Chapter 2.1: 8086 Architecture

Microprocessors

The 8086 at a glance | System buses | BIU and EU | Prefetch queue | Fetch–decode–execute | Register set | Segment registers and IP | Segmented memory | Physical address calculation

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/microprocessors/slides/Chapter_02_01_8086_Architecture.html

Learning Outcomes

General features

Explain the 8086's data and address bus widths, how much memory it can address, and its place in the x86 family.

Internal structure

Describe the roles of the BIU and EU, the prefetch queue and the idea of pipelining.

Registers

Group the 14 registers; explain the roles of the segment registers and IP.

Address calculation

Compute a 20-bit physical address as `segment × 10h + offset`, by hand and step by step.

OVERVIEW

The 8086 at a Glance

- A **16-bit** microprocessor introduced by Intel in **1978**; the first member of the **x86** family.
- Its registers, ALU and data bus are 16 bits; it processes 16 bits of data at a time.
- **20-bit address bus** → 2^{20} = 1,048,576 bytes = **1 MB** of memory.
- **16-bit data bus**: moves 2 bytes at a time between the processor, memory and I/O devices.
- **CISC** (complex instruction set) architecture: variable-length (1–6 byte) instructions, instructions that operate on memory.
- Its instruction set, the basis of the PC, survives through backward compatibility.

16-bit data bus	**20-bit** address bus
1 MB memory	**40 pins** DIP package
5–10 MHz clock	**~29,000** transistors
+5 V single supply	**HMOS** technology

Unlike its sibling the 8088, which has an 8-bit external data bus, the 8086 is 16 bits throughout.

OVERVIEW

The 8086 in the x86 Family

Processor	Year	Internal / external data bus	Address bus	Memory	Note
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)

Every later x86 processor starts up in **real mode** and behaves like an 8086. The programs we write with emu8086 in this course also run in 8086 real mode.

OVERVIEW

Why a 20-Bit Address?

16-bit address — 2^{16} = 65,536 bytes = 64 KB (0000h–FFFFh)

20-bit address — 2^{20} = 1,048,576 bytes = 1 MB (00000h–FFFFFh)

But 8086 registers are 16 bits → one register can reach at most 64 KB.

Solution: two 16-bit values → segment × 16 + offset = 20 bits.

n-bit address

2^n distinct bytes. 1 KB = 2^{10}, 1 MB = 2^{20} bytes.

Data bus ≠ address bus

The data bus is 16 bits (how much data at once), the address bus 20 bits (how much memory).

Hex notation

20 bits = 5 hex digits: the lowest address is `00000h`, the highest `FFFFFh`.

OVERVIEW

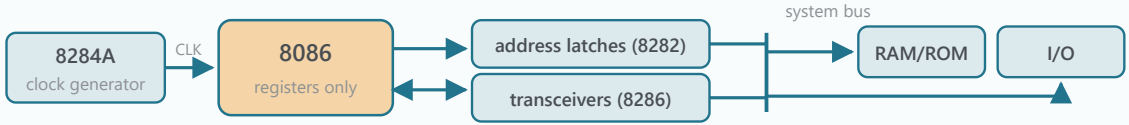
Memory Organization: The CPU Is Not Alone

Microprocessor (8086)

- Puts all the functions of a CPU on a single integrated circuit.
- It has **no RAM or ROM** inside; only registers.
- Intermediate and final results are kept in registers.
- Reaches memory and I/O over the **system bus**.
- Needs external memory, a clock generator and buffer/latch circuits to run.

Microcontroller (e.g. 8051, AVR)

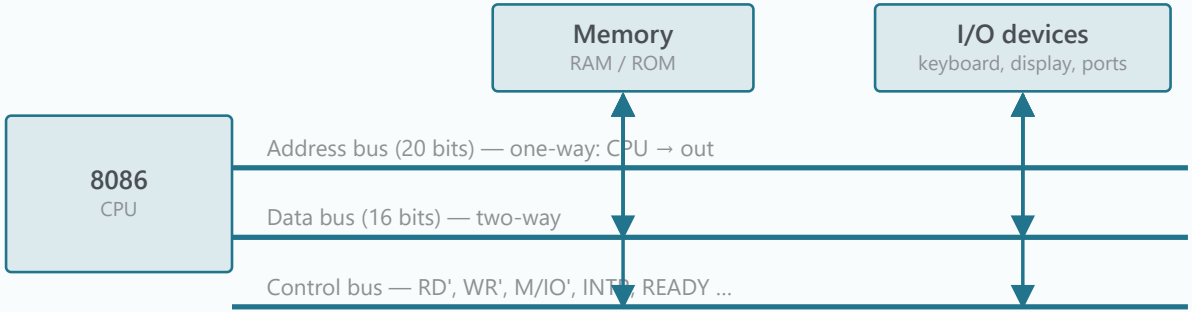
- CPU + program memory (ROM/Flash) + RAM + timers + I/O ports on one chip.
- Can work as a stand-alone system.
- Embedded uses: washing machines, remote controls, sensor nodes.
- Less computing power, lower cost and power consumption.



An 8086-based computer = 8086 + 8284A clock generator + address latches + data bus transceivers + memory + I/O units.

OVERVIEW

System Buses



The “width” of a bus = its number of parallel lines (bits)

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

Code Trace: Memory and I/O over the Buses

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



Data segment **DS = 1000h** (base 10000h). Memory holds **5Ah** at 10200h and I/O port 51h holds **3Ch** (sample values).

Step 1 / 6

VARIABLES

DS = 1000h BX = 0000h AL = 00h

Code Trace: Memory and I/O over the Buses

```
1 org 100h
2 mov ax, 1000h
3 mov ds, ax
▶ mov bx, 0200h
5 mov al, [bx]      ; memory read
6 out 50h, al       ; I/O write
7 in al, 51h        ; I/O read
8 mov [bx+1], al    ; memory write
9 ret
```



BX = 0200h. Instruction fetch only; no bus cycle for data.

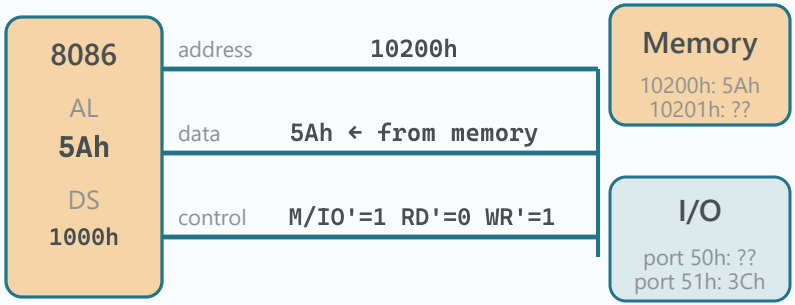
Step 2 / 6

VARIABLES

DS = 1000h BX = 0200h AL = 00h

Code Trace: Memory and I/O over the Buses

```
1 org 100h
2 mov ax, 1000h
3 mov ds, ax
4 mov bx, 0200h
▶ mov al, [bx]      ; memory read
6 out 50h, al       ; I/O write
7 in al, 51h        ; I/O read
8 mov [bx+1], al    ; memory write
9 ret
```



Memory read: the address bus carries the 20-bit **10200h**, **M/IO' = 1**, **RD' = 0**. Memory puts 5Ah on the data bus → AL = **5Ah**.

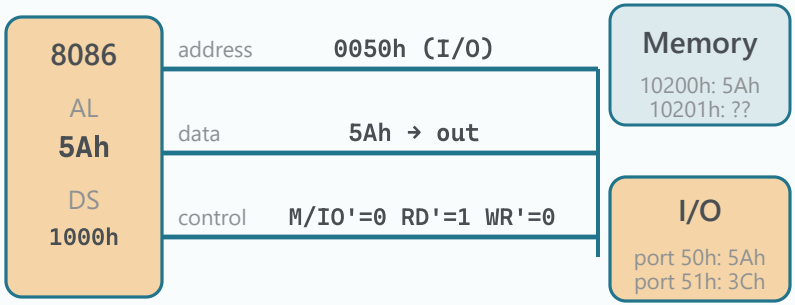
Step 3 / 6

VARIABLES

DS = 1000h | BX = 0200h | AL = 5Ah | M/IO' = 1 | RD' = 0 | WR' = 1

Code Trace: Memory and I/O over the Buses

```
1 org 100h
2 mov ax, 1000h
3 mov ds, ax
4 mov bx, 0200h
5 mov al, [bx]      ; memory read
▶ out 50h, al       ; I/O write
7 in al, 51h        ; I/O read
8 mov [bx+1], al    ; memory write
9 ret
```



I/O write: the address bus carries only the port number (**0050h**, 16 bits), **M/IO' = 0**, **WR' = 0**; AL = 5Ah is on the data bus. Memory ignores this cycle.

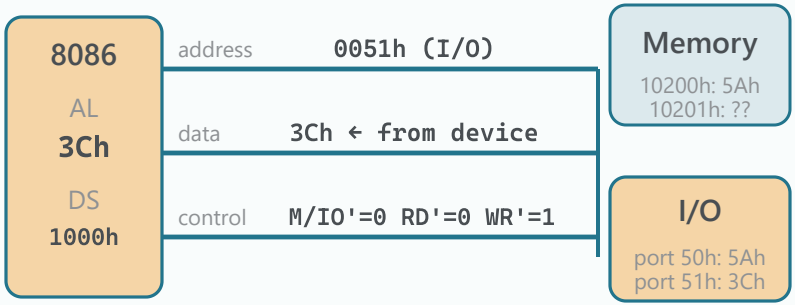
Step 4 / 6

VARIABLES

DS = **1000h** · BX = **0200h** · AL = **5Ah** · M/IO' = **0** · RD' = **1** · WR' = **0**

Code Trace: Memory and I/O over the Buses

```
1 org 100h
2 mov ax, 1000h
3 mov ds, ax
4 mov bx, 0200h
5 mov al, [bx]      ; memory read
6 out 50h, al       ; I/O write
▶ in al, 51h        ; I/O read
8 mov [bx+1], al    ; memory write
9 ret
```



I/O read: port **0051h**, M/IO' = 0, RD' = 0. The device puts 3Ch on the data bus → AL = **3Ch**.

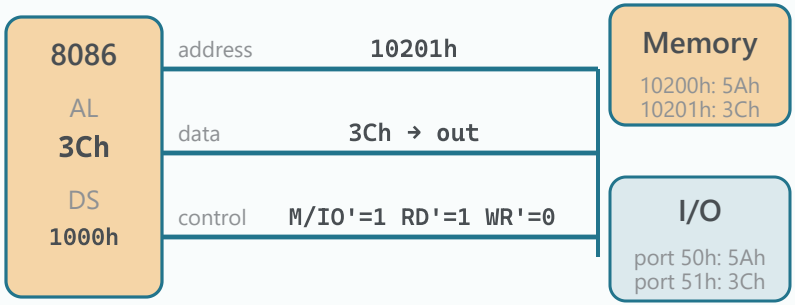
Step 5 / 6

VARIABLES

`DS = 1000h` `BX = 0200h` `AL = 3Ch` `M/IO' = 0` `RD' = 0` `WR' = 1`

Code Trace: Memory and I/O over the Buses

```
1 org 100h
2 mov ax, 1000h
3 mov ds, ax
4 mov bx, 0200h
5 mov al, [bx]      ; memory read
6 out 50h, al       ; I/O write
7 in al, 51h        ; I/O read
▶ mov [bx+1], al    ; memory write
9 ret
```



Memory write: EA = BX + 1 = 0201h → PA = **10201h**, M/IO' = 1, WR' = 0, data 3Ch. Memory and I/O share the same address and data buses; M/IO' tells them apart.

Step 6 / 6

VARIABLES

DS = 1000h | BX = 0200h | AL = 3Ch | M/IO' = 1 | RD' = 1 | WR' = 0

INTERNAL STRUCTURE

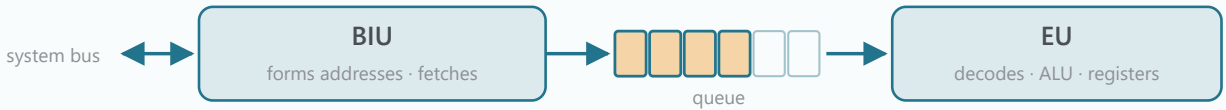
Two Main Units: BIU and EU

Bus Interface Unit (BIU)

- Handles all communication with the outside world: **fetches** instructions from memory.
- Reads operands from memory and writes results to memory or I/O.
- Forms the **20-bit physical address** from the segment registers and IP.
- Fills the 6-byte **prefetch queue**.

