Chapter 3.2: Assembly Language

Microprocessors

Assembly language | Source line format | Instructions and directives | Operand types | Implicit operands | Instruction timing | IP and flags | A first program

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_03_02_Assembly_Language.html

Learning Outcomes

Where assembly fits

Relate assembly language to machine language and to high-level languages.

Line format

Read a source line as label, mnemonic, operands and comment; tell an instruction from a directive.

Operands

Recognize register, immediate and memory operands, and the implicit operands of instructions.

Instruction timing

Compare register and memory forms of an instruction in clock cycles.

IP and flags

Follow how IP advances and which instructions change the flags.

A first program

Write a simple .COM program in emu8086 and single-step through it.

Assembly Language

- A **low-level** language: each instruction corresponds to exactly one machine instruction.
- Instead of raw numbers it uses readable **mnemonics**: `MOV`, `ADD`, `JMP`...
- Writing it **requires** knowing the machine: registers, memory, the system bus.
- It is the best way to **understand** how a computer is organized and how a program really runs.
- Every processor family has its own assembly language; this course uses the **Intel 8086** and **emu8086**.

One line, three views

Assembly: `MOV AX, 5`

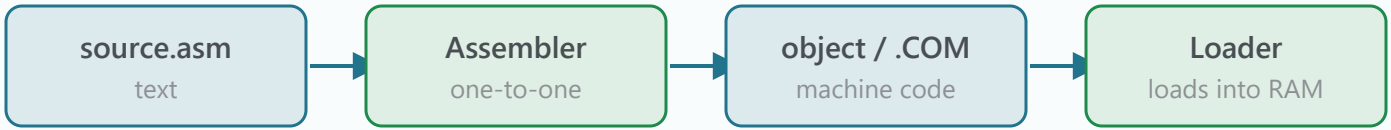
Machine code in hex: `B8 05 00`

What the CPU sees:

```
10111000 00000101 00000000
```

The 16-bit value 0005h is stored low byte first (05 00).

The Assembler and emu8086



For .EXE files a linker sits in between: .obj → .exe

Assembler

Translates mnemonics and labels into machine code and variable names into addresses. Examples: MASM, TASM, NASM.

emu8086

Editor + assembler + 8086 emulator. Shows registers, memory and flags *step by step*.

.COM / .EXE

.COM is the simple single-segment format, at most 64 KB (ORG 100h). .EXE may use several segments.

Examples in this course use emu8086 syntax; most also assemble with MASM/TASM.

Why Assembly?

A program written in assembly language:

Pros

+ Is **smaller** (no unnecessary code).
+ Is **faster**, since it does only what it has to.
+ Can do what high-level languages cannot: port I/O, interrupt handling, direct access to flags.
+ Is indispensable for boot loaders, device drivers and embedded systems.

Cons

– **Not portable**: 8086 code does not run on ARM.
– Tedious to write, read and debug.
– Unproductive for large projects; modern compilers usually generate code that is fast enough.

In practice a **mixed** approach is common: the program is written in a high-level language and only small, time-critical parts in assembly.

Source Line Format

```
[label:]  mnemonic  [dest], [source]  [; comment]
```



Line	Label	Mnemonic	Operands	Comment
`start: MOV AX, 5 ; AX ← 5`	`start`	`MOV`	`AX` , `5`	`; AX ← 5`
`INC AX`	—	`INC`	`AX`	—
`RET`	—	`RET`	none	—

- With two operands the order is **destination, source**: `MOV AX, BX` → AX ← BX.
- The assembler is case-insensitive: `mov ax, bx` and `MOV AX, BX` are the same.
- Everything from `;` to the end of the line is a comment and produces no machine code.

Instruction or Directive?

Line	Kind	What it puts 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 stands for 5
`END`	directive	nothing; the source ends here
`PUTC 'A'`	macro call	code for the instructions in the macro body

Instruction
An operation the processor carries out at run time. Each one becomes a machine instruction.

Directive
Information for the assembler: where code goes, how much space to reserve, what names stand for.

Macro
A named text pattern replaced by several lines at assembly time.

Labels and Names

```
start:  MOV CX, 3    ; code label
again:  INC AX
        LOOP again   ; an address
        RET
num     DB 7         ; data name
```

- A code label ends with a colon and stands for the address of an instruction (its offset within CS); it generates no code itself.
- A variable name has no colon; the assembler records its name, type (DB, DW…) and address.
- A name starts with a letter or underscore, never a digit; letters, digits and underscores may follow.
- Instruction, register and directive names (MOV, AX, DB, END…) are reserved.
- Case does not matter: `again` and `AGAIN` are the same label; defining both gives a “duplicate definition” error.

Name	`_count`	`sum2`	`2sum`	`cx`	`last-val`
Valid?	yes	yes	starts with a digit	register name	– is an operator

Operand Types and Implicit Operands

Register
`MOV AX, BX`: the value comes from inside the processor; the fastest kind.

