a common pin. How this pin is wired distinguishes the two types. In a **common-cathode** (CC) display the common pin is at ground; giving 1 to a segment pin lights the LED. In a **common-anode** (CA) display the common pin is at +5 V; giving 0 to a segment pin lights the LED. The code for a common anode is therefore the bitwise inverse of the common-cathode code and is obtained with `NOT AL`. The LEDs are connected not directly to the port pins but each through a current-limiting resistor.

The segment code is computed from the segments that have to light. The bits of port A are in the order dp g f e d c b a: bit 0 = a, bit 6 = g, bit 7 = dp. For the digit '3', segments a, b, c, d, and g light; bits 0, 1, 2, 3, and 6 are one: 0100 1111b = 4Fh. For a common anode its inverse 1011 0000b = B0h is sent.

Digit	Segments lit	Common cathode (CC)	Common anode (CA)
0	a b c d e f	3Fh	C0h
1	b c	06h	F9h
2	a b d e g	5Bh	A4h
3	a b c d g	4Fh	B0h

Digit	Segments lit	Common cathode (CC)	Common anode (CA)
4	b c f g	66h	99h
5	a c d f g	6Dh	92h
6	a c d e f g	7Dh	82h
7	a b c	07h	F8h
8	a b c d e f g	7Fh	80h
9	a b c d f g	6Fh	90h

Table 10.2.18: Segment codes of the digits (in the order dp g f e d c b a).

The table is not limited to digits. Hexadecimal digits and some letters can also be written with seven segments. For 'A', a b c e f g light: 0111 0111b = 77h. For a lowercase 'b', c d e f g: 0111 1100b = 7Ch. The minus sign is segment g alone: 0100 0000b = 40h.

10.2.5.4 Code table and XLAT

Converting a digit to a segment code is a table lookup. The 8086 has a one-byte instruction for this: **XLAT** (translate). `XLAT` performs AL ← DS:[BX + AL]. BX points to the start of the table and AL holds the index; the byte in the table is written into AL.

```
; digit (0–9) to segment code
mov  bx, offset segtab
mov  al, 3          ; digit
xlat                ; AL = [BX + AL] = 4Fh
; not al            ; if common anode
mov  dx, 310h       ; 8255 port A
out  dx, al
ret
segtab db 3Fh,06h,5Bh,4Fh,66h
       db 6Dh,7Dh,07h,7Fh,6Fh
```

In the example AL = 3, and `XLAT` fetches byte number 3 (counting from 0) of the code table, 4Fh. If the display is common-anode, the `NOT AL` in the comment line is enabled and AL becomes B0h. The result is written to port A of the 8255. The only restriction of `XLAT` is that the table has at most 256 bytes; AL is an 8-bit index. The instruction does not touch the flags and takes 11 clock cycles.

The advantage of the table method over computation is clear. There is no short way to derive the segment code from the digit with a formula; ten separate comparisons would be needed. The table gives the result for every digit in the same time with a single instruction. If the display changes (for example if the segment pins are wired to the port in a different order), only the table is rewritten.

Common mistake

Giving the digit to the table as ASCII. The code of the character '3' read from the keyboard is 33h = 51; `XLAT` reads 51 bytes past the start of the table and fetches a random byte from outside the table. An ASCII digit is first converted to a number with `SUB AL, 30h`.

10.2.5.5 Multiplexed scanning

Connecting a separate 8-bit port to each digit of a four-digit display would need 32 pins. **Multiplexing** reduces this number to 12. The segment pins of the same name on the four digits are tied together and go to the shared port A. The common pin of each digit is connected through its own transistor to one bit of port C. At any moment the program lights only one digit: it writes the code of that digit to port A and sets only that digit's bit to 1 in port C.

When the digits are lit one after another quickly, the eye sees all four lit together. If about 4 ms is given to each digit, a full pass takes 16 ms, so the scan rate is 1 / 16 ms ≈ 62 Hz. The human eye does not notice flicker above about 50 Hz. If the scan slows down (for example 20 ms per digit), a clear flicker is seen on the display.

```
scan:
mov  si, 0
next_dig:
mov  dx, 312h       ; port C: digit select
mov  al, 0
out  dx, al         ; blank all digits first
mov  al, digits[si]
mov  bx, offset segtab   ; table above
xlat
mov  dx, 310h       ; port A: segments
out  dx, al
mov  al, select[si] ; 01h, 02h, 04h, 08h
mov  dx, 312h
out  dx, al         ; only this digit lit
call delay_4ms
inc  si
cmp  si, 4
jb   next_dig
jmp  scan
delay_4ms:          ; ≈ 4 ms delay
mov  cx, 1000
loop $
ret
digits db 1, 2, 3, 4
select db 01h, 02h, 04h, 08h
```

The most important line of the loop is the `OUT DX, AL` that blanks all digits before the digit is changed (AL = 0, DX = 312h). Without this line, at the moment the new code is written to port A the previous digit would still be lit. The new code would appear for an instant on the previous digit too and leave a faint trace there; this is called **ghosting**. The correct order is: blank everything, write the segment code, light only the wanted digit, wait.

The `select` table holds the byte to be written to port C for each digit. 01h, 02h, 04h, and 08h set PC0, PC1, PC2, and PC3 to 1 respectively. These values could also have been produced by shifting 1 left by SI (`MOV CX, SI` · `MOV AL, 1` · `SHL AL, CL`); the table is both shorter and faster.

The following abbreviated program scans the number 2026 once across the display. The delay routine is removed, and a single pass and `RET` replace the endless loop.

```
org 100h
mov  bx, offset segtab
mov  si, 0
next_dig:
mov  dx, 312h      ; port C (digit)
mov  al, 0
out  dx, al        ; all blank
mov  al, digits[si]
xlat               ; segment code
mov  dx, 310h      ; port A (segments)
out  dx, al
mov  al, select[si]
mov  dx, 312h
out  dx, al        ; one digit lit
inc  si
cmp  si, 4
jb   next_dig
ret
segtab db 3Fh,06h,5Bh,4Fh,66h
       db 6Dh,7Dh,07h,7Fh,6Fh
digits db 2, 0, 2, 6
select db 01h, 02h, 04h, 08h
```

BX is loaded once outside the loop, because it does not change in the loop. The code is 36 bytes and the tables follow right behind it: `segtab` is at 0124h, `digits` at 012Eh, and `select` at 0132h. Table 10.2.19 gives the important rows of the trace; the port columns show the last value written to the port up to that moment.

Step	Instruction	SI	AL	Port A	Port C	Explanation
1	`mov bx, offset segtab`	0000h	00h	—	—	BX = 0124h: the start of the code table
2	`mov si, 0`	0000h	00h	—	—	SI = 0: first position
3	`mov dx, 312h`	0000h	00h	—	—	DX = 312h: port C, digit select
4	`mov al, 0`	0000h	00h	—	—	AL = 0
5	`out dx, al`	0000h	00h	—	00h	all digits blank: no ghosting

Step	Instruction	SI	AL	Port A	Port C	Explanation
6	`mov al, digits[si]`	0000h	02h	—	00h	AL ← digits[0] = 2
7	`xlat`	0000h	5Bh	—	00h	AL ← [BX + 2] = 5Bh
8	`mov dx, 310h`	0000h	5Bh	—	00h	DX = 310h: port A, segments
9	`out dx, al`	0000h	5Bh	5Bh	00h	the segment code goes out on the port
10	`mov al, select[si]`	0000h	01h	5Bh	00h	AL ← select[0] = 01h
11	`mov dx, 312h`	0000h	01h	5Bh	00h	DX = 312h
12	`out dx, al`	0000h	01h	5Bh	01h	only position 1 lit: "2"
13	`inc si`	0001h	01h	5Bh	01h	SI = 1
…	CMP, JB; position 2: port C ← 00h, digit 0					
20	`xlat`	0001h	3Fh	5Bh	00h	XLAT: AL = 3Fh
22	`out dx, al`	0001h	3Fh	3Fh	00h	port A ← 3Fh
25	`out dx, al`	0001h	02h	3Fh	02h	only position 2 lit: "0"
…	position 3: digit 2					
33	`xlat`	0002h	5Bh	3Fh	00h	XLAT: AL = 5Bh
38	`out dx, al`	0002h	04h	5Bh	04h	only position 3 lit: "2"
…	position 4: digit 6					
46	`xlat`	0003h	7Dh	5Bh	00h	XLAT: AL = 7Dh
51	`out dx, al`	0003h	08h	7Dh	08h	only position 4 lit: "6"
52	`inc si`	0004h	08h	7Dh	08h	SI = 4
53	`cmp si, 4`	0004h	08h	7Dh	08h	equal: CF = 0
54	`jb next_dig`	0004h	08h	7Dh	08h	the loop ends
55	`ret`	0004h	08h	7Dh	08h	in a real program the scan starts over

Table 10.2.19: Trace of the program that scans 2026 on a four-digit display by multiplexing (common-cathode codes).

In the trace the value of port A changes while port C is 00h. For example, at step 20, when XLAT brings 3Fh, port A is still 5Bh, but because port C = 00h no digit is lit. After port A becomes 3Fh (step 22), port C is made 02h (step 25). Each digit lights while its own segment code is on the port and only it is selected.

10.2.6 Thermometer and thermostat

10.2.6.1 The device

The thermometer device of emu8086 represents the temperature of a room and the heater in it. It uses two ports. `IN AL, 125` reads the current air temperature in degrees Celsius; the value is a signed byte, so it can be between −128 and 127 °C. `OUT 127, AL` controls the heater: AL = 1 turns the heater on, AL = 0 turns it off. While the heater is on the temperature rises slowly, and while it is off it falls.

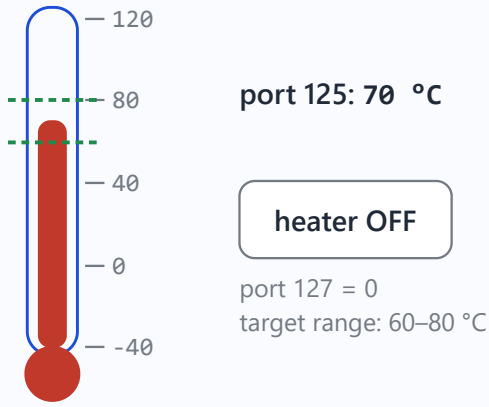


Figure 10.2.15: The thermometer device: the temperature read from port 125 is 70 °C and the heater is off (port 127 = 0). The target range is 60–80 °C.

The goal is a **thermostat** that keeps the temperature in the range 60–80 °C. The rule is simple: if the temperature drops below 60 the heater is switched on, if it rises above 80 it is switched off, and in between the heater is left alone. This "hands-off" zone is left deliberately. If a single threshold (say 70 °C) were used, the heater would switch on and off several times a second while the temperature oscillated around the threshold. The gap between the two thresholds is called **hysteresis**; household thermostats, refrigerators, and air conditioners work the same way.

10.2.6.2 The thermostat program

```
; make DS equal to CS:
mov ax, cs
mov ds, ax
start:
in  al, 125
cmp al, 60
jl  low
cmp al, 80
jle ok
jg  high
low:
mov al, 1
out 127, al      ; heater on
jmp ok
high:
mov al, 0
out 127, al      ; heater off
ok:
jmp start        ; endless loop
```

The first two lines set DS equal to CS. Because a constant cannot be written directly into a segment register, the value goes into AX first. There are no variables in this program, so the lines are not strictly needed; they are left over from the original emu8086 sample. In a .COM program DS already equals CS.

The loop begins with `IN AL, 125`. Two comparisons follow. `CMP AL, 60` performs AL − 60 and only sets the flags. `JL low` jumps if the result is "less", that is if SF ≠ OF. If it does not jump, the temperature is at least 60. `CMP AL, 80` and `JLE ok` test the second threshold: if ZF = 1 or SF ≠ OF the temperature is at most 80 and the program returns to the loop without touching the heater. If neither jumps, the temperature is above 80; `JG high` goes to the code that switches the heater off.

The condition of the `JG high` line is in fact always true. If `JLE` did not jump, the temperature must be above 80; `JG` therefore always jumps, and `JMP high` could have been written instead. The line cannot be deleted altogether: the `low` block sits right below it and the program would turn the heater on in a hot room. `JG` is preferred because it states the condition openly to the reader.

All the comparisons are made with **signed** jumps (`JL`, `JLE`, `JG`), because the temperature can be negative. Unsigned jumps (`JB`, `JBE`, `JA`) count the byte as a number between 0 and 255. This difference reverses the behavior of the program at negative temperatures.

Theorem 10.2.1 (Signed comparison)

If a and b are compared as signed numbers after `CMP a, b`, then a < b holds if and only if SF ≠ OF. a ≤ b holds when ZF = 1 or SF ≠ OF. In an unsigned comparison, a < b holds if and only if CF = 1.

10.2.6.3 Code trace

The trace assumes that port 125 returns 55, 70, and 85 °C in turn and that the heater is off at the start. The three values test the three paths of the program once each: cold, in range, and hot.

Step	Instruction	AL	ZF	SF	OF	CF	Heater	Explanation
1	`mov ax, cs`	00h	0	0	0	0	off	AX = CS = 0700h
2	`mov ds, ax`	00h	0	0	0	0	off	DS = 0700h; a constant cannot be written directly into a segment register
3	`in al, 125`	37h	0	0	0	0	off	55 °C = 37h
4	`cmp al, 60`	37h	0	1	0	1	off	55 − 60 = −5: SF = 1, OF = 0
5	`jl low`	37h	0	1	0	1	off	SF ≠ OF: jumps (low)
6	`mov al, 1`	01h	0	1	0	1	off	AL = 1
7	`out 127, al`	01h	0	1	0	1	on	the heater is switched on
8	`jmp ok`	01h	0	1	0	1	on	jump to ok
9	`jmp start`	01h	0	1	0	1	on	new measurement
10	`in al, 125`	46h	0	1	0	1	on	70 °C = 46h
11	`cmp al, 60`	46h	0	0	0	0	on	70 − 60 = 10: SF = 0, OF = 0
12	`jl low`	46h	0	0	0	0	on	SF = OF: does not jump

Step	Instruction	AL	ZF	SF	OF	CF	Heater	Explanation
13	`cmp al, 80`	46h	0	1	0	1	on	70 − 80 = −10: SF = 1, ZF = 0
14	`jle ok`	46h	0	1	0	1	on	ZF = 1 or SF ≠ OF: jumps; the heater is left alone
15	`jmp start`	46h	0	1	0	1	on	new measurement
16	`in al, 125`	55h	0	1	0	1	on	85 °C = 55h
17	`cmp al, 60`	55h	0	0	0	0	on	85 − 60 = 25
18	`jl low`	55h	0	0	0	0	on	does not jump
19	`cmp al, 80`	55h	0	0	0	0	on	85 − 80 = 5: SF = 0, ZF = 0
20	`jle ok`	55h	0	0	0	0	on	does not jump
21	`jg high`	55h	0	0	0	0	on	ZF = 0 and SF = OF: jumps (high)
22	`mov al, 0`	00h	0	0	0	0	on	AL = 0
23	`out 127, al`	00h	0	0	0	0	off	the heater is switched off
24	`jmp start`	00h	0	0	0	0	off	new measurement

Table 10.2.20: Trace of the thermostat program; port 125 returns 55, 70, and 85 °C in turn, and the heater is off at the start.

In the first measurement 55 − 60 = −5. Since the result is negative, SF = 1; there is no signed overflow, so OF = 0. Because SF ≠ OF, `JL` jumps and the heater is switched on. CF is also 1, because 55 < 60 unsigned as well; for these values the signed and unsigned interpretations give the same result.

In the second measurement 70 − 60 = 10 gives SF = OF = 0 and `JL` does not jump. 70 − 80 = −10 sets SF = 1; `JLE` jumps and the program returns without touching the heater. Because the heater was switched on in the previous round, it stays on; the temperature keeps rising. In the third measurement 85 passes both comparisons: `JG` jumps on the condition ZF = 0 and SF = OF, and the heater is switched off.

The interactive thermostat in the slides runs the same program over time: the temperature rises and falls according to the state of the heater, and the lines the program is executing are highlighted. You can see the temperature moving back and forth between 60 and 80 while the heater changes state only at the thresholds.

10.2.6.4 Negative temperatures and unsigned jumps

Let the temperature be −10 °C. Port 125 returns the two's complement of −10, F6h. For the signed program, `CMP AL, 60` computes −10 − 60 = −70. Since the result is greater than −128 there is no overflow: SF = 1, OF = 0. `JL` jumps and the heater is switched on. This is the correct behavior.

If the program had been written with unsigned jumps (`JB low`, `JBE ok`, `JA high`), F6h would count as 246. 246 < 60 is false: CF = 0 and `JB` does not jump. 246 ≤ 80 is false too; `JA` jumps and the heater is *switched off*. While the room freezes the thermostat keeps the heater off. The same error appears at every temperature between −1 and −128, because all of them are greater than 127 when read as unsigned.

Common mistake

Choosing a jump instruction without checking whether the value read from the device is signed or unsigned. The rule comes from the meaning of the data: quantities that can be negative, such as temperature, acceleration, or a speed difference, are compared with the `JL` / `JG` family; quantities that are never negative, such as addresses, counters, ASCII codes, and status bytes, are compared with the `JB` / `JA` family.

10.2.7 Printer

10.2.7.1 The DOS printer service and control characters

The printer of emu8086 is attached to port 130, but the program does not write to the port directly. Instead it uses the DOS printer service: **INT 21h, AH = 05h** sends the character in DL to the printer (the PRN device). The service also does the job of waiting for the printer to be ready. On a real PC this service uses the BIOS INT 17h routine, which in turn uses the registers of the parallel port. The program does not have to know any of these details.

Most of the bytes sent to the printer are characters to be printed. Some are **control characters**: they print nothing on the paper but move the print head or the paper (Table 10.2.21).

Kod	Ad	Effect
0Ah (10)	LF (line feed)	one line down
0Dh (13)	CR (carriage return)	return to the start of the line
09h (9)	HT (horizontal tab)	to the next horizontal tab stop (every 8 columns)
0Ch (12)	FF (form feed)	eject the page, move to a new page

Table 10.2.21: Control characters that command the printer.

The names of these characters come from the typewriter. CR returns the typewriter's carriage (the sled that carries the paper) to the start of the line; LF pushes the paper up by one line. On a typewriter the two motions were made with separate levers, and the printer likewise expects them as separate characters. To move to a new line both are needed. The usual order is CR + LF, but LF + CR gives the same result: the head first goes down, then to the start. If only CR is sent, the head returns to the start of the same line and the following characters are printed over the old ones. Vertical tab (VT) is 0Bh and is not used in this program.

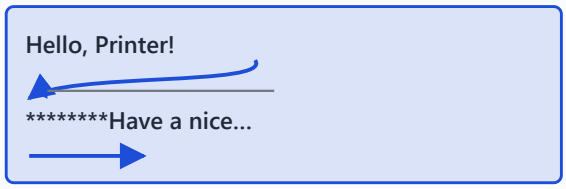


Figure 10.2.16: Control characters on the printer paper: LF and CR take the head to the start of the next line; CR alone returns to the start of the same line; HT jumps to column 8; FF ejects the page.

10.2.7.2 Data section and length computation

```
org 100h
jmp start
msg  db "Hello, Printer!", 0Ah, 0Dh
     db "***************"
     db 13, 9           ; CR and horizontal tab (HT)
     db "Have a nice printing day!"
msg_end db 0
msg2 db "press any key to eject the page.$"
```

The text is split over several `DB` lines, but in memory it is a single unbroken sequence of bytes. The number of characters to print is not counted by hand; the assembler is made to compute it:

> offset msg_end − offset msg = 15 + 2 + 15 + 2 + 25 = 59 = 3Bh bytes.

"Hello, Printer!" is 15 characters, LF with CR 2, the asterisks 15, CR with HT 2, and "Have a nice printing day!" 25. Since `msg` starts at 0102h, `msg_end` is at 0102h + 3Bh = 013Dh. The 0 byte at `msg_end` is not printed; the label is there only to mark the address right after the last character. If the text changes, the length is computed correctly by itself.

`msg2` ends with '$'. This string goes not to the printer but to the screen and is printed with INT 21h/09h; service 09h writes up to the '$' character. The 33-byte msg2 starts at 013Eh and ends at 015Eh; the code therefore starts at 015Fh, and the `JMP start` (`EB 5D`) in the first line jumps there.

10.2.7.3 The print loop

```
start:
mov dl, 12        ; form feed: new page
mov ah, 5
int 21h
mov si, offset msg
mov cx, offset msg_end - offset msg
print:
mov dl, [si]
mov ah, 5         ; DOS printer function
int 21h
inc si            ; next character
loop print
mov dx, offset msg2
mov ah, 9
int 21h           ; message on screen
mov ax, 0
int 16h           ; wait for a key
mov dl, 12        ; eject the page
mov ah, 5
int 21h
ret
```

To start with a clean page the program first sends a form feed. Then SI takes the start of the string and CX the number of characters. The loop body reads one character into DL, writes 05h into AH, and calls the service. `MOV AH, 5` is repeated in every round; because INT 21h services may change AH, this is the safe way. `INC SI` moves on to the next character; `LOOP print` decrements CX and goes back to the top if it is not zero. The loop runs exactly 59 times.

After the loop "press any key to eject the page." is written to the screen and a key is awaited with INT 16h/00h. When the user presses a key, a final form feed ejects the page from the printer. The program ends with `RET`.

Step	Instruction	CX	DL	SI	To printer	Explanation
1	`jmp start`	0000h	00h	0000h		jumps over the data bytes to 015Fh
2	`mov dl, 12`	0000h	0Ch	0000h		DL = 12 = 0Ch: form feed
3	`mov ah, 5`	0000h	0Ch	0000h		AH = 05h: DOS printer output
4	`int 21h`	0000h	0Ch	0000h	FF	the printer moves to a new page
5	`mov si, offset msg`	0000h	0Ch	0102h		SI = 0102h: msg
6	`mov cx, offset msg_end - offset msg`	003Bh	0Ch	0102h		013Dh − 0102h = 3Bh = 59 characters

Step	Instruction	CX	DL	SI	To printer	Explanation
7	`mov dl, [si]`	003Bh	48h	0102h		DL ← [0102h] = 'H'
8	`mov ah, 5`	003Bh	48h	0102h		AH = 05h
9	`int 21h`	003Bh	48h	0102h	'H'	'H' goes to the printer
10	`inc si`	003Bh	48h	0103h		SI = 0103h
11	`loop print`	003Ah	48h	0103h		CX = 58 ≠ 0: back to print
...	continues character by character					
84	`int 21h`	002Ch	0Ah	0111h	LF	byte 16: LF, the paper advances one line
89	`int 21h`	002Bh	0Dh	0112h	CR	byte 17: CR, start of line
...	15 asterisks					
169	`int 21h`	001Bh	0Dh	0122h	CR	byte 33: CR; no LF, so the head returns to the start of the same line
174	`int 21h`	001Ah	09h	0123h	HT	byte 34: HT, jumps to the eight-column tab stop
...	"Have a nice printing day" is printed over the asterisks					
297	`mov dl, [si]`	0001h	21h	013Ch		DL ← [013Ch] = '!'
299	`int 21h`	0001h	21h	013Ch	'!'	last character
300	`inc si`	0001h	21h	013Dh		SI = 013Dh = msg_end
301	`loop print`	0000h	21h	013Dh		CX = 0: the loop ends
302	`mov dx, offset msg2`	0000h	3Eh	013Dh		DX = address of msg2 (013Eh)
304	`int 21h`	0000h	3Eh	013Dh		message on screen (up to the $)
306	`int 16h`	0000h	3Eh	013Dh		waits for a key
309	`int 21h`	0000h	0Ch	013Dh	FF	last form feed: the paper is ejected
310	`ret`	0000h	0Ch	013Dh		the program ends

Table 10.2.22: Trace of the printer program (selected steps; the "To printer" column is the character sent to the printer at that step).

The "To printer" column of the trace shows the byte sent to the printer at each `INT 21h`. The LF at step 84 and the CR at step 89 bring the paper to the start of the second line. After the CR at step 169 there is no LF: the head returns to the start of the same line, at the asterisks. The HT at step 174 moves the head to column 8 (counting from 0). "Have a nice printing day!" therefore starts after the first eight asterisks and is printed over the remaining seven.

10.2.7.4 What appears on paper?

When the program ends, the page has two lines:

```
Hello, Printer!
********Have a nice printing day!
```

On the second line the first eight asterisks are visible; asterisks 9 to 15 are printed on the same place as the letters of the words "Have a ". On a real dot-matrix printer two characters would appear on top of each other at these places; the paper view of emu8086 shows the upper character. This combination of carriage return and tab characters was used deliberately in the typewriter era to get underlined or bold text: the same line was printed twice.

> **Common mistake**
>
> Writing the character count by hand: `MOV CX, 59`. When a single letter is added to the text, the number becomes wrong; either the last character is not printed or the bytes beyond `msg_end` (here the start of msg2) go to the printer. The difference `offset msg_end - offset msg` is computed at assembly time and is updated automatically with the text.

10.2.8 Robot

10.2.8.1 Ports, commands, and status bits

The robot of emu8086 moves in a grid world. The world has walls and lamps. The robot can examine the cell in front of it, go forward, turn left or right, and switch the lamp in front of it on and off. The world is edited with the toolbox in the device window: walls and lamps are added and the robot's position is changed.