Execution Unit (EU)

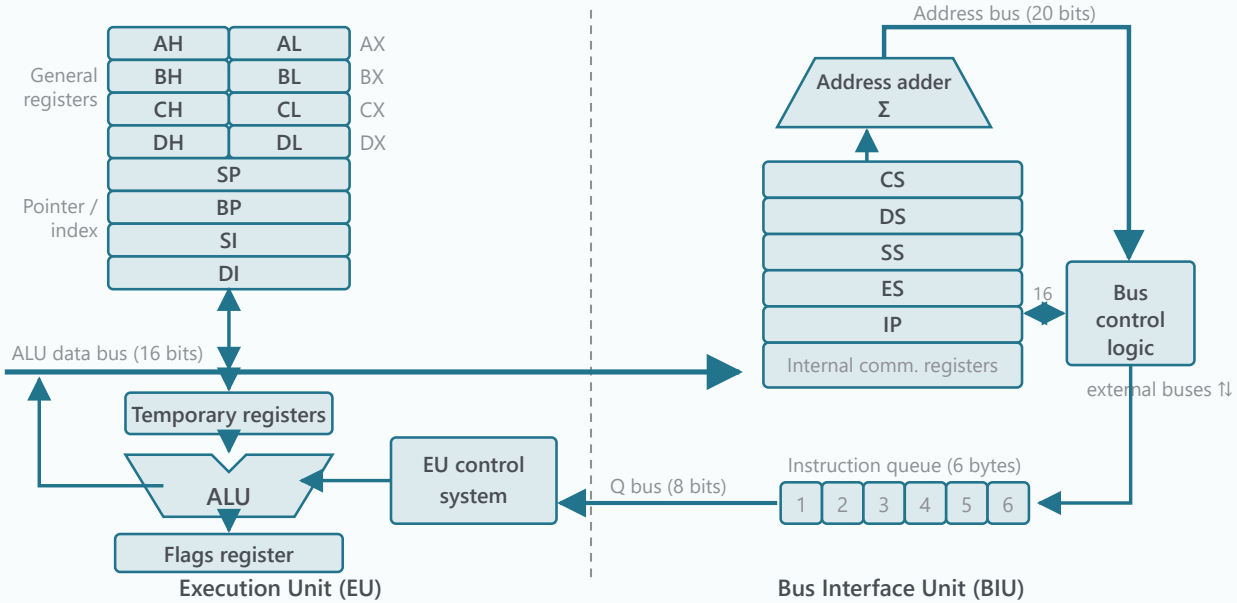
- Takes the next instruction byte from the queue, **decodes** it and **executes** it.
- Performs arithmetic and logic operations in the ALU.
- Updates the general-purpose registers and flags.
- Asks the BIU when it needs an operand in memory; it is not connected to the external buses.



The two units work in **parallel**: while the EU executes one instruction, the BIU fetches the next instruction bytes. This is the 8086's simple form of **pipelining**.

INTERNAL STRUCTURE

8086 Block Diagram

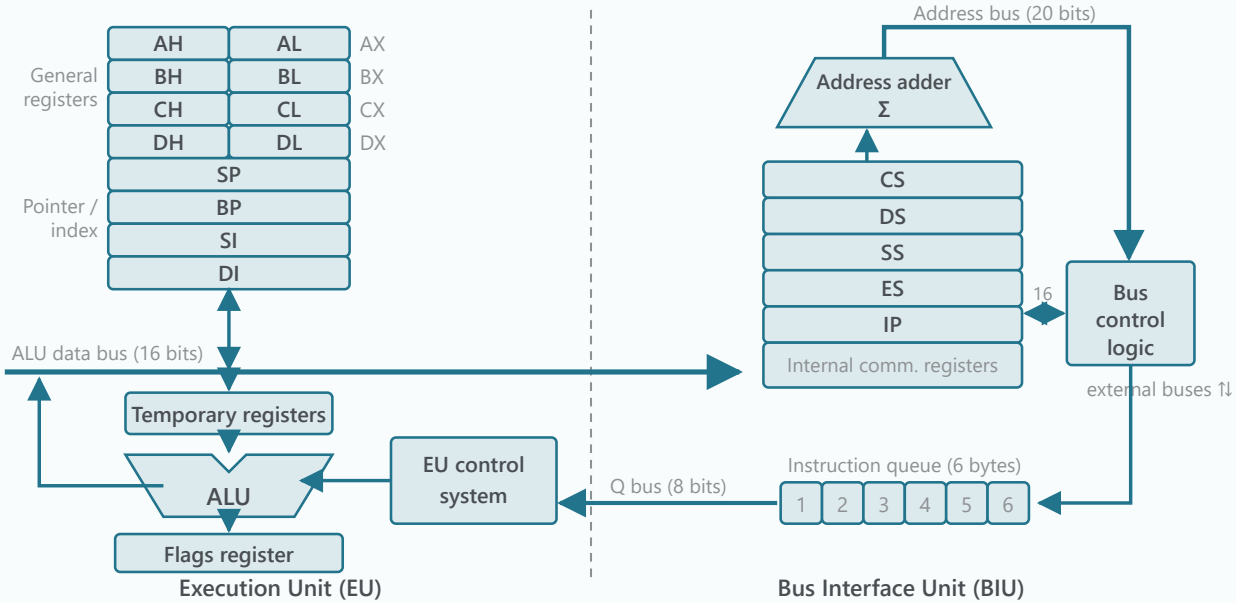


Left: EU
general registers, ALU, flags, control system

Right: BIU
segment registers, IP, address adder, queue, bus control

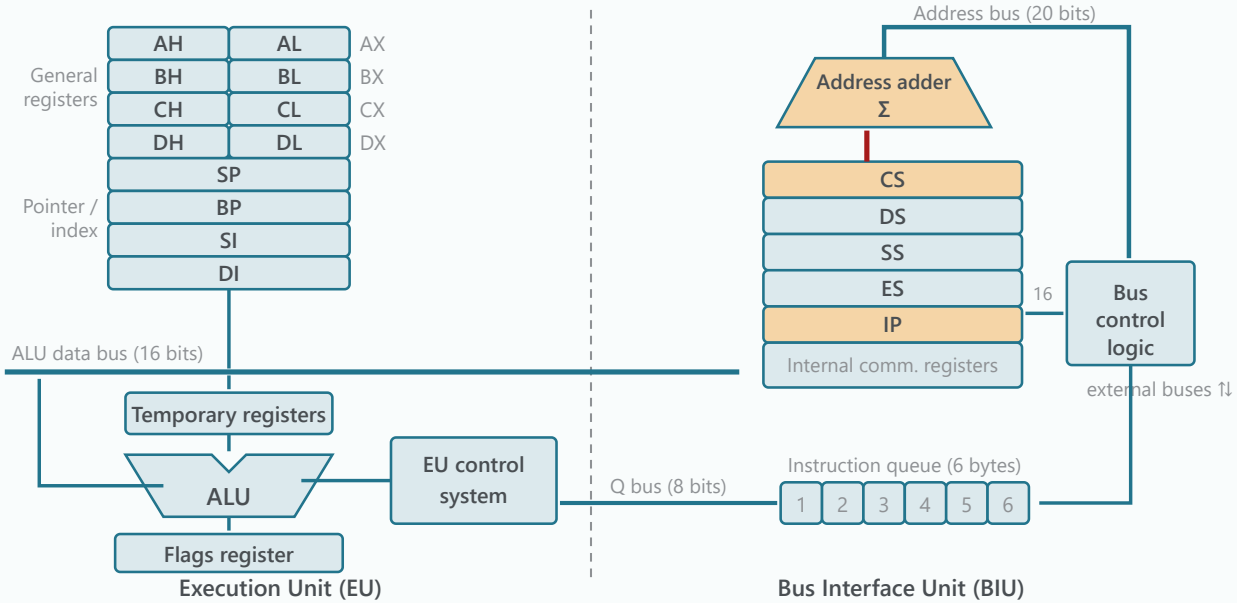
Bridge
the EU takes instruction bytes from the queue over the 8-bit Q bus

How the 8086 Works



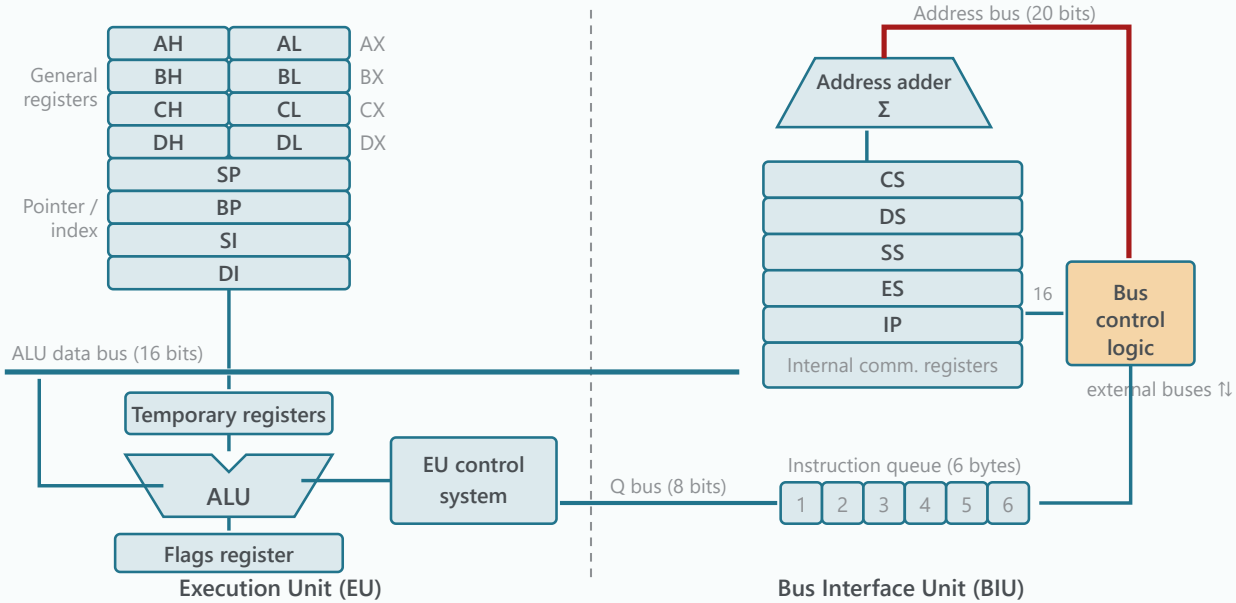
The big picture. Let us follow one instruction step by step: `ADD AX, BX`.

How the 8086 Works



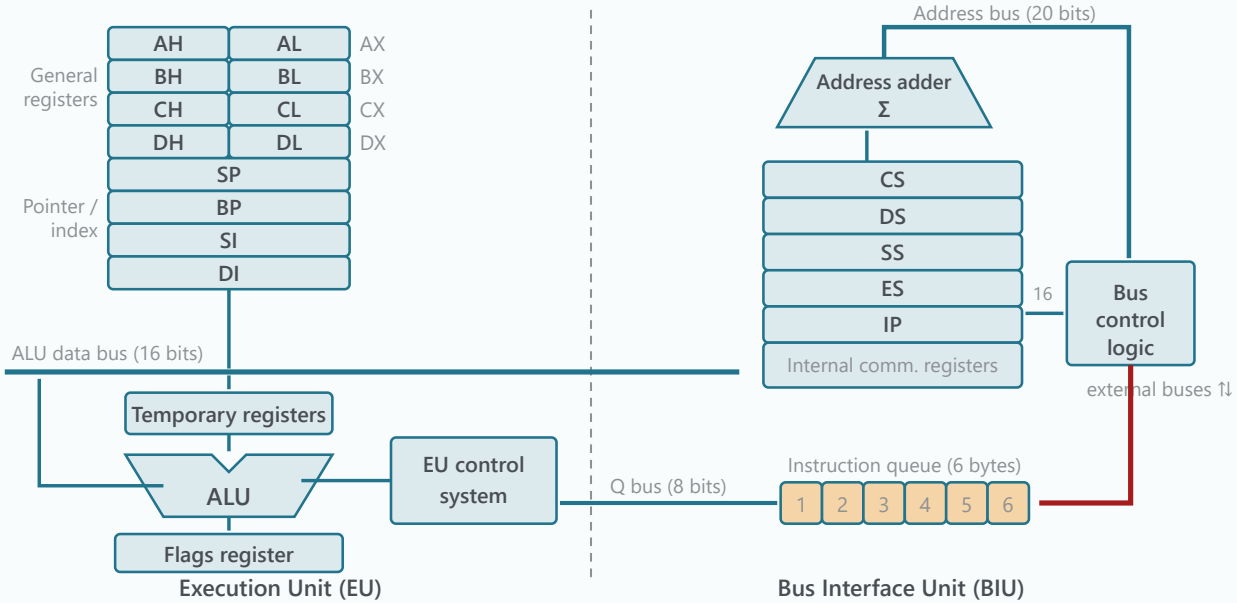
1. The BIU computes the physical address of the instruction to fetch: **CS × 10h + IP** (address adder).