Immediate
`MOV AX, 5`: the value is part of the instruction bytes; it can only be a source.

Memory
`MOV AX, [BX]`: the value is read from memory in a bus cycle.

Instruction	Explicit operand	Implicit operand(s)
`MUL BL`	BL	AL (multiplicand), AX (product)
`DIV BL`	BL	AX (dividend), AL (quotient), AH (remainder)
`LOOP again`	again	CX (counter)
`CBW`	—	AL → AX
`PUSH AX`	AX	SS:SP (top of stack)
`XLAT`	—	AL (index and result), BX (table address)

An implicit operand is not written, yet it changes: `MUL BL` also overwrites AH. A program that needs to keep a register's value must know the implicit operands of the instructions it uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
▶  MOV AL, 12
▶  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
9  ADD AX, 1
10 LOOP again      ; CX - 1
11 RET
```

AX	BX	CX	CF	ZF
000Ch	000Ah	0000h	0	0

AX = 12 (dec)

AL = 12, BL = 10. AH starts at 0.

Step 1 / 8

VARIABLES

AX = 000Ch BX = 000Ah CX = 0000h

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
▶ 4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL        ; count in CL
7  MOV CX, 3         ; LOOP counter
8  again:
9  ADD AX, 1
10 LOOP again        ; CX - 1
11 RET
```



`AX = 120 (dec)`

implicit: AL, AX

`MUL BL` names only BL; the multiplicand is implicitly AL and the product implicitly AX: AX = 120 = **0078h**. The product fits in 8 bits, so CF = 0.

Step 2 / 8

VARIABLES

AX = 0078h BX = 000Ah CX = 0000h

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
▶ MOV CL, 2
▶ SHR AX, CL       ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
9  ADD AX, 1
10 LOOP again      ; CX - 1
11 RET
```



`AX = 30 (dec)`

count in CL

`SHR AX, CL` : a shift count above 1 must be in CL. 120 ÷ 4 = 30 → AX = **001Eh**.

Step 3 / 8

VARIABLES

`AX = 001Eh` `BX = 000Ah` `CX = 0002h`

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
▶  MOV CX, 3       ; LOOP counter
8  again:
9  ADD AX, 1
10 LOOP again      ; CX - 1
11 RET
```



`AX = 30 (dec)`

CX = 3: the implicit counter of LOOP. CL is part of CX, so the old count 2 is gone.

Step 4 / 8

VARIABLES

`AX = 001Eh` `BX = 000Ah` `CX = 0003h`

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
▶  ADD AX, 1
▶  LOOP again      ; CX - 1
11 RET
```



`AX = 31 (dec)`

Pass 1: AX + 1 = 31; `LOOP` decrements CX: CX = 2, not zero → `again`.

Step 5 / 8

VARIABLES

`AX = 001Fh` `BX = 000Ah` `CX = 0002h`

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
▶  ADD AX, 1
▶  LOOP again      ; CX - 1
11 RET
```



`AX = 32 (dec)`

Pass 2: AX + 1 = 32; `LOOP` decrements CX: CX = 1, not zero → `again`.

Step 6 / 8

VARIABLES

`AX = 0020h` `BX = 000Ah` `CX = 0001h`

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
▶  ADD AX, 1
▶  LOOP again      ; CX - 1
11 RET
```



`AX = 33 (dec)`

Pass 3: AX + 1 = 33; `LOOP` decrements CX: CX = 0, the loop ends.

Step 7 / 8

VARIABLES

`AX = 0021h` `BX = 000Ah` `CX = 0000h`

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

Code Trace: Implicit Operands of MUL, SHR and LOOP