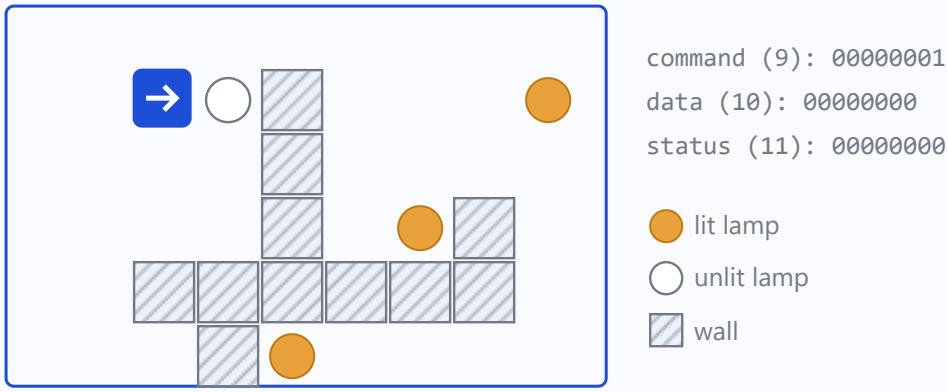


Figure 10.2.17: The robot world: walls (hatched), lit and unlit lamps, the robot, and the current bytes of the three ports.

The robot uses three ports, and this trio is a pattern often seen in real devices as well. Port 9 is the **command** port: the program writes what the robot should do here (OUT). Port 10 is the **data** port: the result of the examine command is read from here (IN). Port 11 is the **status** port: whether the robot is busy, whether new data is available, and whether the last command gave an error are read from here (IN).

Command	Effect	Data	Ahead
0	do nothing	255 (FFh)	wall
1	move forward	0	empty
2	turn left	7	lit lamp
3	turn right	8	unlit lamp
4	examine what is ahead		
5	turn the lamp on		
6	turn the lamp off		

Table 10.2.23: Robot commands (port 9) and examine results (port 10).

Bit	If 1
bit 0	new data in the data register
bit 1	robot busy (command in progress)
bit 2	error in the last command

Table 10.2.24: The robot status byte (port 11).

Each bit of the status byte carries a separate piece of information, and several can be 1 at the same time. 03h = 0000 0011b says that the robot is busy and also that there is unread data in the data register. 05h = 0000 0101b shows that the new-data and error bits are set together. The program

therefore does not compare the status with a number (`CMP AL, 2`); it tests only the bit it cares about (`TEST AL, 010b`).

10.2.8.2 Find the lamp and light it

The following program makes the robot wander through the world and light the unlit lamps it meets. On every round it examines what is ahead. If there is a wall it turns right, if there is an unlit lamp it lights it, and otherwise it goes forward.

```
#start=robot.exe#
main_loop: CALL wait_ready
        MOV  AL, 4          ; examine
        OUT  9, AL
        CALL wait_data
        IN   AL, 10
        CMP  AL, 0FFh       ; wall?
        JE   turn
        CMP  AL, 8          ; lamp off?
        JNE  advance
        CALL wait_ready
        MOV  AL, 5          ; lamp on
        OUT  9, AL
        JMP  main_loop
turn:   MOV  AL, 3          ; turn right
        JMP  send
advance: MOV AL, 1          ; forward
send:   PUSH AX             ; keep command
        CALL wait_ready
        POP  AX
        OUT  9, AL
        JMP  main_loop

wait_ready PROC
b1:     IN   AL, 11
        TEST AL, 010b       ; busy?
        JNZ  b1
        RET
wait_ready ENDP

wait_data PROC
v1:     IN   AL, 11
        TEST AL, 001b       ; new data?
        JZ   v1
        RET
wait_data ENDP
```

The program rests on two wait routines. `wait_ready` tests bit 1 of the status byte: as long as the busy bit is 1 (ZF = 0), `JNZ` jumps back. When the robot finishes the previous command, the routine returns. `wait_data` tests bit 0 and returns on the opposite condition: it waits with `JZ` as long as the new-data bit is 0 (ZF = 1). The two routines have the same structure; only the mask and the jump condition differ.

Before every command the program waits for the busy bit to clear. After the examine (4) command it waits for the data-ready bit; only then is port 10 read. If the value read is FFh, `JE turn` goes to the turn-right code. If it is 8, `JNE` does not jump and the lamp is lit. For any other value `JNE advance` goes to the go-forward code. The "turn right" and "go forward" paths join at the `send` label: the command in AL is written to the port and the loop starts over.

The waiting jobs are routines called with `CALL`. On each call the return address is pushed on the stack (SP decreases by 2) and `RET` takes it back. Instead of writing the same wait loop three times in the program, a single routine called from three places makes the code shorter and reduces the chance of errors.

10.2.8.3 Preserving AL

There is a subtlety on the `send` path. AL carries the robot's command (1 or 3). But before the command is sent `wait_ready` is called, and that routine overwrites AL with the status byte through `IN AL, 11`. When the routine returns, AL is no longer the command but the last status read. For this reason the command is saved on the stack with `PUSH AX` before the call and restored with `POP AX` after it. Because `PUSH` does not accept an 8-bit register, AX is pushed instead of AL.

Table 10.2.25 traces one round with a wall in front of the robot. It is assumed that port 11 returns 00h (ready), 01h (data available), 03h (busy and data available), and 01h in turn, and that port 10 returns FFh (wall).

Step	Instruction	AL	SP	ZF	Port 9	Explanation
1	`main_loop: CALL wait_ready`	00h	FFFCh	0	—	return address 0103h onto the stack
2	`b1: IN AL, 11`	00h	FFFCh	0	—	status = 00h
3	`TEST AL, 010b`	00h	FFFCh	1	—	bit 1 = 0: not busy, ZF = 1
4	`JNZ b1`	00h	FFFCh	1	—	does not jump
5	`RET`	00h	FFFEh	1	—	return: SP = FFFEh
6	`MOV AL, 4`	04h	FFFEh	1	—	command 4: examine
7	`OUT 9, AL`	04h	FFFEh	1	04h	the robot examines what is ahead
8	`CALL wait_data`	04h	FFFCh	1	04h	waits for data
9	`v1: IN AL, 11`	01h	FFFCh	1	04h	status = 01h: new data available
10	`TEST AL, 001b`	01h	FFFCh	0	04h	bit 0 = 1: ZF = 0
11	`JZ v1`	01h	FFFCh	0	04h	does not jump
12	`RET`	01h	FFFEh	0	04h	return
13	`IN AL, 10`	FFh	FFFEh	0	04h	data = FFh: wall
14	`CMP AL, 0FFh`	FFh	FFFEh	1	04h	equal: ZF = 1

Step	Instruction	AL	SP	ZF	Port 9	Explanation
15	`JE turn`	FFh	FFFEh	1	04h	jumps to turn
16	`turn: MOV AL, 3`	03h	FFFEh	1	04h	command 3: turn right
17	`JMP send`	03h	FFFEh	1	04h	send
18	`send: PUSH AX`	03h	FFFCh	1	04h	AX = 0003h onto the stack: SP = FFFCh
19	`CALL wait_ready`	03h	FFFAh	1	04h	return address onto the stack: SP = FFFAh
20	`b1: IN AL, 11`	03h	FFFAh	1	04h	status = 03h: AL overwritten
21	`TEST AL, 010b`	03h	FFFAh	0	04h	bit 1 = 1: busy, ZF = 0
22	`JNZ b1`	03h	FFFAh	0	04h	read again
23	`b1: IN AL, 11`	01h	FFFAh	0	04h	status = 01h
24	`TEST AL, 010b`	01h	FFFAh	1	04h	bit 1 = 0: ZF = 1
25	`JNZ b1`	01h	FFFAh	1	04h	does not jump
26	`RET`	01h	FFFCh	1	04h	return: SP = FFFCh, AL = 01h
27	`POP AX`	03h	FFFEh	1	04h	AL = 03h is back
28	`OUT 9, AL`	03h	FFFEh	1	03h	"turn right" goes to the robot
29	`JMP main_loop`	03h	FFFEh	1	03h	the loop starts over

Table 10.2.25: Trace of the program with a wall in front of the robot; port 11 returns 00h, 01h, 03h, 01h in turn, and port 10 returns FFh.

At step 18 AX = 0003h is pushed and SP drops to FFFCh. The `CALL` at step 19 also pushes the return address: SP = FFFAh. Inside the routine AL becomes 03h (busy) and then 01h. The `RET` at step 26 takes the return address; AL holds 01h. The `POP AX` at step 27 brings back 03h, and at step 28 the right value, 3, goes to the port. The stack is balanced too: SP returns to where it started, FFFEh.

Had `PUSH AX` and `POP AX` been deleted, 01h would go to the port in the same scenario: "go forward". The robot would try to walk into the wall and the error bit (bit 2) would be set. Since the program does not check this error, on the next round it examines again, sees the wall again, and repeats the same error: the robot gets stuck in front of the wall.

Common mistake

Relying, after a routine call, on the old value of a register the routine uses. `wait_ready` changes AL, and there is no way to tell this from the routine's name. Either the routine saves the registers it changes itself (`PUSH` on entry, `POP` on exit), or the caller puts the value it wants to protect on the stack. The robot program uses the second way.

The program has two more shortcomings. First, the error bit is never read. A "forward" command into a wall sets bit 2; a robust program should read the status byte after every command and test this bit. Second, there is the cost of polling: the processor is completely busy while waiting for each movement of the robot. In a real robot controller this waiting is done with interrupts, and in between the processor handles sensor data.

10.2.9 The simple I/O test device

10.2.9.1 Ports 110 and 112

The test device (*simple io test*) is a window made for trying out the `IN` and `OUT` instructions. Port 110 holds a byte and port 112 holds a word. The values typed into the window become input to the program; for a hexadecimal value an `h` is appended (for example `A7h`). The values the program writes to the ports also appear in the window. The input ports are refreshed every 100 ms. A new value typed into the window is therefore read within 100 ms at the latest, even while the program is running.

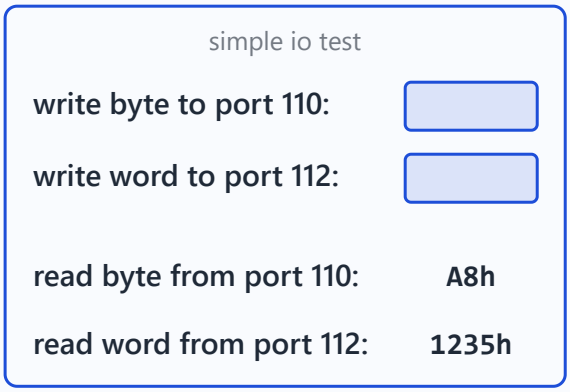


Figure 10.2.18: The test device window. The program has written A8h to port 110 and 1235h to port 112; the window shows these values.

The window is very valuable as a debugging tool. While writing a device program you can see here whether the value going to the port is correct; input values can also be supplied by hand without a device. In the example in the slides, A7h is entered in the window for port 110 and 1234h for port 112. The program adds 1 to the first and writes A8h back, and adds 1 to the second and writes 1235h back.

10.2.9.2 Code trace: byte and word I/O

```
org 100h
in  al, 110      ; read byte
inc al
out 110, al      ; write byte
in  ax, 112      ; read word
add ax, 1
out 112, ax      ; write word
ret
```

When the same program is run with boundary values, the behavior of the flags comes out. Let FFh be entered in the window for port 110 and 7FFFh for port 112.

Step	Instruction	AX	ZF	SF	OF	CF	I/O	Explanation
1	`in al, 110`	00FFh	0	0	0	0	110 → FFh	byte read: AL = FFh
2	`inc al`	0000h	1	0	0	0		FFh + 1 = 00h: ZF = 1; INC does not touch CF
3	`out 110, al`	0000h	1	0	0	0	110 ← 00h	port 110 = 00h in the window
4	`in ax, 112`	7FFFh	1	0	0	0	112 → 7FFFh	word read: AL ← port 112, AH ← port 113
5	`add ax, 1`	8000h	0	1	1	0		7FFFh + 1 = 8000h: the sum of two positives is negative, OF = 1
6	`out 112, ax`	8000h	0	1	1	0	112 ← 8000h	port 112 = 8000h in the window
7	`ret`	8000h	0	1	1	0		0000h popped from the stack → IP
8	`INT 20h (PSP:0000)`	8000h	0	1	1	0		the program ends

Table 10.2.26: Trace of the test device program; port 110 = FFh and port 112 = 7FFFh are entered in the window.

The first part is an 8-bit overflow. The 9th bit of the result FFh + 1 = 100h does not fit in AL, and AL = 00h. ZF = 1. The `INC` instruction does not change CF; CF stays at its previous value, 0. Had the same operation been done with `ADD AL, 1`, CF would be 1. 00h is written to the port.

The second part is a signed overflow. `IN AX, 112` takes AL from port 112 and AH from port 113: AX = 7FFFh, +32,767 as signed. The result of `ADD AX, 1` is 8000h. The sum of two positive numbers appears as a negative number: OF = 1, SF = 1. There is no overflow in the unsigned sum (32,767 + 1 = 32,768 < 65,536), so CF = 0. The window shows port 112 = 8000h. Had the same word been written to the LED display, it would show −32,768.

10.2.10 The common pattern

The seven devices look very different from one another: one lights lamps, one turns a motor, one reads a temperature. Their programs, however, are variations of the same skeleton (Table 10.2.27).

Device	Port	Basic idea
Traffic lights	4	Each bit is a lamp; state table and INT 15h/86h wait.
Stepper motor	7	Bits 0–2 magnets, bit 7 ready; half step 11.25°, full step 22.5°.
LED display	199	Word, signed decimal display.
Thermometer	125, 127	Read, signed comparison, heater on/off.
Printer	130	INT 21h/05h; control characters LF, CR, HT, FF.

Device	Port	Basic idea
Robot	9, 10, 11	Command, data, status; poll the busy and data bits.
Test device	110, 112	Byte and word I/O experiment.
8255 PPI (real)	310h–313h	The control word (for example 82h) sets the port directions; port > 255 → DX.

Table 10.2.27: Devices, their ports, and their basic ideas.

The common pattern has four steps: read the status, test the relevant bit, write or read the data if ready, go back to the loop. In the traffic lights the "status" is time, and waiting replaces testing. In the thermostat the status is the temperature read itself. In the stepper motor and the robot the status is bits in a separate port. In every case `IN` and `OUT` work only with AL/AX, and DX is used if the port is above 255. Most devices are driven in an endless loop.

10.2.11 Exercises and solutions

Exercise 1. What is the port 4 value that lights only the yellow lamps in all directions?

The yellow lamps are at bits 3k + 1: bits 1, 4, 7, and 10. The word is 2 + 16 + 128 + 1024 = 1170 = 0000 0100 1001 0010b = **0492h**.

Exercise 2. What must CX and DX be for a 2.5-second wait with INT 15h/86h?

2.5 s = 2,500,000 µs = 002625A0h. The high word is CX = **0026h** and the low word DX = **25A0h**. In the source code write `mov cx, 26h` and `mov dx, 25A0h`.

Exercise 3. What hexadecimal value is written to AX to show −1 on the LED display?

The 16-bit two's complement of −1 is 10000h − 1 = **FFFFh**. When `mov ax, -1` is written, the assembler produces the same value.

Exercise 4. How is the thermostat adapted to the range 20–25 °C? What would happen at −10 °C if `JB` were used instead of `JL`?

Only two constants change: `cmp al, 20` · `jl low` · `cmp al, 25` · `jle ok` · `jg high`. −10 °C arrives from the port as F6h. In a signed comparison −10 < 20 and the heater is switched on. `JB`, however, counts F6h as 246; 246 < 20 is false, so it does not jump and the heater is not switched on. If the unsigned `JA` is used as well, the heater is switched off.

Exercise 5. Which lines of the stepper motor program must change to turn the motor only clockwise and in full steps?

BX must point to the full-step table at the start: `mov bx, offset datcw_fs`. The part that changes the table (from `inc cx` to the last `jmp next_step`) is deleted, and right after `mov si, 0` comes a plain `jmp next_step`. Since CX is no longer used, the line `mov cx, 0` is unnecessary too.

Exercise 6. How is the shortest program that prints "Hello" on the printer and ejects the page written?

One could write `MOV DL, 'H'` · `MOV AH, 5` · `INT 21h` for every character; with a string and a loop it is shorter:

```
org 100h
mov  si, offset msg
mov  cx, 6          ; "Hello" + FF
again: mov dl, [si]
mov  ah, 5
int  21h
inc  si
loop again
ret
msg  db "Hello", 12
```

The 12 (FF) added to the end of the string ejects the page, so there is no need for a separate form feed call.

Exercise 7. On an 8 MHz 8086, what is N for a delay of about 1 ms with `mov cx, N` / `loop`?

1 ms is 8,000 clock cycles. The equation 17N − 8 = 8,000 gives N ≈ 471.06. With N = **471**, 17 · 471 − 8 = 7,999 cycles, that is 0.999875 ms.

Exercise 8. What is the control word of the 8255 for A input, B output, and all of C input (mode 0)?

D7 = 1, D6 D5 = 00, D4 = 1 (A input), D3 = 1 (C upper input), D2 = 0, D1 = 0 (B output), D0 = 1 (C lower input): 1 00 1 1 0 0 1b = **99h**.

Exercise 9. What are the codes of the digit '5' and the letter 'H' on a common-anode display (in the order dp g f e d c b a)?

For '5' the common-cathode code is 6Dh = 0110 1101b; its inverse is 1001 0010b = **92h**. For 'H', b, c, e, f, and g light: 0111 0110b = 76h; the common-anode code is 1000 1001b = **89h**.

10.2.12 Summary

- A peripheral shows itself to the processor through its registers; the address of these registers is called an I/O port. The port numbers are determined not by the processor but by the system's address decoder.
- On the 8086 a port is accessed only with `IN` and `OUT`. Data passes only through AL (byte) or AX (word). A port from 0 to 255 is given in the instruction as a constant; a larger one is given in DX. The word form uses ports n and n + 1 together.

- Port-mapped I/O uses a separate 64 K address space (M/$\overline{IO}$ = 0); in memory-mapped I/O the device is at memory addresses and is reached with every memory instruction (for example the B800h text screen).
- There are three ways of coordinating with a device: polling (the processor reads the status bit in a loop), interrupts (the device alerts the processor), and DMA (a controller moves the data without the processor).
- The 8255 PPI has three 8-bit ports and a control register. A control word with D7 = 1 sets the mode and direction of the ports (1 = input); a bit set/reset word with D7 = 0 changes a single bit of C.
- In the traffic lights each lamp is a bit of the word at port 4 (pole k: red 3k, yellow 3k + 1, green 3k + 2). The states are kept in a table and written to the port in turn using SI.
- INT 15h/86h waits CX:DX microseconds. The delay of `MOV CX, N` / `LOOP` is 17N − 8 clock cycles and depends on the clock speed; N = 281 for 1 ms at 4.77 MHz.
- In the emu8086 stepper motor, bits 0–2 carry the magnets and bit 7 the ready information. The program polls bit 7 before every byte and reads the table with `[BX][SI]`.
- A real unipolar motor is driven through a driver such as the ULN2003; there are single-phase, two-phase, and half-step sequences. The motor is open loop and its speed is set by the step interval.
- The LED display shows the word at port 199 as a signed decimal. On a 7-segment display the code is fetched from a table with `XLAT`; for a common anode the code is inverted. In multiplexed scanning all digits are blanked before switching digits.
- Because the thermostat can read negative temperatures it uses signed jumps (`JL`, `JLE`, `JG`); unsigned jumps take negative values for large positive numbers.
- Characters are sent to the printer with INT 21h/05h; LF, CR, HT, and FF control the paper and head. The length is computed at assembly time with `offset msg_end - offset msg`.
- The robot works with command, data, and status ports; because the wait routines overwrite AL, the command is saved on the stack before the call.

10.2.13 Key terms

peripheral I/O port port-mapped / memory-mapped I/O address decoder polling handshaking interrupt direct memory access (DMA) programmable peripheral interface (8255 PPI) control word software delay clock cycle stepper motor seven-segment display segment (display) control character emulator

10.2.14 Test Yourself

1. What is the port 4 word that lights red + yellow on the lower-right and upper-left poles and only red on the other two poles?

   Answer

   Lower right: bits 0 and 1; upper right: bit 3; upper left: bits 6 and 7; lower left: bit 9. The total is 1 + 2 + 8 + 64 + 128 + 512 = 715 = 0000 0010 1100 1011b = **02CBh**.

2. What must CX and DX be to wait 3 seconds with INT 15h/86h?

   Answer

   3,000,000 µs = 002DC6C0h: CX = **002Dh**, DX = **C6C0h**. Because the DX value starts with a letter, it is written `0C6C0h` in the source.

3. Why does `IN AL, 300h` not assemble? How is the same job written?

   Answer

   In the constant-port form the port number is written in a single byte, and 300h does not fit in this byte. The correct form is `MOV DX, 300h` · `IN AL, DX`.

4. What happens to the ports if 9Bh is written to the control register of the 8255? What is the bit set/reset word that sets PC4 to 1?

   Answer

   9Bh = 1 00 1 1 0 1 1b: mode 0, A input, C upper input, B input, C lower input; all ports are inputs. For PC4 = 1, D3–D1 = 100 and D0 = 1: 0000 1001b = **09h**.

5. On a 4.77 MHz 8086 how long does `mov cx, 500` / `wait: loop wait` take?

   Answer

   17 · 500 − 8 = 8,492 clock cycles; 8,492 ÷ 4.77 ≈ 1,780 µs ≈ **1.78 ms**.

6. What are the common-cathode and common-anode segment codes of the letter 'E'?

   Answer

   For 'E', a, d, e, f, and g light: bits 0, 3, 4, 5, 6. Common cathode 0111 1001b = **79h**, common anode 1000 0110b = **86h**.

7. In the thermostat program, if the temperature read is exactly 80 °C, which jump is taken and what happens to the heater?

   Answer

   The result of `CMP AL, 60` is positive and `JL` does not jump. The result of `CMP AL, 80` is zero: ZF = 1 and `JLE ok` jumps. The heater is not touched; it stays in its previous state.

8. The robot's status port returns 06h. What do the `wait_ready` and `wait_data` routines do?

   Answer

   06h = 0000 0110b: bit 1 (busy) and bit 2 (error) are 1, bit 0 (data) is 0. `wait_ready` keeps waiting because the busy bit is 1; `wait_data` also waits because the data bit is 0. Neither routine looks at the error bit.

9. On the test device with 7Fh entered for port 110, `IN AL, 110` · `INC AL` runs. What are AL and OF, SF, ZF?

   Answer

   AL = **80h**. One more than +127 appears as −128 in 8 bits: OF = 1, SF = 1, ZF = 0. `INC` does not touch CF.

10. If the status register of a memory-mapped device were at B800:0010, which instructions would be enough to wait for bit 7?

    Answer

    After setting ES = B800h: `wait: TEST BYTE PTR ES:[0010h], 80h` · `JZ wait`. With a memory-mapped device `TEST` is applied directly to the memory operand; there is no need to read into AL.

10.2.15 Further reading

- Barry B. Brey, *The Intel Microprocessors*, 8th edition: the basic I/O interface chapter covers IN/OUT, address decoding, the 8255, and driving 7-segment displays, with circuit diagrams.
- Intel, *8255A Programmable Peripheral Interface* datasheet: timing diagrams for modes 0, 1, and 2 and the original definition of the control word.
- Texas Instruments, *ULN2003A* datasheet: the current limits of the Darlington channels and how to wire the COM pin with inductive loads.
- emu8086 help documentation, *I/O ports and hardware interrupts* and *virtual devices*: the port layout and sample programs of the virtual devices.

10.2.16 References

1. emu8086 help documentation: *I/O ports and hardware interrupts*, *Virtual devices*, and the sample programs (traffic_lights.asm, stepper_motor.asm, led_display_test.asm, thermometer.asm, printer.asm, robot.asm).
2. Intel, *The 8086 Family User's Manual*, 1979 (IN/OUT instructions, the I/O address space, and instruction timings).
3. Intel, *8255A Programmable Peripheral Interface* datasheet.
4. Texas Instruments, *ULN2003A* datasheet.
5. M. Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
6. R. Brown, *Ralf Brown's Interrupt List* (INT 15h/86h, INT 21h/05h, INT 16h/00h).
7. B. B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, 2009, Chapter 11 (Basic I/O interface).

Instruction Encoding and String Instructions

Every line written in assembly language ends up as a few bytes. The processor sees only these bytes: it reads them from memory, puts them in the prefetch queue, decodes them, and executes them. This note first divides the 8086 instruction set into functional classes. It then shows, bit by bit, how an instruction is turned into machine code: prefix, opcode, the D and W bits, the ModR/M byte, displacement, and immediate value. The reverse of the same rules, recovering an instruction from bytes in memory, is worked step by step as well, and an example shows how decoding that starts at a wrong address turns the same bytes into completely different instructions. Next come the data-transfer instructions XLAT, LEA, LDS, and LES. The second half of the note is devoted to the string instructions: how MOVS, CMPS, SCAS, LODS, and STOS work together with SI, DI, CX, and the direction flag. The REP, REPE, and REPNE prefixes, overlapping blocks, and timings follow. The last sections cover the encoding of JMP, the processor-control instructions, and reference tables showing which instruction changes which flag. Every piece of code is traced together with the registers, flags, and memory.