How the 8086 Works



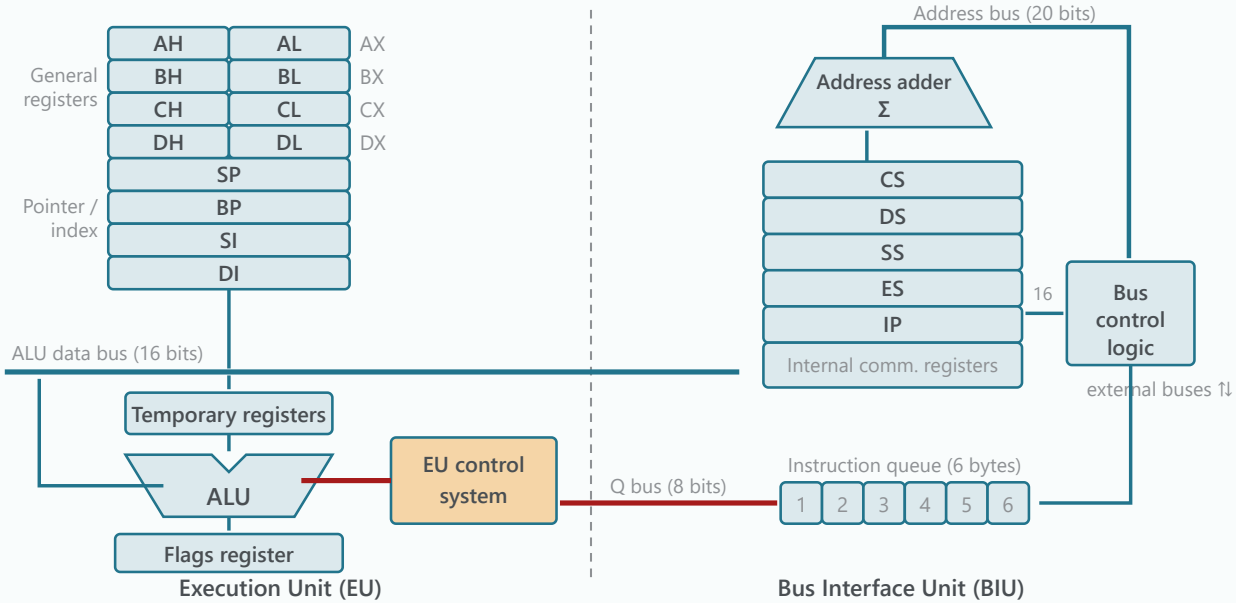
2. The 20-bit address goes onto the external address bus; the bus control logic starts a **memory read cycle**.

How the 8086 Works



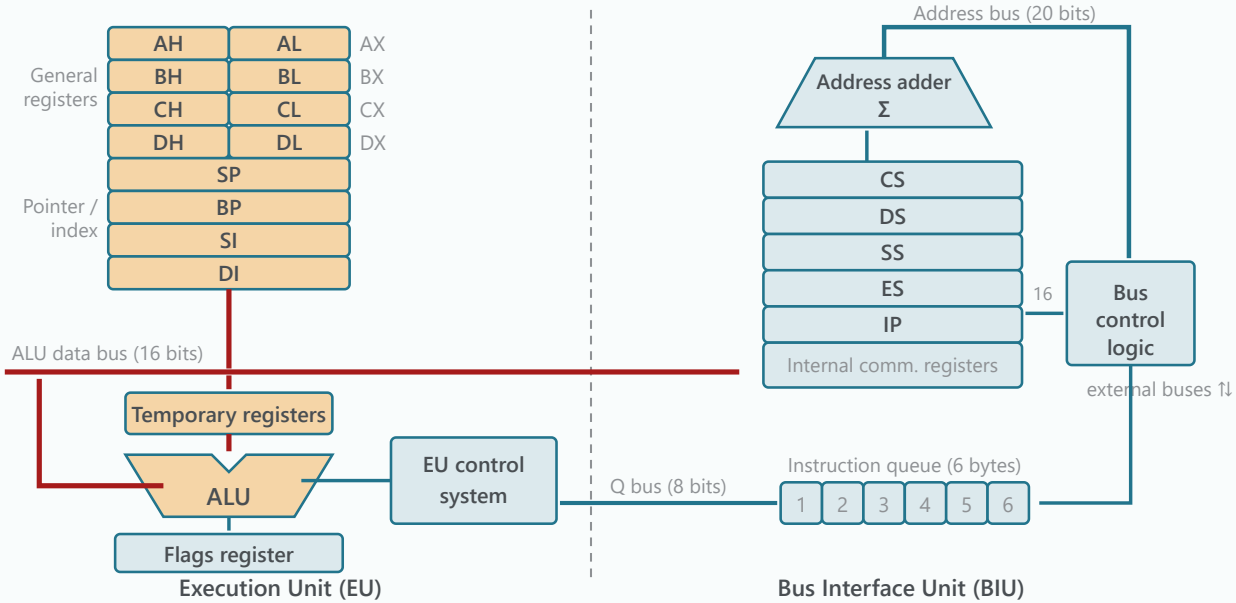
3. The incoming instruction bytes enter the 6-byte **FIFO queue**; IP advances by the number of bytes fetched.

How the 8086 Works



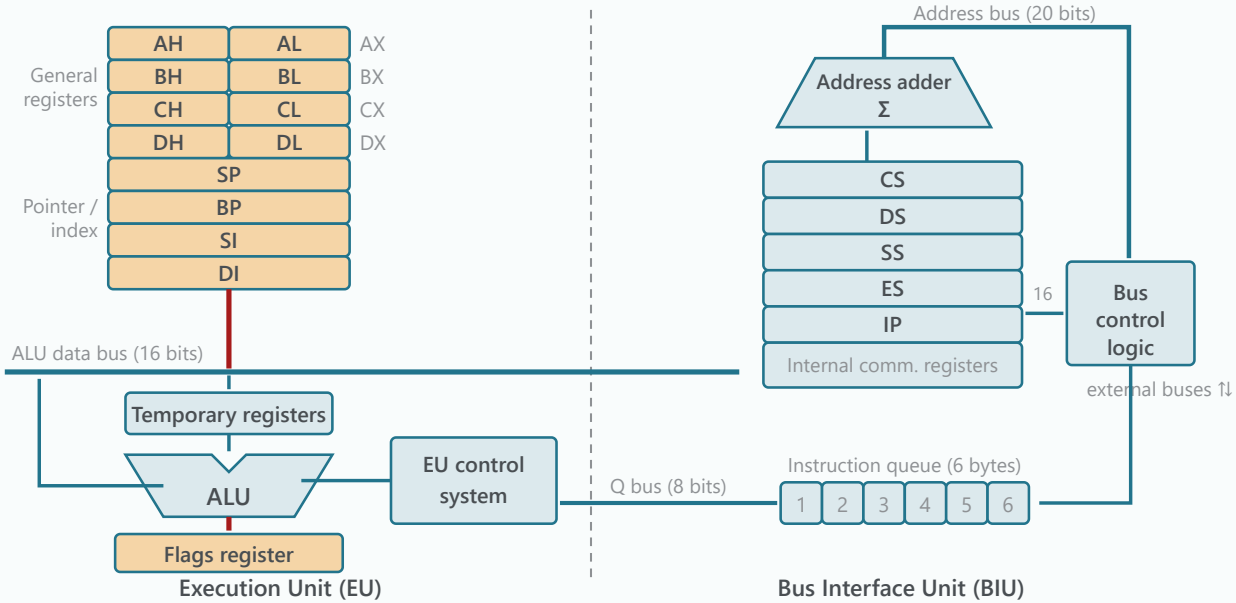
4. When its turn comes, the EU takes the opcode over the Q bus; the control system **decodes** the instruction.

How the 8086 Works



5. Control signals move AX and BX over the ALU data bus into the temporary registers, then into the **ALU**; the add takes place.

How the 8086 Works



6. The result is written to AX and the **flags** (CF, ZF, SF, OF, PF, AF) are updated. Meanwhile the BIU keeps fetching the next bytes.

Bus Interface Unit (BIU)

- Interfaces with external memory and I/O devices over the **system bus**.
- Forms the **20-bit physical address** from a segment register and an offset (address adder).
- **Fetches** instructions from memory and writes them to the queue.
- Reads operands the EU requests from memory/I/O and writes results.
- Manages the 6-byte **prefetch queue** → makes pipelining possible.

Inside the BIU

- Segment registers: `CS`, `DS`, `SS`, `ES`
- Instruction pointer: `IP`
- Address adder (Σ)
- 6-byte instruction queue
- Bus control logic

INTERNAL STRUCTURE

The Prefetch Queue

- A 6-byte **FIFO** (first in, first out) queue: bytes are used in the order fetched.
- While the EU executes, the BIU prefetches the next instruction bytes → **pipelining**.
- Once **2 bytes are free** in the queue, the BIU fetches the next word.
- A **branch** (JMP, CALL, RET, conditional jump, interrupt) **flushes** the queue; the prefetched bytes are on the wrong path.
- 8086 instructions are 1–6 bytes (without prefixes); the queue holds the longest in full.



Gain

The EU mostly takes instructions from the queue without waiting; memory latency is hidden.

Loss

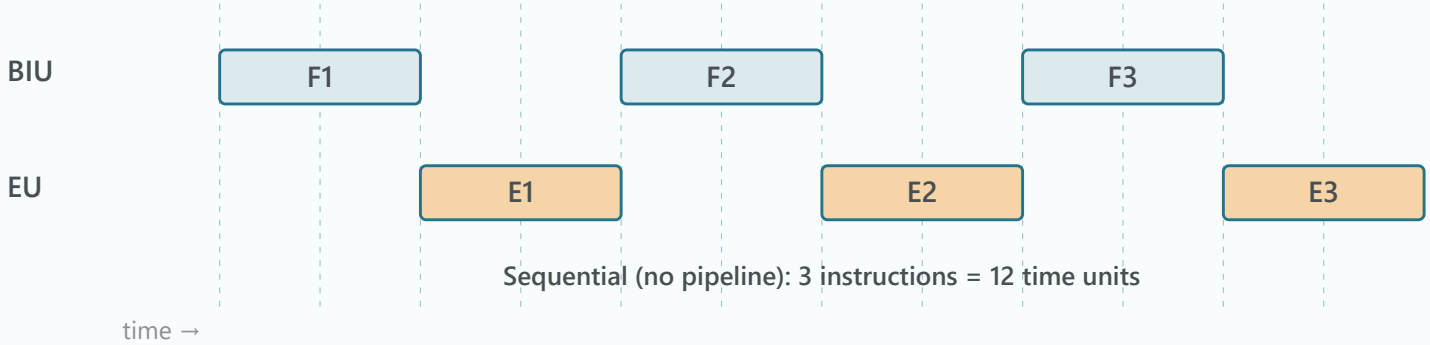
A branch empties the queue; the EU waits until it refills.

The 8088

8-bit external bus → a 4-byte queue; fetches once 1 byte is free.

Pipelining

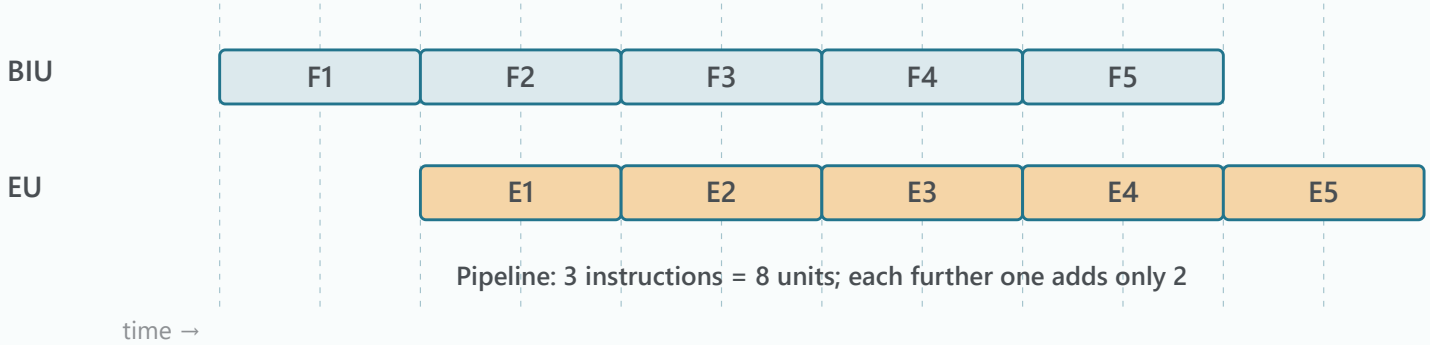
F = fetch (BIU), E = decode + execute (EU). Each box is assumed to take 2 time units.



Without a pipeline: fetch (F) first, then execute (E); the BIU and EU take turns sitting idle.

Pipelining

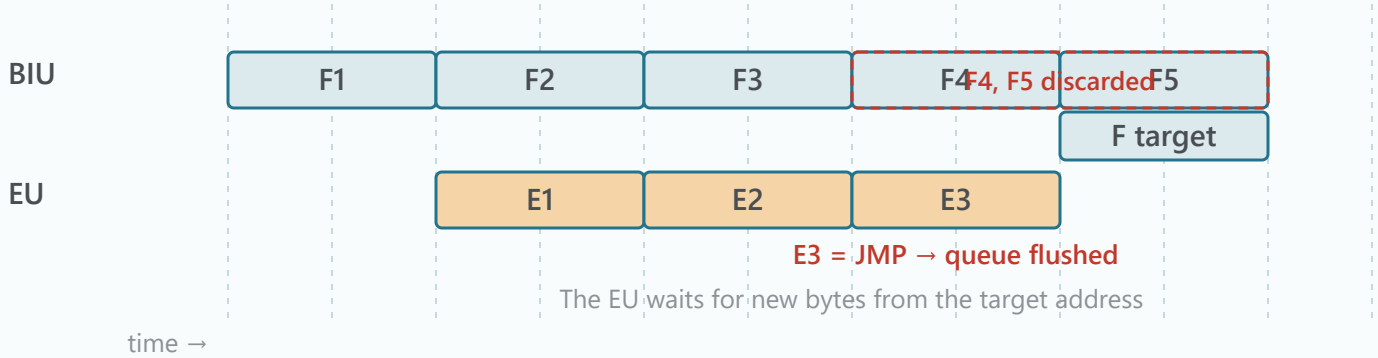
F = fetch (BIU), E = decode + execute (EU). Each box is assumed to take 2 time units.



