Chapter 1.1: Introduction

Microprocessors

Microprocessor vs. microcontroller | Fetch · decode · execute | CPU structure | Clock speed | CISC · RISC · EPIC | Special-purpose processors | Evolution of Intel x86 | Modern CPU features

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_01_01_Introduction.html

Learning Outcomes

Basic concepts

Define the microprocessor, the microcontroller and the buses of a computer system.

Instruction cycle

Trace the fetch–decode–execute cycle and the role of the program counter (IP/PC) step by step.

Performance

Relate clock frequency, period, CPI and execution time.

Architecture types

Compare CISC, RISC and EPIC with vector, SIMD, DSP and GPU processors.

The x86 family

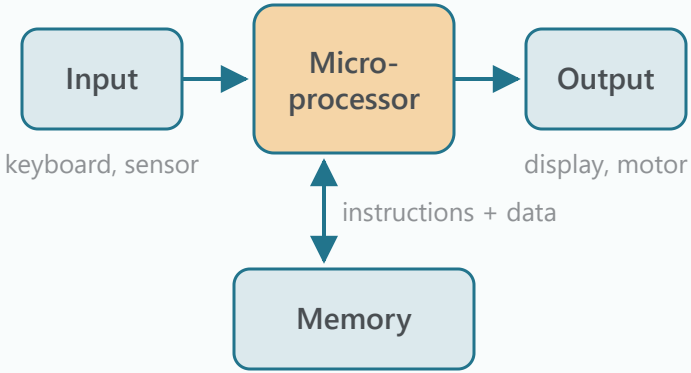
Trace Intel processors from the 4004 to the Core i7 and place the 8086 among them.

Modern features

Recognize cache, multicore, virtualization and power management.

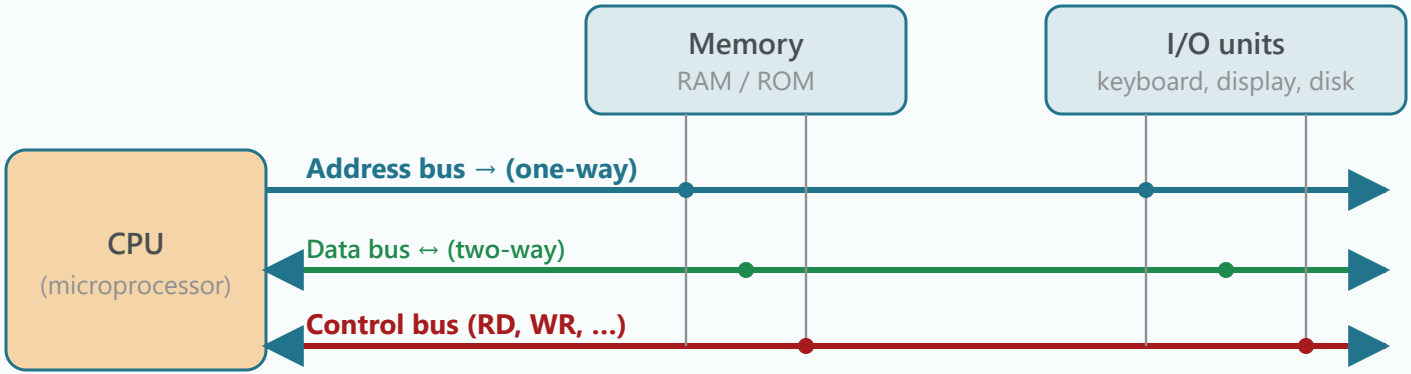
What Is a Microprocessor?

- The computer's **brain**: a central processing unit (CPU) built on a single chip.
- A **programmable** device; instructions in memory tell it what to do.
- Basic flow:
  - takes **input**,
  - performs **arithmetic and logic** operations,
  - produces the desired **output**.
- It fetches and executes machine-code instructions.



Example: on a calculator the keys are the **input**, the addition is the **processing**, and the result on the display is the **output**.

The Computer System and Its Buses



Address bus

Selects which memory cell or I/O port is accessed. n lines → 2^n addresses. 8086: **20 bits → 1 MB**.

Data bus

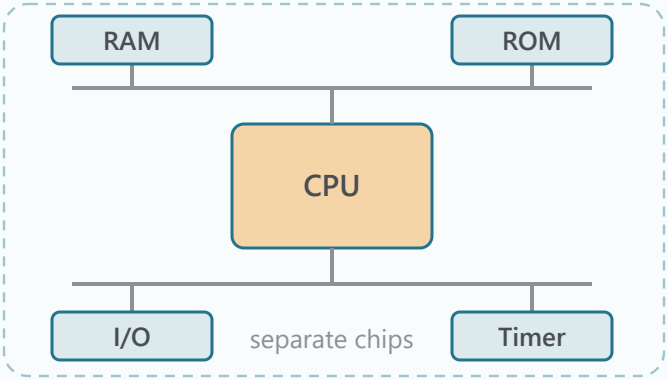
Carries the data being read or written. Its width is the number of bits moved at once. 8086: **16 bits**.

Control bus

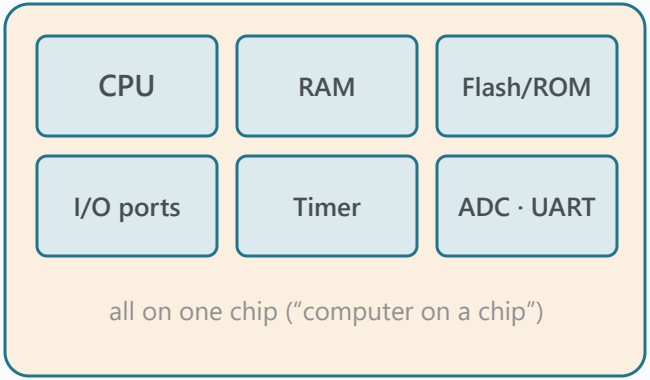
Timing and control signals: read/write, memory or I/O, clock, interrupts.