LEARNING OUTCOMES

Recognizing instruction classes
Divide the roughly one hundred instructions of the 8086 into the classes data transfer, arithmetic, logic and shift, string, control transfer, and processor control; explain the typical use of each class.

Encoding instructions
Find the opcode, the D and W bits, the ModR/M byte, the displacement, and the immediate value of an assembly instruction and write out its machine code byte by byte; recognize the short special forms and the prefixes.

Decoding machine code
Split a byte sequence in memory into instructions from the correct starting address; show why decoding that starts at a wrong address produces different instructions.

Using address-loading instructions
Translate through a table with XLAT; get an address with LEA and the contents with MOV; load a far pointer from memory into the segment and offset registers with LDS and LES.

Tracing string instructions
Use MOVS, CMPS, SCAS, LODS, and STOS with the REP, REPE, and REPNE prefixes; show how SI, DI, CX, ZF, and DF change at every step; choose the correct direction for overlapping blocks.

Reading the reference tables
Compute the encoding and the relative displacement of the JMP forms; explain how the processor-control instructions relate to processor pins; read from the flag-effects table which flags an instruction changes.

11.1 Overview of the instruction set

11.1.1 What is an instruction set?

The list of all the instructions a processor can execute is called its **instruction set**. The instruction set is the contract between the processor and the person who programs it. The programmer knows which instructions exist, which operands each accepts, and what each does to the registers, memory, and flags. There is no need to know how the circuits inside the processor are built. Two different processors that execute the same instruction set run the same program; that is why the 8086, the 8088, and the later x86 processors can run programs written for the 8086.

Depending on how they are counted, the 8086 instruction set has about a hundred instructions. The programmer writes them with **mnemonics** such as `MOV`, `ADD`, and `JMP`. A mnemonic is a name that is easy for a person to remember; the processor does not recognize it. The assembler converts every line to the numeric form the processor understands, the **machine code**. The same mnemonic can turn into many different machine codes depending on its operands. For example `MOV` uses separate codes for register to register, memory to register, immediate to memory, and moves to a segment register. So while the 8086 has about a hundred mnemonics, the number of distinct opcodes runs to a few hundred.

The instructions of the 8086 are of variable length. The shortest instruction is a single byte (`NOP` = 90h, `INC AX` = 40h). The longest reaches six bytes without prefixes (`MOV WORD PTR [1234h], 5678h` = C7 06 34 12 78 56). One or more prefix bytes can also be added in front. Variable length uses memory efficiently: frequently used instructions are kept short and rarely used ones long. The price is that the processor can tell where an instruction ends only by decoding its bytes in order.

11.1.2 Six functional classes

The easiest way to memorize the instructions is to group them by what they do. Table 11.1 divides the 8086 instructions into six classes. The classes are not just a memory aid: instructions in the same class affect the flags in similar ways and are often encoded in similar ways. Data-transfer instructions do not touch the flags; arithmetic and logic instructions write the flags that describe the result; control-transfer instructions read the flags but do not change them.

Class	What it does	Instructions
1. Data transfer	copying data between registers, memory, the stack, and I/O	`MOV PUSH POP XCHG IN OUT XLAT LEA LDS LES LAHF SAHF PUSHF POPF`
2. Arithmetic	the four operations, BCD adjustment, sign extension	`ADD ADC INC AAA DAA SUB SBB DEC NEG CMP AAS DAS MUL IMUL AAM DIV IDIV AAD CBW CWD`
3. Logic, shift, rotate	bit-level operations	`NOT AND OR XOR TEST SHL SAL SHR SAR ROL ROR RCL RCR`
4. String processing	scanning and copying memory blocks with SI and DI	`MOVSB/W CMPSB/W SCASB/W LODSB/W STOSB/W` ; prefixes `REP REPE REPZ REPNE REPNZ`
5. Control transfer	branches, loops, procedures, and interrupts	`JMP` , `Jcc` (`JE JNE JA JB JG JL` …), `LOOP LOOPE LOOPNE JCXZ CALL RET RETF INT INTO IRET`
6. Processor control	setting and clearing flags, managing the processor state	`CLC STC CMC CLD STD CLI STI HLT WAIT ESC LOCK NOP`

Table 11.1: The six functional classes of the 8086 instruction set and the instructions in each.

This note covers in detail the address-loading instructions of the first class (XLAT, LEA, LDS, LES), all of the fourth class, the encoding of JMP from the fifth class, and all of the sixth class. The flag effects of the arithmetic and logic instructions are given together in the reference tables at the end of the note. The encoding rules, on the other hand, are common to all the classes.

11.1.3 How do instruction bytes reach the processor?

To understand why the encoding rules are what they are, we need to look at the path an instruction follows inside the processor. The inside of the 8086 is divided into two units. The **bus interface unit** (BIU) manages the external buses. It reads instruction bytes from the address pointed to by CS:IP and puts them in the six-byte **prefetch queue**. As soon as two bytes are free in the queue it starts a new read. Because the data bus of the 8086 is 16 bits wide, a read that starts at an even address brings two bytes in a single bus cycle.

The **execution unit** (EU) takes bytes from the head of the queue one at a time. If the first byte is a prefix, it records it in a latch and moves on to the next byte. The first real opcode byte determines the type of the instruction and how many bytes follow. If the opcode needs a ModR/M byte, the EU takes a second byte from the queue. The MOD field of ModR/M says whether the displacement is 0, 1, or 2 bytes. The W bit in the opcode and the type of the instruction also determine the length of the immediate value. Once the EU has taken these bytes it knows where the instruction ends; the next byte is the first byte of a new instruction.

This sequential decoding explains two important properties of machine code. First, the length of an instruction follows from its first byte or two; that is why the important information (opcode, D, W, MOD) always comes first. Second, the processor does not distinguish code from data. Whatever byte CS:IP points to is taken as an opcode. If control jumps to a wrong address, the same bytes are executed as completely different instructions. This subject is worked out with an example in the section on decoding machine code.

Definition 11.1

Machine code is the byte sequence of an instruction as stored in memory and executed directly by the processor. The **instruction format** defines the fields of this sequence and their order. **Encoding** is producing machine code from an assembly instruction; **decoding** is recovering the assembly instruction from machine code.

11.2 Instruction format

11.2.1 Fields

The general instruction format of the 8086 is shown in Figure 11.1. An instruction consists of at most five fields, and these fields always appear in the same order:

1. **Prefixes** (optional): a segment prefix (26h, 2Eh, 36h, 3Eh), a repeat prefix (F3h, F2h), or LOCK (F0h). It changes the behavior of the instruction.
2. **Opcode byte** (always present): says what the instruction does. In most two-operand instructions 6 bits of this byte carry the operation and the last two bits carry the D and W bits.
3. **ModR/M byte** (if one of the operands can be a register or memory): selects the two operands with the MOD, REG, and R/M fields.
4. **Displacement** (0, 1, or 2 bytes): a constant added to the memory address; the low byte comes first.
5. **Immediate** (0, 1, or 2 bytes): a constant operand given inside the instruction; the low byte comes first.

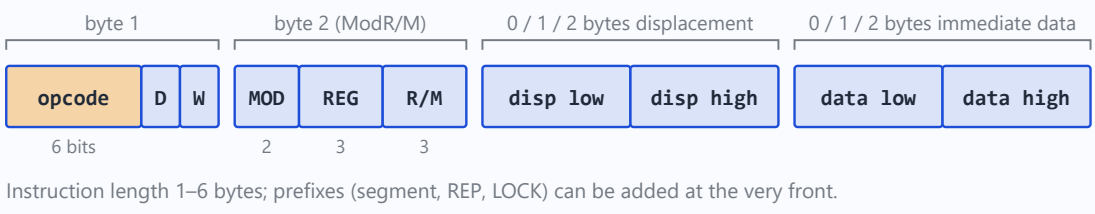


Figure 11.1: The 8086 instruction format. The first byte carries the 6-bit opcode and the D and W bits, and the second byte (ModR/M) carries the MOD, REG, and R/M fields. Then come 0–2 bytes of displacement and

0–2 bytes of immediate value.

Displacement and immediate bytes are always written in **little-endian** order: the low byte of a 16-bit number goes to the lower address and the high byte to the next address. `MOV AX, 1234h` is therefore encoded as B8 **34 12**. In an assembly line numbers are written in the order people read them; in memory and in machine-code listings they appear reversed. This reversal comes from the layout of the 16-bit data bus of the 8086. A byte at an even address travels on the low half of the data bus (D0–D7) and a byte at an odd address on the high half (D8–D15). When the low byte of a word sits at an even address it connects directly to the low half of the register.

11.2.2 The D and W bits

In most two-operand instructions one operand is a register and the other a register or memory. The 8086 describes the two operands with a single ModR/M byte: the REG field selects the register, and the R/M field selects a register or a memory address. Two questions remain: is the operation on a byte or a word, and is the result written to REG or to R/M? The last two bits of the opcode byte answer these two questions.

Definition 11.2

The **W bit** (*word*, bit 0 of the opcode) selects the operand size: W = 0 means a byte (8-bit) operation, W = 1 a word (16-bit) operation. The same three bits in the REG field point to one of AL…BH when W = 0 and to one of AX…DI when W = 1.

The **D bit** (*direction*, bit 1 of the opcode) selects the direction: if D = 1 the REG field is the *destination* and the R/M field is the source; if D = 0 the REG field is the *source* and the R/M field is the destination.

The D bit has an interesting consequence in two-register instructions. In `MOV AX, BX` both operands are registers; which goes into the REG field and which into the R/M field is free. If D = 1 is chosen, the destination AX goes into the REG field and the source BX into the R/M field: 100010 **1 1** = 8Bh, ModR/M = 11 000 011 = C3h. If D = 0 is chosen, the source BX goes into the REG field and the destination AX into the R/M field: 100010 **0 1** = 89h, ModR/M = 11 011 000 = D8h. Table 11.2 puts the two encodings side by side, together with `ADD AX, BX` in the same situation.

Instruction	D	Opcode	ModR/M (MOD REG R/M)	Machine code
`MOV AX, BX`	1	100010 1 1 = 8Bh	11 000 (AX) 011 (BX) = C3h	`8B C3`
`MOV AX, BX`	0	100010 0 1 = 89h	11 011 (BX) 000 (AX) = D8h	`89 D8`
`ADD AX, BX`	1	000000 1 1 = 03h	11 000 (AX) 011 (BX) = C3h	`03 C3`
`ADD AX, BX`	0	000000 0 1 = 01h	11 011 (BX) 000 (AX) = D8h	`01 D8`

Table 11.2: The two valid encodings of two-register instructions. Both do the same job; the difference is only in the D bit and in the swapped REG and R/M fields.

Both encodings are valid and the processor executes both in the same way: the result is written to AX and BX is unchanged. Which form is produced depends on the assembler: MASM usually produces the D = 1 form (`8B C3` , `03 C3`), NASM the D = 0 form (`89 D8` , `01 D8`). Both do the same job and have the same length. For this reason the same instruction can appear with two different codes in course material: that 01 D8 is written for `ADD AX, BX` when the 8086 architecture is explained and 03 C3 in assembler output is not a contradiction. When one of the operands is memory there is no choice. In `MOV [BX], AL` memory can be expressed only in the R/M field, so AL goes into the REG field and D = 0 is forced (88 07). In `MOV AL, [BX]` D = 1 (8A 07).

Common mistake

Reading the D bit as "the first operand" or "the left operand". The meaning of the D bit is tied to the REG field: D = 1 says that the register in REG is the destination. In the instruction line the destination is always on the left; but in machine code whether the destination is REG or R/M depends on D.

11.2.3 The REG field

The middle three bits of the ModR/M byte are the REG field. Three bits are enough to select eight registers. Which eight registers are meant depends on the W bit. Table 11.3 gives these codes. The same table also contains the two-bit codes of the segment registers. Segment registers appear in only two instruction families (MOV starting with 8Ch and 8Eh, and PUSH and POP) and are written in the REG field with a leading 0.

Kod	W = 0 (byte)	W = 1 (word)	Segment register
`000`	AL	AX	ES (00)
`001`	CL	CX	CS (01)
`010`	DL	DX	SS (10)
`011`	BL	BX	DS (11)
`100`	AH	SP	–
`101`	CH	BP	–
`110`	DH	SI	–
`111`	BH	DI	–

Table 11.3: Register codes of the REG field (and of the R/M field when MOD = 11). The codes of the segment registers are two bits.

The order has a logic. The first four codes give AX, CX, DX, BX and their low halves. The last four codes select SP, BP, SI, DI in a word operation and the high halves of the first four registers (AH, CH, DH, BH) in a byte operation. So code 100 means AH when W = 0 and SP when W = 1. Overlooking the W bit in decoding leads to mistaking AH for SP. These codes are not used only inside ModR/M. In the single-

byte short forms too, the last three bits of the opcode carry the same register code: `INC` 40h + r, `PUSH` 50h + r, `MOV r8, imm8` B0h + r, `MOV r16, imm16` B8h + r. `PUSH BX` is therefore 50h + 3 = 53h, and `MOV SI, 1000h` is B8h + 6 = BEh followed by 00 10.

11.2.4 The MOD field

The two leftmost bits of the ModR/M byte are the MOD field. MOD says how the R/M field is to be interpreted and how many bytes of displacement the instruction contains. Table 11.4 gives the four cases.

MOD	Meaning	Displacement
`00`	memory operand, no displacement; the only exception is R/M = 110: 16-bit direct address	0 bytes (2 bytes if R/M = 110)
`01`	memory operand + 8-bit displacement; d8 is sign-extended to 16 bits	1 byte
`10`	memory operand + 16-bit displacement	2 bytes
`11`	register operand: the R/M field is a register code as well	none

Table 11.4: The four values of the MOD field.

The sign extension in the MOD = 01 case has a practical benefit. If the displacement is between −128 and +127 the assembler uses a single byte. In `MOV AX, [BX+DI-2]`, −2 = FEh fits in one byte and the instruction takes three bytes as 8B 41 FE. The processor extends FEh to FFFEh and adds it to BX + DI; in 16-bit addition, adding FFFEh gives the same result as subtracting 2. If the displacement is outside this range MOD = 10 is chosen and two bytes are needed: `MOV CL, [BX+SI+1234h]` = 8A 88 34 12.

11.2.5 The R/M field and the effective address

If MOD is not 11, the R/M field selects an addressing form. The processor computes the **effective address** (EA) according to this form. The effective address is the offset of the operand within the segment. The physical address is found by adding EA to the segment register multiplied by 16. Table 11.5 gives the eight R/M codes with the three MOD values and the default segment.

R/M	MOD = 00	MOD = 01	MOD = 10	Default segment
`000`	[BX+SI]	[BX+SI+d8]	[BX+SI+d16]	DS
`001`	[BX+DI]	[BX+DI+d8]	[BX+DI+d16]	DS
`010`	[BP+SI]	[BP+SI+d8]	[BP+SI+d16]	SS
`011`	[BP+DI]	[BP+DI+d8]	[BP+DI+d16]	SS
`100`	[SI]	[SI+d8]	[SI+d16]	DS
`101`	[DI]	[DI+d8]	[DI+d16]	DS
`110`	**[d16] direct**	[BP+d8]	[BP+d16]	DS / SS

R/M	MOD = 00	MOD = 01	MOD = 10	Default segment
`111`	[BX]	[BX+d8]	[BX+d16]	DS

Table 11.5: The R/M field and the effective address. In all forms that contain BP the default segment is SS; with MOD = 00, R/M = 110 it is a direct address and the segment is DS.

Three rules follow from the table. First, only BX, BP, SI, and DI can be address registers; a base register (BX or BP) and an index register (SI or DI) can be used together. There is no R/M code for `[AX]`, `[CX]`, `[SI+DI]`, or `[BX+BP]`, and the assembler treats them as errors. Second, every form that contains BP uses the stack segment (SS). BP is designed for reaching parameters and local variables in a stack frame. Third, the combination MOD = 00 and R/M = 110 is reserved not for [BP] but for the 16-bit direct address. When you write just `[BP]`, the assembler therefore uses MOD = 01 and d8 = 0: `MOV AL, [BP]` is three bytes (8A 46 00), whereas `MOV AL, [BX]` is two bytes (8A 07).

Computing the effective address takes time, and this time is added to the time of the instruction. On the 8086 the microcode of the execution unit computes the EA step by step with the ALU; the time therefore varies with the addressing form. Table 11.6 gives the EA times from the Intel 8086 manual. The total time of an instruction with a memory operand is written as "base time + EA". For example `MOV reg, mem` takes 8 + EA clock cycles; for `MOV CL, [BX+SI+12h]` this is 8 + 11 = 19 cycles. On a 5 MHz 8086 one clock cycle is 200 ns, so the instruction takes about 3.8 µs.

Addressing form	Example	EA time
base or index only	`[BX]`, `[SI]`, `[DI]`	5
displacement only (direct)	`[1234h]`, `var1`	6
base + index	`[BP+DI]`, `[BX+SI]` / `[BP+SI]`, `[BX+DI]`	7 / 8
displacement + base or index	`[BX+4]`, `[BP+2]`, `array[SI]`	9
displacement + base + index	`[BP+DI+d]`, `[BX+SI+d]` / `[BP+SI+d]`, `[BX+DI+d]`	11 / 12

Table 11.6: Effective address calculation times on the 8086 (clock cycles). If there is a segment prefix, 2 cycles are added.

One more number matters for every word read from or written to memory. If the word is at an even address, the BIU moves it in a single bus cycle (4 clock cycles). If it is at an odd address, two separate byte cycles are needed and 4 cycles are added to the time. The 8086 selects the low byte lane with A0 = 0 for an even address and the high byte lane with BH$\bar{E}$ = 0 for an odd address. Because the two bytes of a word at an odd address fall in different lanes and at different word addresses, two cycles are unavoidable. This is why the `EVEN` directive of the assembler is used to align word arrays to even addresses.

11.2.6 Short special forms

The general format can encode every instruction but does not always give the shortest code. The designers of the 8086 set aside separate opcodes that need no ModR/M byte for the most frequently used operations. These short forms save one or two bytes and are sometimes faster. Table 11.7

collects the most important ones.

Form	Opcode	Example	Kod	General form
INC / DEC r16	40h+r / 48h+r	`INC AX`	`40`	`FF C0` (2 bytes)
PUSH / POP r16	50h+r / 58h+r	`PUSH BX`	`53`	`FF F3` (2 bytes)
XCHG AX, r16	90h+r	`XCHG AX, BX`	`93`	`87 C3` (2 bytes)
MOV r8, imm8	B0h+r	`MOV AL, 5`	`B0 05`	`C6 C0 05` (3 bytes)
MOV r16, imm16	B8h+r	`MOV AX, 1234h`	`B8 34 12`	`C7 C0 34 12` (4 bytes)
MOV AL/AX ↔ memory	A0h–A3h	`MOV AX, [2000h]`	`A1 00 20`	`8B 06 00 20` (4 bytes)
ADD AL/AX, imm	04h / 05h	`ADD AL, 7`	`04 07`	`80 C0 07` (3 bytes)
ADD r/m16, imm8 (sign-extended)	83h /0	`ADD BX, 5`	`83 C3 05`	`81 C3 05 00` (4 bytes)

Table 11.7: Short forms that use no ModR/M byte. "r" is the register code written in the last three bits of the opcode.

The 83h form in the last row shows another idea. The opcodes 80h, 81h, and 83h perform eight different arithmetic and logic instructions (ADD, OR, ADC, SBB, AND, SUB, XOR, CMP) with an immediate value. The REG field of ModR/M says which operation is performed, because these instructions have no second register operand. In manuals this is written `83 /0`: the "/0" says that 000 goes into the REG field, that is the operation is ADD. For `ADD BX, 5` ModR/M = 11 000 011 = C3h and a single byte 05 follows. The processor sign-extends 05h to 0005h and adds it to BX. The same "operation selector" layout exists in the FEh and FFh families: `FF /4` is indirect near JMP, `FF /6` is PUSH memory, `FE /0` is byte INC. This layout reappears when the encoding of JMP is explained.

Common mistake

Assuming that an instruction has only one machine code. `MOV AL, [1000h]` can be encoded both as A0 00 10 (3 bytes) and as 8A 06 00 10 (4 bytes); both are correct. The assembler picks the short form. When encoding by hand, checking first whether a short form exists avoids confusion on exams and when comparing with listing output.

11.3 Encoding step by step

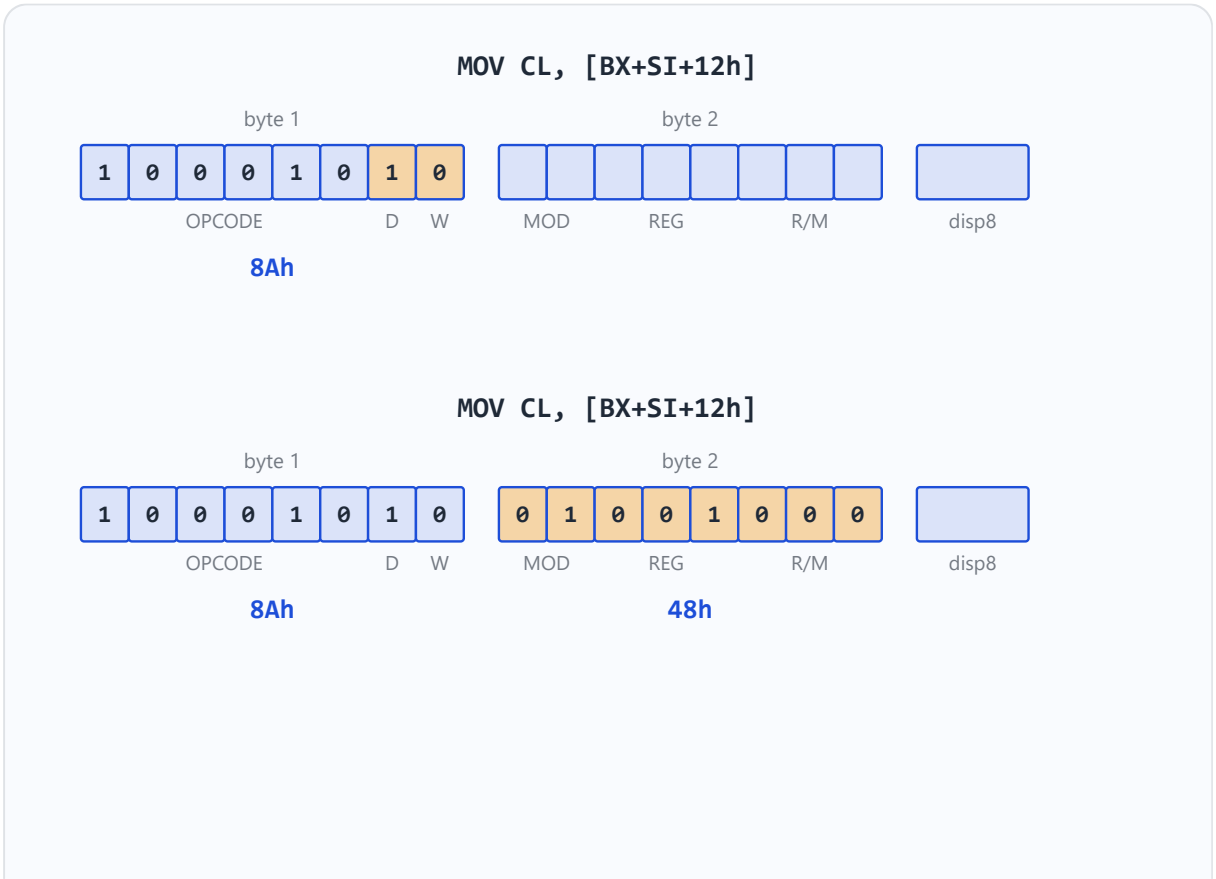
11.3.1 The encoding method

Encoding an assembly instruction by hand repeats on paper the job the assembler does. The work always proceeds in the same order:

1. **Choose the format.** Look at the types of the operands: register–register, register–memory, immediate–register, immediate–memory. If a short special form exists (AL/AX with memory, a register with an immediate value, INC r16 ...), use it; otherwise go to the general format.
2. **Write the opcode.** Write the 6-bit code of the instruction family (100010 for MOV, 000000 for ADD). Then add the D and W bits. D is chosen by whether the register in the REG field is the destination; W by the operand size.
3. **Build the ModR/M byte.** Write the code of the register operand into the REG field. If the other operand is a register, MOD = 11 and its code goes into R/M. If it is memory, R/M is chosen according to the addressing form and MOD = 00, 01, or 10 according to the size of the displacement.
4. **Add the displacement and the immediate.** The displacement comes first, then the immediate; both are written low byte first.
5. **Put the prefixes in front.** If a segment other than the default is used, a segment prefix; if a string instruction is repeated, the REP prefix; if the bus is locked, LOCK is added at the very front.

11.3.2 Example: `MOV CL, [BX+SI+12h]`

This instruction copies the byte at address DS:(BX + SI + 12h) into CL. One operand is a register (CL) and the other is memory. There is no short form, because the short memory form exists only for AL and AX and only for a direct address. The general format is used: MOV reg, r/m. Figure 11.2 shows the three stages of the encoding.