```
1  ORG 100h
2  MOV AL, 12
3  MOV BL, 10
4  MUL BL          ; AX = AL × BL
5  MOV CL, 2
6  SHR AX, CL      ; count in CL
7  MOV CX, 3       ; LOOP counter
8  again:
9  ADD AX, 1
10 LOOP again      ; CX - 1
▶  RET
```

AX	BX	CX	CF	ZF
0021h	000Ah	0000h	0	0

AX = 33 (dec)

RET. The program never decrements CX explicitly, yet LOOP changed it three times: AX = **0021h** = 33.

Step 8 / 8

VARIABLES

AX = **0021h** BX = **000Ah** CX = **0000h**

AX, CL and CX were “hidden” operands in this program; the special roles of the registers come from these implicit uses.

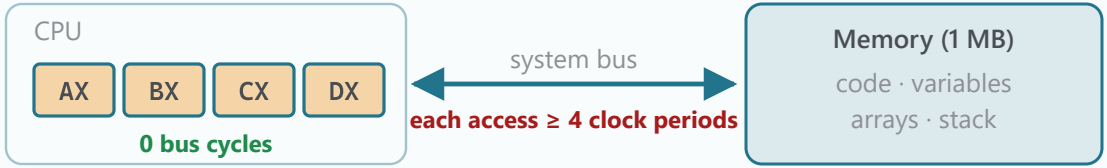
Register or Memory?

Register

- Inside the CPU; no bus cycle needed.
- Very few of them (the 8086 has 14 16-bit registers).
- Ideal for intermediate results and counters.

Memory

- Outside the CPU; each access takes at least one bus cycle (4 clock periods on the 8086).
- Large: up to 1 MB.
- Holds the program code, variables, arrays and the stack.



Good assembly code keeps frequently used values in **registers** and goes to memory only when it must. Keeping a loop counter in `CX` instead of memory, for example, speeds up the loop.

14 registers: 8 general-purpose + 4 segment + IP + FLAGS.

Instruction Timing: Register vs. Memory

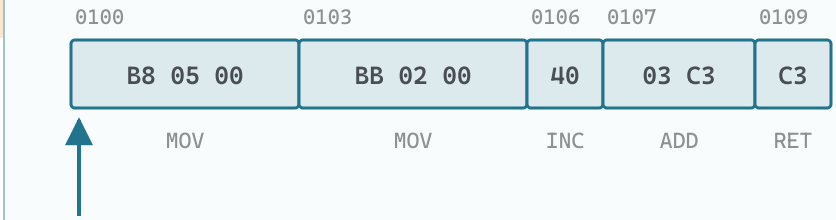
Instruction (8086)	Clocks
`MOV AX, BX`	2
`MOV AX, 5`	4
`MOV AX, num`	10
`MOV CX, [BX]`	8 + 5 = 13
`ADD AX, BX`	3
`ADD AX, num`	9 + 6 = 15
`INC CX`	2
`INC count` (word)	15 + 6 = 21

- A memory operand adds the **effective address calculation** (EA) time: direct address 6, `[BX]` 5, `[BX+SI]` 7 clocks.
- Instructions that write memory take longer: `INC count` reads, then writes (two bus cycles).
- Loading AX from a direct address uses the short accumulator form: 10 clocks.
- In a loop that runs 1000 times, incrementing a counter in CX costs 1000 × 2 = 2000 clocks; in memory, 1000 × 21 = 21,000. At 5 MHz that is 4.2 ms instead of 0.4 ms.

These are best-case times from the Intel 8086 data sheet; word access at an odd address and a flushed prefetch queue make them longer. The ratio holds: a memory-operand instruction is several times slower than its register form.

Code Trace: How Does IP Advance?

```
1 ▶ ORG 100h
2   MOV AX, 5       ; 3 bytes
3   MOV BX, 2       ; 3 bytes
4   INC AX          ; 1 byte
5   ADD AX, BX      ; 2 bytes
6   RET             ; 1 byte
```



The program is loaded at CS:0100h; **IP = 0100h** points to the first instruction.

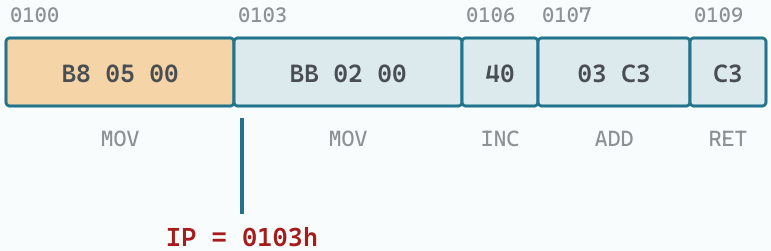
Step 1 / 6

VARIABLES

IP = **0100h** AX = **0000h** BX = **0000h** SP = **FFFEh**

Code Trace: How Does IP Advance?

```
1 ORG 100h
▶ MOV AX, 5      ; 3 bytes
3 MOV BX, 2      ; 3 bytes
4 INC AX         ; 1 byte
5 ADD AX, BX     ; 2 bytes
6 RET            ; 1 byte
```



MOV AX, 5 (B8 05 00) is fetched: **IP = 0103h**; once executed, AX = 0005h.

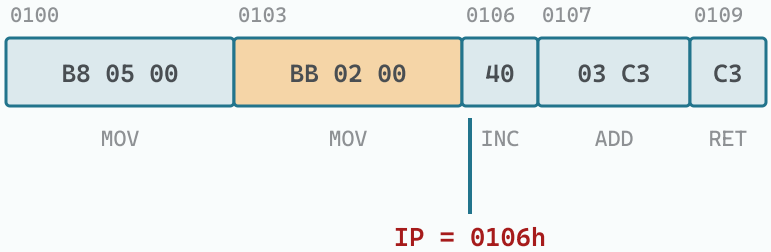
Step 2 / 6

VARIABLES

IP = 0103h AX = 0005h BX = 0000h SP = FFFEh

Code Trace: How Does IP Advance?