Microprocessor vs. Microcontroller

Microprocessor (MPU) based system



Microcontroller (MCU)



Microprocessor

Just the CPU; memory and peripherals are added outside. Flexible and powerful. E.g. **8086**, Core i7, Ryzen.

Microcontroller

CPU + memory + peripherals on one chip. Cheap, low-power, dedicated to one job. E.g. 8051, PIC, AVR (Arduino), STM32.

Microprocessor or Microcontroller?

Criterion	Microprocessor (MPU)	Microcontroller (MCU)
Contents	CPU (ALU, control unit, registers)	CPU + RAM + Flash + I/O + timers + ADC…
Memory	external, large (GB)	on chip, small (KB–MB)
Clock speed	high (GHz)	low–medium (MHz)
Power	high (W–100 W)	very low (mW, µW in sleep)
Cost	high; needs extra chips	low; one chip is enough
Software	usually an OS + many applications	one fixed program (firmware)
Use	PC, server, laptop	washing machine, remote, car ECU, IoT

Our example processor is the **8086**: its buses, registers and instructions work the same way in MCUs.

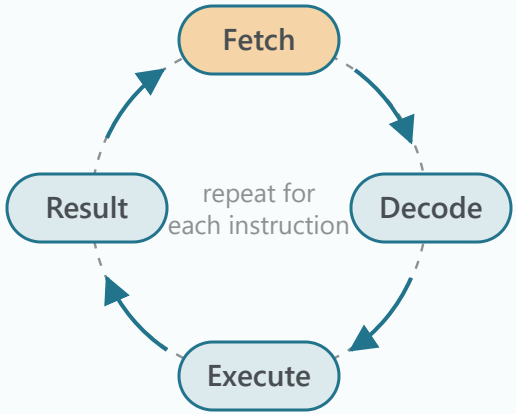
Main Tasks: The Instruction Cycle

1 · Fetch
Reads the instruction from memory at the address in the PC.

2 · Decode
Decodes the opcode: which operation, which operands?

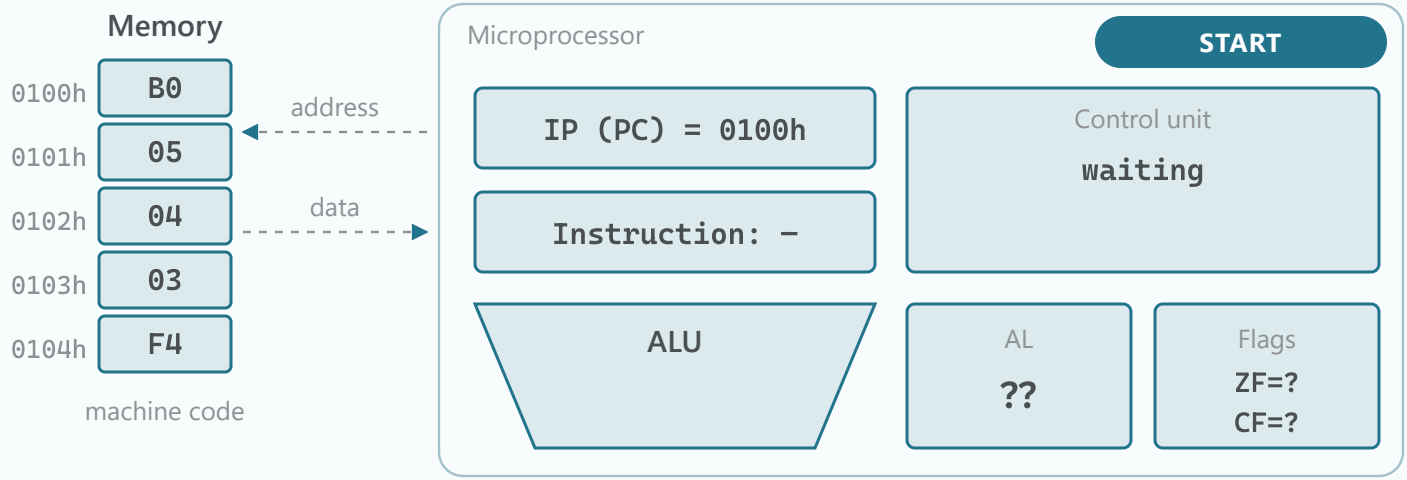
3 · Execute
Carries out the decoded instruction (ALU, data move, jump).

4 · Result
Writes the result to a register or memory and updates the flags.