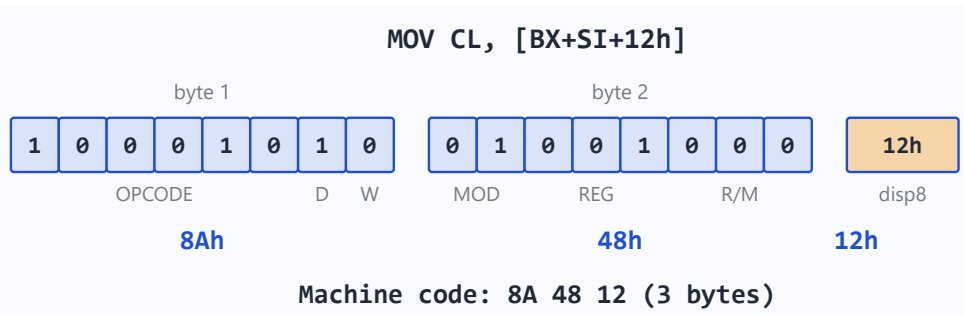


Figure 11.2: The encoding of `MOV CL, [BX+SI+12h]`. At the top the first byte (opcode 100010, D = 1, W = 0 → 8Ah), in the middle the ModR/M byte (MOD = 01, REG = 001, R/M = 000 → 48h), at the bottom the 8-bit displacement (12h) and the result.

1. **Opcode.** The MOV family between a register and r/m is 100010.
2. **D and W.** The destination CL is a register and goes into the REG field; since REG is the destination, D = 1. CL is a byte register; W = 0. The first byte is 1000 1010b = 8Ah.
3. **REG.** The code of CL is 001 in the W = 0 column.
4. **R/M and MOD.** The R/M code of the form [BX+SI+d] is 000. The displacement 12h fits in the range −128…+127; MOD = 01 (8-bit displacement). ModR/M = 01 001 000b = 48h.
5. **Displacement.** 12h is added as a single byte.

The result is **8A 48 12** and the instruction is three bytes. Had the displacement been a word such as 1234h, MOD = 10 would be chosen, ModR/M would be 10 001 000b = 88h, and the instruction would take four bytes as 8A 88 34 12. The time of the instruction is 8 + EA; since the EA time of the form [BX+SI+d] is 11 cycles, the total is 19 clock cycles. Most of this time goes to the address calculation and to the BIU fetching the byte from memory.

Example 11.1

Let us encode `MOV [SI+20h], CX`. The source CX goes into the REG field and the destination memory is in R/M: since REG is the source, D = 0, and since it is a word operation, W = 1. The first byte is 100010 0 1 = 89h. REG = 001 (CX), R/M = 100 ([SI+d]), and since 20h fits in one byte, MOD = 01. ModR/M = 01 001 100b = 4Ch. The result is **89 4C 20**.

By the same method, `ADD BX, [1234h]`: the ADD family is 000000, BX is the destination and is in REG, so D = 1, W = 1 → 03h. REG = 011 (BX). For a direct address MOD = 00, R/M = 110 → ModR/M = 00 011 110b = 1Eh. The address 34 12 follows: **03 1E 34 12**.

11.3.3 Example instructions and their lengths

Table 11.8 gives ten instructions of different lengths together with their machine codes. The table shows when all the fields of the instruction format appear: opcode only (1 byte), opcode and immediate, opcode and ModR/M, prefix and ModR/M, and ModR/M with displacement and

immediate together.

Instruction	Machine code	Bytes	Explanation
`NOP`	`90`	1	actually `XCHG AX, AX` (90h + 0)
`INC AX` / `PUSH BX`	`40` / `53`	1	register code inside the opcode (40h + r, 50h + r)
`MOV AL, 5`	`B0 05`	2	B0h + r, immediate byte
`INT 21h`	`CD 21`	2	interrupt number inside the instruction
`MOV AX, 1234h`	`B8 34 12`	3	low byte first (little-endian)
`MOV CL, [BX+SI+12h]`	`8A 48 12`	3	ModR/M + d8
`MOV ES:[BX], AL`	`26 88 07`	3	26h = ES prefix
`ADD [BX+DI+1000h], DX`	`01 91 00 10`	4	MOD = 10, d16
`MOV WORD PTR [1234h], 5678h`	`C7 06 34 12 78 56`	6	longest unprefixed form: d16 + imm16
`REP MOVSB`	`F3 A4`	2	F3h = REP prefix

Table 11.8: Example instructions, their machine codes, and lengths.

It is useful to open up a few rows. In `ADD [BX+DI+1000h], DX` the source DX is in the REG field (D = 0, W = 1 → 01h). ModR/M = 10 010 001b = 91h: MOD = 10 (16-bit displacement), REG = 010 (DX), R/M = 001 ([BX+DI+d]). The displacement 1000h is added in little-endian order as 00 10. This instruction both reads and writes memory: the BIU first reads the word, the ALU does the addition, and the BIU writes the result back to the same address. On the 8086 the time of this form is 16 + EA, that is 16 + 12 = 28 clock cycles.

In `MOV WORD PTR [1234h], 5678h` all the fields are present together. C7h is the word form of the "move immediate to r/m" family. ModR/M = 00 000 110b = 06h: the REG field here is not a register but an operation selector, and it is 000 for MOV; MOD = 00, R/M = 110 means a direct address. Then comes the address (34 12) and finally the immediate value (78 56). This instruction is six bytes, the greatest length an unprefixed 8086 instruction can reach. The `WORD PTR` in the assembly line is necessary; without it the assembler cannot tell whether 5678h is to be written as a byte or a word. In this example the word can be guessed because 5678h does not fit in a byte, but in a line like `MOV [1234h], 5` the size is really ambiguous and an error is given.

11.3.4 The encoding forms of MOV

MOV is the most frequently used instruction of the 8086 and hides eight separate encoding forms behind a single mnemonic. Table 11.9 gives these forms, the fields that follow, an example of each, and the timings from the Intel 8086 manual.

Form	Opcode (8 / 16 bit)	Followed by	Example	Kod	Time
register → r/m	`88` / `89`	ModR/M (+ disp)	`MOV [BX], AL`	`88 07`	2 (register), 9 + EA

Form	Opcode (8 / 16 bit)	Followed by	Example	Kod	Time
r/m → register	`8A` / `8B`	ModR/M (+ disp)	`MOV CX, [SI]`	`8B 0C`	2 (register), 8 + EA
immediate → r/m	`C6` / `C7`	ModR/M (REG = 000) + disp + imm	`MOV BYTE PTR [BX], 5`	`C6 07 05`	10 + EA
immediate → register	`B0+r` / `B8+r`	imm8 / imm16	`MOV SI, 1000h`	`BE 00 10`	4
memory → AL / AX	`A0` / `A1`	16-bit address	`MOV AX, [2000h]`	`A1 00 20`	10
AL / AX → memory	`A2` / `A3`	16-bit address	`MOV [1000h], AL`	`A2 00 10`	10
segment register → r/m16	`8C`	ModR/M (REG = sreg)	`MOV AX, ES`	`8C C0`	2, 9 + EA
r/m16 → segment register	`8E`	ModR/M (REG = sreg)	`MOV DS, AX`	`8E D8`	2, 8 + EA

Table 11.9: The encoding forms of MOV. Times are in clock cycles; EA is the effective address time.

The first two rows of the table are the D = 0 and D = 1 cases of the same family: the only difference between 88h/89h and 8Ah/8Bh is bit 1. In `MOV CX, [SI]` CX is the destination and is in REG: 8Bh; ModR/M = 00 001 100b = 0Ch. In `MOV [BX], AL` AL is the source: 88h; ModR/M = 00 000 111b = 07h. In the third row the REG field selects no register but means "000 = MOV"; since the second operand is an immediate value, REG is free. The short form A0h–A3h for moves between AL/AX and a direct address skips the ModR/M byte and saves one byte. The short form takes 10 cycles; the general form takes 8 + 6 = 14 for a read and 9 + 6 = 15 for a write.

11.3.5 MOV with segment registers

Segment registers are kept apart from the general-purpose registers and are moved with only two MOV forms. 8Ch copies from a segment register to r/m16, and 8Eh from r/m16 to a segment register. In both forms the first bit of the REG field is 0 and the last two bits are the code of the segment register: ES = 00, CS = 01, SS = 10, DS = 11. Figure 11.3 opens up the second byte of `MOV DS, AX`: MOD = 11 (register), REG = 011 (DS), R/M = 000 (AX), that is 1101 1000b = D8h. The instruction is **8E D8**.

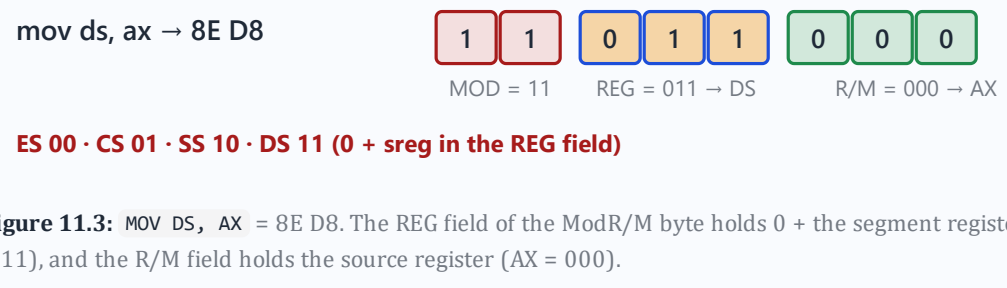


Figure 11.3: `MOV DS, AX` = 8E D8. The REG field of the ModR/M byte holds 0 + the segment register code (DS = 11), and the R/M field holds the source register (AX = 000).

Some forms do not exist at all. There is no opcode that writes an immediate value into a segment register; `MOV DS, 1234h` is invalid and the value is first loaded into a general register such as AX. There is no memory-to-memory MOV either, because the ModR/M byte can describe only one memory operand. A job that needs two memory operands is the special task of the string instructions (MOVS, CMPS). `MOV CS, AX` may look encodable but is not used: an instruction that changes CS without IP would make the program fetch its next instruction from a completely different segment. CS changes only, together with IP, through far JMP, far CALL, RETF, INT, and IRET. The 8086 has one more feature: after a MOV or POP that writes SS, no interrupt is accepted until the next instruction ends. This prevents an interrupt from slipping in between `MOV SS, AX` and the `MOV SP, …` that follows it and using a wrong stack.

Example 11.2

`MOV ES, DX` : the 8Eh family, REG = 0 + 00 (ES) = 000, MOD = 11, R/M = 010 (DX). ModR/M = 11 000 010b = C2h. The instruction is **8E C2**. The opposite `MOV AX, ES` is encoded with 8Ch and ModR/M = 11 000 000b = C0h: **8C C0**.

11.3.6 Direct address: MOD = 00, R/M = 110

The only irregularity in the R/M table is the combination MOD = 00 and R/M = 110. Had the table been regular, this combination would have meant [BP]. The designers of the 8086 reserved it for the 16-bit direct address, because reaching variables by name is far more common than [BP]. `MOV [1000h], BL` is therefore encoded like this:

Byte	Bits	Meaning
`88h`	`100010 0 0`	MOV r/m ↔ reg; D = 0 (REG is the source), W = 0 (byte)
`1Eh`	`00 011 110`	MOD = 00, REG = 011 (BL), R/M = 110 → direct address
`00 10`	`0001 0000 0000 0000`	16-bit address 1000h (low byte first)

Table 11.10: The bits of the instruction `MOV [1000h], BL` = 88 1E 00 10.

The instruction is four bytes and the segment is DS. For BL there is no short form like A2h; the short form is only for AL and AX. Had AL been written to the same address, the assembler would have produced A2 00 10 (3 bytes). `88 06 00 10` , written in the general form, does the same job: ModR/M 06h = 00 000 110b, REG = 000 (AL). Table 11.11 compares these forms with [BP].

Instruction	Kod	Explanation
`MOV [1000h], AL`	`A2 00 10`	short form for AL, 3 bytes
`MOV [1000h], AL`	`88 06 00 10`	general form, 4 bytes (same job): ModR/M 06h = 00 000 110
`MOV [1000h], BL`	`88 1E 00 10`	no short form for BL
`MOV [BP], BL`	`88 5E 00`	5Eh = 01 011 110: [BP+d8], d8 = 0; segment SS

Table 11.11: Encodings of the direct address and of [BP].

This rule concerns not only the assembler but also the decoder. A decoder that sees the byte 1Eh must notice that MOD = 00 and R/M = 110 and read two address bytes after it. A decoder that forgets the rule takes 88 1E for a two-byte `MOV [BP], BL`. The address bytes then become the start of a new instruction: 00 10 is read as `ADD [BX+SI], DL` (00h = ADD r/m8, r8; 10h = 00 010 000b). All the instruction boundaries after that shift. This is a small example of the general problem treated in the next section.

11.3.7 Prefix bytes

A prefix is not an instruction by itself; it changes the behavior of the instruction that follows. The 8086 has three kinds of prefix and Table 11.12 lists all of them.

Prefix	Byte	Effect
ES: / CS: / SS: / DS:	`26` / `2E` / `36` / `3E`	changes the default segment of the memory operand (001*ss*110 pattern)
REP / REPE / REPZ	`F3`	repeats the string instruction while CX ≠ 0 (and ZF = 1 for CMPS/SCAS)
REPNE / REPNZ	`F2`	repeats the string instruction while CX ≠ 0 and ZF = 0
LOCK	`F0`	keeps the LOC$\bar{K}$ pin active for the duration of the instruction; closes the bus to other processors

Table 11.12: The 8086 prefix bytes.

The bit pattern of the segment prefixes is interesting: 26h = 001 **00** 110, 2Eh = 001 **01** 110, 36h = 001 **10** 110, 3Eh = 001 **11** 110. The middle two bits are the same segment register code used in the MOV forms. `MOV AL, ES:[BX]` is therefore encoded as 26 8A 07: the prefix, then the code of `MOV AL, [BX]` itself. The assembler does not produce an unnecessary prefix. If `MOV AL, DS:[BX]` is written, no prefix is added because DS is already the default segment, and the code stays 8A 07. `MOV AL, DS:[BP]`, however, needs the prefix, because the default for BP is SS.

A prefix adds 2 clock cycles to the time of the instruction and takes one byte. Another cost shows up with interrupts. If an interrupt arrives while a string instruction with a REP prefix is being executed, the 8086 can abandon the work and go to the interrupt. To restart the instruction on return it pushes the address of the prefix onto the stack. However the 8086 remembers only the *last* prefix: if REP and a segment prefix are used together (such as `REP ES: MOVSB`), the earlier prefix is lost on return. For this reason no segment prefix is added to a REP string instruction in 8086 programs; if necessary, interrupts are disabled with CLI.

Example 11.3

Let us decode the bytes `2E FF 27`. 2Eh is the CS prefix. FFh is a family whose operation is selected by the REG field of ModR/M. 27h = 00 100 111b: MOD = 00, REG = 100 (/4: indirect near JMP), R/M = 111 ([BX]). The instruction is `JMP WORD PTR CS:[BX]`: IP is replaced by the

word at address CS:[BX]. This prefix is needed when the jump table is kept in the code segment.

11.4 Decoding machine code

11.4.1 The decoding method

Decoding (*disassembly*) is the reverse of encoding: recovering assembly instructions from bytes in memory. Debuggers, the memory window of emu8086, and listing output do this job. Decoding by hand is a way to understand how a program sits in memory and to answer the "which instructions are these bytes?" question that is often asked on exams. The method imitates what the EU of the processor does:

1. **Know the starting point.** Decoding must start at the first byte of an instruction. The start of the program (0100h for .COM) or a jump target is a reliable starting point.
2. **Classify the first byte.** If the byte is a prefix (26h, 2Eh, 36h, 3Eh, F0h, F2h, F3h), note it and move on to the next byte. Otherwise find the instruction family from the opcode table.
3. **Is it a short form?** For codes such as 40h–5Fh, 90h–97h, and B0h–BFh the register is inside the opcode; there is no ModR/M. The length of the immediate value that follows is known from the rule of the family.
4. **Open up the ModR/M.** If the family needs ModR/M, the next byte is split into the MOD, REG, and R/M fields. The length of the displacement follows from MOD and R/M (do not forget the special case MOD = 00 and R/M = 110).
5. **Take the remaining bytes.** The displacement and the immediate are read little-endian. The instruction ends here; the next byte is the first byte of a new instruction.

11.4.2 Example: a twelve-byte fragment

Let the following twelve bytes, starting at 0100h in memory, be part of a program: `8B 4E 04 03 C3 A3 00 02 F3 AB EB FE`. Decoding starts at 0100h. Table 11.13 gives the result; the steps are explained below.

Address	Bytes	Decoding	Instruction
0100h	`8B 4E 04`	8Bh = 100010 1 1; 4Eh = 01 001 110 → MOD = 01, REG = CX, R/M = [BP+d8]; d8 = 04h	`MOV CX, [BP+04h]`
0103h	`03 C3`	03h = 000000 1 1 (ADD, D = 1, W = 1); C3h = 11 000 011 → AX ← AX + BX	`ADD AX, BX`
0105h	`A3 00 02`	A3h = AX → memory short form; address 0200h	`MOV [0200h], AX`
0108h	`F3 AB`	F3h = REP prefix; ABh = STOSW	`REP STOSW`

Address	Bytes	Decoding	Instruction
010Ah	`EB FE`	EBh = short JMP; FEh = −2 → target 010Ch − 2 = 010Ah	`JMP 010Ah` (to itself)

Table 11.13: Decoding of the bytes 8B 4E 04 03 C3 A3 00 02 F3 AB EB FE.

The first byte 8Bh belongs to the MOV r/m ↔ reg family and carries D = 1, W = 1: the REG field is the 16-bit destination register. The second byte 4Eh = 0100 1110b, split into its fields, is 01 | 001 | 110. Since MOD = 01, one byte of displacement will follow. REG = 001 with W = 1 is CX. R/M = 110 with MOD = 01 means [BP+d8]; with MOD = 00 it would have been a direct address. The third byte 04h is the displacement. The instruction is `MOV CX, [BP+04h]`, with segment SS. This form is the typical pattern for reading a procedure parameter from the stack.

The fourth byte 03h is the D = 1, W = 1 form of the ADD family. In the fifth byte C3h = 11 000 011b, MOD = 11, so both operands are registers: REG = AX (destination), R/M = BX. The instruction is `ADD AX, BX`. It was shown above that the same instruction can also be encoded as 01 D8; here we meet the D = 1 form. The sixth byte A3h is a short form that uses no ModR/M: it writes AX to a direct address. The next two bytes give the address: 00 02 → 0200h. The ninth byte F3h is a prefix; the instruction is completed by the next byte. ABh = 1010 101 1b is the W = 1 form of the STOS family. The instruction is `REP STOSW` and takes only two bytes. The last two bytes are EB FE. EBh is the short JMP and FEh is −2 as signed. The target is found by adding −2 to 010Ch, the address of the next instruction: 010Ah, that is the instruction itself. These two bytes are the endless loop that stops a program where it is.

The result: twelve bytes are five instructions. The lengths of the instructions are 3, 2, 3, 2, and 2 bytes, and each length was derived by looking only at the first byte or two. That the byte count comes out exactly at the end of decoding (0100h + 12 = 010Ch) is a check that the boundaries were found correctly.

11.4.3 Decoding from a wrong offset

The first step of the decoding method was to know the starting point. An example shows why this step cannot be skipped. Let the following eleven bytes be in memory starting at offset 0000h: `B4 02 B2 41 CD 21 B8 00 4C CD 21`. Decoded from offset 0, these bytes are a familiar program that writes 'A' to the screen and ends the program:

```
MOV AH, 02h     ; B4 02
MOV DL, 41h     ; B2 41  ('A')
INT 21h        ; CD 21  write character
MOV AX, 4C00h   ; B8 00 4C
INT 21h        ; CD 21  end the program
```

B4h = B0h + 4 is `MOV AH, imm8` and takes one immediate byte: 02h. B2h = B0h + 2 is `MOV DL, imm8`: 41h = 'A'. CDh is the INT instruction and takes the interrupt number from the byte that follows. B8h is `MOV AX, imm16` and takes two bytes: 00 4C → 4C00h. The final CD 21 calls the DOS program-termination service with AH = 4Ch. The decoding is exactly the code the programmer wrote.

Now suppose IP slips by one byte; for example, a faulty jump goes to 0001h. The processor takes 02h as an opcode. Figure 11.4 shows the two decodings one above the other and Table 11.14 gives a byte-by-byte explanation of the second.

Figure 11.4: Decoding the same eleven bytes from offset 0 (top) and from offset 1 (bottom). In the decoding from offset 1, the 'A' and the opcode of INT are lost inside a displacement; at the last CD 21 the two decodings fall on the same boundary again.

Offset	Bytes	Decoding	Instruction
0001h	`02 B2 41 CD`	02h = ADD r8, r/m8 (D = 1, W = 0); B2h = 10 110 010 → MOD = 10 (d16), REG = DH, R/M = [BP+SI]; d16 = CD41h	`ADD DH, [BP+SI+CD41h]`
0005h	`21 B8 00 4C`	21h = AND r/m16, r16 (D = 0, W = 1); B8h = 10 111 000 → MOD = 10, REG = DI, R/M = [BX+SI]; d16 = 4C00h	`AND [BX+SI+4C00h], DI`
0009h	`CD 21`	INT 21h	`INT 21h`

Table 11.14: Decoding that starts at offset 1.

Decoding from offset 1 gives three instructions. The first takes B2h as ModR/M and, because MOD = 10, reads two bytes of displacement. The character 'A' (41h) and the opcode of INT (CDh) are thereby lost inside a number, CD41h. This instruction adds the byte at address SS:(BP + SI + CD41h) to DH; it only reads. The second instruction is more dangerous: 21h is an AND with D = 0 and writes its result to the r/m operand in memory. The word at DS:(BX + SI + 4C00h) is ANDed with DI and corrupted. At the final CD 21 the two decodings fall on the same boundary again; the decoding has resynchronized by itself. But this time AX = 4C00h was never loaded. INT 21h is called with whatever happens to be in AH, with an unpredictable result.

The example teaches two things. First, the meaning of the bytes is determined not by the bytes but by where IP starts. The same eleven bytes are a harmless "write A and exit" program from one start and a memory-corrupting instruction sequence from one byte further. Second, in x86 code decoding

usually returns to the correct boundary by itself after a few instructions. This makes a wrong decoding hard to notice: after a few meaningless instructions everything looks proper again.

Example 11.4

Let us decode the bytes `8A 1E 34 12` from offset 0. 8Ah = 100010 1 0 (D = 1, W = 0), 1Eh = 00 011 110 (MOD = 00, REG = BL, R/M = 110 → direct address). The instruction is `MOV BL, [1234h]`. If we start at offset 1, 1Eh is the one-byte `PUSH DS` (the segment PUSH of the form 06h + 8 × 3), and then 34 12 becomes the two-byte `XOR AL, 12h`. The four bytes this time give two separate instructions, and an unexpected word is pushed onto the stack.

11.4.4 Code and data: the processor's view

On the 8086 there is no hardware that separates code from data. The code segment and the data segment are just numbers in segment registers, and in .COM programs all four segment registers point to the same segment. The program puts its data right behind its code. If a JMP lands in the data area by mistake, or if the return address on the stack is corrupted and RET returns there, the processor starts executing data bytes as instructions. A variable such as `num db 3Ch`, read as an opcode, becomes 3Ch = `CMP AL, imm8` and swallows the following byte as an immediate value.

Disassemblers run into the same problem. A disassembler can guess which of the bytes in memory are code and which are data only by following the flow of the program. A listing file does not have this problem, because the assembler knows from the source what every line is. The most reliable way to inspect the machine code of a program is therefore to look at the listing the assembler produces. The .lst file created after compilation in emu8086 gives the address and the bytes of every line side by side.

Common mistake

Forgetting the `RET` or `INT 21h/4Ch` at the end of the program. When the code ends the processor does not stop; IP keeps advancing and executes the variables right behind it as instructions. The program then behaves meaninglessly or hangs. Defining the variables before the code and jumping over them with a JMP, or making sure that every exit path explicitly ends the program, prevents this mistake.

11.5 Data-transfer instructions

11.5.1 Overview

Data-transfer instructions copy a value from one place to another: from register to register, from memory to a register, to the stack, or to an I/O port. What these instructions have in common is that no calculation result passes through the ALU. They therefore *do not change* the flags. The two exceptions are SAHF and POPF, which are designed to write the flags directly. Table 11.15 summarizes the instructions of this class.

Instruction	Operation	Example	Kod
`MOV h, k`	d ← s (copy)	`MOV BX, [SI]`	`8B 1C`
`PUSH k` / `POP h`	SP ← SP − 2, [SS:SP] ← k / h ← [SS:SP], SP ← SP + 2	`PUSH AX`	`50`
`XCHG a, b`	a ↔ b (one must be a register)	`XCHG AL, [BX]`	`86 07`
`IN` / `OUT`	AL/AX ↔ I/O port	`IN AL, 60h`	`E4 60`
`XLAT`	AL ← [DS:BX + AL]	table translation	`D7`
`LEA r, m`	r ← **address** of m (EA)	`LEA SI, [BX+DI+4]`	`8D 71 04`
`LDS` / `LES r, m`	r ← [m], DS/ES ← [m + 2] (far pointer)	`LES DI, far_ptr`	`C4 3E …`
`LAHF` / `SAHF`	AH ← low byte of FLAGS / the reverse		`9F` / `9E`
`PUSHF` / `POPF`	push FLAGS onto the stack / pop it from the stack		`9C` / `9D`

Table 11.15: Data-transfer instructions.