```
1 ORG 100h
2 MOV AX, 5      ; 3 bytes
▶ MOV BX, 2      ; 3 bytes
4 INC AX         ; 1 byte
5 ADD AX, BX     ; 2 bytes
6 RET            ; 1 byte
```



Three more bytes: **IP = 0106h**, BX = 0002h.

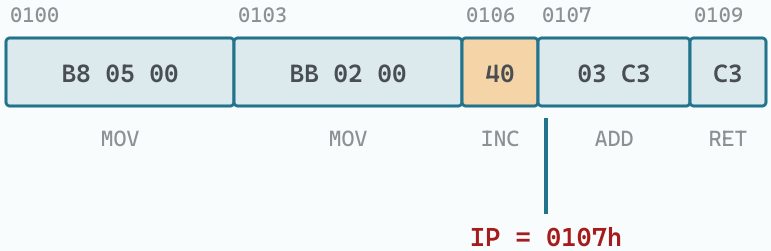
Step 3 / 6

VARIABLES

IP = **0106h** AX = **0005h** BX = **0002h** SP = **FFFEh**

Code Trace: How Does IP Advance?

```
1 ORG 100h
2 MOV AX, 5      ; 3 bytes
3 MOV BX, 2      ; 3 bytes
▶ INC AX         ; 1 byte
5 ADD AX, BX     ; 2 bytes
6 RET            ; 1 byte
```



INC AX is a single byte (40h): **IP = 0107h**, AX = 0006h.

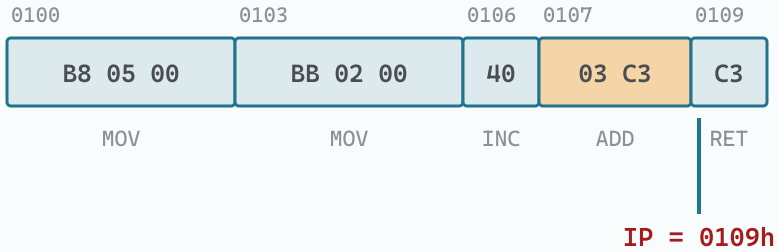
Step 4 / 6

VARIABLES

IP = **0107h** AX = **0006h** BX = **0002h** SP = **FFFEh**

Code Trace: How Does IP Advance?

```
1 ORG 100h
2 MOV AX, 5      ; 3 bytes
3 MOV BX, 2      ; 3 bytes
4 INC AX         ; 1 byte
▶ ADD AX, BX     ; 2 bytes
6 RET            ; 1 byte
```



ADD AX, BX is 2 bytes: **IP = 0109h**, AX = 0008h.

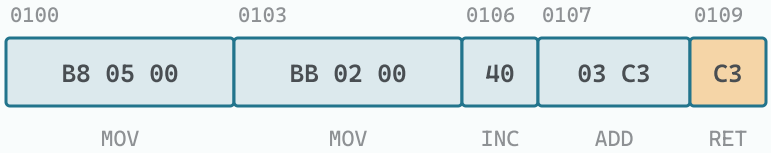
Step 5 / 6

VARIABLES

IP = **0109h** AX = **0008h** BX = **0002h** SP = **FFFEh**

Code Trace: How Does IP Advance?

```
1 ORG 100h
2 MOV AX, 5      ; 3 bytes
3 MOV BX, 2      ; 3 bytes
4 INC AX         ; 1 byte
5 ADD AX, BX     ; 2 bytes
▶ RET            ; 1 byte
```



IP = 0000h (back to PSP)

RET pops 0000h off the stack: **IP = 0000h**, SP = 0000h. The INT 20h at PSP:0000 ends the program.

Step 6 / 6

VARIABLES

IP = 0000h AX = 0008h BX = 0002h SP = 0000h

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
▶ STC               ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



CF only

`STC` : CF = 1. It sets a flag directly.

Step 1 / 8

VARIABLES

`AL = 00h` `CF = 1` `ZF = 0` `SF = 0`

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
▶  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



no flags

`MOV AL, 0FFh` : AL changed, none of the flags did. MOV does not affect flags.

Step 2 / 8

VARIABLES

AL = **FFh** | CF = **1** | ZF = **0** | SF = **0**

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
▶  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



CF kept

`INC AL` : FFh + 1 = 00h → ZF = 1, AF = 1, PF = 1. CF is still 1: INC and DEC never touch the carry flag.

Step 3 / 8

VARIABLES

`AL = 00h` `CF = 1` `ZF = 1` `SF = 0`

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
▶  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



no flags

`MOV CX, 0` : the result is zero, but ZF is not recomputed; it keeps the value from the previous instruction.

Step 4 / 8

VARIABLES

`AL = 00h` `CF = 1` `ZF = 1` `SF = 0`

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
▶  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



CF kept

`DEC AL` : 00h − 1 = FFh → SF = 1, ZF = 0. CF is still 1: DEC does not record the borrow in CF.

Step 5 / 8

VARIABLES

AL = **FFh** | CF = **1** | ZF = **0** | SF = **1**

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
▶  ADD AL, 1        ; new CF
8  CLC
9  NOT AL           ; flags kept
10 RET
```



all updated

`ADD AL, 1` : FFh + 1 = 100h → AL = 00h. ADD rewrites every status flag: this time CF = 1 is ADD's own carry.