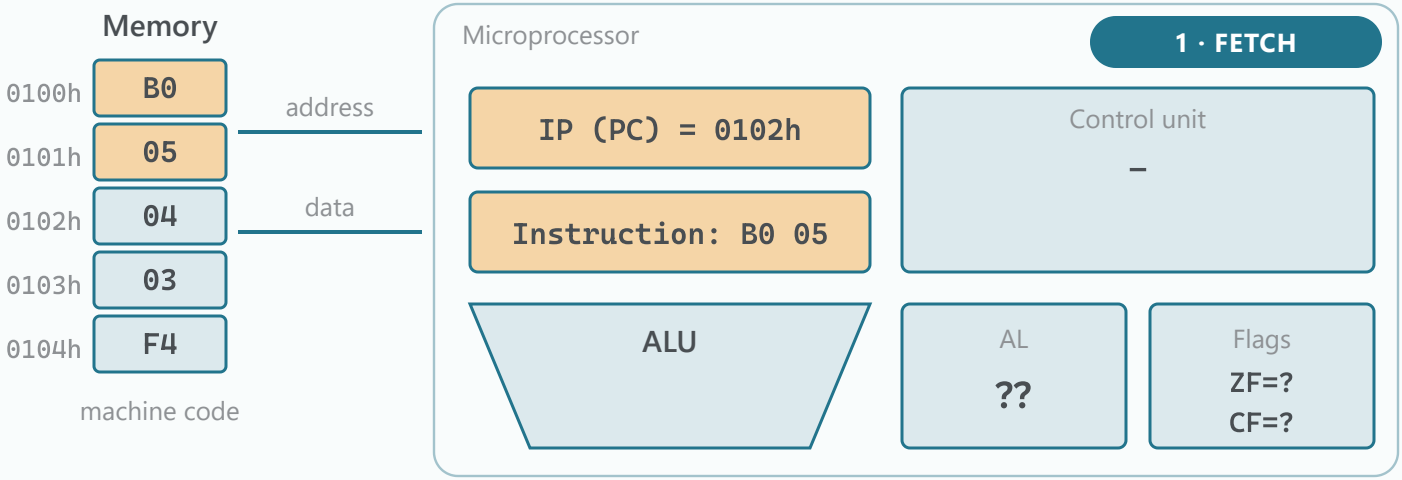
The cycle repeats nonstop while the processor is on. Fast processors **overlap** the stages (pipelining).

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



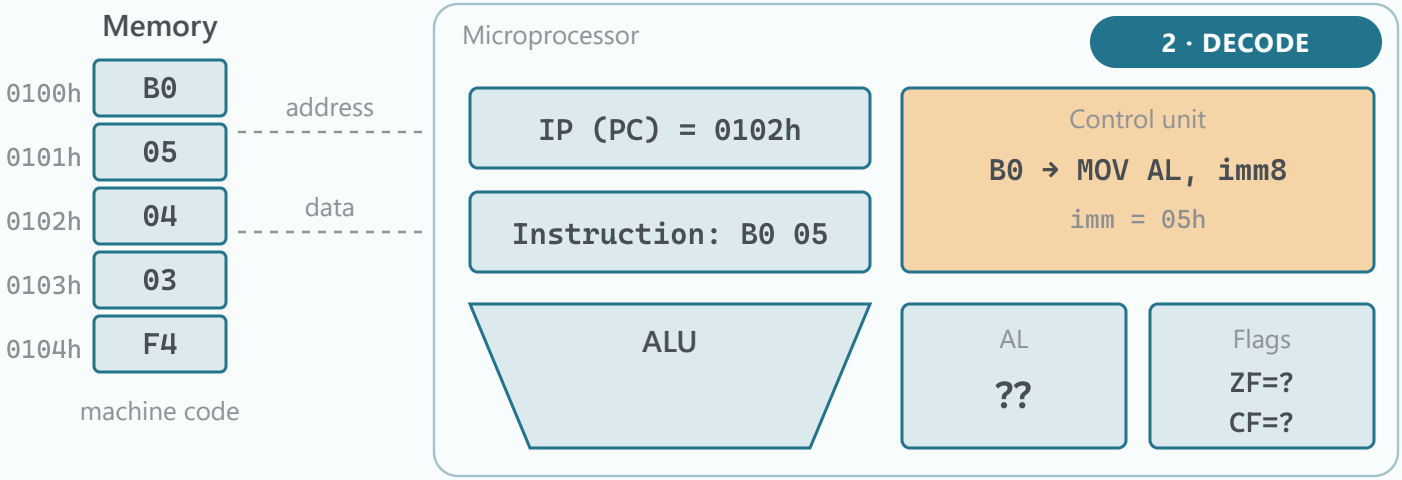
The program is loaded into memory. **IP** (the 8086's name for the program counter) holds the address of the first instruction, **0100h**.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



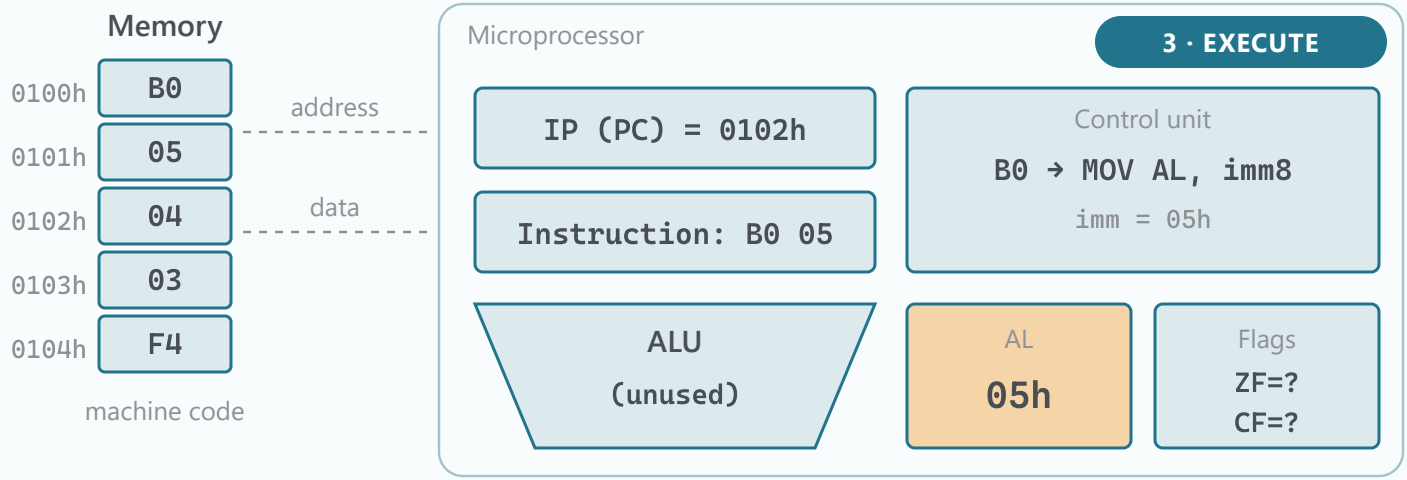
Fetch: IP goes onto the address bus and **B0 05** is read from memory. IP grows by the instruction length (2 bytes) → **0102h**.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