The encoding forms of MOV were covered in detail above. The register forms of PUSH and POP are single bytes (50h + r, 58h + r); those of the segment registers are single bytes too (PUSH ES = 06h, PUSH CS = 0Eh, PUSH SS = 16h, PUSH DS = 1Eh). XCHG swaps two operands and removes the need for a temporary register; swapping AX with a register is a single byte of the form 90h + r. IN and OUT are the only instructions that reach the I/O address space. If the port number is 0–255 it is given inside the instruction as a constant (E4h, E5h, E6h, E7h); larger port numbers are given in DX (ECh, EDh, EEh, EFh). While these instructions execute, the M/I$\bar{O}$ pin of the 8086 is pulled to 0 and the address bus selects an I/O port instead of memory.

LAHF (*load AH from flags*) copies the low byte of the flag register into AH. This byte contains SF (bit 7), ZF (bit 6), AF (bit 4), PF (bit 2), and CF (bit 0). SAHF does the reverse and writes these five bits in AH into the flags. The two instructions were included to ease the flag handling of programs ported from the 8080 processor. OF, DF, IF, and TF can be reached only as a group with PUSHF/POPF.

11.5.2 XLAT: table translation

Many programs translate one value to another through a table: a hexadecimal digit to an ASCII character, a key code to a letter, a number to the bit pattern of a seven-segment display. The general way to do this is to put the value in an index register and read from the table. The 8086 has set aside a

special one-byte instruction for this frequent job.

Definition 11.3

XLAT (*translate*) uses the value in AL as an index into a **translation table**: AL ← [DS:BX + AL]. BX holds the start address of the table; AL is added as an unsigned number (0–255). The table can be at most 256 bytes. The instruction is a single byte (D7h), does not change the flags, and takes 11 clock cycles.

The effective address calculation of XLAT fits none of the ordinary addressing forms: the 8-bit AL is added to the 16-bit BX. XLAT therefore has no ModR/M byte, and its operands are implicit. Some assemblers also accept the notation `XLAT hex_tab` or `XLATB`; the operand is only for documentation and does not change the code. The one difference is that a segment prefix is added if the operand is in another segment (`XLAT ES:hex_tab`). Putting the address of the table into BX is the programmer's job.

11.5.3 Code trace: printing a byte in hexadecimal with XLAT

The following program writes a byte in memory to the screen as two hexadecimal digits. The byte is 3Ch; "3C" should appear on the screen. A byte is split into two halves (nibbles); each half is a number between 0 and 15 and is translated by XLAT into one of the characters '0'...'F'.

```
org 100h
mov bx, offset hex_tab
mov al, num       ; AL = 3Ch
mov cl, 4
shr al, cl         ; high nibble
xlat               ; AL = [BX+AL]
mov dl, al
mov ah, 2
int 21h            ; print DL
mov al, num
and al, 0Fh        ; low nibble
xlat
mov dl, al
int 21h
ret
num  db 3Ch
hex_tab db '0123456789ABCDEF'
```

The program starts at 0100h. The total length of the instructions is 28 bytes; `num` therefore sits at 011Ch and `hex_tab` at 011Dh. Table 11.16 gives the state after each instruction.

Address	Kod	Instruction	AL	AH	BX	CL	DL	Output
0100h	`BB 1D 01`	`MOV BX, OFFSET hex_tab`	00h	00h	**011Dh**	00h	00h	
0103h	`A0 1C 01`	`MOV AL, num`	**3Ch**	00h	011Dh	00h	00h	
0106h	`B1 04`	`MOV CL, 4`	3Ch	00h	011Dh	**04h**	00h	
0108h	`D2 E8`	`SHR AL, CL`	**03h**	00h	011Dh	04h	00h	
010Ah	`D7`	`XLAT`	**33h** ('3')	00h	011Dh	04h	00h	
010Bh	`8A D0`	`MOV DL, AL`	33h	00h	011Dh	04h	**33h**	
010Dh	`B4 02`	`MOV AH, 2`	33h	**02h**	011Dh	04h	33h	
010Fh	`CD 21`	`INT 21h`	33h	02h	011Dh	04h	33h	**3**
0111h	`A0 1C 01`	`MOV AL, num`	**3Ch**	02h	011Dh	04h	33h	3
0114h	`24 0F`	`AND AL, 0Fh`	**0Ch**	02h	011Dh	04h	33h	3
0116h	`D7`	`XLAT`	**43h** ('C')	02h	011Dh	04h	33h	3
0117h	`8A D0`	`MOV DL, AL`	43h	02h	011Dh	04h	**43h**	3
0119h	`CD 21`	`INT 21h`	43h	02h	011Dh	04h	43h	**3C**
011Bh	`C3`	`RET`	43h	02h	011Dh	04h	43h	3C

Table 11.16: Trace of the XLAT program. Each row is the state after the instruction executes; at the start all registers are 0.

The first line loads the offset of the table into BX: 011Dh. `MOV AL, num` uses the short form for AL (A0h) and reads the direct address 011Ch. Because on the 8086 a shift count greater than 1 can be given only in CL, 4 is put in CL. `SHR AL, CL` shifts 0011 1100b right by four bits: 0000 0011b = 03h. The last bit shifted out is 1, so CF = 1; because the shift count is greater than 1, OF is undefined. These flags do not affect the flow of the program.

The first XLAT uses AL = 03h as the index. The address read is DS:(011Dh + 03h) = DS:0120h; there sits the fourth character of the table, '3' (33h). AL is now 33h. It is copied to DL, 02h is put in AH, and INT 21h is called. DOS service 02h writes the character in DL and puts the same value in AL; a "3" appears on the screen.

In the second half the byte is read again and `AND AL, 0Fh` clears the upper nibble: 3Ch AND 0Fh = 0Ch. AND sets CF and OF to 0. The second XLAT reads DS:(011Dh + 0Ch) = DS:0129h; there is 'C' (43h). Since AH is still 02h it is not reloaded; INT 21h is called with DL = 43h and "3C" appears on the screen. RET ends the program through the INT 20h in the PSP.

The gain of XLAT in this program is clear. The same translation could also have been done with preparation like `MOV SI, AX` and an instruction such as `MOV AL, hex_tab[SI]`. But extending AL into a 16-bit index register costs extra instructions. XLAT does the job in one byte and 11 cycles. Another benefit of the table method is that it needs no comparison and branching: instead of the logic "if the digit is less than 10 add '0', otherwise add 'A' − 10", a single table read is enough.

Example 11.5

With DS = 1000h, BX = 0200h, and AL = 05h, which physical address does XLAT read? The effective address is BX + AL = 0200h + 05h = 0205h. The physical address is 1000h × 10h + 0205h = 10000h + 0205h = **10205h**. The byte at that address is loaded into AL; BX is unchanged.

Common mistake

Thinking that AH must be cleared before XLAT, or the opposite, forgetting to clear the extra bits in AL. XLAT uses only AL; it does not look at AH. But all eight bits of AL take part in the index. Had `AND AL, 0Fh` been forgotten in the program above, the index would have been 3Ch = 60 and a byte far beyond the table would have been read.

11.5.4 LEA: address or contents?

A memory operand has two sides: its address and the contents at that address. MOV reads the contents. Sometimes only the address is needed: putting the start of a string into SI, or handing the address of an array element to a procedure. The 8086 offers a separate instruction for this.

Definition 11.4

LEA (*load effective address*) computes the effective address of the memory operand and writes it to the destination register: r16 ← EA. It does not access memory and does not change the flags. Its code is 8Dh + ModR/M; its time is 2 + EA clock cycles.

For a variable whose address is known at assembly time, `LEA SI, msg` and `MOV SI, OFFSET msg` give the same result. emu8086 therefore compiles LEA with a direct address to the same three-byte code as `MOV r16, imm16`: `LEA BX, vals` = BB 14 01. Real MASM compiles the same line to a four-byte code beginning with 8D 1E; the extra byte also shifts the addresses of the variables that follow. The real strength of LEA shows in operands whose address is computed from registers at run time. `LEA SI, [BX+DI+4]` makes SI = BX + DI + 4 with the current values of BX and DI (8D 71 04). The OFFSET operator cannot do this, because it is a constant whose value is computed at assembly time.

That LEA neither accesses memory nor changes flags also makes it an addition tool. `LEA AX, [BX+SI]` performs AX = BX + SI in a single instruction; BX and SI are unchanged and the flags are preserved. The same job with `ADD` needs two instructions (`MOV AX, BX` / `ADD AX, SI`) and disturbs the flags. This feature is useful when an address has to be incremented between a CMP and a conditional jump.

```
mov bx, 0010h
mov di, 0004h
lea ax, [bx+di+2]  ; AX = 0016h (address)
mov ax, [bx+di+2]  ; AX = [0016h] (contents)
```

These two lines use the same operand and compute the same effective address (0010h + 0004h + 2 = 0016h). LEA writes this number to AX and stops. MOV, however, makes the BIU read a word from DS:0016h and writes the value read to AX. Their machine codes differ only in the opcode: 8D 41 02 and 8B 41 02.

11.5.5 LDS and LES: the far pointer

An offset alone is not enough to specify a place in memory; a segment is needed too. A four-byte value that holds both the segment and the offset is called a **far pointer**. A far pointer is stored in memory in a fixed order: the offset first (low word), then the segment (high word). Each word is itself little-endian. A far pointer to address B800:0000 sits in memory as the bytes 00 00 00 B8.

Definition 11.5

LDS r16, m32 loads a far pointer from memory: r16 ← [m], DS ← [m + 2]. **LES r16, m32** does the same with ES. The opcodes are C5h (LDS) and C4h (LES), followed by ModR/M; the second operand must be memory. The flags do not change; the time is 16 + EA clock cycles.

LDS and LES read two words in a single instruction. Doing the same with MOV needs two instructions: one loads the offset into a general register, the other the segment into a segment register. Their use together with the string instructions is especially common: LDS SI prepares the full address of the source and LES DI the full address of the destination in one instruction. The typical pattern for writing to an area outside the program's own segment, such as screen memory, is `LES DI, screen` followed by `STOSB` or `STOSW`.

Example 11.6

Let the bytes of the variable `far_ptr dd` in memory be 34 12 00 B8. After `LES DI, far_ptr` the first word (34 12 → 1234h) is loaded into DI and the second word (00 B8 → B800h) into ES: **DI = 1234h, ES = B800h**. ES:DI now points to the physical address B8000h + 1234h = B9234h. Had the bytes been 00 20 00 10, `LDS SI, …` would make SI = 2000h and DS = 1000h, and DS:SI would point to 12000h.

11.5.6 Code trace: LEA, MOV, and LES

The following program shows together the difference between LEA and MOV and the loading of a far pointer with LES. At the end the program writes 'A' to the top-left corner of the screen.

```
org 100h
lea bx, vals        ; BX ← address
mov di, 2
lea ax, [bx+di+2]   ; AX ← EA
mov ax, [bx+di+2]   ; AX ← [EA]
les di, screen       ; far pointer
mov al, 'A'
stosb               ; ES:[DI] ← AL
ret
vals   dw 1111h, 2222h, 3333h
screen dw 0000h, 0B800h
```

The lengths of the instructions are 3 + 3 + 3 + 3 + 4 + 2 + 1 + 1 = 20 bytes. Since the program starts at 0100h, `vals` sits at 0114h and `screen` at 0114h + 6 = 011Ah. Table 11.17 gives the trace, and Figure 11.5 shows that LEA and MOV take different things from the same EA.

Address	Kod	Instruction	AX	BX	DI	ES	Memory
0100h	`BB 14 01`	`LEA BX, vals`	0000h	**0114h**	0000h	0700h	
0103h	`BF 02 00`	`MOV DI, 2`	0000h	0114h	**0002h**	0700h	
0106h	`8D 41 02`	`LEA AX, [BX+DI+2]`	**0118h**	0114h	0002h	0700h	not read
0109h	`8B 41 02`	`MOV AX, [BX+DI+2]`	**3333h**	0114h	0002h	0700h	DS:0118h is read
010Ch	`C4 3E 1A 01`	`LES DI, screen`	3333h	0114h	**0000h**	**B800h**	DS:011Ah–011Dh are read
0110h	`B0 41`	`MOV AL, 'A'`	**3341h**	0114h	0000h	B800h	
0112h	`AA`	`STOSB`	3341h	0114h	**0001h**	B800h	[B8000h] ← 41h
0113h	`C3`	`RET`	3341h	0114h	0001h	B800h	

Table 11.17: Trace of the LEA, MOV, and LES program. At the start CS = DS = ES = SS = 0700h.

Figure 11.5: Four moments of the LEA and LES program. Top left: LEA AX, [BX+DI+2] gives only the address (0118h). Top right: with the same EA, MOV reads the contents (3333h). Bottom left: LES DI, screen loads the far pointer (DI = 0000h, ES = B800h). Bottom right: STOSB writes 'A' to B8000h and increments DI by 1.

The first instruction is compiled in emu8086 as the three-byte `MOV BX, 0114h`; BX = 0114h and memory is not read. After `MOV DI, 2` the two instructions use the same operand. EA = BX + DI + 2 = 0114h + 0002h + 0002h = 0118h. LEA writes this number to AX. MOV, however, reads the word at DS:0118h: 3333h, the third element of `vals`. The difference in machine code is only in the opcode (8Dh and 8Bh); the ModR/M (41h = 01 000 001: [BX+DI+d8], REG = AX) and the displacement (02h) are the same.

The ModR/M byte of `LES DI, screen` is 3Eh = 00 111 110b: REG = 111 (DI), MOD = 00 and R/M = 110 (direct address 011Ah). The processor reads four bytes from DS:011Ah: 00 00 00 B8. The first word 0000h goes to DI and the second, B800h, to ES. At this point the previous value of DI (2) is lost. `MOV AL, 'A'` changes only AL; AH keeps the 33h left over from the earlier MOV and AX = 3341h. STOSB writes AL to address ES:DI. The physical address is B800h × 10h + 0000h = B8000h; this is the character cell at the top-left corner of the screen in text mode. Since DF = 0, DI is incremented by one.

Common mistake

Continuing to access the program's own variables after changing DS with LDS. After `LDS SI, ptr` DS points to another segment; an instruction such as `MOV AL, counter` now reads from the wrong segment. The fix is to save DS with PUSH DS before the LDS and restore it with POP DS when the work is done, or to prefer LES, which uses ES:DI for the destination.

11.6 Arithmetic, logic, and shift instructions: summary

The arithmetic and logic instructions are a topic in their own right; here only their place in the instruction set and the common layout of their encodings are summarized. Table 11.18 gives the groups.

Group	Instructions	Explanation
Addition	`ADD ADC INC`	ADC also adds the carry (CF); INC does not change CF
Subtraction	`SUB SBB DEC NEG CMP`	SBB also subtracts the borrow (CF); NEG = 0 − d; CMP does not write the result

Group	Instructions	Explanation
Multiplication	`MUL IMUL`	unsigned / signed; implicit AL or AX
Division	`DIV IDIV`	unsigned / signed; quotient and remainder
BCD adjust	`DAA DAS AAA AAS AAM AAD`	corrects the binary result to decimal
Sign extension	`CBW CWD`	AL → AX, AX → DX:AX
Logic	`AND OR XOR NOT TEST`	clear, set, and complement bits; TEST does not write the result
Shift / rotate	`SHL SAL SHR SAR ROL ROR RCL RCR`	shift count 1 or CL; the bit shifted out goes to CF

Table 11.18: Arithmetic, logic, and shift instructions.

The eight two-operand arithmetic-logic instructions (ADD, OR, ADC, SBB, AND, SUB, XOR, CMP) share the same encoding pattern. Bits 5–3 of the opcode select the operation: ADD = 000, OR = 001, ADC = 010, SBB = 011, AND = 100, SUB = 101, XOR = 110, CMP = 111. The register–r/m forms follow the pattern *00ooo0dw*: 00–03h for ADD, 28–2Bh for SUB, 38–3Bh for CMP. The forms with an immediate value are 80h, 81h, and 83h, and the same three-bit operation number is written into the REG field of ModR/M this time. Thus `SUB BX, 5` = 83 EB 05 (11 101 011: REG = 101 = SUB) and `CMP BX, 5` = 83 FB 05. Once this layout has been seen, the twenty-four forms of the eight instructions collapse into a single table.

The processor does addition and subtraction with a single adder circuit and does not know whether the number is signed or unsigned. It therefore produces flags for both interpretations: CF shows unsigned overflow and OF shows signed overflow. The programmer decides which flag is meaningful. Which flag each of these instructions changes, and how, is collected in the reference table at the end of the note.

11.7 String instructions

11.7.1 Why do string instructions exist?

Doing the same job over and over on bytes or words that lie consecutively in memory is very common in programs. Examples: copying a text from one place to another, comparing two passwords, searching a text for a character, filling the screen with blanks. These jobs can also be done with MOV, CMP, and INC. But several instructions are needed for each element: read from the source, write to the destination, increment the two addresses, decrement the counter, go back to the top of the loop. Moreover, because MOV cannot copy from memory to memory, every element passes through a temporary register.

For jobs of this kind the 8086 offers five **string instructions**. The operands of these instructions are implicit and are not written in the instruction. The source is always DS:SI and the destination always ES:DI. After doing its job the instruction advances SI and DI to the next element by itself. Each

instruction is therefore a single byte and takes care of all the address work of a loop within itself. Used together with a repeat prefix, a single two-byte instruction can process a block of thousands of bytes.

Definition 11.6

A **string instruction** is an instruction that processes one byte (B version) or one word (W version) at the source pointed to by DS:SI and/or at the destination pointed to by ES:DI. When it finishes, it increments or decrements the index registers it used by 1 or 2 according to the direction flag. The segment of the source can be changed with a segment prefix; the destination is always in ES.

11.7.2 The five string instructions

Table 11.19 gives the five string instructions, what they do, the registers they update, their flag effects, their opcodes, and their times when executed alone.

Instruction	Operation	Updated	Flags	Kod	Time
`MOVSB / MOVSW`	[ES:DI] ← [DS:SI]	SI, DI	–	`A4` / `A5`	18
`CMPSB / CMPSW`	[DS:SI] − [ES:DI]	SI, DI	like CMP	`A6` / `A7`	22
`SCASB / SCASW`	AL/AX − [ES:DI]	DI	like CMP	`AE` / `AF`	15
`LODSB / LODSW`	AL/AX ← [DS:SI]	SI	–	`AC` / `AD`	12
`STOSB / STOSW`	[ES:DI] ← AL/AX	DI	–	`AA` / `AB`	11

Table 11.19: String instructions. The code column shows the byte / word version; the time is in clock cycles for a single execution.

It helps to divide the instructions into two groups. MOVS, LODS, and STOS move data and do not touch the flags. CMPS and SCAS compare: they perform a subtraction without writing the result anywhere and set the flags as CMP does. All the opcodes follow the pattern 1010 xxxw; the last bit is W and distinguishes the byte and word versions. That the codes are only a single byte also explains why string instructions are so frequently used: the prefetch queue can take several string instructions in one read.

From the hardware point of view, string instructions make heavy use of the BIU. MOVSB needs one memory-read and one memory-write bus cycle; each cycle takes at least four clock cycles. CMPSB does two reads, SCASB and LODSB one read each, and STOSB one write. MOVSW moves two bytes in the same two cycles at even addresses; copying large blocks with the word version is therefore about twice as fast. If the source or destination is at an odd address, every word is split into two byte cycles and this gain shrinks.

A note on syntax: MASM and emu8086 also accept the form with operands. When `MOVS dst, src` is written, the assembler produces MOVSB or MOVSW by looking at the type of the variables. If a segment prefix is written on the source operand (`MOVS dst, ES:src`), that prefix is added to the code. The operands only determine the size and the prefix; loading the addresses into SI and DI is still the programmer's job.

11.7.3 The direction flag and the step

Whether a string instruction increments or decrements SI and DI is determined by the **direction flag** (DF). DF is bit 10 of the flag register and is a control flag: arithmetic instructions do not change it. It is set with two instructions:

- `CLD` (*clear direction flag*, FCh): DF = 0. String instructions **increment** SI and DI; the operation proceeds from a low address to a higher one.
- `STD` (*set direction flag*, FDh): DF = 1. String instructions **decrement** SI and DI; the operation proceeds from a high address to a lower one.

The step size depends on the version of the instruction: ±1 for the B version and ±2 for the W version. The update is always made *after* the operation. Therefore, after a string instruction finishes, SI and DI point not to the last element processed but to the next one. When a search instruction finds a match, the address of the match is DI − 1 (byte, DF = 0).

> **Common mistake**
>
> Assuming the value of DF. DOS usually leaves DF at 0 when it starts a program, but another procedure, a library, or an interrupt handler may have set DF to 1. Every piece of code that uses string instructions must set the direction explicitly with `CLD` or `STD` before starting. A procedure that uses STD must also restore DF with `CLD` before returning.

The source and destination of string instructions can be in different segments. In .COM programs CS, DS, ES, and SS point to the same segment, and DS:SI and ES:DI look at the same memory. In .EXE programs DOS sets ES not to the program's data segment but to the PSP. In these programs ES must be set explicitly, like DS:

```
mov ax, @data
mov ds, ax
mov es, ax       ; for the string destination
```

11.7.4 Code trace: copying with MOVSB

The following program copies the four-byte text 'ABCD' to a four-byte area filled with dots. The copy is done with a LOOP loop and a single MOVSB in each round, so that every execution of the instruction can be traced separately.

```
org 100h
cld
lea si, src          ; DS:SI source
lea di, dst          ; ES:DI destination
mov cx, 4
copy_lp:
movsb               ; [DI]←[SI], SI++, DI++
loop copy_lp         ; = rep movsb
ret
src db 'ABCD'
dst db 4 dup('.')
```

The code is 14 bytes long: CLD (1), two LEAs (3 + 3), MOV CX (3), MOVSB (1), LOOP (2), RET (1). `src` therefore starts at 010Eh and `dst` at 0112h. The code of LOOP is E2 FD: FDh = −3, 3 back from the address of the next instruction 010Dh, that is the address of MOVSB, 010Ah.

Step	Instruction	SI	DI	CX	dst	Explanation
1	`CLD`	0000h	0000h	0000h	`....`	DF = 0: forward
2	`LEA SI, src`	**010Eh**	0000h	0000h	`....`	BE 0E 01
3	`LEA DI, dst`	010Eh	**0112h**	0000h	`....`	BF 12 01
4	`MOV CX, 4`	010Eh	0112h	**0004h**	`....`	counter
5	`MOVSB`	**010Fh**	**0113h**	0004h	`A...`	[0112h] ← [010Eh] = 41h
6	`LOOP copy_lp`	010Fh	0113h	**0003h**	`A...`	CX ≠ 0 → jump
7	`MOVSB`	**0110h**	**0114h**	0003h	`AB..`	[0113h] ← 42h
8	`LOOP copy_lp`	0110h	0114h	**0002h**	`AB..`	jump
9	`MOVSB`	**0111h**	**0115h**	0002h	`ABC.`	[0114h] ← 43h
10	`LOOP copy_lp`	0111h	0115h	**0001h**	`ABC.`	jump
11	`MOVSB`	**0112h**	**0116h**	0001h	`ABCD`	[0115h] ← 44h
12	`LOOP copy_lp`	0112h	0116h	**0000h**	`ABCD`	CX = 0 → fall through
13	`RET`	0112h	0116h	0000h	`ABCD`	the program ends

Table 11.20: Trace of the MOVSB loop. The flags do not change at any step.

Two things happen in every MOVSB step. The BIU first reads the byte at DS:SI and then writes it to ES:DI; the byte passes through no general register. Then SI and DI are incremented by one. After the last MOVSB SI = 0112h. This is the byte right behind the source and happens to be the start of `dst`.

DI = 0116h points one past the destination. The flag register does not change from start to finish. LOOP does not touch the flags either; it decrements CX and jumps if it is not zero.

The same job is done with a single instruction using `MOV CX, 4` and `REP MOVSB`, and the result registers are the same: SI = 0112h, DI = 0116h, CX = 0. The difference is the time. The LOOP version spends 18 cycles on MOVSB and 17 cycles on a taken LOOP for each byte; the last LOOP is not taken and takes 5 cycles. For four bytes that is 4 × 18 + 3 × 17 + 5 = 128 cycles. REP MOVSB takes 9 + 17 × 4 = 77 cycles. The difference approaches a factor of two as the block grows.

11.7.5 Code trace: converting to upper case with LODSB and STOSB

When LODS and STOS are used together, a string is read, each element is processed in a register, and the result is written back. This pattern is the typical way to transform an array in place. The following program converts the text 'micro' to upper case. In ASCII the difference between a lowercase and an uppercase letter is only bit 5 ('m' = 6Dh = 0110 1101b, 'M' = 4Dh = 0100 1101b). `AND AL, 0DFh` clears this bit.

```
org 100h
cld               ; DF = 0: forward
lea si, msg
mov di, si        ; write in place
mov cx, 5
next_ch:
lodsb             ; AL = [SI], SI++
and al, 0DFh      ; upper case
stosb             ; [DI] = AL, DI++
loop next_ch
ret
msg db 'micro'
```

The code is 16 bytes and `msg` sits at 0110h. `MOV DI, SI` (8B FE) puts the destination at the same place as the source. In a .COM program ES = DS, so DS:SI and ES:DI point to the same byte. LODSB advances the read address by one, and STOSB, one step behind, overwrites the same byte. Table 11.21 gives the first round in detail and the later rounds in summary.