Step 6 / 8

VARIABLES

AL = 00h | CF = 1 | ZF = 1 | SF = 0

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
▶  CLC
9  NOT AL           ; flags kept
10 RET
```

AL

00h

CF	ZF	SF	AF	PF
0	1	0	1	1

CF only

`CLC` : CF = 0.

Step 7 / 8

VARIABLES

AL = 00h | CF = 0 | ZF = 1 | SF = 0

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

Code Trace: Which Instructions Touch the Flags?

```
1  ORG 100h
2  STC              ; CF = 1
3  MOV AL, 0FFh     ; flags kept
4  INC AL           ; CF untouched
5  MOV CX, 0        ; ZF kept
6  DEC AL
7  ADD AL, 1        ; new CF
8  CLC
▶  NOT AL           ; flags kept
10 RET
```



no flags

`NOT AL` : AL = FFh, yet ZF is still 1. NOT changes no flags; to test the result, follow it with `TEST AL, AL` or `CMP`.

Step 8 / 8

VARIABLES

`AL = FFh` `CF = 0` `ZF = 1` `SF = 0`

MOV, NOT, XCHG, LEA, PUSH and POP leave the flags alone; INC and DEC change all but CF. Before a conditional jump, check which instruction last set the flag.

.COM Program Template

```
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

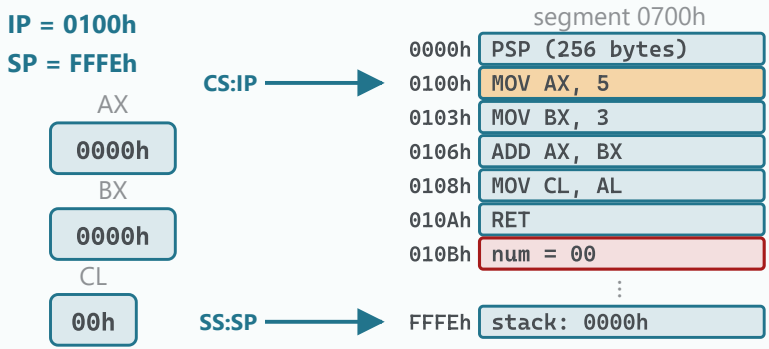
num DB 0            ; variables go after RET
```

- `ORG 100h`: a .COM program is loaded at offset 0100h of its segment; the first 256 bytes (the PSP) belong to DOS.
- Instructions execute top to bottom, **in order**.
- `RET` returns to the operating system.
- Variables go at the **end** (after RET) so the CPU never executes them as instructions.
- A .COM program starts with `CS = DS = ES = SS`, `IP = 0100h`, `SP = FFFEh`.

In emu8086, **new → COM template** opens the template; **emulate** loads the program into the emulator, and **single step** shows how the registers change after each instruction.

Code Trace: A .COM Program in Memory

```
1  ORG 100h          ; code at 0100h
2  MOV AX, 5
3  MOV BX, 3
4  ADD AX, BX        ; AX = 8
5  MOV CL, AL        ; CL = 08h
6  RET               ; back to DOS
7  num DB 0          ; data
```



The program is loaded at 0700:0100. CS = DS = ES = SS = 0700h, **IP = 0100h**, **SP = FFFEh**; the top of the stack holds 0000h.

Step 1 / 6

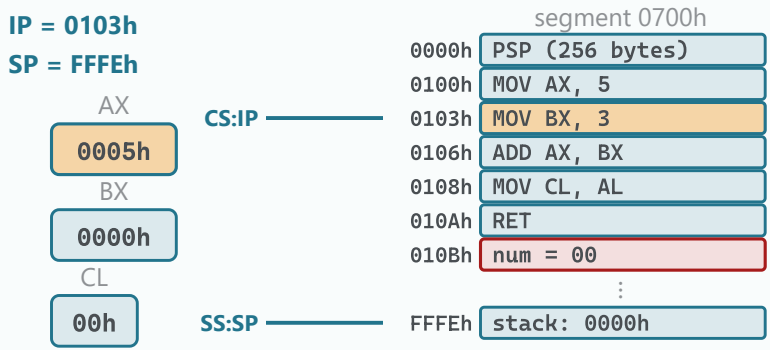
VARIABLES

AX = **0000h** BX = **0000h** CL = **00h** IP = **0100h** SP = **FFFEh**

The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

Code Trace: A .COM Program in Memory