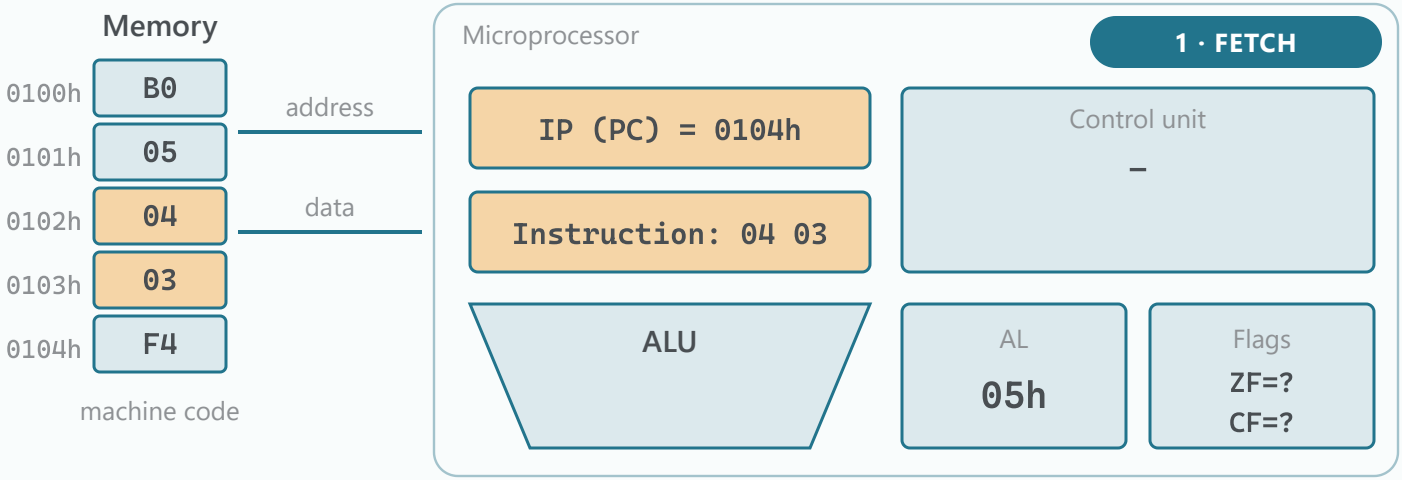
Decode: the control unit decodes opcode **B0** as “load an 8-bit constant into AL”; the constant is 05h.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



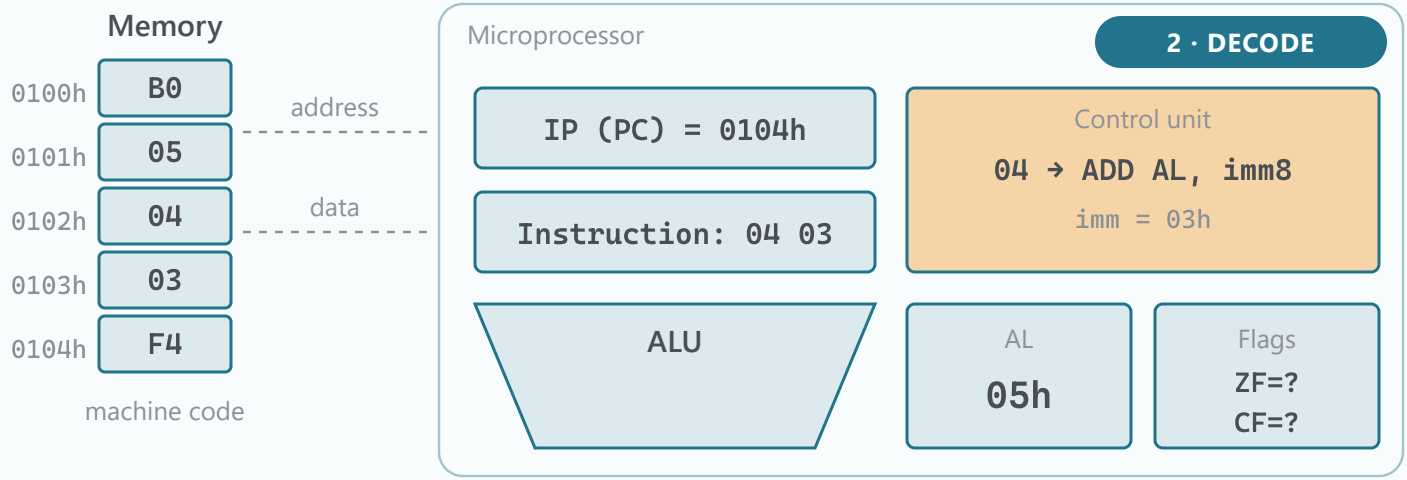
Execute / result: 05h is written to AL. MOV does **not affect** the flags.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