8086: while the EU executes **E1**, the BIU fetches **F2**. Both units work at once; throughput roughly doubles.

Pipelining

F = fetch (BIU), E = decode + execute (EU). Each box is assumed to take 2 time units.



Branching: if E3 is a **JMP**, the prefetched F4 and F5 are invalid. The queue is emptied and refilled from the target address.

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left



EU now:

– (waiting)

IP (next instruction) = 0100h

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

Start (or after a branch): the queue is empty, the EU waits. CS:IP = CS:0100h.

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

B8	34	12	01	D8	41

EU now:

`decoding MOV AX,1234h`

EU

`IP (next instruction) = 0100h`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The BIU fetches word by word (B8 34, 12 01, D8 41); the queue fills. The EU starts decoding as soon as the first byte arrives.

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

01	D8	41			

EU

EU now:

`MOV AX,1234h`

`IP (next instruction) = 0103h`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The EU took the 3-byte **MOV AX,1234h** from the queue. The queue has 3 free slots (≥ 2).

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

01	D8	41	EB	05	

EU now:

`MOV AX,1234h`

EU

`IP (next instruction) = 0103h`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

While the EU executes, the BIU fetches the next word (**EB 05**). No waiting.

INTERNAL STRUCTURE

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

41	EB	05	90	90	

EU now:

`ADD AX,BX`

EU

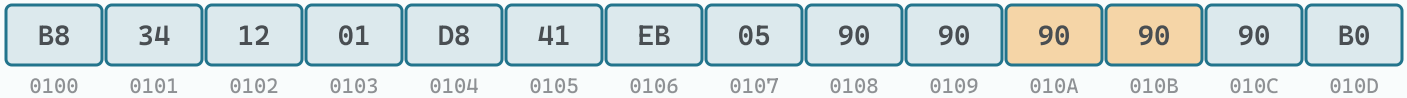
`IP (next instruction) = 0105h`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The EU took **01 D8** (ADD AX,BX); the BIU fetched **90 90**.

The Queue Step by Step

Memory (CS:address → byte)



Queue (6 bytes, FIFO) — output on the left



EU now:

`INC CX`

EU

`IP (next instruction) = 0106h`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The EU took **41** (INC CX); the BIU fetched two more NOP bytes, and the queue is full.

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

EU

EU now:

`JMP SHORT 010Dh`

`IP (next instruction) = 010Dh`

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The EU executes **EB 05**: IP = 0108h + 5 = **010Dh**. The 90s in the queue are invalid → **the queue is flushed**.

The Queue Step by Step

Memory (CS:address → byte)

B8	34	12	01	D8	41	EB	05	90	90	90	90	90	B0
0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	010A	010B	010C	010D

Queue (6 bytes, FIFO) — output on the left

B0					

EU now:

– (waiting)

EU

IP (next instruction) = 010Dh

B8 34 12 = MOV AX,1234h · 01 D8 = ADD AX,BX · 41 = INC CX · EB 05 = JMP SHORT 010Dh · 90 = NOP · B0 … = MOV AL,…

The BIU starts fetching again at 010Dh (an odd address, so 1 byte first). The EU waits for the byte.

Prefetch and Decode Functions

The prefetch function

- Fetches instructions from memory and keeps them in a **buffer** (the queue).
- A **program counter** holds the address of the next byte to fetch (on the 8086, IP and a counter inside the BIU do this).
- The EU takes instructions from the buffer instead of memory → memory access time is hidden.
- More instructions execute per unit of time.

The decode function

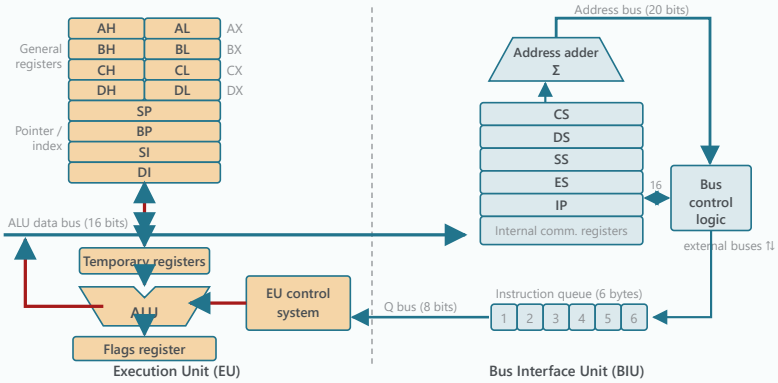
- Turns machine code into control signals/micro-operations.
- Runs in parallel with prefetching.
- Decodes complex instructions such as conditional jumps, CALL and RET.
- Steers register-to-register and memory-to-register transfers.

On the 8086 these are not separate units: prefetching happens in the **BIU**, decoding in the **EU**'s control system (with a microcode ROM). Separate prefetch and decode units appear with the 80286.

Execution Unit (EU)

Main parts of the EU:

- General-purpose registers
- Arithmetic logic unit (ALU)
- Special-purpose registers (SP, BP, SI, DI)
- Instruction register and decoder
- FLAGS (status) register
- Control system



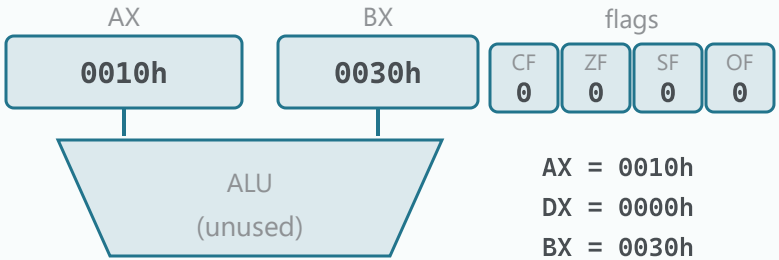
Arithmetic Logic Unit (ALU)

- Performs 8- and 16-bit **arithmetic** and **logic** operations.
- Arithmetic: add, subtract, multiply, divide, increment/decrement, compare.
- Logic: AND, OR, XOR, NOT, shifts and rotates.
- Updates the **flags** according to each result.

Instruction	Operation	Size
`ADD AX, BX`	AX ← AX + BX	16 bits
`SUB AL, 5`	AL ← AL − 5	8 bits
`MUL BL`	AX ← AL × BL	8 → 16
`DIV CX`	AX ← DX:AX ÷ CX, DX ← remainder	32 → 16
`AND AL, 0Fh`	clears the upper 4 bits of AL	8 bits
`XOR AX, AX`	AX ← 0 (ZF = 1)	16 bits

Code Trace: What Does the ALU Do?

```
1  org 100h
▶  mov ax, 0010h
▶  mov bx, 0030h
4  add ax, bx       ; AX ← AX + BX
5  sub al, 5        ; AL ← AL − 5
6  mov bl, 4
7  mul bl           ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx           ; AX ← DX:AX ÷ CX
11 and al, 0Fh      ; clear upper 4
   bits
12 xor ax, ax       ; AX ← 0
13 ret
```



The operands are loaded into registers: AX = 0010h, BX = 0030h. MOV does not use the ALU; the flags do not change.

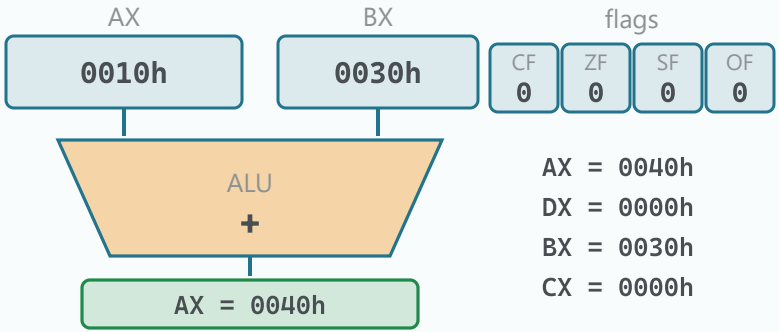
Step 1 / 8

VARIABLES

AX = 0010h | BX = 0030h | CX = 0000h | DX = 0000h | CF = 0 | ZF = 0

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
▶  add ax, bx        ; AX ← AX + BX
5  sub al, 5         ; AL ← AL − 5
6  mov bl, 4
7  mul bl            ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx            ; AX ← DX:AX ÷ CX
11 and al, 0Fh       ; clear upper 4
   bits
12 xor ax, ax        ; AX ← 0
13 ret
```



`add ax, bx` : the ALU does a 16-bit add → AX = **0040h**. No carry (CF = 0), result not zero (ZF = 0).

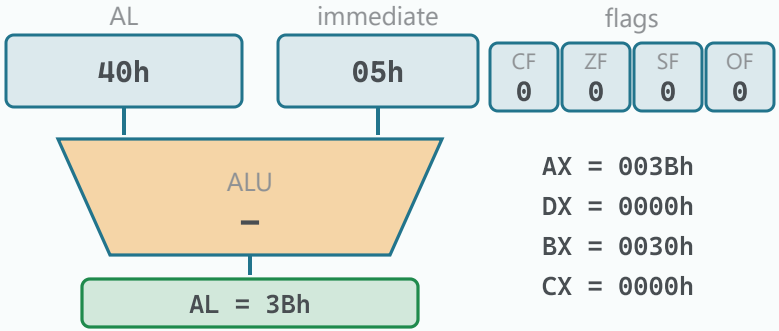
Step 2 / 8

VARIABLES

AX = 0040h | BX = 0030h | CX = 0000h | DX = 0000h | CF = 0 | ZF = 0

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx       ; AX ← AX + BX
▶  sub al, 5        ; AL ← AL - 5
6  mov bl, 4
7  mul bl           ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx           ; AX ← DX:AX ÷ CX
11 and al, 0Fh      ; clear upper 4
   bits
12 xor ax, ax       ; AX ← 0
13 ret
```



`sub al, 5` : an 8-bit subtract; only AL changes → AL = **3Bh** (AX = 003Bh). No borrow: CF = 0.

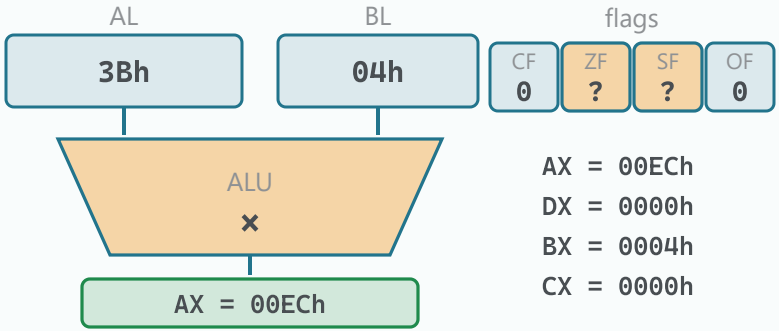
Step 3 / 8

VARIABLES

AX = 003Bh | BX = 0030h | CX = 0000h | DX = 0000h | CF = 0 | ZF = 0

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx        ; AX ← AX + BX
5  sub al, 5         ; AL ← AL − 5
▶  mov bl, 4
▶  mul bl            ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx            ; AX ← DX:AX ÷ CX
11 and al, 0Fh       ; clear upper 4
   bits
12 xor ax, ax        ; AX ← 0
13 ret
```



BL = 4. `mul bl` : the product of 8-bit operands is 16 bits: 3Bh × 04h = **00ECh** → AX. AH = 0, so CF = OF = 0; SF and ZF are undefined (?).

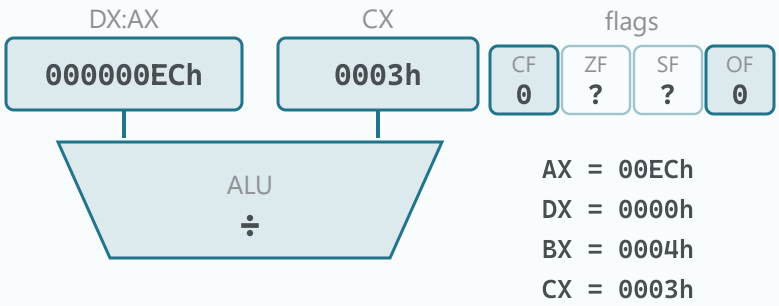
Step 4 / 8

VARIABLES

AX = 00ECh BX = 0004h CX = 0000h DX = 0000h CF = 0 ZF = ?

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx        ; AX ← AX + BX
5  sub al, 5         ; AL ← AL − 5
6  mov bl, 4
7  mul bl            ; AX ← AL × BL
▶  mov dx, 0
▶  mov cx, 3
10 div cx            ; AX ← DX:AX ÷ CX
11 and al, 0Fh       ; clear upper 4
   bits
12 xor ax, ax        ; AX ← 0
13 ret
```



Setting up the divide: the dividend is the 32-bit **DX:AX** pair, so DX = 0; the divisor is CX = 3.