```
1  ORG 100h        ; code at 0100h
▶  MOV AX, 5
3  MOV BX, 3
4  ADD AX, BX      ; AX = 8
5  MOV CL, AL      ; CL = 08h
6  RET             ; back to DOS
7  num DB 0        ; data
```



`MOV AX, 5` (3 bytes): AX = 0005h, IP = **0103h**.

Step 2 / 6

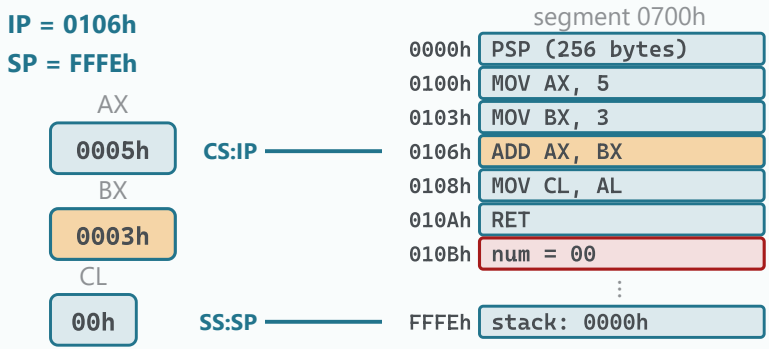
VARIABLES

`AX = 0005h` `BX = 0000h` `CL = 00h` `IP = 0103h` `SP = FFFEh`

The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

Code Trace: A .COM Program in Memory

```
1  ORG 100h        ; code at 0100h
2  MOV AX, 5
▶  MOV BX, 3
4  ADD AX, BX      ; AX = 8
5  MOV CL, AL      ; CL = 08h
6  RET             ; back to DOS
7  num DB 0        ; data
```



`MOV BX, 3` (3 bytes): BX = 0003h, IP = **0106h**.

Step 3 / 6

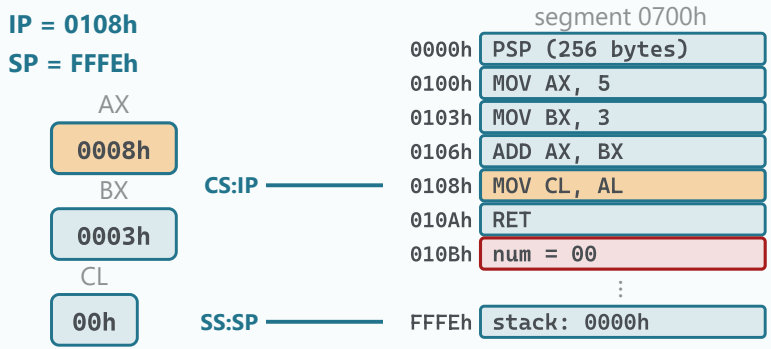
VARIABLES

`AX = 0005h` `BX = 0003h` `CL = 00h` `IP = 0106h` `SP = FFFEh`

The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

Code Trace: A .COM Program in Memory

```
1 ORG 100h          ; code at 0100h
2 MOV AX, 5
3 MOV BX, 3
▶ ADD AX, BX        ; AX = 8
5 MOV CL, AL        ; CL = 08h
6 RET               ; back to DOS
7 num DB 0          ; data
```



`ADD AX, BX` (2 bytes): AX = **0008h**, IP = 0108h.

Step 4 / 6

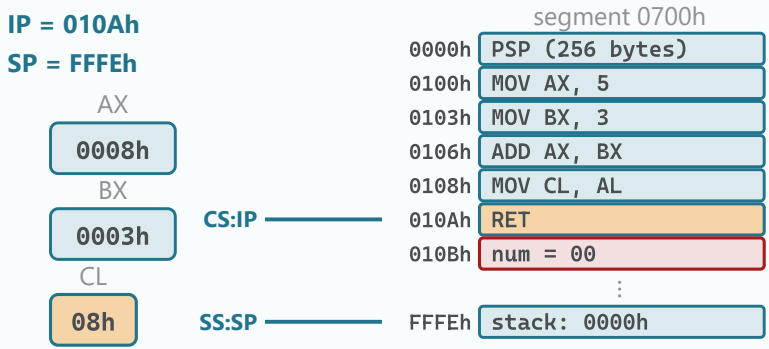
VARIABLES

AX = **0008h** BX = **0003h** CL = **00h** IP = **0108h** SP = **FFFEh**

The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

Code Trace: A .COM Program in Memory

```
1 ORG 100h          ; code at 0100h
2 MOV AX, 5
3 MOV BX, 3
4 ADD AX, BX        ; AX = 8
▶ MOV CL, AL        ; CL = 08h
6 RET               ; back to DOS
7 num DB 0          ; data
```



`MOV CL, AL` (2 bytes): CL = **08h**, IP = 010Ah.

Step 5 / 6

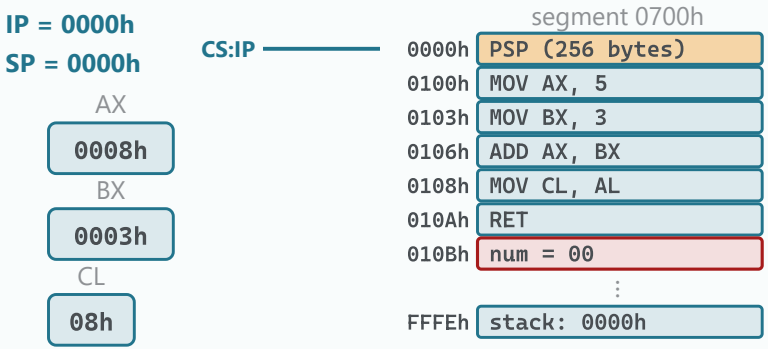
VARIABLES

`AX = 0008h` `BX = 0003h` `CL = 08h` `IP = 010Ah` `SP = FFFEh`

The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

Code Trace: A .COM Program in Memory