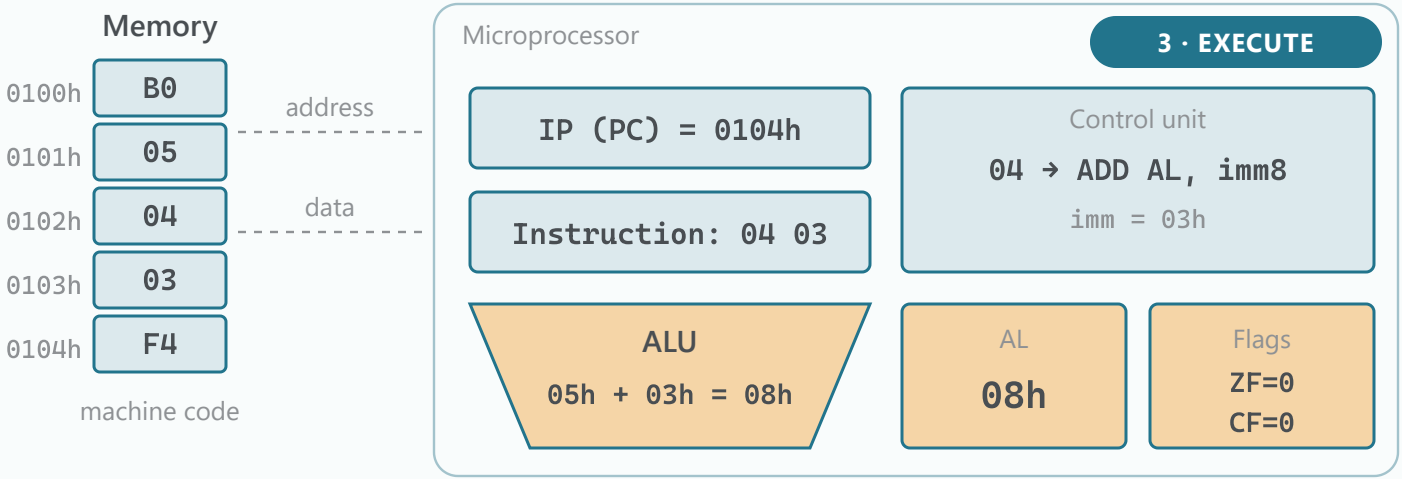
The cycle starts again: **04 03** is fetched, IP → **0104h**.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



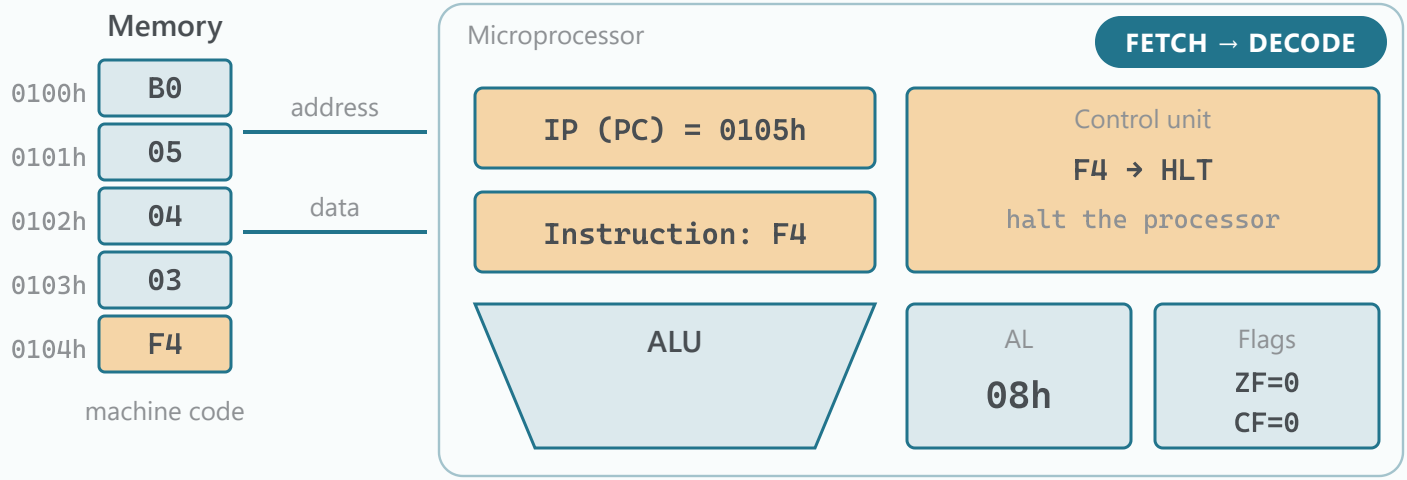
04 = “add an 8-bit constant to AL”. Operands: AL (05h) and 03h.

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



The **ALU** adds: 05h + 03h = **08h** → AL. The result is not zero (**ZF=0**), no carry (**CF=0**).

Instruction Cycle: `MOV AL,5` · `ADD AL,3` · `HLT`



F4 = HLT is fetched and decoded; the processor stops until an interrupt arrives. The same cycle repeats for every instruction.

Basic Functions

Arithmetic and logic operations

Add, subtract, compare, AND/OR/XOR, shift. Done by the **ALU** (arithmetic logic unit). Newer processors also handle floating-point numbers (FPU).

```
ADD AX, BX
CMP AL, 10
AND AL, 0Fh
```

Data movement

Data is copied between registers, memory and I/O. This is the most common operation in programs.

```
MOV AX, [1234h]
MOV [BX], AL
IN AL, 60h
```

Program counter (PC)

Holds the address of the next instruction. Writing a new value to the PC is a **jump**; conditional jumps make **decisions**. On the 8086 it is called **IP**.

```
JMP label
JZ is_zero
CALL subroutine
```

Every program is a combination of these three functions (compute · move · change the order).