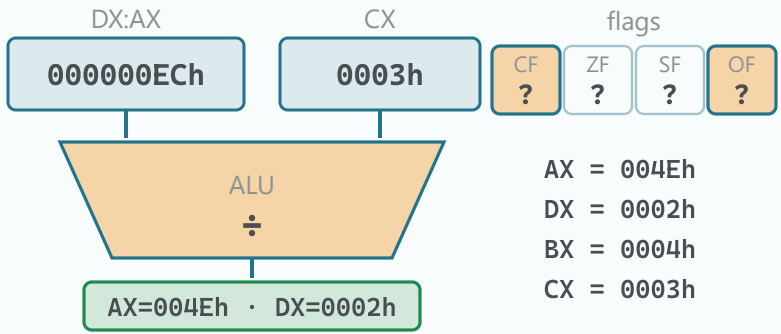
Step 5 / 8

VARIABLES

AX = **00ECh** | BX = **0004h** | CX = **0003h** | DX = **0000h** | CF = **0** | ZF = **?**

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx        ; AX ← AX + BX
5  sub al, 5         ; AL ← AL − 5
6  mov bl, 4
7  mul bl            ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
▶  div cx            ; AX ← DX:AX ÷ CX
11 and al, 0Fh       ; clear upper 4
   bits
12 xor ax, ax        ; AX ← 0
13 ret
```



`div cx` : 236 ÷ 3 = 78 remainder 2 → quotient AX = **004Eh**, remainder DX = **0002h**. Flags are undefined after DIV.

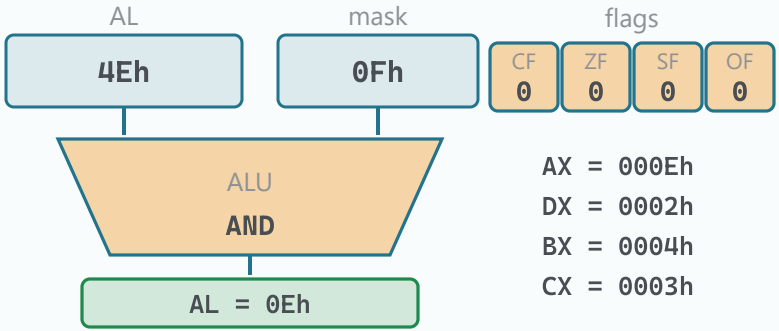
Step 6 / 8

VARIABLES

AX = 004Eh | BX = 0004h | CX = 0003h | DX = 0002h | CF = ? | ZF = ?

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx        ; AX ← AX + BX
5  sub al, 5         ; AL ← AL − 5
6  mov bl, 4
7  mul bl            ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx            ; AX ← DX:AX ÷ CX
▶  and al, 0Fh       ; clear upper 4 bits
12 xor ax, ax        ; AX ← 0
13 ret
```



`and al, 0Fh` : 0100 1110 AND 0000 1111 = 0000 1110 → AL = **0Eh**. The upper 4 bits are cleared; AND sets CF = OF = 0.

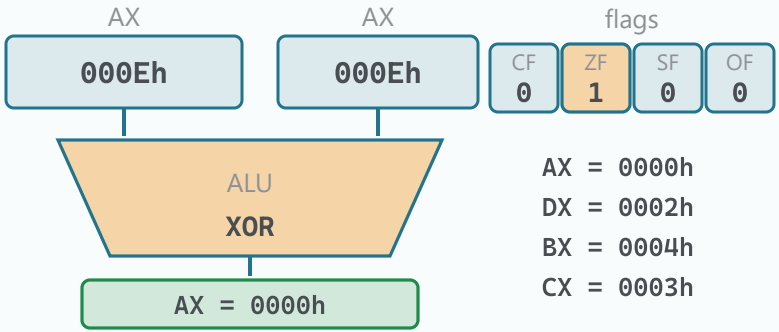
Step 7 / 8

VARIABLES

AX = 000Eh | BX = 0004h | CX = 0003h | DX = 0002h | CF = 0 | ZF = 0

Code Trace: What Does the ALU Do?

```
1  org 100h
2  mov ax, 0010h
3  mov bx, 0030h
4  add ax, bx       ; AX ← AX + BX
5  sub al, 5        ; AL ← AL − 5
6  mov bl, 4
7  mul bl           ; AX ← AL × BL
8  mov dx, 0
9  mov cx, 3
10 div cx           ; AX ← DX:AX ÷ CX
11 and al, 0Fh      ; clear upper 4
   bits
▶  xor ax, ax       ; AX ← 0
13 ret
```



`xor ax, ax` : any bit XORed with itself is 0 → AX = **0000h**, **ZF = 1**. A shorter (2-byte) way to clear than `mov ax, 0`.

Step 8 / 8

VARIABLES

AX = 0000h | BX = 0004h | CX = 0003h | DX = 0002h | CF = 0 | ZF = 1

INTERNAL STRUCTURE

Instruction Register, Decoder and Control

Instruction register

The EU takes the next instruction from the queue; its opcode is held here while it is decoded. Programs cannot access this register.

Instruction decoder

Decodes the opcode: which operation, which operands, which addressing mode? Passes that information to the control circuitry.

Control unit

Directs the flow of instructions and data inside the processor; sequences the fetch–decode–execute cycle and coordinates the other parts.

The control unit **fetches** instructions (through the BIU), **decodes** and **executes** them, updates the processor state (registers, flags), and checks for **interrupt requests** (INTR, NMI) at the end of each instruction. Error conditions (e.g. divide by zero) are handled by special interrupts.

The 8086 has no power management; those features came much later (e.g. the 386SL). The 8086's control unit is microcoded.

Instruction Lengths

8086 instructions have **variable length**: opcode + (optional) addressing byte + displacement + immediate data.

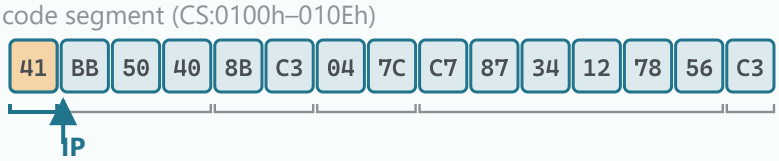
Instruction	Machine code (hex)	Bytes	Description
`INC CX`	`41`	1	opcode only
`MOV AX, BX`	`8B C3`	2	opcode + ModR/M byte
`ADD AL, 7Ch`	`04 7C`	2	short AL form + 8-bit immediate
`MOV BX, 4050h`	`BB 50 40`	3	opcode + 16-bit immediate (little-endian)
`MOV WORD PTR [BX+1234h], 5678h`	`C7 87 34 12 78 56`	6	ModR/M + 16-bit displacement + 16-bit immediate

Immediate values are stored **little-endian** (low byte first): `4050h` → `50` first, then `40`.

A register-to-register MOV is 2 bytes (`8B C3` or the equivalent `89 D8`).

Code Trace: Bytes and IP

```
1 org 100h
▶ inc cx
3 mov bx, 4050h
4 mov ax, bx
5 add al, 7Ch
6 mov word ptr [bx+1234h], 5678h
7 ret
```



INC CX

1 byte · opcode

IP: 0100h → 0101h

The program starts at 0100h. `inc cx` is just an opcode: **41** (1 byte) → IP = **0101h**. CX = 1.

Step 1 / 6

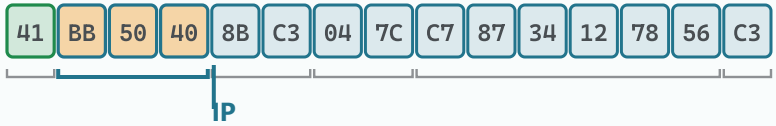
VARIABLES

IP = **0101h** CX = **0001h** BX = **0000h** AX = **0000h**

Code Trace: Bytes and IP

```
1 org 100h
2 inc cx
▶ mov bx, 4050h
4 mov ax, bx
5 add al, 7Ch
6 mov word ptr [bx+1234h], 5678h
7 ret
```

code segment (CS:0100h–010Eh)



`MOV BX,4050h`

3 bytes · opcode + 16-bit immediate

`IP: 0101h → 0104h`

BB 50 40: BB = “16-bit constant into BX”; the constant is little-endian (50 first, then 40). 3 bytes → IP = **0104h**.

Step 2 / 6

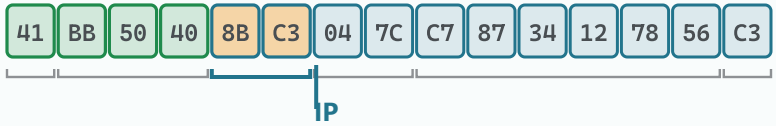
VARIABLES

`IP = 0104h` `CX = 0001h` `BX = 4050h` `AX = 0000h`

Code Trace: Bytes and IP

```
1 org 100h
2 inc cx
3 mov bx, 4050h
▶ mov ax, bx
5 add al, 7Ch
6 mov word ptr [bx+1234h], 5678h
7 ret
```

code segment (CS:0100h–010Eh)



`MOV AX,BX`

2 bytes · opcode + ModR/M

`IP: 0104h → 0106h`

8B C3: 8B = MOV r16, r/m16; ModR/M C3 = 11 000 011 → mod 11 (register), reg AX, r/m BX. 2 bytes → IP = **0106h**.

Step 3 / 6

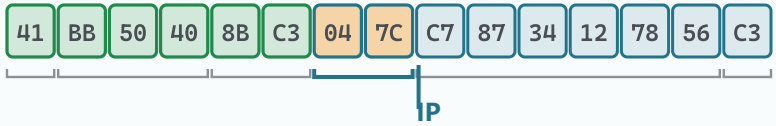
VARIABLES

`IP = 0106h` `CX = 0001h` `BX = 4050h` `AX = 4050h`

Code Trace: Bytes and IP

```
1 org 100h
2 inc cx
3 mov bx, 4050h
4 mov ax, bx
▶ add al, 7Ch
6 mov word ptr [bx+1234h], 5678h
7 ret
```

code segment (CS:0100h–010Eh)



`ADD AL,7Ch`

2 bytes · short AL form + 8-bit immediate

`IP: 0106h → 0108h`

04 7C: the short AL form + an 8-bit constant. AL = 50h + 7Ch = **CCh**, AX = 40CCh. IP = **0108h**.

Step 4 / 6

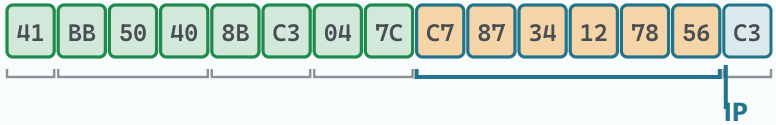
VARIABLES

`IP = 0108h` `CX = 0001h` `BX = 4050h` `AX = 40CCh`

Code Trace: Bytes and IP

```
1 org 100h
2 inc cx
3 mov bx, 4050h
4 mov ax, bx
5 add al, 7Ch
▶ mov word ptr [bx+1234h], 5678h
7 ret
```

code segment (CS:0100h–010Eh)



`MOV WORD PTR [BX+1234h],5678h`

6 bytes · opcode + ModR/M + disp + imm

`IP: 0108h → 010Eh`

DS:5284h



data: 5678h

6 bytes: C7 + ModR/M 87 (mod 10: 16-bit displacement, r/m BX) + **34 12** + **78 56**. EA = 4050h + 1234h = 5284h; 5678h is written there. IP = **010Eh**.

Step 5 / 6

VARIABLES

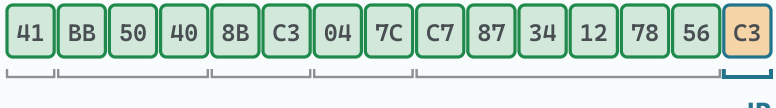
`IP = 010Eh` `CX = 0001h` `BX = 4050h` `AX = 40CCh`

`[5284h] = 5678h`

Code Trace: Bytes and IP

```
1 org 100h
2 inc cx
3 mov bx, 4050h
4 mov ax, bx
5 add al, 7Ch
6 mov word ptr [bx+1234h], 5678h
▶ ret
```

code segment (CS:0100h–010Eh)



RET

1 byte · opcode

IP: 010Eh → 0000h

DS:5284h



data: 5678h

C3 = RET (1 byte); IP gets 0000h from the stack (the COM program returns to DOS). Six instructions took 15 bytes: lengths range from 1 to 6 bytes.