Instruction	AL	SI	DI	CX	msg
`CLD`, `LEA SI, msg`	00h	0110h	0000h	0000h	`micro`
`MOV DI, SI`, `MOV CX, 5`	00h	0110h	0110h	0005h	`micro`
`LODSB`	**6Dh** ('m')	**0111h**	0110h	0005h	`micro`
`AND AL, 0DFh`	**4Dh** ('M')	0111h	0110h	0005h	`micro`
`STOSB`	4Dh	0111h	**0111h**	0005h	`Micro`
`LOOP next_ch`	4Dh	0111h	0111h	**0004h**	`Micro`
end of round 2	49h ('I')	0112h	0112h	0003h	`MIcro`

Instruction	AL	SI	DI	CX	msg
end of round 3	4Bh ('K')	0113h	0113h	0002h	`MICro`
end of round 4	52h ('R')	0114h	0114h	0001h	`MICRo`
end of round 5	4Fh ('O')	0115h	0115h	0000h	`MICRO`

Table 11.21: Trace of the LODSB / STOSB loop.

In this program only AND changes the flags: it sets CF and OF to 0, sets SF and ZF according to the result, and sets PF according to the parity of the result. Because 4Dh = 0100 1101b has four 1s, PF = 1 in the first round; because 49h = 0100 1001b has three 1s, PF = 0 in the second. LODSB, STOSB, and LOOP leave these flags as they are. The program also works correctly on a text that starts with a capital: 'M' is already a letter whose bit 5 is 0 and AND does not change it. But care is needed with characters that are not letters. For example a digit ('1' = 31h) comes out of this operation as 11h; in a real program one first tests whether the character is in the range 'a'–'z'.

The time of each round is LODSB (12) + AND AL, imm8 (4) + STOSB (11) + LOOP (17) = 44 cycles. Since the last LOOP takes 5 cycles, the total is 5 × 27 + 4 × 17 + 5 = 208 clock cycles. REP cannot be put on this loop, because REP repeats only a single string instruction; since an AND operation is needed in between, the loop is built with LOOP.

11.7.6 Summing a word array with LODSW

In the W versions the step is 2. The following program reads an array of four words with LODSW and adds them up in BX.

```
org 100h
cld
lea si, arr
mov cx, 4
xor bx, bx        ; sum = 0
top:
lodsw             ; AX = [SI], SI += 2
add bx, ax
loop top
ret
arr dw 100h, 200h, 300h, 400h
```

Tur	address read by LODSW	AX	SI	BX	CX (after LOOP)
1	010Fh	0100h	0111h	0100h	0003h
2	0111h	0200h	0113h	0300h	0002h
3	0113h	0300h	0115h	0600h	0001h
4	0115h	0400h	0117h	**0A00h**	0000h

Table 11.22: Trace of the LODSW sum (the state after the ADD of each round). `arr` starts at 010Fh.

Since the code takes 15 bytes, the array starts at an odd address, 010Fh. Each LODSW increments SI by 2 and moves to the next word of the array. The result is BX = 0A00h (256 + 512 + 768 + 1024 = 2560 = 0A00h). Because the array is at an odd address, each LODSW needs two separate byte bus cycles. Had an `EVEN` directive been put in front of the array, the array would shift to 0110h and each word would be read in a single cycle.

11.7.7 CMPS and SCAS: comparison

CMPS and SCAS perform a subtraction as CMP does, discard the result, and set the six status flags. The order of the operands matters for interpreting CF and SF in the result:

- **CMPS**: [DS:SI] − [ES:DI]. Source minus destination. Comparing 'R' (52h) with 'S' (53h) gives 52h − 53h = FFh: CF = 1, SF = 1, ZF = 0. CF = 1 shows, in the unsigned interpretation, that the source is less than the destination.
- **SCAS**: AL (or AX) − [ES:DI]. The searched value minus memory. With AL = 'S' and 'R' in memory, 53h − 52h = 01h: CF = 0, ZF = 0.

This order may look contrary to the general rule of Intel syntax that the destination is written on the left: in CMPS the left side is the source. After CMPS, unsigned jumps such as JA and JB are read as "the source is greater/less than the destination". SCAS advances only DI, because its source operand is a register. CMPS, like MOVS, addresses two memory strings at once and advances SI and DI together.

11.8 Repeat prefixes

11.8.1 REP, REPE, and REPNE

The real power of a string instruction shows when it is combined with a **repeat prefix**. The prefix repeats the string instruction that follows until the CX counter reaches zero or until a condition fails. Table 11.23 gives the three prefixes and their continuation conditions.

Prefix	Byte	Continuation condition	Use
`REP`	`F3`	CX ≠ 0	MOVS, STOS (meaningless with LODS)
`REPE / REPZ`	`F3`	CX ≠ 0 and ZF = 1	CMPS, SCAS: "continue while equal"
`REPNE / REPNZ`	`F2`	CX ≠ 0 and ZF = 0	CMPS, SCAS: "continue while not equal"

Table 11.23: Repeat prefixes.

REP and REPE are the same byte (F3h). The processor tells the difference not from the prefix but from the instruction that follows: since MOVS, LODS, and STOS produce no flags, the ZF condition is not tested for them; for CMPS and SCAS it is. Bit 0 of F2h is 0 and that of F3h is 1, and this bit says which value ZF must have for the repetition to continue. The execution of a prefixed string instruction follows this order:

1. If CX = 0, stop. The instruction may not be executed at all.
2. If an interrupt is pending, accept it (the return address is the address of the prefix).
3. Execute the string instruction once; update SI and DI.
4. CX ← CX − 1. This decrement does not change the flags.
5. If it is REPE or REPNE and the instruction is CMPS or SCAS, test ZF; if the condition is not met, stop. If it is met, go back to step 1.

The importance of the first step is great. A `REP MOVSB` that starts with CX = 0 copies no bytes at all. LOOP behaves the opposite way: a LOOP loop entered with CX = 0 first brings CX down to FFFFh and runs 65,536 rounds. This difference lets REP be used safely on a block whose length is computed: if the length comes out 0, nothing happens.

Definition 11.7

A **repeat prefix** is a prefix that makes the string instruction that follows repeat CX times or until a ZF condition fails. CX decreases by one with every repetition. When the instruction finishes, CX shows the number of repetitions not performed, and SI and DI point one past the last element processed.

11.8.2 Block copy and filling the screen

The two most common uses of REP are copying a block of memory and filling a block with a single value. The first piece below copies 100 bytes and the second fills the whole screen with blanks.

```
; copy a 100-byte block
cld
lea si, src
lea di, dst
mov cx, 100
rep movsb

; fill the screen with blanks
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h   ; ' ' + attribute
mov cx, 2000    ; 80×25
rep stosw
```

In the first piece REP MOVSB runs a hundred times. At the end CX = 0, SI and DI are 100 (64h) more than their starting values, and a copy of the source block is at the destination. When this piece is put into a program and run in emu8086, for example if the source starts at 010Dh and the destination at 0171h, SI = 0171h and DI = 01D5h are seen at the end. According to the 8086 manual the time is 9 +

17 × 100 = 1709 clock cycles. Had the same block been copied with the word version (`MOV CX, 50` / `REP MOVSW`), the time would fall to 9 + 17 × 50 = 859 cycles; a MOVSW round takes the same as a MOVSB round but moves two bytes each time.

The second piece fills the screen memory in text mode. The color text screen starts at B800:0000. Each character cell is two bytes: at the even address the ASCII code of the character, at the odd address the attribute (color) byte. The low byte of AX = 0720h is AL = 20h (blank) and the high byte is AH = 07h (light gray on black). Because STOSW writes AX in little-endian order, 20h falls on the even address and 07h on the odd one; each STOSW therefore clears one cell completely. For 80 × 25 = 2000 cells, CX = 2000. When the instruction finishes DI = 2000 × 2 = 4000 = 0FA0h, CX = 0, and the last cell of the screen holds the values 20h, 07h at B8F9Eh–B8F9Fh. The time is 9 + 10 × 2000 = 20,009 cycles; at 5 MHz about 4 ms.

11.8.3 Code trace: searching for a character with REPNE SCASB

REPNE SCASB searches a string for a character: the condition "continue while not equal" stops when the character is found or when CX runs out. The following program searches the text 'pushing' for the letter 'h' and puts the position it finds (counting from 0) in DI.

```
org 100h
cld
lea di, msg
mov cx, 7          ; length
mov al, 'h'        ; target
repne scasb        ; search
jne not_found
dec di             ; match position
sub di, offset msg
not_found:
ret
msg db 'pushing'
```

The code is 19 bytes and `msg` starts at 0113h. `REPNE SCASB` is two bytes (F2 AE). Table 11.24 shows the four repetitions of the instruction separately. In each repetition AL is compared with the byte at ES:DI, DI is incremented by one, and CX is decremented by one.

Instruction	Comparison	DI	CX	ZF	CF	Explanation
`CLD`		0000h	0000h	0	0	forward
`LEA DI, msg`		**0113h**	0000h	0	0	BF 13 01
`MOV CX, 7`		0113h	**0007h**	0	0	at most 7 comparisons
`MOV AL, 'h'`		0113h	0007h	0	0	AL = 68h
`REPNE SCASB` (1)	68h – 'm' (6Dh)	0114h	0006h	0	1	not equal → continue
`REPNE SCASB` (2)	68h – 'e' (65h)	0115h	0005h	0	0	continue

Instruction	Comparison	DI	CX	ZF	CF	Explanation
`REPNE SCASB` (3)	68h − 'r' (72h)	0116h	0004h	0	1	continue
`REPNE SCASB` (4)	68h − 'h' (68h)	**0117h**	**0003h**	**1**	0	equal → stop
`JNE not_found`		0117h	0003h	1	0	ZF = 1 → no jump
`DEC DI`		**0116h**	0003h	0	0	address of the match
`SUB DI, OFFSET msg`		**0003h**	0003h	0	0	'h' at index 3

Table 11.24: Trace of the REPNE SCASB program. Each repetition of REPNE SCASB is shown on a separate row.

In the fourth repetition 'h' is found: 68h − 68h = 0 and ZF = 1. Because the REPNE condition (ZF = 0) fails, the instruction stops. DI points one past the match (0117h), because the update is made after the comparison. CX = 3 is the number of characters of the string left unscanned. Since there can be two reasons for the instruction to finish, a test is needed afterwards. If ZF = 1 the character has been found. If ZF = 0, CX has run down to zero and the whole string was scanned in vain; in that case `JNE not_found` jumps. Testing CX = 0 alone is not enough: if the searched character is exactly in the last position, both CX = 0 and ZF = 1.

`DEC DI` takes DI back to the match itself: 0116h. `SUB DI, OFFSET msg` subtracts the start address and gives 0116h − 0113h = 3: 'h' is at index 3, counting from 0. DEC and SUB change the flags according to their own results; since the result is not zero, ZF returns to 0. The time of the instruction is 9 + 15 × 4 = 69 clock cycles.

11.8.4 Comparing two strings with REPE CMPSB

REPE CMPSB compares two strings "as long as they are equal" and stops at the first difference. The following program compares the first six characters of the strings 'ANKARA' and 'ANTALY' and leaves the index of the first difference in SI.

```
org 100h
cld
lea si, str1
lea di, str2
mov cx, 6
repe cmpsb
je same            ; all equal
dec si
sub si, offset str1 ; index of first difference
same:
ret
str1 db 'ANKARA'
str2 db 'ANTALY'
```

The code is 20 bytes; `str1` starts at 0114h and `str2` at 011Ah. The first repetition compares 'A' with 'A': difference 0, ZF = 1, SI = 0115h, DI = 011Bh, CX = 5. The second repetition compares 'N' with 'N': ZF = 1, CX = 4. In the third repetition 'K' (4Bh) − 'T' (54h) = F7h. ZF = 0 and REPE stops. At this moment SI = 0117h, DI = 011Dh, CX = 3. To interpret the result as unsigned, look at CF: CF = 1, that is 'K' < 'T'. This says that ANKARA comes before ANTALY in dictionary order. `JE same` does not jump; `DEC SI` and `SUB SI, OFFSET str1` give 0116h − 0114h = 2: the first difference is at index 2.

Here too there are two reasons for the instruction to finish. Had the strings been completely equal, all six repetitions would end with ZF = 1 and CX would be 0; then `JE same` would jump. A password or command-name comparison is done with this pattern.

11.8.5 Overlapping blocks and choosing the direction

If the source and destination overlap in memory, the direction of copying changes the result. Suppose we want to shift a text two bytes to the right: `msg db '12345', 2 dup('.')`. The destination is two bytes to the right of the source and the two blocks overlap in three bytes.

```
std                 ; backward
lea si, msg+4      ; last byte of the source
lea di, msg+6      ; last byte of the destination
mov cx, 5
rep movsb
```

With STD the copy proceeds from end to start: first '5' is written to position 6, then '4' to position 5, and '3' to position 4. Every source byte is read before anything is written over it. The result is '1212345': the text has shifted right by two and the old characters remain in the first two positions. Had the same job been done with CLD, giving SI = msg and DI = msg+2, the first step would write '1' to position 2, that is over the '3'. The third step would read this new '1' and carry it forward. The result would be '1212121' and the text would be corrupted. Table 11.25 compares the two results.

Direction	Initial SI / DI	Final SI / DI	Result
STD (correct)	0111h / 0113h	010Ch / 010Eh	`1212345`
CLD (wrong)	010Dh / 010Fh	0112h / 0114h	`1212121`

Table 11.25: Copying the text '12345..' two bytes to the right. When the program starts at 0100h, `msg` is at 010Dh.

Rule (overlapping copy)

If the destination overlaps on the right of the source (at a higher address), the copy is done from end to start: STD, with SI and DI pointing to the last element of the blocks. If the destination overlaps on the left of the source, it is done from start to end: CLD, with SI and DI pointing to the first element of the blocks. If the blocks do not overlap, either direction is correct.

11.8.6 Timing, interrupts, and hardware

A string instruction with REP is distinctly faster than a loop doing the same job. Table 11.26 gives a few comparisons according to the timings in the Intel 8086 manual. The source of the gain is that the loop control (decrementing CX, testing, jumping) is done inside the microcode and that the prefetch queue is not flushed on every round. When a LOOP or JMP is taken, the queue is flushed and the BIU has to start reading again from the new address; an instruction with REP never changes IP.

Task	With a loop	With REP
copy 4 bytes	MOVSB + LOOP: 128	REP MOVSB: 77
copy 100 bytes	MOVSB + LOOP: 3488	REP MOVSB: 1709; REP MOVSW (50 words): 859
fill 2000 words	–	REP STOSW: 20.009
search for the character at position 4 in 7 bytes	–	REPNE SCASB: 69

Table 11.26: Timings of string jobs (clock cycles). For an instruction with REP the time is 9 + n × the time per round.

A long REP instruction can take thousands of cycles. The 8086 does not hold interrupts off during this time. At the start of each repetition it looks at whether an interrupt is pending. If there is one, it abandons the work done so far; SI, DI, and CX keep their current values. The return address pushed onto the stack is the address of the prefix. When the interrupt handler returns with IRET, the processor restarts the prefixed instruction and the work continues with the remaining CX value. This scheme keeps interrupt latency short but also brings the limitation described above: since the 8086 remembers only one prefix, REP and a segment prefix should not be used together.

Common mistake

Using REP with LODS or REPE with MOVS. `REP LODSB` is valid code but overwrites AL in every repetition; only the last byte remains in AL at the end. `REPE MOVSB` is the same as REP MOVSB, because MOVS produces no ZF. The pairing of prefix and instruction: REP for MOVS and STOS, REPE or REPNE for CMPS and SCAS.

11.9 Control transfer: the encoding of JMP

11.9.1 Kinds of JMP

The unconditional jump JMP loads a new value into IP (and also into CS in a far jump). The 8086 has five encoding forms of JMP. The forms are distinguished by how the target is given: as an offset inside the instruction, as an absolute address inside the instruction, or in a register or memory. Table 11.27 gives all five and their times.

Type	Kod	Range	Example	Time
Short	`EB d8`	−128 … +127 bytes	`JMP SHORT again`	15
Near	`E9 d16`	same segment, ±32 K	`JMP start`	15
Far	`EA IP CS`	any segment	`JMP FAR PTR target`	15
Indirect near	`FF /4`	IP ← register or memory	`JMP BX`, `JMP table[BX]`	11 / 18 + EA
Indirect far	`FF /5`	CS:IP ← memory (4 bytes)	`JMP DWORD PTR [SI]`	24 + EA

Table 11.27: The encoding forms of JMP. Times are in clock cycles.

Short and near jumps are encoded **relative**: the instruction carries not the address of the target but the distance of the target from the next instruction. The processor adds this distance to IP. Since IP already points to the next instruction while the instruction is executing, the calculation is: target = address of the next instruction + displacement. In the short form the distance is a single byte and is sign-extended to 16 bits. A far jump, on the other hand, carries the absolute address of the target: first IP (2 bytes), then CS (2 bytes), both little-endian. A far JMP is therefore five bytes. `JMP FAR PTR far_lbl` is encoded as EA 56 02 00 07 if the target is 0700:0256h.

The indirect forms belong to the FFh family, and the REG field of ModR/M says which operation is performed: /4 is indirect near JMP, /5 indirect far JMP. `JMP BX` = FF E3 (11 100 011: MOD = 11, REG = 100, R/M = BX). `JMP DWORD PTR [SI]` = FF 2C (00 101 100: REG = 101, R/M = [SI]). In these two lines it is plain that the REG field shows not a register but the type of the operation.

11.9.2 Computing the relative displacement

Figure 11.6 shows a short jump. The `EB 10` at 0120h is two bytes; the next instruction is at 0122h. The target is 0122h + 10h = 0132h.

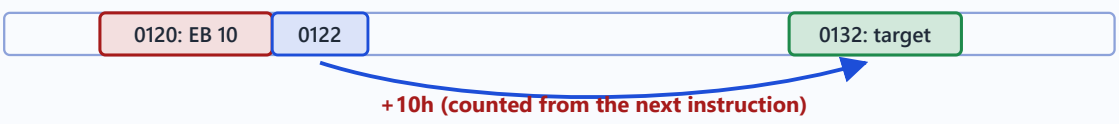


Figure 11.6: Relative jump: target = address of the next instruction + displacement. For EB 10 at 0120h,

0122h + 10h = 0132h.

The reverse calculation is what the assembler does for every jump: displacement = target − address of the next instruction. If the result is negative it is written in two's complement. Three examples:

1. **Jump to itself.** If a two-byte short JMP at 010Ah jumps to itself, the next instruction is at 010Ch. 010Ah − 010Ch = −2 = FEh. The code is **EB FE**; it is the endless loop that stops a program where it is.
2. **Backward jump.** If the short JMP at 0150h jumps to 0130h, the next instruction is at 0152h. 0130h − 0152h = −22h. The two's complement of −22h is 100h − 22h = DEh. The code is **EB DE**.
3. **Near jump.** If a JMP at 0122h goes to a label at 0251h, the distance does not fit the short range. The near form is three bytes and the next instruction is at 0125h. 0251h − 0125h = 012Ch. The code is **E9 2C 01**.

Relative encoding has an important consequence: when the code is moved to another address in memory, the bytes of short and near jumps do not change, because the jumping instruction and its target shift together. This is convenient for a system such as DOS that does not know in advance where the program will be loaded. A far jump, however, carries an absolute CS value, so in .EXE programs DOS has to fix it up during loading. On the 8086 the conditional jumps (Jcc) have only the short form. If the target is outside the −128…+127 range, the assembler inverts the condition and places a near JMP behind it.

Every jump pays one more cost besides its time. When a jump is taken, the bytes in the BIU's prefetch queue are no longer valid, because the next instruction is somewhere else. The queue is flushed and the BIU starts reading from the new CS:IP. Most of the 15-cycle time of JMP comes from this refill. That conditional jumps take only 4 cycles when not taken is also because the queue is preserved.

11.9.3 Indirect jumps and the jump table

The indirect near JMP makes it possible to choose the target at run time. A menu or a state machine keeps the addresses of the possible targets in an array of words and jumps to one of them with an index. This structure is called a **jump table**; it is the machine-level counterpart of the `switch` statement of high-level languages.

```
mov bx, 2           ; choice (0, 1, 2)
shl bx, 1           ; word index: × 2
jmp table[bx]       ; IP ← table[BX]
table dw s0, s1, s2
s0: mov dx, 0
    ret
s1: mov dx, 1
    ret
s2: mov dx, 2
    ret
```

Every element of the table is two bytes; the choice number is therefore multiplied by two with SHL. When the program starts at 0100h, `JMP table[BX]` is encoded as FF A7 09 01: ModR/M A7h = 10 100 111b (MOD = 10, REG = 100 = /4, R/M = [BX+d16]), then the address of the table, 0109h. With BX = 4 the processor reads the word at DS:010Dh; there is the address of s2, 0117h. IP ← 0117h and the program ends with DX = 2. It must be tested beforehand that the choice number does not exceed the table size; otherwise the code bytes behind the table are taken for addresses and the processor jumps to a random place.

11.10 Processor-control instructions

11.10.1 Flag instructions

Processor-control instructions do no calculation and move no data; they change the state of the processor. Some of them set or clear a single flag; all of these flag instructions are single bytes and take 2 clock cycles. Table 11.28 gives all the instructions with their codes and times.

Instruction	Kod	Effect	Time
`CLC` / `STC`	`F8` / `F9`	CF ← 0 / CF ← 1	2
`CMC`	`F5`	CF ← ¬CF	2
`CLD` / `STD`	`FC` / `FD`	DF ← 0 / 1 (string direction)	2
`CLI` / `STI`	`FA` / `FB`	IF ← 0 / 1 (INTR interrupts)	2
`HLT`	`F4`	stop until an interrupt or RESET arrives	2
`WAIT`	`9B`	wait until the TEST̄ pin becomes active (8087 coordination)	3 + 5n
`ESC`	`D8` – `DF`	pass the instruction to the coprocessor (8087)	2 / 8 + EA
`LOCK`	`F0`	prefix: for the duration of the next instruction the LOCK̄ pin is active, the bus is locked	2
`NOP`	`90`	do nothing	3

Table 11.28: Processor-control instructions.

Each flag instruction touches a single flag; the others do not change. The reason such small instructions exist separately is that the program itself deliberately sets these three flags (CF, DF, IF). CF can be cleared before starting a multiword addition (CLC, then a chain of ADC). DF selects the direction before string instructions. IF determines whether interrupts are accepted. There are no separate set/clear instructions for OF, SF, ZF, AF, and PF; they are written with POPF (and, except for OF, with SAHF).

A common use of CLC and STC is returning success or failure information from a procedure. If the procedure finds an error it returns with STC, if not with CLC. The caller decides right afterwards with `JC error` or `JNC ok`. Most of the INT 21h services of DOS report errors this way too. The following procedure divides AX by BL, but if BL = 0 it returns with CF = 1 without causing a divide error:

```
divide:
    cmp bl, 0
    jne ok
    stc          ; error: divisor is zero
    ret
ok:
    div bl       ; AL = quotient, AH = remainder
    clc          ; success
    ret
```

When called with AX = 7 and BL = 0 the procedure returns with CF = 1 and AX unchanged. When called with AX = 7 and BL = 3, DIV produces AL = 2 (quotient) and AH = 1 (remainder); AX = 0102h and the return is with CF = 0. Because DIV leaves all the flags undefined, CLC is needed here: without it the value of CF would be unpredictable. Since RET does not change the flags, CF reaches the caller as it is.

11.10.2 The interrupt flag: CLI and STI

IF determines whether the processor accepts the maskable interrupts coming from the INTR pin. When IF = 0 is made with CLI, INTR is ignored; the non-maskable interrupt from the NMI pin and interrupts called by software with INT n still work. The region between CLI and STI is a critical region in which an interrupt must not see a half-finished data structure. The typical example is moving the stack:

```
cli              ; interrupts off
mov ss, ax
mov sp, 0FFFEh
sti              ; interrupts on
```

If an interrupt arrives when SS has changed but SP still has its old value, the processor pushes FLAGS, CS, and IP to a wrong address and corrupts memory. The 8086 already holds off interrupts for the single instruction after one that writes SS. But this protection is only one instruction long; the CLI/STI pair shows the intent openly and is safe even when there are more instructions in between. STI has a similar property: interrupts are accepted after the instruction following STI finishes. So in the `STI` / `IRET` pair at the end of an interrupt handler, no new interrupt can slip in before IRET executes.

11.10.3 HLT, WAIT, ESC, LOCK, and NOP

These instructions manage the processor's relation to the outside world and are directly tied to the pins of the 8086.