Code Trace: How Does IP Change?

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

`mov cx, 3` is 3 bytes; after it runs, IP points to the next instruction (0103h).

Step 1 / 9

VARIABLES

`IP = 0103h` `CX = 0003h` `AX = ????`

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
▶ 0103  mov  ax, 0      ; sum = 0
3 0106  again: add ax, cx
4 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

The sum is cleared. IP again advances by the instruction length → 0106h.

Step 2 / 9

VARIABLES

IP = 0106h | CX = 0003h | AX = 0000h

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
▶ 0106  again: add ax, cx
4 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

AX = 0 + 3 = 3. `add ax, cx` is 2 bytes → IP = 0108h.

Step 3 / 9

VARIABLES

IP = **0108h** CX = **0003h** AX = **0003h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
3 0106  again: add ax, cx
▶ 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

`loop` : CX = CX − 1 = 2 ≠ 0 → **jump**. IP gets 0106h; the sequential flow is broken.

Step 4 / 9

VARIABLES

IP = **0106h** CX = **0002h** AX = **0003h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
▶ 0106  again: add ax, cx
4 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

AX = 3 + 2 = 5.

Step 5 / 9

VARIABLES

IP = **0108h** CX = **0002h** AX = **0005h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
3 0106  again: add ax, cx
▶ 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

CX = 1 ≠ 0 → jump to 0106h again.

Step 6 / 9

VARIABLES

IP = **0106h** CX = **0001h** AX = **0005h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
▶ 0106  again: add ax, cx
4 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

AX = 5 + 1 = 6.

Step 7 / 9

VARIABLES

IP = 0108h CX = 0001h AX = 0006h

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
3 0106  again: add ax, cx
▶ 0108  loop again      ; CX--, jump if CX≠0
5 010A  hlt             ; halt
```

CX = 0 → **no** jump; IP moves on to the next instruction (010Ah).

Step 8 / 9

VARIABLES

IP = **010Ah** CX = **0000h** AX = **0006h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Code Trace: How Does IP Change?