```
1 ORG 100h          ; code at 0100h
2 MOV AX, 5
3 MOV BX, 3
4 ADD AX, BX        ; AX = 8
5 MOV CL, AL        ; CL = 08h
▶ RET               ; back to DOS
7 num DB 0          ; data
```



`RET` pops 0000h: IP = 0000h, SP = 0000h. The `INT 20h` at PSP:0000 ends the program; `num` at 010Bh is never executed.

Step 6 / 6

VARIABLES

`AX = 0008h` `BX = 0003h` `CL = 08h` `IP = 0000h` `SP = 0000h`

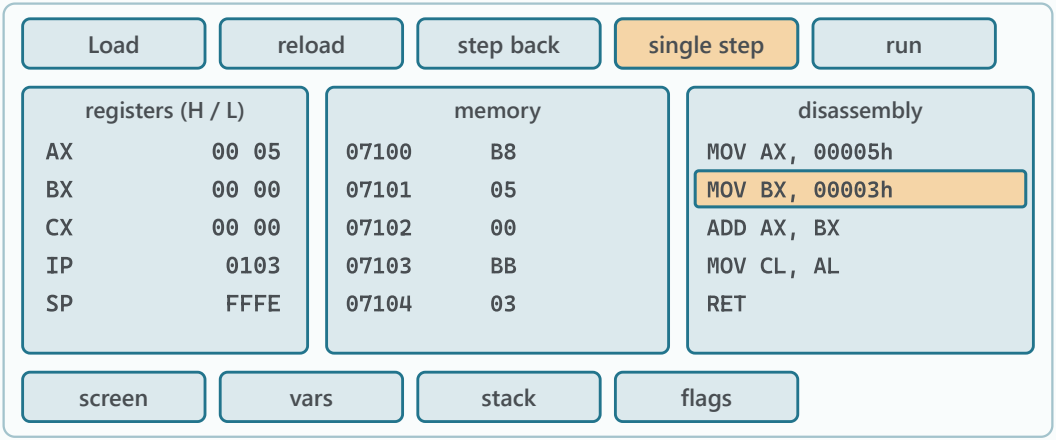
The instructions are 3, 3, 2, 2 and 1 bytes long; IP advances by each length. The data sits after RET, so it is never executed.

The Program in Memory, Byte by Byte

Offset	Bytes	Source line	Note
0100h	`B8 05 00`	`MOV AX, 5`	B8 + register number (AX = 0); immediate little-endian
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`	one-byte instruction
010Bh	`00`	`num DB 0`	data, not an instruction

- The .COM file is exactly these 12 bytes: there is no header; DOS copies them into memory starting at 0100h.
- `ORG 100h` produces no bytes; it only sets where the first byte goes.
- One instruction can have several encodings: `01 D8` is also valid for `ADD AX, BX`. MASM picks 03 C3; both do the same thing.
- Nothing in memory tells code from data: had IP reached 010Bh, the 00 there would have been decoded as an instruction.

The emu8086 Emulator Window



- **single step** executes one instruction; changed registers and memory show up at once.
- **step back** undoes the last step, so you can trace a bug back to its cause.
- The highlighted line in the disassembly is the **next** instruction (CS:IP), not the one just executed.
- The disassembly shows constants in hex with 5 digits (00005h).
- Extra windows: screen output, variables, stack and flags.

The memory column shows physical addresses: 0700:0100 = 07100h. The drawing shows only the main parts of the window.

Syntax Rules

Number bases

`25` or `25d` decimal · `19h` hex · `00011001b` binary · `31o` octal. All the same value: 25.

Hex starting with a letter

Prefix a `0`: `0FFh`, `0B800h`. Written as `FFh`, the assembler takes it for a *name* (label or variable).

Characters

`'A'` → ASCII code 41h. `MOV CL, 'A'` and `MOV CL, 41h` produce the same machine code.

Signed numbers

`-1` is stored as FFh in 8 bits and FFFFh in 16 bits (two's complement).

Case

The assembler ignores case: `mov`, `MOV`, `Mov` are the same; so are `var1` and `VAR1`.

Comments

Anything after `;` is a comment. Good habit: say "why", not "what".

Common Mistakes

Wrong	Why?	Correct
`MOV AL, BX`	Sizes differ (8 ↔ 16 bits)	`MOV AL, BL` or `MOV AX, BX`
`MOV DS, 1234h`	No immediate into a segment register	`MOV AX, 1234h` · `MOV DS, AX`
`MOV AL, [AX]`	Only BX, SI, DI, BP can address memory	`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	use `JMP`
`MOV AL, 300`	300 > 255, does not fit in 8 bits	`MOV AX, 300`

These rules come from the operand restrictions of MOV.

More Syntax Errors

Wrong	Why?	Correct
`MOV CS, AX`	only JMP, CALL, RET, INT change CS	a far jump
`MOV ES, DS`	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 unknown	`INC BYTE PTR [BX]`
`MOV AL, [BX+BP]`	two base registers together	`[BX+SI]` or `[BP+DI]`
`ADD 5, AX`	destination cannot be immediate	`ADD AX, 5`
`PUSH 5`	not before the 80186	`MOV AX, 5` · `PUSH AX`
`SHL AX, 2`	8086 count is 1 or CL only	`MOV CL, 2` · `SHL AX, CL`

PUSH immediate and SHL AX, 2 are valid from the 80186 on. MASM rejects them when assembling for the 8086; emu8086 accepts them, but the code will not run on a real 8086.

Assembly Error or Logic Error?

Assembly (syntax) error

The assembler finds it and reports the line number; no program is produced.

- `MOV AL, BX` : size mismatch
- `JMP strt` : undefined label
- `MOV AL, 300` : constant too large

Logic error

The program assembles and runs, but the result is wrong; the assembler cannot tell.

- Forgetting RET: data executed as code
- Using `JA` for signed numbers
- ADD instead of ADC: the carry is lost
- Not loading DS in an .EXE

Logic errors are found by single-stepping: after each instruction, compare the expected register and flag values with the emulator's. The first instruction where they differ is where the bug is.

Lab: First Programs

.COM program

CODE · 12 bytes