- **HLT** (*halt*) puts the processor into the halt state. No instruction is executed and no bus cycle is started; the processor waits without fetching the next instruction. There are three ways out of the state: an interrupt from the INTR pin (if IF = 1), an interrupt from the NMI pin, or RESET. When it leaves through an interrupt, the address of the instruction after HLT is pushed onto the stack; when the handler returns, the program continues after the HLT. That is why HLT is used to mean "wait for the next interrupt". An HLT after CLI makes the processor wakeable only by NMI or RESET.
- **WAIT** polls the TEST̄ input of the 8086. As long as the pin is 1 the processor keeps waiting and samples the pin again every five clock cycles. This pin is usually connected to the BUSY output of the 8087 floating-point coprocessor. WAIT keeps the 8086 from reading the result of the 8087 before it has finished a calculation.
- **ESC** (*escape*) instructions are those beginning with 11011xxx (D8h–DFh). The 8086 does not execute these instructions itself. If there is a memory operand, it computes the effective address and does a read from that address. The 8087, which watches the bus, captures both the instruction and the address and does the calculation itself. The three bits in the opcode and the REG field of ModR/M combine to give the six-bit instruction code of the 8087.
- **LOCK** is a prefix. As long as the instruction that follows it is executing, the LOCK̄ output of the 8086 (in maximum mode) is held at 0. The bus arbiter does not let another processor or a DMA controller take the bus during this time. An instruction made of a read and a write thus completes without another bus master slipping in between.
- **NOP** does nothing and takes 3 cycles. Its code is 90h, that is it is actually `XCHG AX, AX`. It is used for short delays in timing loops, for filling gaps in code alignment, and in debugging for disabling an instruction without deleting it.

The best-known use of LOCK is building a **semaphore** to protect a resource shared by several processors. If the `mutex` byte in memory is 0 the resource is free. To take the resource a processor executes `LOCK XCHG AL, mutex` with AL = 1. The exchange is done in a single instruction and, because of LOCK, no other processor can slip in. If the old value that arrives in AL is 0, the resource has been taken; if it is 1, someone else has it and the loop tries again:

```
wait_lock:
    mov al, 1
    lock xchg al, mutex   ; F0 86 06 …
    test al, al
    jnz wait_lock           ; someone else has it
    ; critical section
    mov mutex, 0            ; release the resource
```

Had the read and write been done with two separate instructions (`MOV AL, mutex` / `MOV mutex, 1`), two processors could read 0 at the same time and both think they had taken the resource. On a single-processor system CLI/STI gives the same protection; LOCK is for multiprocessor systems.

11.11 Flag effects

11.11.1 Computational instructions

The correct working of a program often depends on knowing which instruction changes which flag. Conditional jumps look at the result of the last instruction that changed the flags; any instruction in between can spoil this result. Table 11.29 gives the effect of the arithmetic, logic, and shift instructions on the six status flags.

Instruction	OF	SF	ZF	AF	PF	CF
`ADD ADC SUB SBB CMP NEG`	*	*	*	*	*	*
`CMPS SCAS`	*	*	*	*	*	*
`INC DEC`	*	*	*	*	*	–
`MUL IMUL`	*	?	?	?	?	*
`DIV IDIV`	?	?	?	?	?	?
`DAA DAS`	?	*	*	*	*	*
`AAA AAS`	?	?	?	*	?	*
`AAM AAD`	?	*	*	?	*	?
`AND OR XOR TEST`	0	*	*	?	*	0
`SHL SAL SHR SAR`	*[1]	*	*	?	*	*
`ROL ROR RCL RCR`	*[1]	–	–	–	–	*

Table 11.29: Flag effects of the computational instructions. * varies with the result · 0 cleared · – unchanged · ? undefined · [1] OF is defined only when the shift count is 1. DF, IF, and TF are not affected by these instructions.

A few rows of the table need special attention. INC and DEC preserve CF; this property is designed so that incrementing a counter or an index in a multiword addition loop does not spoil the carry of ADC. MUL and IMUL set only CF and OF meaningfully: together they say whether the upper half of the result is needed. After DIV and IDIV no flag can be trusted. Logic instructions always set CF and OF to 0. `TEST AX, AX` is therefore a frequently used test: it does not change AX, sets ZF, SF, and PF according to AX, and clears CF. Rotate instructions change only CF and OF. In shifts and rotates, if the count is 0 (CL = 0) no flag changes.

An "undefined" (?) flag means that the processor writes some value to that flag but this value is not documented. In emu8086 or on a real processor a particular value is seen in this flag, but it may be different on another processor model. A correctly written program does not decide by looking at an

undefined flag.

11.11.2 Other instructions

Table 11.30 collects the effects of the remaining instructions on the flags. The main message of this table is that the great majority of instructions do not touch the flags at all.

Instructions	Flags affected
`MOV XCHG LEA LDS LES XLAT PUSH POP IN OUT LAHF PUSHF CBW CWD NOT`	none
`MOVS LODS STOS`, `REP`	none
`JMP Jcc LOOP LOOPE LOOPNE JCXZ CALL RET`	none (only read)
`SAHF`	SF ZF AF PF CF ← AH
`POPF`, `IRET`	all (from the stack)
`INT n`, `INTO`	IF ← 0, TF ← 0 (old FLAGS on the stack)
`CLC STC CMC`	CF
`CLD STD`	DF
`CLI STI`	IF

Table 11.30: Flag effects of the other instructions.

Two rows may come as a surprise. NOT is a logic instruction but changes no flag; a program that wants to set the flags uses `XOR AX, 0FFFFh` instead. LOOP also decrements CX but, unlike DEC, does not touch the flags. LOOPE and LOOPNE read ZF and do not write it. INT n pushes the flags onto the stack before changing them; since IRET restores them, the flags of the program are back to their old state after returning from the interrupt handler.

11.11.3 What can come between CMP and a conditional jump?

The place where the flag tables are most often used is between a comparison and the conditional jump that depends on it. If an instruction that changes flags comes in between, the jump looks at the result of that instruction, not of the comparison. In the following piece AL = 20 and BX = FFFFh:

```
cmp al, 10
add bx, 2       ; bug: corrupts the flags
jb small         ; if AL < 10
mov dx, 1
ret
small:
mov dx, 2
ret
```

CMP computes 20 − 10 and produces CF = 0; AL is not less than 10 and JB must not jump. But the `ADD BX, 2` in between computes FFFFh + 2 = 1 0001h: the result does not fit in 16 bits and CF = 1. JB looks at this CF, jumps, and the program ends in the wrong branch with DX = 2. If `LEA BX, [BX+2]` is written in place of ADD, BX is again 0001h but the flags do not change; JB sees the result of CMP, does not jump, and DX = 1, the correct result, comes out. The general rule: between CMP and Jcc only instructions that do not affect the flags, such as MOV, LEA, XCHG, PUSH, and POP, can come. If a counter has to be incremented, either move it before the CMP or do it with LEA.

11.12 Common mistakes

This section lists in one place the mistakes met throughout the note. After each, it is given why it occurs and how to prevent it.

- **Taking the D bit for "the left operand".** D = 1 says that the register in the REG field is the destination. In two-register instructions both encodings are valid (8B C3 and 89 D8); in instructions with memory, which side the memory is on decides D.
- **Skipping the W bit and reading the wrong register.** REG = 100 is AH when W = 0 and SP when W = 1. In decoding, look at W first.
- **Encoding [BP] with MOD = 00.** MOD = 00 and R/M = 110 is the direct address. [BP] is encoded with MOD = 01 and d8 = 0 and is three bytes.
- **Starting decoding at a wrong address.** A one-byte shift changes all the boundaries; decoding starts from the start of a program or from a known jump target.
- **Forgetting little-endian order.** The displacement, the immediate, and each word of a far pointer are written low byte first. In a far pointer the offset comes first, then the segment.
- **Confusing LEA with MOV.** `LEA AX, [BX+DI+2]` gives the address, `MOV AX, [BX+DI+2]` the contents.
- **Assuming DF.** Before a string instruction the direction is set explicitly with CLD or STD.
- **Not setting ES in an .EXE program.** The string destination is always ES:DI, and DOS does not set ES to the data segment.
- **Looking only at CX after REPNE/REPE.** ZF tells why the instruction stopped; when the searched character is in the last position, CX = 0 and ZF = 1 occur together.
- **The wrong direction with overlapping blocks.** If the destination is to the right of the source, copy from end to start with STD.
- **Putting a flag-corrupting instruction between CMP and Jcc.** Instructions such as INC, ADD, SUB, and AND set the flags anew; only instructions that do not affect flags may come in between.
- **Trusting an undefined flag.** ZF after MUL, any flag after DIV, and OF after a shift by more than 1 are undefined.

11.13 Exercises and solutions

The nine exercises in this section are the exercises at the end of the slides. The solution to each is given step by step with the method described in the note.

Exercise 1. Encode `MOV DX, [BP+DI+8]` (opcode 100010, D = 1, W = 1).

Solution. The first byte is 100010 1 1 = 8Bh. The destination DX gives REG = 010. The R/M code of the form [BP+DI+d] is 011. Since 8 fits in one byte, MOD = 01. ModR/M = 01 010 011b = 53h. The displacement is 08h. The code is **8B 53 08**; because it contains BP, the default segment is SS.

Exercise 2. With DS = 1000h, BX = 0200h, AL = 05h, the byte at which physical address does `XLAT` load into AL?

Solution. EA = BX + AL = 0200h + 05h = 0205h. The physical address is 1000h × 10h + 0205h = **10205h**.

Exercise 3. The bytes of the variable `far_ptr dd` in memory are 34 12 00 B8. What are ES and DI after `LES DI, far_ptr`?

Solution. In a far pointer the offset is stored first, then the segment. The first word 34 12 → 1234h, the second word 00 B8 → B800h. **DI = 1234h, ES = B800h**.

Exercise 4. How many bytes does `REP MOVSB` copy when CX = 0?

Solution. **None**. REP tests CX before every repetition; if CX = 0 the string instruction is not executed even once. SI, DI, and memory do not change.

Exercise 5. CLD, CX = 5, SI = 0100h, DI = 0200h; `REPE CMPSB` finds a difference at the 3rd byte. What are CX, SI, DI, and ZF afterwards?

Solution. Three comparisons are made: the first two are equal (ZF = 1), the third differs (ZF = 0) and REPE stops. In each comparison SI and DI increase by one and CX decreases by one: **CX = 2, SI = 0103h, DI = 0203h, ZF = 0**. The differing byte is at SI − 1 = 0102h.

Exercise 6. The `JMP SHORT` at address 0150h jumps to 0130h. What is the machine code?

Solution. The instruction is two bytes; the next instruction is at 0152h. The distance is 0130h − 0152h = −22h. In one byte −22h = 100h − 22h = DEh. The code is **EB DE**.

Exercise 7. Write code that counts the spaces in a string with `REPNE SCASB`.

Solution. REPNE SCASB stops when it finds a space; DI points behind the space found and CX shows the remaining length. If the counter is incremented and the instruction run again, the search continues where it left off. The loop ends when the search finishes without finding (ZF = 0) or when the string is exhausted (CX = 0).

```
org 100h
cld
xor bx, bx          ; counter = 0
lea di, msg
mov cx, 19          ; length
mov al, ' '
search:
repne scasb
jne done            ; no spaces left
inc bx              ; one more space
jcxz done           ; string finished
jmp search
done:
ret
msg db 'a short sample text'
```

When run in emu8086 the program ends with BX = 3: 'a short sample text' is 19 characters and contains three spaces. The JCXZ test covers the case where the last character is a space. In that case REPNE SCASB finds the space in the last repetition; ZF = 1 and CX = 0. The counter is incremented and JCXZ leaves the loop. Without this test, the REPNE entered again with CX = 0 would not run at all and ZF would stay 1; JNE would not jump, the counter would increase without stopping, and the loop would never end. When the end of the text is changed to 'tex ' the program counts four spaces correctly.

Exercise 8. Which instruction are the bytes `8A 1E 34 12`? What happens if the same bytes are decoded shifted by one byte (from offset 1)?

Solution. 8Ah = 100010 1 0: D = 1, W = 0. 1Eh = 00 011 110: MOD = 00, REG = 011 (BL), R/M = 110 → direct address. Then 34 12 → 1234h. The instruction is `MOV BL, [1234h]`. When decoded from offset 1, 1Eh is the one-byte `PUSH DS`, and then 34 12 is the two-byte `XOR AL, 12h`.

Exercise 9. Encode `MOV ES, DX` and `MOV AL, [0020h]`.

Solution. `MOV ES, DX`: the 8Eh family; ModR/M = 11 000 010b = C2h (REG = 0 + 00 = ES, R/M = DX). The code is **8E C2**. `MOV AL, [0020h]`: the short form between AL and a direct address is A0h, followed by the address in little-endian: **A0 20 00**.

11.14 Summary

- **Instruction set**: the roughly one hundred instructions of the 8086 fall into six classes: data transfer, arithmetic, logic and shift, string, control transfer, and processor control. There is one mnemonic but many opcodes.
- **Instruction format**: prefixes + opcode (6 bits, D, W) + ModR/M (MOD, REG, R/M) + 0–2 bytes of displacement + 0–2 bytes of immediate. An unprefixed instruction is 1–6 bytes; numbers are written little-endian.

- **D and W**: W = 0 byte, W = 1 word; D = 1 REG is the destination, D = 0 REG is the source. Two-register instructions have two valid codes: `MOV AX, BX` = 8B C3 or 89 D8; `ADD AX, BX` = 03 C3 or 01 D8.
- **ModR/M**: MOD = 11 register, 00/01/10 memory (0/1/2 bytes of displacement). MOD = 00, R/M = 110 is the direct address; [BP] is therefore encoded with d8 = 0. The segment of forms containing BP is SS. The EA time is 5–12 cycles.
- **Short forms**: 40h+r, 50h+r, 90h+r, B0h+r, B8h+r, A0h–A3h, 04h/05h, 83h; one or two bytes shorter than the general form of the same job. In the 80h/81h/83h and FEh/FFh families the REG field selects the operation (/0 … /7).
- **Decoding**: from the starting point the first byte, ModR/M, displacement, and immediate are read in order. Decoding that starts at a wrong address turns the same bytes into other instructions; the processor does not distinguish code from data.
- **Data transfer**: does not change the flags (except SAHF and POPF). XLAT: AL ← [DS:BX + AL]. LEA gives the address, MOV the contents; in emu8086 LEA with a direct address is compiled as the three-byte MOV r16, imm16. LDS/LES load a far pointer (offset first, then segment).
- **String instructions**: MOVS, CMPS, SCAS, LODS, STOS; source DS:SI, destination ES:DI. DF = 0 (CLD) increments, DF = 1 (STD) decrements; the step is 1 for B and 2 for W. CMPS = [SI] − [DI], SCAS = AL/AX − [DI].
- **Repeat prefixes**: REP (F3h) while CX ≠ 0; REPE (F3h) and REPNE (F2h) with a ZF condition. If CX = 0 the instruction does not run at all. After it finishes, ZF says why the instruction stopped. With overlapping blocks, use STD if the destination is on the right.
- **JMP**: short (EB d8) and near (E9 d16) are relative; target = next instruction + distance. Far (EA IP CS) and indirect (FF /4, FF /5) use an absolute address. A jump flushes the prefetch queue.
- **Processor control**: CLC/STC/CMC, CLD/STD, CLI/STI are single-flag; HLT, WAIT, ESC, and LOCK work with the processor's pins (INTR/NMI/RESET, TEST̄, 8087, LOCK̄); NOP = XCHG AX, AX.
- **Flag effects**: moves, string moves, jumps, and NOT do not change flags; INC/DEC preserve CF; logic instructions make CF = OF = 0; the flags are undefined after DIV. No flag-affecting instruction should come between CMP and Jcc.

11.15 Key terms

The terms introduced in this chapter:

instruction set mnemonic machine code instruction format opcode ModR/M byte displacement immediate value effective address (EA) default segment prefix, segment override prefix little-endian prefetch queue data transfer instruction translation table (XLAT) far pointer string instruction direction flag (DF) repeat prefix (REP) short / near / far jump relative addressing processor control instruction clock cycle listing file

11.16 Test Yourself

1. What is the machine code of `MOV [BX+SI+0Ah], DX`?

> **Answer**
>
> DX is the source and goes into the REG field: D = 0, W = 1 → 89h. REG = 010 (DX), R/M = 000 ([BX+SI+d]), and since 0Ah fits in one byte, MOD = 01. ModR/M = 01 010 000b = 50h. The code is **89 50 0A**.

2. Which instruction are the bytes `C6 46 FE 07`? In which segment is the memory operand?

> **Answer**
>
> C6h is the MOV that moves an immediate byte to r/m8 (the REG field must be 000). 46h = 01 000 110b: MOD = 01 (d8), R/M = 110 → [BP+d8]. d8 = FEh = −2, and the immediate value is 07h. The instruction is `MOV BYTE PTR [BP-2], 7`; because it contains BP, the segment is SS. This is the pattern for assigning a value to a local variable in a stack frame.

3. Why is `MOV AL, [BX]` two bytes while `MOV AL, [BP]` is three bytes?

> **Answer**
>
> The combination MOD = 00 and R/M = 110 is reserved for the direct address. [BP] is therefore encoded with MOD = 01 and d8 = 00: 8A 46 00. For [BX] there is no such conflict: 8A 07.

4. The near JMP (E9) at 0200h jumps to 0100h. What is the machine code?

> **Answer**
>
> The instruction is three bytes; the next instruction is at 0203h. 0100h − 0203h = −103h; in 16 bits 10000h − 103h = FEFDh. Written little-endian: **E9 FD FE**.

5. `squares db 0, 1, 4, 9, 16, 25, 36, 49, 64, 81` is defined. What is AL after `LEA BX, squares`, `MOV AL, 7`, `XLAT`?

> **Answer**
>
> XLAT does AL ← [DS:BX + 7]; element 7 of the table (counting from 0) is 49. **AL = 49 = 31h**. The table method gives the squares of small numbers without a multiplication, in 11 cycles.

6. The bytes of the variable `ptr` in memory are 00 20 00 10. What are SI and DS after `LDS SI,`

`ptr`? Which physical address does DS:SI point to?

> **Answer**
>
> The first word is the offset: **SI = 2000h**; the second word is the segment: **DS = 1000h**. The physical address is 10000h + 2000h = **12000h**. From this moment on DS no longer points to the program's own data.

7. With CLD, SI = 0200h, DI = 0300h, CX = 3, how many bytes does `REP MOVSW` copy? What are SI, DI, and CX at the end?

> **Answer**
>
> Three words, that is **6 bytes**, are copied. In each repetition SI and DI increase by 2: **SI = 0206h, DI = 0306h, CX = 0**.

8. STD, DI = 0309h, CX = 10, AL = 'x'. Memory from 0300h to 0309h holds '0123x5x789'. What are DI, CX, and ZF after `REPNE SCASB`?

> **Answer**
>
> The scan proceeds from end to start: 0309h '9', 0308h '8', 0307h '7', 0306h 'x'. A match is found at the fourth comparison, ZF = 1, and the instruction stops. Since DI decreases after every comparison, **DI = 0305h**, **CX = 6**, **ZF = 1**. The address of the match is DI + 1 = 0306h; with STD the scan finds not the first 'x' but the last one.

9. Which of the following instructions can be placed between `CMP AX, BX` and `JE equal`: `INC CX`, `MOV CX, 0`, `XOR CX, CX`, `LEA CX, [BX+1]`, `PUSH AX`?

> **Answer**
>
> `MOV CX, 0`, `LEA CX, [BX+1]`, **and** `PUSH AX` do not change the flags and can be placed there. `INC CX` sets ZF, SF, OF, AF, and PF anew; `XOR CX, CX` makes ZF = 1 and makes JE always jump. This side effect of XOR, the short way to zero CX, leads to a bug here.

10. What is the difference in behavior, when CX = 0, between `REP STOSB` and `MOV CX, 0` / `again: STOSB` / `LOOP again`?

> **Answer**
>
> REP tests CX first; with CX = 0, STOSB does not run at all and memory is unchanged. In the loop with LOOP, STOSB runs once, LOOP brings CX down to FFFFh, and the loop runs 65,536 times; a 64 KB area starting at ES:DI is filled with AL.

11.17 Further reading

- Intel, *The 8086 Family User's Manual* (1979): the instruction set chapter gives the flag effect and clock-cycle time of each instruction, and the instruction encoding chapter gives the opcode tables and the ModR/M layout first hand.
- B. B. Brey, *The Intel Microprocessors* (8th edition), the chapters on data-transfer and string instructions: machine-language encoding, MOVS/STOS/LODS examples, and details that extend to later x86 processors.
- K. R. Irvine, *Assembly Language for x86 Processors* (7th edition): the chapter on string instructions and array processing; many examples of comparison with REPE CMPSB and table-driven search.
- emu8086 help documentation, "8086 instructions": a short description of each instruction, its valid operand forms, and the flag table; the explanations of the environment in which the examples in the slides were tried.

11.18 References

1. Intel, *The 8086 Family User's Manual*, 1979 — Chapter 2: the instruction set, Chapter 4: instruction encoding.
2. B. B. Brey, *The Intel Microprocessors*, 8th edition, Pearson, 2009.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7th edition, Pearson, 2014.
4. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.
5. emu8086 help documentation: "8086 instructions" and "8086 assembler tutorial".

APPENDIX A

Glossary of Terms

The terms used in the book, each with a short definition. The number next to a term is the chapter where it is explained in detail. A searchable version of the glossary is on the course page.

A

above / below 7
The outcomes of an unsigned comparison (JA, JB).

arithmetic instruction 6
Instructions that perform addition, subtraction, multiplication, and division.

B

BCD adjustment 6
The operation that converts the result of a binary addition or subtraction into valid BCD (DAA, AAA, and so on).

C

comparison (CMP) 7
The operation that subtracts two operands and sets only the flags.

D

data transfer instruction 11
Instructions that copy data among registers, memory, the stack, and I/O (MOV, XCHG, PUSH, and so on).

dividend / divisor / quotient / remainder 6
The elements of a division operation.

F

far pointer 11
A 4-byte address that holds an offset and a segment together; in memory the offset is stored first, then the segment (LDS, LES).

flow control 7
Changing the order in which instructions execute by means of jumps and loops.

G

greater / less 7
The outcomes of a signed comparison (JG, JL).

I

instruction format 11
The layout of the fields of a machine instruction (opcode, ModR/M, displacement, immediate data).

instruction set 11
All the instructions a processor can execute.

intrasegment / intersegment jump 7
A jump that changes only IP (short and near) versus a jump that also changes CS and moves to another segment (far).

J

jump 7
Continuing execution from a different address (JMP, Jcc).

jump displacement 7
The signed difference between the jump target and the address of the instruction after the jump.

L

label 7
A name given to an address and used as the target of a jump or call.

logical / arithmetic shift 6
Right shifts that put 0 into the vacated bit and that preserve the sign bit (SHR / SAR).

logical instruction 6
Instructions that perform bitwise AND, OR, XOR, NOT, and TEST operations.

loop 7
Repeating a group of instructions until a condition is met (LOOP, CX).

M

masking 6
Using AND/OR/XOR to clear, set, or toggle selected bits of a value.

ModR/M byte 11
The byte in instruction encoding, made up of the MOD, REG, and R/M fields, that specifies the operands and the addressing mode.

multiplication / division 6
The operation of the MUL/IMUL and DIV/IDIV instructions; the result is in AX or DX:AX.

N

nested loop 7
A loop contained in the body of another loop.

O

opcode 11
The part of a machine instruction that specifies which operation is to be performed.

P

packed / unpacked BCD 6
Formats that store two or one BCD digits in a byte.

prefix 11
A byte added in front of an instruction that changes its behavior (segment, REP, LOCK).

processor control instruction 11
Instructions that set or clear flags and halt the processor (CLC, STI, HLT, NOP, and so on).

R

repeat prefix (REP) 11
A prefix that makes a string instruction repeat CX times or until a condition fails.

rotate 6
A shift in which the bit that leaves re-enters at the other end (ROL, ROR, RCL, RCR).

S

shift 6
Shifting bits left or right (SHL, SHR, SAR); 0 or the sign bit enters the vacated bits.

short / near / far jump 11
Jumps made with an 8-bit distance, with a 16-bit distance within the same segment, and to another segment.

sign extension 6
Moving a signed value to a wider register by copying the sign bit (CBW, CWD).

string instruction 11
Instructions that operate on consecutive memory elements through DS:SI and ES:DI (MOVS, CMPS, SCAS, LODS, STOS).

T

translation table (XLAT) 11
The instruction and its table that replace AL with the entry in the table pointed to by BX.

U

unconditional / conditional jump 7
A jump that is always taken versus one taken only when the flags satisfy a condition.

APPENDIX B

Prerequisites for Microprocessors

MICROPROCESSORS · PREREQUISITES

Microprocessor programming brings together number systems, digital logic, computer architecture, assembly language, memory addressing, interrupts, stack-based execution, input/output and basic electronics.

Core preparation

The soundest preparation is hands-on familiarity with binary and hexadecimal numbers, Boolean logic, programming, the organization of the CPU and memory, and basic digital electronics.

NUMBERS · ARCHITECTURE · ASSEMBLY

Core prerequisites

These topics lay the groundwork for the 8086 programming model, registers and pins, memory access with segment:offset, arithmetic and logic instructions, procedures, the stack, interrupts, I/O ports and BIOS/DOS services.

BIN

The binary number system

Bits · powers of two · unsigned values

Microprocessors represent data and instructions as binary bit patterns.

TECHNICAL SIGNIFICANCE

Binary arithmetic is the foundation of registers, memory, opcodes, flags and digital logic.

- Converting between decimal and binary.
- Recognizing the powers of two.
- The ranges of 8-bit and 16-bit unsigned values.

Related topics: registers, opcodes, masks, addresses, flags.

HEX

The hexadecimal number system

Base 16 · a short way to write binary values

The hexadecimal system is a way of writing binary values compactly.