```
1 0100  mov  cx, 3      ; count = 3
2 0103  mov  ax, 0      ; sum = 0
3 0106  again: add ax, cx
4 0108  loop again      ; CX--, jump if CX≠0
▶ 010A  hlt             ; halt
```

`hlt`: the processor halts. Result AX = 3 + 2 + 1 = **6**.

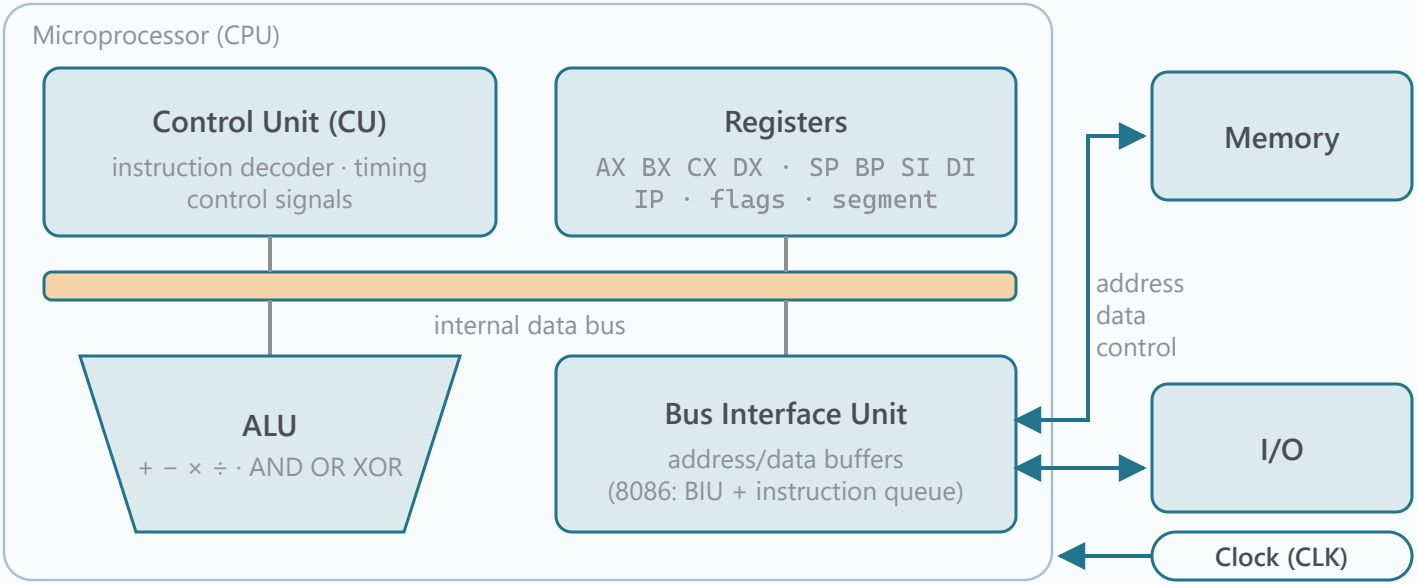
Step 9 / 9

VARIABLES

IP = **010Bh** CX = **0000h** AX = **0006h**

The numbers on the left are the instruction addresses (offsets, hex). `loop` does not change the flags.

Microprocessor Structure



The control unit decodes instructions and tells the other units **what to do and when**; the ALU computes; registers are the fastest temporary storage; the bus interface talks to the outside world.

Building Blocks

ALU

Performs arithmetic (+, −) and logic (AND, OR, NOT, XOR) on binary numbers, plus compares and shifts. Sets the **flags** (zero, carry, sign, overflow) from the result.

Control unit

Decodes the instruction and times which register connects to which bus, which operation the ALU performs, and the memory read/write signals.

Registers

Small, very fast storage cells inside the processor: general-purpose (data), address/pointer, status (flags) and program counter registers.

Bus interface

Drives the address, data and control buses. On the 8086 the **BIU** prefetches instructions into a 6-byte **queue**.

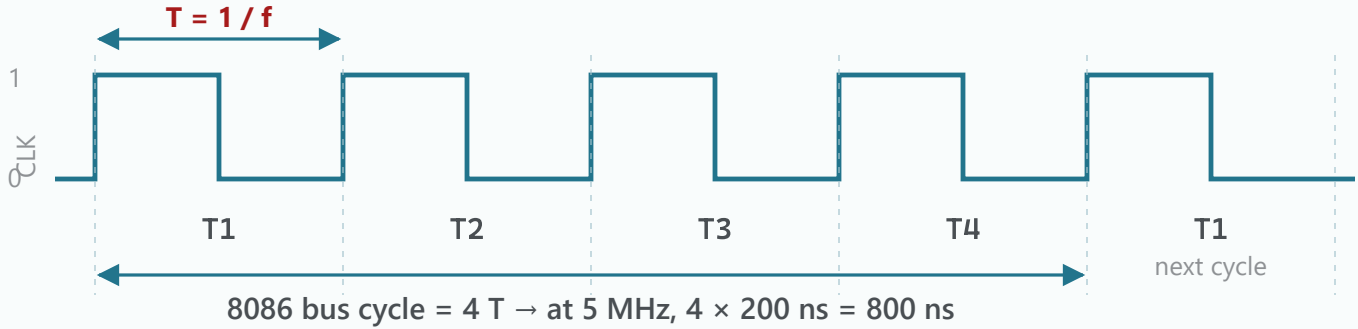
Clock and timing

Every operation advances in step with the clock ticks. One tick lasts **T = 1/f**.

Cache, FPU, cores

Modern processors also put cache, a floating-point unit, several cores and a graphics unit on the same chip.

The Clock Signal and Its Period



- **Clock frequency** f: ticks (cycles) per second, in Hz.
- **Period** T = 1 / f: the length of one tick.
- An instruction may take several clock cycles; on the 8086 a memory access takes at least **4 T** (T1–T4).

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

MICROPROCESSOR STRUCTURE · INTERACTIVE

Calculator: Clock Speed, Period and Execution Time

8086 · 5 MHz | 80386 · 33 MHz | Pentium · 66 MHz | Core i7 · 3.33 GHz

Clock frequency f (MHz)	Cycles per instruction (CPI)	Instruction count N (millions)
5	10	1

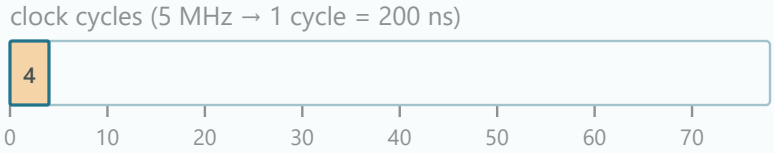
Period T	8086 bus cycle (4 T)	Speed (MIPS = f / CPI)	Execution time
200 ns	800 ns	0.5	2 s

T = 1 / 5 MHz = **200 ns** · time = N × CPI / f = $1 \cdot 10^6 \times 10 / 5 \cdot 10^6$ Hz = **2 s**

The CPI values are rough examples: 8086 instructions take ~10 cycles on average, while modern processors can finish several instructions per cycle (CPI < 1).

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **4 cycles × 200 ns = 0.8 µs**

this one: MOV imm → 4 cycles

CX = 4 · AX = 0

Every instruction takes a set number of **clock cycles**. On a 5 MHz 8086 one cycle is T = 1/f = **200 ns**. `mov cx, 4` : 4 cycles.

Step 1 / 11

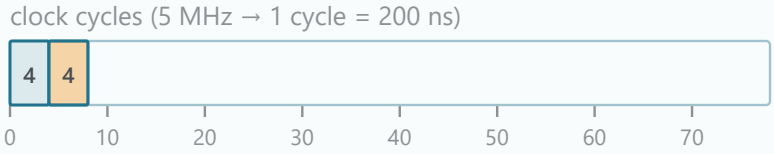
VARIABLES

`CX = 4` `AX = 0` `cycles = 4` `time = 0.8 µs`

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **8 cycles × 200 ns = 1.6 µs**