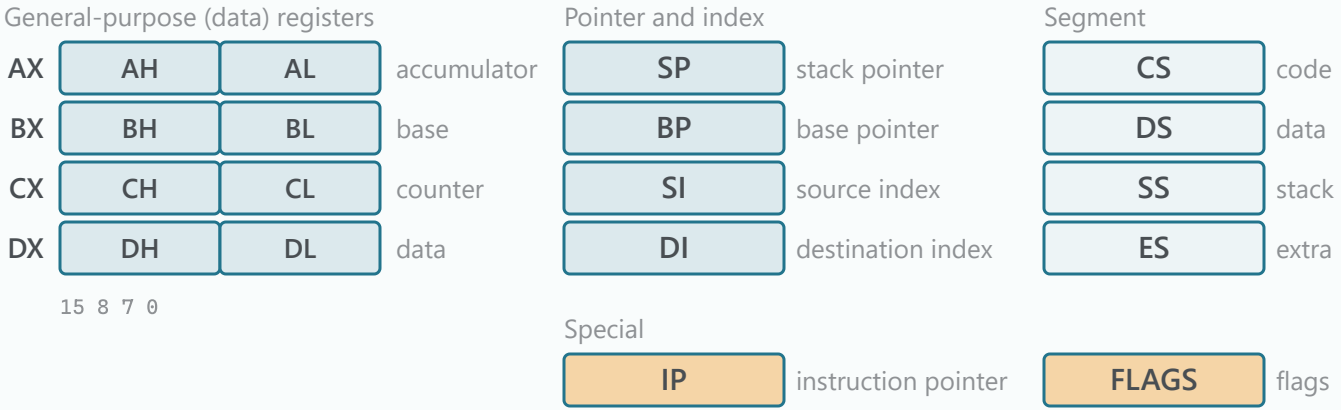
Step 6 / 6

VARIABLES

IP = **0000h** CX = **0001h** BX = **4050h** AX = **40CCh**

[5284h] = **5678h**

The Register Set



4 + 4 + 4 + 2 = 14 registers of 16 bits

- **General-purpose**: hold data and take part in arithmetic/logic; each splits into two 8-bit halves.
- **Segment registers**: point to the bases of the four active segments in memory. **Special**: IP points to the next instruction, FLAGS reflects the last result.

Internal Registers and Word Size

- The 8086 is a 16-bit **integer** (scalar, non-vector) processor; floating point needs a separate coprocessor (the 8087).
- It comes in a 40-pin **dual in-line package** (DIP).
- The size of the on-chip registers sets how much data the processor handles **at once** (the word size) → 16 bits.
- It gives the programmer **14 registers of 16 bits**.
- Registers are the fastest storage in the processor: a memory access needs a bus cycle (≥ 4 clocks), a register access does not.

8 bits	**16 bits**	**32 bits (DX:AX)**
0 … 255 (FFh)	0 … 65,535 (FFFFh)	0 … 4,294,967,295

Segment Registers

- Segments live **in memory**; segment registers live **inside the processor**.
- Each segment register holds a segment's **start address / 16** (segment base = register × 10h).

CS Code Segment	Program code; `CS:IP` is the next instruction
DS Data Segment	Variables and data
SS Stack Segment	The stack; `SS:SP`, `SS:BP`
ES Extra Segment	Extra data; string destination `ES:DI`

Rules

- A segment register cannot be loaded with an immediate: `MOV DS, 2000h` is **invalid**; use `MOV AX, 2000h`, then `MOV DS, AX`.
- `CS` cannot be changed with MOV; JMP FAR, CALL FAR, RET FAR, INT and IRET change it.
- There is no direct segment-to-segment MOV (`MOV ES, DS` is invalid).

Instruction Pointer (IP)

- A 16-bit register.
- Holds the offset of the **next instruction** in the code segment.
- Advances by each instruction's length as its bytes are fetched.
- Branch instructions (JMP, Jcc, CALL, RET, LOOP, INT) load IP with a **new value**.
- Programs cannot read or write IP directly (there is no `MOV IP, …`).
- Physical address of the next instruction: **CS × 10h + IP**.

Example

CS = `4321h`, IP = `1000h`

```
  43210h   ; CS × 10h
+ 01000h   ; IP
--------
  44210h
```

The next instruction is at physical address **44210h**.

Segmented Memory Architecture

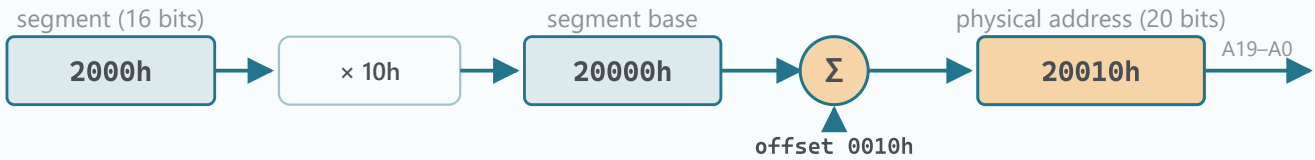
- The 8086 addresses memory in **segments**.
- Every memory address is given as two 16-bit parts: **segment : offset**, e.g. `2000h:0010h`.
- The **segment register** points to the start of the segment.
- The **offset** is the distance of the byte from the start of the segment (0000h–FFFFh).
- The address adder in the BIU combines the two into a **20-bit physical address**.

Logical vs. physical address

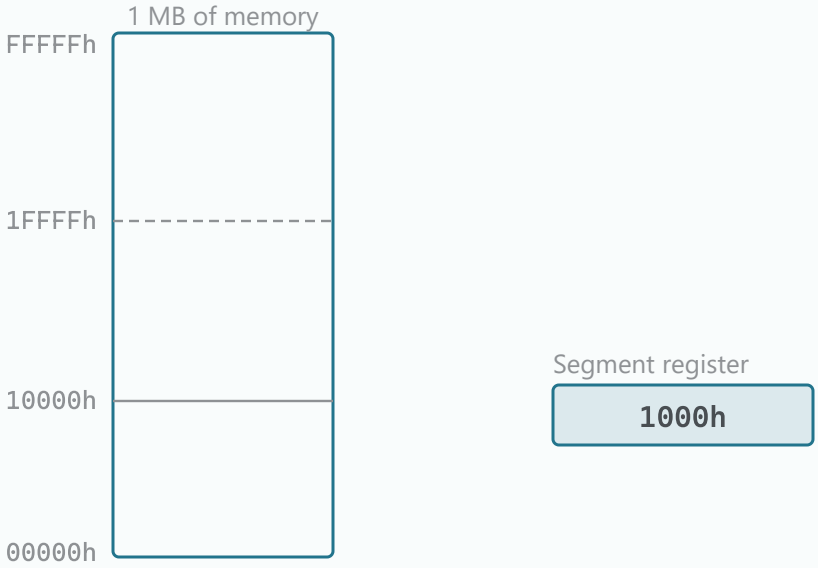
Logical address: the `segment:offset` pair the program uses.

Physical address: the actual 20-bit address placed on the address bus.

PA = segment × 10h + offset

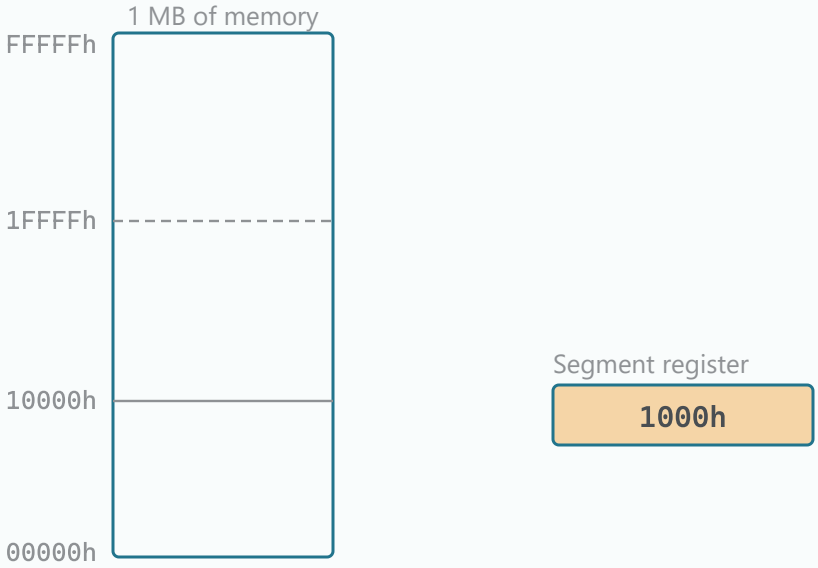


Segment and Offset



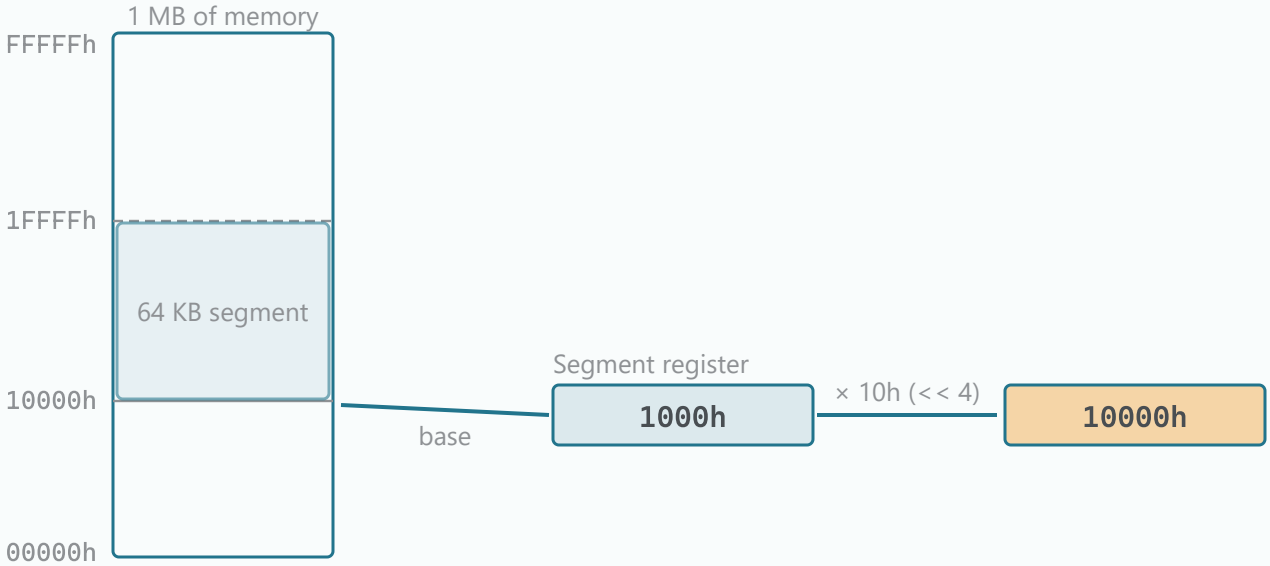
Memory is 1 MB: 00000h–FFFFFh. We will build a 20-bit address from a segment register and an offset.

Segment and Offset



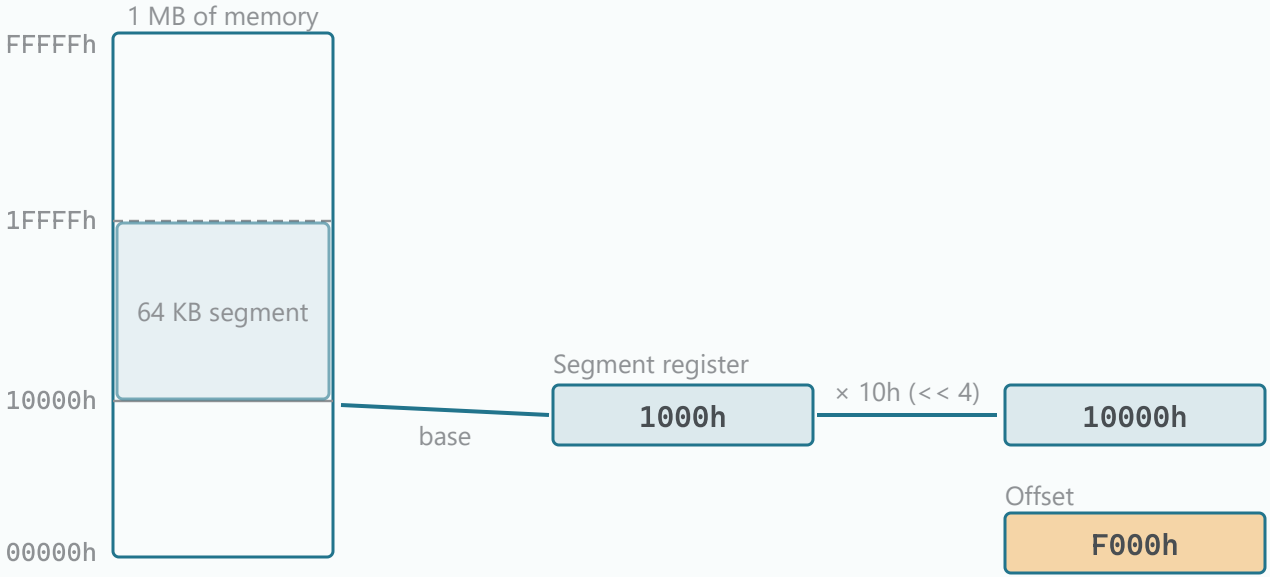
The segment register holds **1000h** (16 bits).