```
 1 ORG 100h          ; code at 0…
 2 MOV AX, 5
 3 MOV BX, 3
 4 ADD AX, BX        ; AX = 8
 5 MOV CL, AL        ; CL = 08h
 6 RET               ; back to D…
 7 num DB 0          ; data
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	000C	CH	00	CL	0C
DX	0000	DH	00	DL	00

SI	0000	DI	0000	BP	0000
SP	FFFE	CS	0700	IP	0100
DS	0700	ES	0700	SS	0700

FLAGS

O	D	I	T	S	Z	A	P	C
0	0	1	0	0	0	0	0	0

CS:IP = 0700:0100 · step 0

MEMORY DS = 0700h

```
0108 8A C8 C3 00 00 00 00 00
0110 00 00 00 00 00 00 00 00
0118 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00
0128 00 00 00 00 00 00 00 00
```

STACK SS:SP

```
FFF6 0000
FFF8 0000
FFFA 0000
FFFC 0000
FFFE 0000 ◀ SP
```

OUTPUT

Program loaded: 12 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (000Ch). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Assembly

A low-level language that maps one-to-one onto machine code: small, fast code and direct hardware access, but not portable.

Line format

`[label:] mnemonic dest, source ; comment`. Directives (ORG, DB, EQU) produce no machine code.

Operands

Registers are fastest; immediates can only be sources; memory operands need a bus cycle. MUL, LOOP and CBW use implicit operands.

First program

A .COM program starts at 0100h with `ORG 100h`; data goes after RET.

IP

CS:IP points to the next instruction; IP advances by the instruction's length.

Flags

MOV, NOT and XCHG leave the flags alone; INC and DEC do not touch CF; ADD rewrites them all.

SUMMARY

Exercises

1. With AX = 0ABCDh, what are AX, AL and BX after `MOV AH, 0`?
2. With AX = 1234h and BL = 03h, what is AX after `MUL BL`? What happens to the 12h in AH?
3. On an 8 MHz 8086, a loop runs 2000 times. How many clocks and milliseconds are saved by using `INC CX` (2 clocks) instead of `INC count` (21 clocks)?
4. With AL = 80h, give AL, CF, ZF, SF and OF after `ADD AL, 80h`.
5. A program starts at 0100h. What is IP after four instructions of 3, 2, 1 and 4 bytes?
6. Why are `MOV CX, AL` and `MOV ES, 0B800h` wrong? Fix them.

▶ **Answers**

References

- emu8086 documentation, *8086 assembler tutorial for beginners* (part 1: numbering systems, part 2: memory access).
- Intel, *8086 16-Bit HMOS Microprocessor* data sheet.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson.
- Muhammad Ali Mazidi, Janice G. Mazidi, *The 80x86 IBM PC and Compatible Computers*, Pearson.

The End

Chapter 3.2: Assembly Language

Previous: Chapter 3.1: Programming Languages · Next: Chapter 4.1: Memory Access