TECHNICAL SIGNIFICANCE

8086 addresses, machine-code bytes, register values and bit masks are usually written in hexadecimal.

- Converting decimal↔hexadecimal and binary↔hexadecimal.
- One hexadecimal digit corresponds to four bits.
- Reading byte and word values in hexadecimal.

Related topics: addresses, memory dumps, opcodes, debugging.

2C

Signed numbers and two's complement

Negative integers · overflow

Processors need a way to encode negative integers in binary.

TECHNICAL SIGNIFICANCE

Two's complement representation lets the same adder hardware serve both signed and unsigned arithmetic.

- The sign bit and the ranges of signed values.
- Finding the two's complement of a negative value.
- Telling carry apart from signed overflow.

Related topics: ADD/SUB, the sign flag (SF), the overflow flag (OF), conditional jumps.

ASC

ASCII character codes

Characters · bytes · 30h–39h

Every character read from the keyboard or written to the screen is carried as a one-byte ASCII code.

TECHNICAL SIGNIFICANCE

Writing a number to the screen or reading one from the keyboard requires converting between a digit and its ASCII code. DOS and BIOS services take and return characters as these codes.

- Digits: '0'–'9' = 30h–39h. The value of a digit is found by subtracting 30h from its code.
- Upper case letters 'A'–'Z' = 41h–5Ah, lowercase letters 'a'–'z' = 61h–7Ah. The difference is 20h, that is only bit 5.
- Control characters: carriage return (CR) is 0Dh, line feed (LF) is 0Ah. The space is 20h and '$' is 24h.

MHz

Units and prefixes

Frequency · period · KB and MB

Clock speeds, instruction times and memory sizes are given in units with prefixes.

TECHNICAL SIGNIFICANCE

Computing the running time of a piece of code requires the clock period, and interpreting the width of the address bus requires knowing memory sizes in binary prefixes.

Related topics: INT 21h character services, routines for reading and printing numbers, strings, DOS strings terminated with '$'.

- The period is the reciprocal of the frequency: T = 1/f. With a 5 MHz clock, one clock period is 200 ns.
- Time prefixes: 1 ms = 10^{-3} s, 1 µs = 10^{-6} s, 1 ns = 10^{-9} s.
- Time = number of clock cycles × clock period.
- Memory uses binary prefixes: 1 KB = 2^{10} = 1024 bytes, 1 MB = 2^{20} bytes. n address lines give 2^n different addresses; 20 lines mean an address space of 1 MB.

Related topics: the clock signal, instruction cycle counts, delay loops, address bus width, the memory map.

BOO

Boolean logic

AND · OR · XOR · NOT

Arithmetic and control circuits are built from Boolean operations.

TECHNICAL SIGNIFICANCE

Logical instructions expose these operations directly to assembly programs.

- Basic truth tables.
- Bit masking.
- Toggling bits with XOR; clearing a register by XORing it with itself.

Related topics: masks, flags, device control, bit fields.

DIG

Digital logic basics

Gates · multiplexers · flip-flops

A CPU consists of digital circuits that store and transform binary values.

TECHNICAL SIGNIFICANCE

Basic logic concepts make it easier to understand registers, ALUs, buses and control signals.

- The difference between combinational and sequential logic.
- Gates, multiplexers, decoders and flip-flops.
- A register as stored binary state.

Related topics: the ALU, the register file, the instruction decoder, the control unit.

Z

Latches and three-state buffers

Latch · buffer · high impedance

When several devices are connected to one bus, these building blocks decide which device drives the bus and when a value is captured.

TECHNICAL SIGNIFICANCE

In processors that put address and data on the same pins one after the other, the address has to be captured in a latch and the data bus has to be isolated with three-state buffers.

- A latch passes its input to its output while the enable input is active; when the enable is removed it holds the last value.

- A three-state output can take the high-impedance (Z) state in addition to 0 and 1. In this state the device is as if disconnected from the bus.
- Only one device should drive a bus at a time. Two outputs driving the bus at the same time is bus contention.
- An active-low signal is active when it is 0; it is shown with a ' at the end of its name or with a bar over it.

Related topics: ALE and address latches, data transceivers, the RD' and WR' signals, memory banks.

PRG

Basic programming

Variables · loops · conditions

Assembly is easier to learn once you are used to the control flow of algorithms.

TECHNICAL SIGNIFICANCE

High-level programming concepts correspond to low-level instruction sequences.

- Assignment and arithmetic expressions.
- Tracing different programming concepts step by step.
- Arrays and functions.

C

C programming concepts

Types · arrays · functions · memory

C is a useful bridge between high-level code and assembly.

Related topics: CMP/Jcc, loops, variables in memory, procedures.

TECHNICAL SIGNIFICANCE

Many processor concepts become clearer when they are related to variables, pointers, function calls and data sizes in C.

- Integer types and arrays.
- Basic pointers and address concepts.
- Function calls and local variables at a conceptual level.

BIT

Bitwise operations

Masks · shifts · rotates

Working with bits is the foundation of low-level programming.

Related topics: stack frames, calling conventions, data representation.

TECHNICAL SIGNIFICANCE

The 8086 has dedicated logic, shift and rotate instructions for bit-level control.

- Building and applying masks.
- Testing individual bits.
- The conceptual difference between logical and arithmetic shifts.

ARC

Computer architecture basics

CPU · memory · buses

Related topics: SHL/SHR, ROL/ROR, device registers, flags.

How the CPU exchanges data with memory and I/O.

TECHNICAL SIGNIFICANCE

Registers, buses, memory and control signals form the execution environment that assembly code sees.

- The basic roles of the CPU, memory and input/output (I/O).
- Address, data and control buses at a conceptual level.
- The fetch–decode–execute cycle.

Related topics: the 8086 bus interface, memory access, instruction execution.

REG

Registers

General-purpose · segment · index · pointer

The 8086 exposes a small set of named registers directly to programs.

TECHNICAL SIGNIFICANCE

Assembly instructions mostly work on the AX, BX, CX and DX registers and on the segment, index and pointer registers.

- Registers are small, fast storage areas inside the CPU.
- The distinction between general-purpose and special-purpose roles.
- 16-bit registers and, where available, 8-bit high/low halves.

Related topics: AX/AH/AL, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES.

MEM

Memory addressing

Addresses · bytes · words

Assembly code reads and writes memory locations whose addresses it specifies explicitly.

TECHNICAL SIGNIFICANCE

Knowing how memory is organized is indispensable for variables, arrays, pointers and instructions.

- Byte-addressable memory.
- Little-endian storage of multi-byte values.
- Recognizing the effective address.

Related topics: data variables, arrays, pointers, instruction operands.

SEG

Segmentation on the 8086

Segment : offset · physical address

The 8086 forms 20-bit physical addresses from 16-bit segment and offset values.

TECHNICAL SIGNIFICANCE

Segmentation explains how the original 8086, whose registers are 16 bits wide, could address up to 1 MB of memory.

- Physical address = segment × 16 + offset.

- Overlapping segments.
- The roles of the CS, DS, SS and ES registers.

Related topics: code, data, stack and extra segments, 20-bit addressing.

ALU

Arithmetic and flags

ADD · SUB · CMP · status flags

Arithmetic instructions update both the value in the destination and the processor's status flags.

TECHNICAL SIGNIFICANCE

Flags are used in comparisons, signed/unsigned decisions, loops and error detection.

- The carry, zero, sign and overflow flags at a conceptual level.
- CMP updates the flags as a subtraction without storing the result of the subtraction.
- The relation between flags and conditional jumps.

JMP

Control flow in assembly

Jumps · loops · labels

Assembly expresses decisions and repetition through labels and branch instructions.

Related topics: ADD, SUB, CMP, Jcc, signed and unsigned comparisons.

TECHNICAL SIGNIFICANCE

High-level control structures turn into comparisons and jumps when compiled.

- Tracing conditional and unconditional jumps step by step.
- Labels as addresses.
- Translating simple if/while logic into low-level steps.

STK

Stack fundamentals

PUSH · POP · SP · LIFO

The stack is used for temporary storage, procedures, return addresses and interrupts.

Related topics: JMP, JE/JZ, JNE/JNZ, signed/unsigned conditional jumps.

TECHNICAL SIGNIFICANCE

Disciplined use of the stack is essential for procedure calls and interrupt handling to work correctly.

- LIFO (last in, first out) order.
- PUSH and POP instructions and their effects.
- The SP and SS registers that hold the state of the stack on the 8086.

PROC

Procedures and calls

CALL · RET · parameters

Procedures divide assembly programs into reusable blocks.

Related topics: procedures, return addresses, parameters, interrupts.

TECHNICAL SIGNIFICANCE

CALL and RET rely on the stack to preserve the return information.

- CALL pushes the return address onto the stack.
- RET returns to the calling code.

- Passing parameters in registers and on the stack at a conceptual level.

Related topics: modularity, the stack, calling conventions.

INT

Interrupts

Hardware · software · vector table

Interrupts transfer control to a service routine in response to hardware or software events.

TECHNICAL SIGNIFICANCE

BIOS/DOS services, timers, keyboards and interaction with devices all rely on interrupts.

- The distinction between hardware and software interrupts.
- The idea of an interrupt vector.
- The need to save and restore the processor state.

Related topics: the INT instruction, the interrupt vector table (IVT), the interrupt service routine (ISR), BIOS/DOS services.

I/O

Input/output

Ports · devices · IN/OUT

Microprocessors communicate with peripherals through explicit I/O mechanisms.

TECHNICAL SIGNIFICANCE

Port-mapped and memory-mapped I/O connect the instructions in software to hardware devices.

- Device registers at a conceptual level.
- The purpose of the IN and OUT instructions on x86.
- Polling and interrupt-driven I/O.

Related topics: ports, peripherals, keyboards, displays, controllers.

ELE

Basic electronics

Voltage · signals · timing

Microprocessors are not only abstract instruction machines but also digital systems that run on electricity.

TECHNICAL SIGNIFICANCE

Basic signal and timing concepts help explain the clock signal, buses, pins and interfacing.

- Binary voltage levels at a conceptual level.
- Clock signals.
- Input/output direction and signal timing.

Related topics: the clock, reset (RESET), bus timing, digital interfaces.

TLS

Assembly tools

Assembler · linker · debugger

Low-level programming requires a toolchain that turns symbolic assembly code into executable machine code.

TECHNICAL SIGNIFICANCE

Knowing the stages of the toolchain makes it easier to trace the source of syntax, symbol and runtime errors.

- The roles of the assembler and the linker.
- Reading a simple memory/register dump.
- Single-stepping and breakpoints at a conceptual level.

Related topics: a MASM/TASM/NASM-style workflow, the linker, the debugger.

MC

Machine code basics

Opcodes · operands · encoding

Assembly instructions are the symbolic representation of encoded machine instructions.

TECHNICAL SIGNIFICANCE

Instruction encoding explains why operands, addressing modes and instruction sizes matter.

- The difference between an opcode and an operand.
- Instructions and bytes in memory.
- The addressing mode affects the encoding.

Related topics: machine code, disassembly, instruction size.

BUS

Bus and timing basics

Address · data · control

The processor communicates with the outside world through address, data and control signals that work together.

TECHNICAL SIGNIFICANCE

Bus timing explains how memory and peripherals know when to read, write or respond.

- The distinction between address information and data information.
- Read/write control signals at a conceptual level.
- Wait states and timing in broad terms.

Related topics: memory cycles, I/O cycles, control pins, interfacing.

APPENDIX C

Important People in Microprocessors

MICROPROCESSORS · HISTORY OF COMPUTING

The history of microprocessors brings together semiconductor technology, instruction set architecture, CPU implementation, memory systems, performance engineering and the development that runs from the first single-chip processors to today's x86 and RISC designs.

From the integrated circuit to today's processors

These cards introduce the people who contributed to the invention of the transistor, semiconductor integration, the first microprocessors, the 8086/x86 family, RISC architectures, memory technology, processor performance, the classification of parallel processors and the first widely used high-level language.

PEOPLE · CHIPS · ARCHITECTURE

Notable contributions

The leading people behind integrated circuits, the first commercial microprocessors, the Intel 8086/x86 family, RISC and ARM architectures, memory technology and processor performance engineering.

FF

Federico Faggin

Intel 4004 · 8080 · silicon-gate design

Led the design of the Intel 4004 and contributed to the development of the first general-purpose microprocessors.

TECHNICAL SIGNIFICANCE

His work helped move CPU functions from boards full of logic circuits into highly integrated single-chip processors.

4004 **8080** **MOS integration**

Faggin developed silicon-gate MOS technology and led the implementation work that made the 4004 usable as a commercial microprocessor. He later contributed to the 8008 and 8080 era.

Technical concepts: 4004, 8080, MOS integration, instruction decoding, datapath integration, semiconductor scaling.

TH

Ted Hoff

Microprocessor architecture · Intel 4004

Proposed a general-purpose, programmable processor architecture for the project that became the Intel 4004.

TECHNICAL SIGNIFICANCE

Replacing many special-purpose logic circuits with a programmable CPU turned the microprocessor into a computing component that could be reused in different products.

4004 **architecture** **stored-program control**

Hoff's architectural contribution was to simplify a chipset designed for a calculator into a programmable processor plus memory and I/O components.

SM

Stanley Mazor

Intel 4004 architecture

Worked on the architectural definition and the instruction set of the Intel 4004.

Technical concepts: 4004, architecture, stored-program control, instruction set, programmable control.

TECHNICAL SIGNIFICANCE

The choices made in an instruction set determine how software expresses data movement, arithmetic, branching and control.

ISA **4004** **opcode design**

Mazor contributed to defining the instructions and the architectural behavior of Intel's first microprocessor family.

MS

Masatoshi Shima

4004 implementation · CPU design

Played a central role in the detailed logic design of the Intel 4004 and of later microprocessors.

Technical concepts: instruction set architecture (ISA), 4004, opcode design, instruction formats, control flow.

TECHNICAL SIGNIFICANCE

Turning an architectural definition into a working chip takes concrete control logic, datapaths, timing and interfaces.

4004 **logic design** **datapath**

Shima translated architectural requirements into detailed processor logic and later worked on the design of other influential CPUs.

Technical concepts: 4004, logic design, datapath, timing, register transfer, chip implementation.

SM

Stephen Morse

Intel 8086 architecture · 1970s

Was the principal architect of the Intel 8086 instruction set and programming model.

TECHNICAL SIGNIFICANCE

The 8086 laid the foundation of the x86 architecture studied in the course.

8086 **x86** **ISA**

The 8086 introduced 16-bit registers, segmented addressing, general-purpose and segment registers, and an instruction set that is the ancestor of today's x86 processors.

Technical concepts: 8086, x86, ISA, segment:offset addressing, registers, the flags register (FLAGS).

BR

Bruce Ravenel

8086 architecture and implementation

Worked on the 8086 design team and in the early development of x86.

TECHNICAL SIGNIFICANCE

Processor architecture is teamwork that brings together instruction-set design, implementation constraints and system compatibility.

8086 **x86** **execution unit**

Within the semiconductor constraints of the late 1970s, the 8086 team balanced transistor budget, performance, code density, bus design and compatibility goals.

BP

Bill Pohlman

Intel 8086 project management

Managed the engineering work of the Intel 8086 development program.

Technical concepts: 8086, x86, execution unit (EU), bus interface unit (BIU), instruction prefetch, segmented memory.

TECHNICAL SIGNIFICANCE

A successful CPU design requires coordination among architecture, logic design, verification, manufacturing and product constraints.

8086 **Intel** **engineering**

The 8086 is the product of engineering carried out under tight constraints: a limited transistor count, 16-bit arithmetic, a 20-bit address bus and a design that eased the move from earlier processors.

SW

Sophie Wilson

ARM instruction set · RISC

Designed the original ARM instruction set architecture.

Technical concepts: 8086, Intel, engineering, bus width, address space, system integration.

TECHNICAL SIGNIFICANCE

ARM showed that a small, regular instruction set can deliver high performance with low hardware complexity and low power consumption.

ARM **RISC** **ISA**

Wilson's work on the ARM architecture emphasized simple instruction decoding, a load/store design, conditional execution and efficient use of registers.

Technical concepts: ARM, RISC, ISA, load/store architecture, registers, conditional execution.

SF

Steve Furber

ARM processor implementation

Co-designed the implementation of the first ARM processor.

TECHNICAL SIGNIFICANCE

His work tied the ideas of RISC architecture to a practical, low-power processor design.

ARM **RISC** **datapath**

Furber helped turn the ARM instruction set into working silicon and later made important contributions to processor and neuromorphic-system research.

Technical concepts: ARM, RISC, datapath, control logic, low-power design.

JH

John Hennessy

RISC architecture · MIPS

Led groundbreaking work on RISC processor architecture and the MIPS family.

TECHNICAL SIGNIFICANCE

RISC research reshaped the balance that processors strike between instruction complexity, pipelining, compiler support and implementation speed.

MIPS **RISC** **pipelining**

Hennessy's work helped load/store instruction sets, large register files and pipelined execution become established design principles.

Technical concepts: MIPS, RISC, pipelining, load/store, joint design of compiler and architecture.

DP

David Patterson

RISC · Berkeley RISC

Led the Berkeley RISC project and helped define today's idea of reduced instruction set computing.

TECHNICAL SIGNIFICANCE

RISC designs showed that simpler instructions allow faster pipelines and more efficient implementations.

RISC **RISC-V** **pipelining**

Patterson also helped establish the quantitative approach to computer architecture and later took part in the development of RISC-V.

Technical concepts: RISC, RISC-V, pipelining, quantitative architecture, instruction set design.

MF

Michael J. Flynn

Flynn's taxonomy · 1966

In 1966 classified processors into four classes, SISD, SIMD, MISD and MIMD, by the number of instruction streams and data streams processed at the same time.

TECHNICAL SIGNIFICANCE

The difference between a scalar core, a vector or array processor, a GPU and a multicore processor can be described in one table with this taxonomy. The SSE and AVX instructions of x86 belong to the SIMD class.

SIMD **MIMD** **parallel architecture**

The taxonomy was published in the paper “Very High-Speed Computing Systems” (*Proceedings of the IEEE*, 1966). Flynn worked on computer architecture at Stanford University.

Technical concepts: SISD, SIMD, MISD, MIMD, instruction stream, data stream, vector processing.

JB

John Bardeen

Transistor · 1947 · Bell Laboratories

Together with Walter Brattain, built the first working transistor, the point-contact transistor, at Bell Laboratories in late 1947.

TECHNICAL SIGNIFICANCE

The transistor does the switching and amplifying that the vacuum tube did with a much smaller, low-power and rugged semiconductor device. Every logic gate in a microprocessor is built from transistors.

transistor **semiconductor** **Bell Laboratories**

Bardeen, Brattain and Shockley shared the 1956 Nobel Prize in Physics for their work on the transistor. In 1972 Bardeen received a second Nobel Prize in Physics for the theory of superconductivity (BCS theory); he is the only person to have won this prize twice.

WB

Walter Brattain

Transistor · 1947 · experimental physics

At Bell Laboratories, built and operated the first point-contact transistor on a germanium crystal together with Bardeen (1947).

Technical concepts: point-contact transistor, germanium, amplification, switching, solid-state electronics.

TECHNICAL SIGNIFICANCE

Turning the theoretical idea into a working device was possible through experimental work on semiconductor surfaces. The transistor was announced in 1948 and began the age of microelectronics.

transistor **germanium** **surface physics**

Brattain was an experimental physicist who studied the electrical behavior of semiconductor surfaces. He shared the 1956 Nobel Prize in Physics with Bardeen and Shockley.

Technical concepts: point-contact transistor, semiconductor surface, germanium crystal, current gain.

WS

William Shockley

Junction transistor · the beginning of Silicon Valley

Led the transistor group at Bell Laboratories and developed the theory of the junction transistor; the device was announced in 1951.

TECHNICAL SIGNIFICANCE

The junction transistor was more robust than the point-contact transistor and better suited to mass production. The transistors in integrated circuits descend from this line.

junction transistor **semiconductor industry** **Fairchild**

In 1956 Shockley founded Shockley Semiconductor Laboratory in Mountain View, California. In 1957 eight researchers left it, among them Robert Noyce and Gordon Moore, and founded Fairchild Semiconductor. Intel was later born from this team. Shockley shared the 1956 Nobel Prize in Physics with Bardeen and Brattain.

RN

Robert Noyce

Integrated circuits · one of the founders of Intel

One of the inventors of the integrated circuit; co-founded Intel.

TECHNICAL SIGNIFICANCE

Integrated circuits made it possible to manufacture dense, reliable processor chips economically.

integrated circuit **Intel** **semiconductor manufacturing**

Noyce's work on planar integrated circuits and semiconductor manufacturing formed the technological base of commercial microprocessors.

GM

Gordon Moore

Semiconductor scaling · Intel

One of the founders of Intel, he stated the scaling trend that was later named Moore's law.

TECHNICAL SIGNIFICANCE

For decades the rapid growth in transistor density determined processor performance, the level of integration, cache size and architectural complexity.

Moore's law **transistor density** **scaling**

Moore's observations became a planning principle in the industry for semiconductor integration and product roadmaps.

Technical concepts: Moore's law, transistor density, scaling, integration, Intel.

JK

Jack Kilby

Integrated circuit

Built and operated one of the first integrated circuits.

TECHNICAL SIGNIFICANCE

Integration replaced hand-wired assemblies with small semiconductor circuits and opened the way to ever more complex digital systems.

integrated circuit **semiconductor integration** **miniaturization**

Kilby's work began the integrated-circuit era in which microprocessors were born.

Technical concepts: integrated circuit, semiconductor integration, miniaturization.

RD

Robert Dennard

DRAM · scaling

Invented the one-transistor DRAM cell and formulated Dennard scaling.

TECHNICAL SIGNIFICANCE

DRAM became the dominant main-memory technology, while the scaling rules influenced the course of processor frequency and power consumption.

DRAM **memory** **Dennard scaling**

Dennard's contributions tie the development of processors directly to memory density and to the power characteristics of transistors.

FW

Fred Weber

AMD K8 · the x86-64 era

Led the major processor architecture work at AMD during the K8 generation.

Technical concepts: DRAM, memory, Dennard scaling, power density, semiconductor physics.

TECHNICAL SIGNIFICANCE

AMD64 extended x86 to 64-bit addressing while keeping compatibility with existing x86 software.

AMD64 **x86-64** **integrated memory controller**

The K8 architecture introduced a 64-bit ISA extension and moved the memory controller onto the processor die.

DM

Dirk Meyer

AMD processors · x86 implementation

Held key management and engineering roles in the development of AMD processors.

Technical concepts: AMD64, x86-64, integrated memory controller, compatibility, 64-bit ISA.

TECHNICAL SIGNIFICANCE

Today's x86 processors have to turn a complex ISA into deeply pipelined, out-of-order internal execution.

x86 **AMD** **microarchitecture**

AMD's designs show that architectural compatibility can coexist with far-reaching innovation in the microarchitecture.

Technical concepts: x86, AMD, microarchitecture, out-of-order execution, compatibility.

TG

Tom Gunter

Motorola 68000 family

Was the chief architect of the Motorola 68000 project; he formed the design team in 1977.

TECHNICAL SIGNIFICANCE

The 68000 family was influential in workstations, personal computers and embedded systems.

Motorola 68000 **ISA design** **registers**

With a relatively orthogonal instruction set and a large set of registers visible to the programmer, it offered a design approach different from that of the early x86 processors.

Technical concepts: Motorola 68000, ISA design, registers, embedded systems.

CP

Chuck Peddle

MOS 6502

Led the development of the MOS Technology 6502.

TECHNICAL SIGNIFICANCE

The inexpensive 6502 helped microprocessors find their way into the first personal computers and embedded products.

6502 **low-cost CPU** **personal computers**

The 6502 was the processor of systems such as the Apple II, Commodore computers and many game consoles.

Technical concepts: 6502, low-cost CPU, personal computers, embedded systems.

BM

Bill Mensch

Motorola 6800 · 6502

Worked on the development of the Motorola 6800 and co-designed the 6502.

TECHNICAL SIGNIFICANCE

His work spans two influential 8-bit processor families of the early period.

6800 **6502** **8-bit processors**

Mensch later founded Western Design Center, the company that continued the development of 6502-derived processors.

Technical concepts: 6800, 6502, 8-bit processors, CPU implementation.

IS

Ivan Sutherland

Computer systems · influence on VLSI education

Contributed through his research and teaching to the development of thinking in computer systems and VLSI.

TECHNICAL SIGNIFICANCE

VLSI design methods made it possible to handle increasingly complex processor chips systematically.

VLSI **computer systems** **integrated design**

His broader work influenced the culture of integrated digital-system design on which today's processor implementation rests.

Technical concepts: VLSI (very large-scale integration), computer systems, integrated design, chip design methods.

JB

John Backus

Fortran · 1957 · IBM

Led the team at IBM that developed Fortran, the first widely used high-level language, and its compiler (1957).

TECHNICAL SIGNIFICANCE

The Fortran compiler showed that programs written in the form of formulas could be turned into machine code nearly as efficient as hand-written code. The shift of programming from assembly to high-level languages began with this work.

Fortran **compiler** **high-level language**

The Backus–Naur form (BNF), used to define the syntax of programming languages, also bears his name. He received the Turing Award in 1977.

Technical concepts: compiler, source code, machine code, syntax, BNF.