Segment and Offset



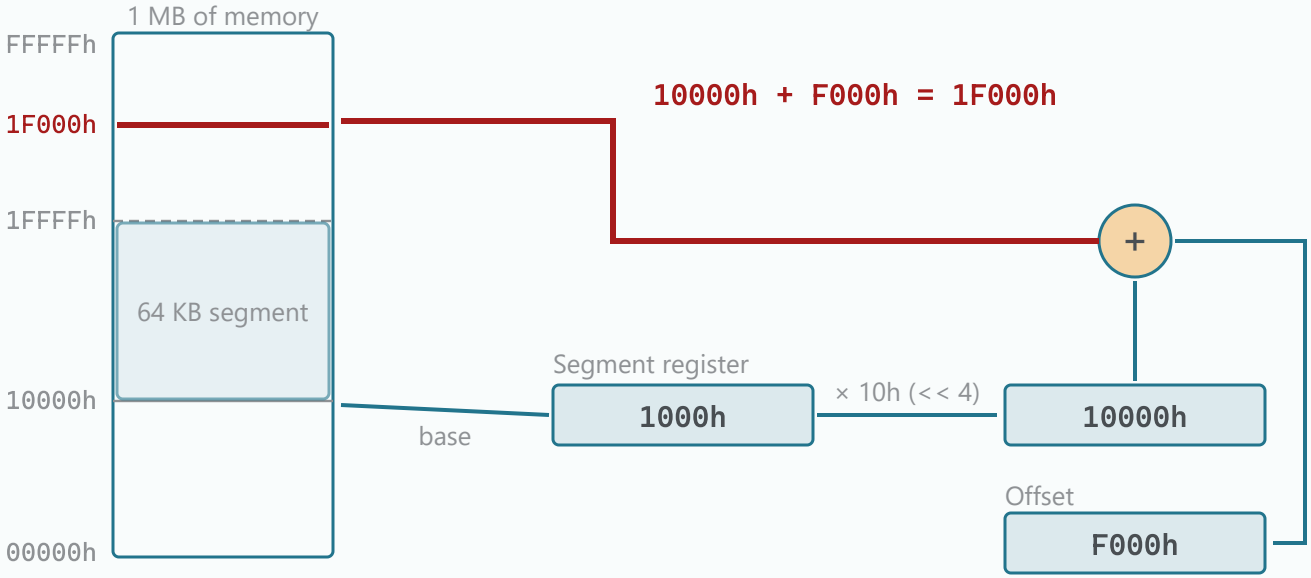
The segment value is **shifted left** 4 bits (× 10h): 1000h → **10000h**. A segment base is always a multiple of 16 (a paragraph boundary).

Segment and Offset



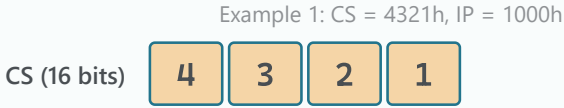
The offset is **F000h**. The largest offset is FFFFh, so the segment spans 10000h–1FFFFh (64 KB).

Segment and Offset



The adder: **10000h + F000h = 1F000h**. This 20-bit value goes onto the address bus.

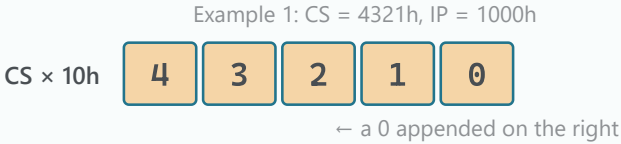
Step by Step: CS:IP → Physical Address



CS = **4321h** (4 hex digits = 16 bits).

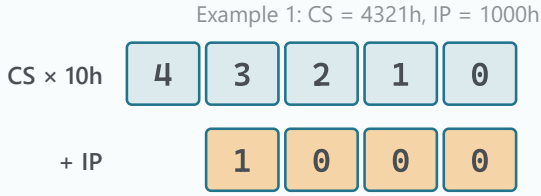
Step 1 / 8

Step by Step: CS:IP → Physical Address



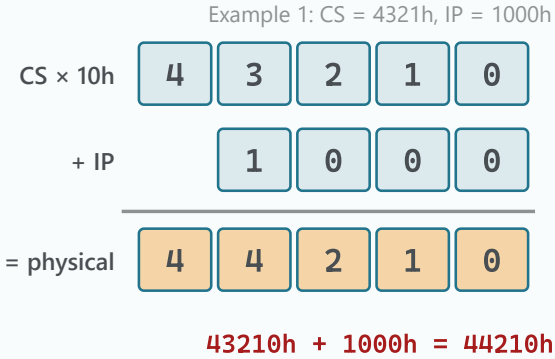
× 10h = append a **0** on the right in hex (shift left 4 bits): **43210h**.

Step by Step: CS:IP → Physical Address



IP = **1000h** is written below, right-aligned (the leftmost digit is blank = 0).

Step by Step: CS:IP → Physical Address



Add the digits from the right: 0+0, 1+0, 2+0, 3+1, 4+0 → **44210h**. No carry.

Step by Step: CS:IP → Physical Address

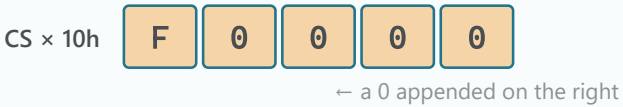
Example 2: CS = F000h, IP = FFF0h



CS = **F000h**, IP = **FFF0h**: the address of the first instruction in the BIOS **after reset** (the 8086 loads CS = FFFFh, IP = 0000h on reset; both give the same byte).

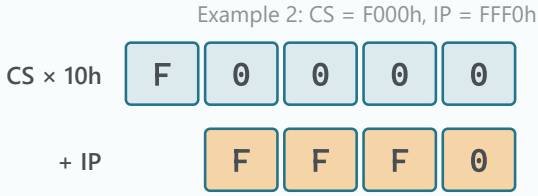
Step by Step: CS:IP → Physical Address

Example 2: CS = F000h, IP = FFF0h



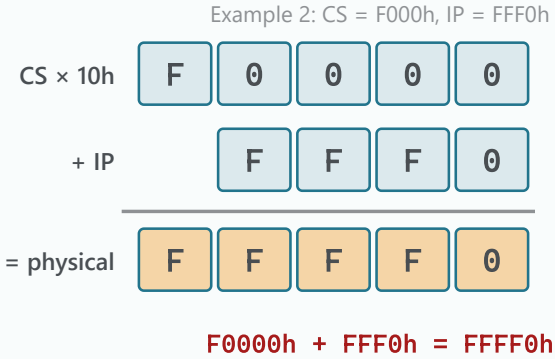
F000h × 10h = **F0000h**.

Step by Step: CS:IP → Physical Address



IP = FFF0h is right-aligned.

Step by Step: CS:IP → Physical Address



F0000h + FFF0h = **FFFF0h**: the last 16 bytes of the 1 MB. After reset the processor takes its first instruction from here (the same address as FFFFh:0000h).

Demo: Physical Address Calculator

Segment : Offset **random**

4321 h : 1000 h

```
  4321h      ; segment
  43210h     ; segment × 10h (shl 4)
+  1000h     ; offset
-----------
  44210h     ; physical address
```

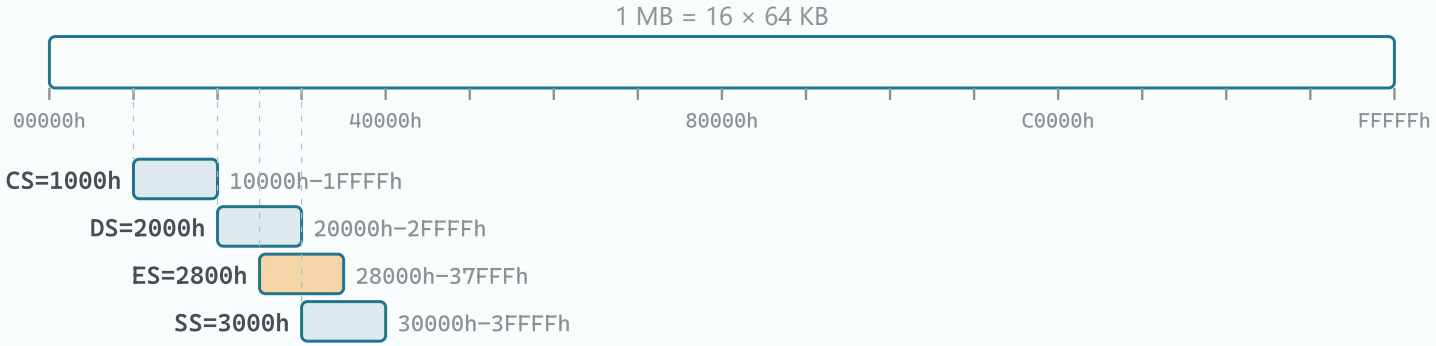


Segment range **43210h–5320Fh** · normalized form **4421h:0000h**

The map shows the segment base and the 64 KB segment range. If the sum passes 1 MB, the 8086 truncates the address to 20 bits (wraparound).

SEGMENTED MEMORY

Memory Segmentation



The ES segment overlaps DS and SS: segments may overlap.

- The 20-bit address bus reaches 1 MB; a segment is at most **64 KB** (16-bit offset).
- Only **4 segments are active** at any time (CS, DS, SS, ES) → at most 256 KB is directly reachable at once. To reach another area, change a segment register.
- Segments are not 16 fixed pieces: one can start at any **paragraph** (16-byte) boundary, and segments may overlap or be disjoint.

1 MB / 64 KB = 16 is only arithmetic; 8086 segment starts are not fixed (65,536 possible starts).

Why Segmentation?

Pros

+ 16-bit registers can address 20 bits (1 MB) of memory.
+ Code, data and stack are kept apart; the program layout is clear.
+ **Relocatability**: a program can be loaded elsewhere in memory just by changing the segment registers.
+ Instructions stay short: most addresses are written as 16-bit offsets.

Cons

– A single array or structure cannot exceed 64 KB (otherwise you must switch segments).
– One physical address has many `segment:offset` forms → comparing pointers gets harder.
– There is no protection between segments: a bad write can corrupt another segment.
– Address arithmetic burdens the programmer.

SEGMENTED MEMORY

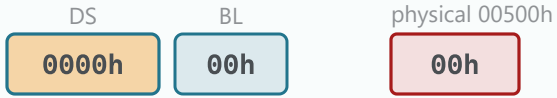
Same Address, Different Segment:Offset

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)

Normalization: to bring the offset into 0–Fh, take segment = the first 4 digits of the PA and offset = the last digit. 12345h → `1234h:0005h` . A physical address has at most 4096 different forms.

Code Trace: The Address Wraps Around at 1 MB

```
1 org 100h
▶ mov ax, 0
▶ mov ds, ax
4 mov byte ptr [0500h], 77h
5 mov ax, 0FFFFh
6 mov ds, ax
7 mov bl, [0510h]    ; FFFF:0510
8 ret
```



DS = 0000h: the segment is at the very start of memory. A segment register cannot take a constant, so the value goes through AX.

Step 1 / 5

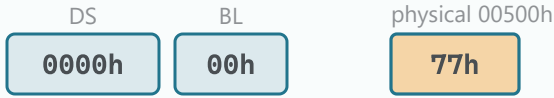
VARIABLES

DS = 0000h BL = 00h

On the 80286 and later, if the 21st address line (A20) is enabled, FFFF:0510 really reaches 100500h; it can be disabled because some old 8086 programs rely on the wraparound.

Code Trace: The Address Wraps Around at 1 MB

```
1 org 100h
2 mov ax, 0
3 mov ds, ax
▶ mov byte ptr [0500h], 77h
5 mov ax, 0FFFFh
6 mov ds, ax
7 mov bl, [0510h]    ; FFFF:0510
8 ret
```



`0000h × 10h + 0500h`
= 00500h

`mov byte ptr [0500h], 77h` : 0000:0500 → 77h is written to physical **00500h**.

Step 2 / 5

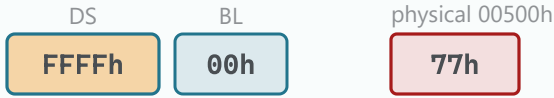
VARIABLES

DS = **0000h** BL = **00h**

On the 80286 and later, if the 21st address line (A20) is enabled, FFFF:0510 really reaches 100500h; it can be disabled because some old 8086 programs rely on the wraparound.

Code Trace: The Address Wraps Around at 1 MB

```
1 org 100h
2 mov ax, 0
3 mov ds, ax
4 mov byte ptr [0500h], 77h
▶ mov ax, 0FFFFh
▶ mov ds, ax
7 mov bl, [0510h]    ; FFFF:0510
8 ret
```



DS = **FFFFh**: the segment starts at FFFF0h, the last 16 bytes of the 1 MB.

Step 3 / 5

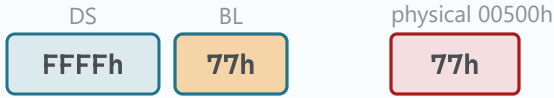
VARIABLES

DS = **FFFFh** BL = **00h**

On the 80286 and later, if the 21st address line (A20) is enabled, FFFF:0510 really reaches 100500h; it can be disabled because some old 8086 programs rely on the wraparound.

Code Trace: The Address Wraps Around at 1 MB

```
1 org 100h
2 mov ax, 0
3 mov ds, ax
4 mov byte ptr [0500h], 77h
5 mov ax, 0FFFFh
6 mov ds, ax
▶ mov bl, [0510h]    ; FFFF:0510
8 ret
```



`FFFF0h + 0510h = 100500h`

bit 21 dropped → 00500h

wrapped

`mov bl, [0510h]` : the sum needs 21 bits; the 8086's 20 address lines carry only 00500h. The same byte is read: BL = **77h**.

Step 4 / 5

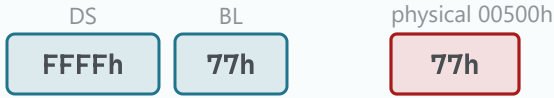
VARIABLES

DS = **FFFFh** BL = **77h**

On the 80286 and later, if the 21st address line (A20) is enabled, FFFF:0510 really reaches 100500h; it can be disabled because some old 8086 programs rely on the wraparound.

Code Trace: The Address Wraps Around at 1 MB

```
1 org 100h
2 mov ax, 0
3 mov ds, ax
4 mov byte ptr [0500h], 77h
5 mov ax, 0FFFFh
6 mov ds, ax
7 mov bl, [0510h]    ; FFFF:0510
▶ ret
```



RET. Addresses from FFFF:0010 to FFFF:FFFF fall into 00000h–0FFEFh at the start of memory.

Step 5 / 5

VARIABLES

DS = **FFFFh** BL = **77h**

On the 80286 and later, if the 21st address line (A20) is enabled, FFFF:0510 really reaches 100500h; it can be disabled because some old 8086 programs rely on the wraparound.

Default Segments

Access type	Segment	Offset source	Can it be overridden?
Instruction fetch	`CS`	`IP`	no
Stack operations (PUSH, POP, CALL, RET)	`SS`	`SP`	no
BP addressing `[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

A **segment override prefix** changes the default segment: `MOV AL, ES:[BX]` → physical address = ES × 10h + BX. The prefix adds 1 byte (`26h`) to the instruction.

Code Trace: One Offset, Three Segments

```
1  org 100h
▶  mov ax, 2000h
▶  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

segment registers

`DS = 2000h`

`ES = 0700h`

`SS = 0700h`

`BX = 0000h · BP = 0000h`

ES segment (not yet)

30010h ??

DS segment

20010h ??

SS segment

07010h ??

A segment register cannot take a constant; **DS = 2000h** goes through AX. In a COM program SS = CS (**0700h** here).

Step 1 / 8

VARIABLES

`DS = 2000h` `ES = 0700h` `SS = 0700h` `BX = 0000h` `BP = 0000h` `AX = 2000h`

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
▶  mov ax, 3000h
▶  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

segment registers

DS = 2000h

ES = 3000h

SS = 0700h

BX = 0000h · BP = 0000h

ES segment	
30010h	??
DS segment	
20010h	??
SS segment	
07010h	??

Likewise **ES = 3000h**. Now there are three segment bases: 07000h (SS), 20000h (DS), 30000h (ES).

Step 2 / 8

VARIABLES

DS = 2000h · ES = 3000h · SS = 0700h · BX = 0000h · BP = 0000h · AX = 3000h

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
▶  mov bx, 0010h
▶  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

segment registers

DS = 2000h

ES = 3000h

SS = 0700h

BX = 0010h · BP = 0010h

ES segment
30010h ??

DS segment
20010h ??

SS segment
07010h ??

BX and BP hold the same offset: **0010h**. The **segment used**, not the offset, decides which byte is accessed.

Step 3 / 8

VARIABLES

DS = 2000h | ES = 3000h | SS = 0700h | BX = 0010h | BP = 0010h | AX = 3000h

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
▶  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

DS × 10h + BX

```
20000h
+ 0010h
-------
20010h
```

write 11h

ES segment
30010h → ??

DS segment
20010h → 11h

SS segment
07010h → ??

`[bx]` uses **DS** by default: 2000h × 10h + 0010h = **20010h** ← 11h.

Step 4 / 8

VARIABLES

DS = **2000h** · ES = **3000h** · SS = **0700h** · BX = **0010h** · BP = **0010h** · AX = **3000h**

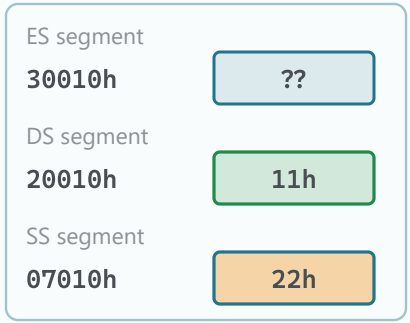
Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
▶  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

SS × 10h + BP

```
07000h
+ 0010h
-------
07010h
```

write 22h



`[bp]` uses **SS** by default: 0700h × 10h + 0010h = **07010h** ← 22h. Same offset, different byte.

Step 5 / 8

VARIABLES

DS = **2000h** · ES = **3000h** · SS = **0700h** · BX = **0010h** · BP = **0010h** · AX = **3000h**

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
▶ mov byte ptr es:[bx], 33h ; ES
11 mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

ES × 10h + BX

```
30000h
+ 0010h
-------
30010h
```

write 33h

ES segment
30010h — 33h

DS segment
20010h — 11h

SS segment
07010h — 22h

The `es:` prefix selects **ES** instead of DS (+1 byte, 26h): 30000h + 0010h = **30010h** ← 33h.

Step 6 / 8

VARIABLES

DS = **2000h** · ES = **3000h** · SS = **0700h** · BX = **0010h** · BP = **0010h** · AX = **3000h**

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
▶  mov al, [bx]               ; DS
12 mov ah, [bp]               ; SS
13 ret
```

DS × 10h + BX

```
  20000h
+ 0010h
--------
  20010h
```

read → AL

ES segment
30010h → 33h

DS segment
20010h → 11h

SS segment
07010h → 22h

`mov al, [bx]` reads DS:0010h again → AL = **11h** (not the 33h at ES:0010h).

Step 7 / 8

VARIABLES

DS = 2000h | ES = 3000h | SS = 0700h | BX = 0010h | BP = 0010h | AX = 3011h

Code Trace: One Offset, Three Segments

```
1  org 100h
2  mov ax, 2000h
3  mov ds, ax
4  mov ax, 3000h
5  mov es, ax
6  mov bx, 0010h
7  mov bp, 0010h
8  mov byte ptr [bx], 11h     ; DS
9  mov byte ptr [bp], 22h     ; SS
10 mov byte ptr es:[bx], 33h  ; ES
11 mov al, [bx]               ; DS
▶  mov ah, [bp]               ; SS
13 ret
```

SS × 10h + BP

```
07000h
+ 0010h
-------
07010h
```

read → AH

ES segment
30010h → 33h

DS segment
20010h → 11h

SS segment
07010h → 22h

`mov ah, [bp]` reads SS:0010h → AH = **22h**. Result AX = **2211h**: three accesses, one offset, three physical addresses.

Step 8 / 8

VARIABLES

DS = 2000h | ES = 3000h | SS = 0700h | BX = 0010h | BP = 0010h | AX = 2211h

Code Trace: Access by Segment:Offset

AX ← 2000h. A segment register cannot take an immediate, so the value goes into AX first.

Step 1 / 10

```
▶ mov ax, 2000h
2  mov ds, ax          ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax          ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]        ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]        ; last byte
```

VARIABLES

AX = **2000h** DS = **?** BX = **0000h**

SI = **0000h** FA = **–**

Code Trace: Access by Segment:Offset

DS ← AX = 2000h. The data segment is now **20000h–2FFFFh**.

Step 2 / 10

```
1  mov ax, 2000h
▶  mov ds, ax          ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax          ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]        ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]        ; last byte
```

VARIABLES

AX = **2000h** | DS = **2000h** | BX = **0000h** | SI = **0000h** | FA = **–**

Code Trace: Access by Segment:Offset

BX ← 0010h (offset).

Step 3 / 10

```
1  mov ax, 2000h
2  mov ds, ax          ; DS = 2000h
▶  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax          ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]        ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]        ; last byte
```

VARIABLES

AX = **2000h** | DS = **2000h** | BX = **0010h** | SI = **0000h** | FA = **–**

Code Trace: Access by Segment:Offset

PA = 2000h×10h + 0010h = **20010h**. 41h ('A') is written there.

Step 4 / 10

```
1  mov ax, 2000h
2  mov ds, ax           ; DS = 2000h
3  mov bx, 0010h
▶  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax           ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]         ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]         ; last byte
```

VARIABLES

AX = **2000h** | DS = **2000h** | BX = **0010h**

SI = **0000h** | FA = **20010h** | [20010h] = **41h**

Code Trace: Access by Segment:Offset

AX ← 2001h.

Step 5 / 10

```
1  mov ax, 2000h
2  mov ds, ax          ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
▶  mov ax, 2001h
6  mov ds, ax          ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]        ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]        ; last byte
```

VARIABLES

AX = **2001h** | DS = **2000h** | BX = **0010h** | SI = **0000h** | FA = **20010h** | [20010h] = **41h**

Code Trace: Access by Segment:Offset

DS ← 2001h: the segment base moved 16 bytes (one paragraph) → **20010h–3000Fh**.

Step 6 / 10

```
1  mov ax, 2000h
2  mov ds, ax           ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
▶  mov ds, ax           ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]         ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]         ; last byte
```

VARIABLES

AX = **2001h** | DS = **2001h** | BX = **0010h**

SI = **0000h** | FA = **20010h** | [20010h] = **41h**

Code Trace: Access by Segment:Offset

BX ← 0000h.

Step 7 / 10

```
1  mov ax, 2000h
2  mov ds, ax           ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax           ; +16 bytes
▶  mov bx, 0000h
8  mov al, [bx]         ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]         ; last byte
```

VARIABLES

AX = **2001h** | DS = **2001h** | BX = **0000h** | SI = **0000h** | FA = **20010h** | [20010h] = **41h**

Code Trace: Access by Segment:Offset

PA = 20010h + 0000h = **20010h** → AL = 41h. `2000h:0010h` and `2001h:0000h` are **the same byte**!

Step 8 / 10

```
1  mov ax, 2000h
2  mov ds, ax           ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax           ; +16 bytes
7  mov bx, 0000h
▶  mov al, [bx]         ; same byte
9  mov si, 0FFFFh
10 mov ah, [si]         ; last byte
```

VARIABLES

- AX = **2041h**
- DS = **2001h**
- BX = **0000h**
- SI = **0000h**
- FA = **20010h**
- [20010h] = **41h**

Code Trace: Access by Segment:Offset

SI ← FFFFh: the largest offset.

Step 9 / 10

```
1  mov ax, 2000h
2  mov ds, ax           ; DS = 2000h
3  mov bx, 0010h
4  mov byte ptr [bx], 41h
5  mov ax, 2001h
6  mov ds, ax           ; +16 bytes
7  mov bx, 0000h
8  mov al, [bx]         ; same byte
▶  mov si, 0FFFFh
10 mov ah, [si]         ; last byte
```

VARIABLES

AX = **2041h** | DS = **2001h** | BX = **0000h**

SI = **FFFFh** | FA = **20010h** | [20010h] = **41h**

Code Trace: Access by Segment:Offset

PA = 20010h + FFFFh = **3000Fh**: the last byte of this segment. AH gets the value in that byte.

Step 10 / 10

```
1 mov ax, 2000h
2 mov ds, ax          ; DS = 2000h
3 mov bx, 0010h
4 mov byte ptr [bx], 41h
5 mov ax, 2001h
6 mov ds, ax          ; +16 bytes
7 mov bx, 0000h
8 mov al, [bx]        ; same byte
9 mov si, 0FFFFh
▶ mov ah, [si]        ; last byte
```

VARIABLES

AX = ??41h | DS = 2001h | BX = 0000h | SI = FFFFh | FA = 3000Fh | [20010h] = 41h

Summary

Overview

16-bit data bus and registers, 20-bit address bus → 1 MB. CISC, variable-length instructions.

BIU + EU

The BIU fetches and forms addresses; the EU decodes and executes. A 6-byte FIFO queue lets them work in parallel.

14 registers

AX, BX, CX, DX · SP, BP, SI, DI · CS, DS, SS, ES · IP, FLAGS.

Segment registers

CS, DS, SS and ES point to the bases of the four active segments; a segment register cannot take an immediate.

Address calculation

PA = segment × 10h + offset. A segment is ≤ 64 KB, starts on a paragraph boundary, and segments may overlap.

Branches

Load IP with a new value and flush the prefetch queue.

SUMMARY

Exercises

1. If CS = `2F00h` and IP = `01A4h`, what is the physical address of the next instruction?
2. Write the physical address `3A2C5h` in normalized `segment:offset` form; with DS = `3000h`, what must the offset be?
3. Why does `MOV DS, 1000h` not assemble? Give the correct form.
4. If the EU executes a `JMP` while the queue holds 6 bytes, what happens to the queue? Why?
5. Can the 8086 execute one instruction while fetching another? Thanks to which units?

▶ **Hints / answers**

References

- Intel, *8086 16-Bit HMOS Microprocessor* data sheet (231455).
- Intel, *The 8086 Family User's Manual*, 1979.
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.
- M. A. Mazidi, J. G. Mazidi, D. Causey, *The x86 PC: Assembly Language, Design, and Interfacing*, 5th ed., Pearson, 2010.
- emu8086 documentation and instruction set reference.

The End

Chapter 2.1: 8086 Architecture

Previous: Chapter 1.2: Number Systems · Next: Chapter 2.2: 8086 Registers