this one: MOV imm → 4 cycles

CX = 4 · AX = 0

`mov ax, 0` : 4 cycles again. Total 8 cycles = 1.6 µs.

Step 2 / 11

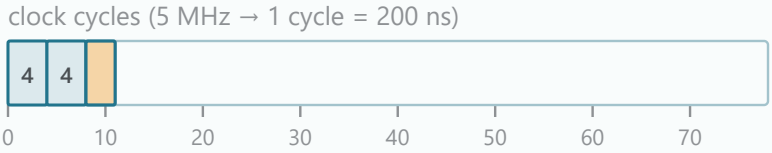
VARIABLES

CX = 4 | AX = 0 | cycles = 8 | time = 1.6 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **11 cycles × 200 ns = 2.2 µs**

this one: ADD reg → 3 cycles

CX = 4 · AX = 4

`add ax, cx` : a register-to-register add takes 3 cycles; short because there is no memory access. AX = 4.

Step 3 / 11

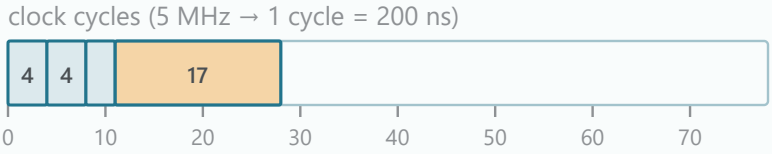
VARIABLES

CX = 4 | AX = 4 | cycles = 11 | time = 2.2 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **28 cycles × 200 ns = 5.6 µs**

this one: LOOP → 17 cycles CX = 3 · AX = 4

`loop` : CX = 3 ≠ 0 → jump. A taken LOOP costs **17 cycles**: the queue is flushed and refilled from the target.

Step 4 / 11

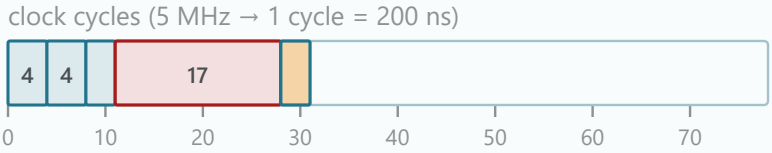
VARIABLES

CX = **3** AX = **4** cycles = **28** time = **5.6 µs**

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **31 cycles × 200 ns = 6.2 µs**

this one: ADD reg → 3 cycles — CX = 3 · AX = 7

`add ax, cx` : a register-to-register add takes 3 cycles; short because there is no memory access. AX = 7.

Step 5 / 11

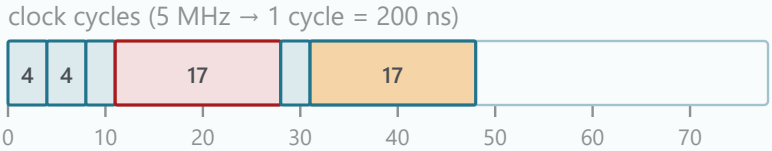
VARIABLES

`CX = 3` `AX = 7` `cycles = 31` `time = 6.2 µs`

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **48 cycles × 200 ns = 9.6 µs**

this one: LOOP → 17 cycles CX = 2 · AX = 7

`loop` : CX = 2 ≠ 0 → jump. A taken LOOP costs **17 cycles**: the queue is flushed and refilled from the target.

Step 6 / 11

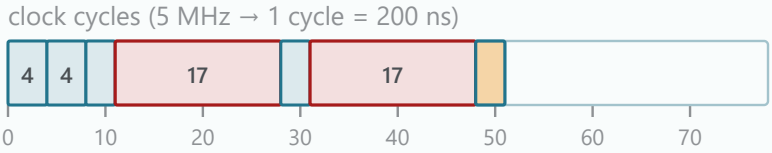
VARIABLES

CX = **2** AX = **7** cycles = **48** time = **9.6 µs**

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **51 cycles × 200 ns = 10.2 µs**

this one: ADD reg → 3 cycles

CX = 2 · AX = 9

`add ax, cx` : a register-to-register add takes 3 cycles; short because there is no memory access. AX = 9.

Step 7 / 11

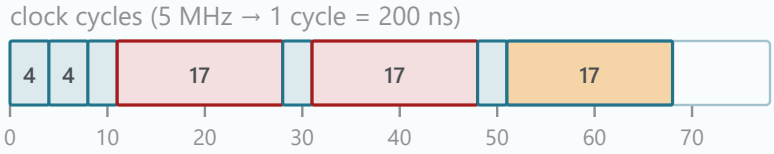
VARIABLES

CX = 2 | AX = 9 | cycles = 51 | time = 10.2 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **68 cycles × 200 ns = 13.6 µs**

this one: LOOP → 17 cycles CX = 1 · AX = 9

`loop` : CX = 1 ≠ 0 → jump. A taken LOOP costs **17 cycles**: the queue is flushed and refilled from the target.

Step 8 / 11

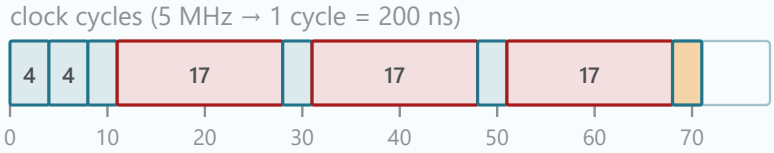
VARIABLES

CX = **1** AX = **9** cycles = **68** time = **13.6 µs**

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **71 cycles × 200 ns = 14.2 µs**

this one: ADD reg → 3 cycles CX = 1 · AX = 10

`add ax, cx` : a register-to-register add takes 3 cycles; short because there is no memory access. AX = 10.

Step 9 / 11

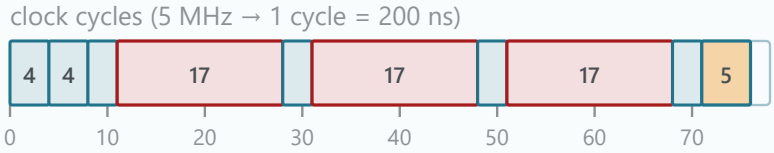
VARIABLES

CX = 1 | AX = 10 | cycles = 71 | time = 14.2 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **76 cycles × 200 ns = 15.2 µs**

this one: LOOP → 5 cycles

CX = 0 · AX = 10

CX = 0 → no jump; a LOOP that falls through takes only **5 cycles**.

Step 10 / 11

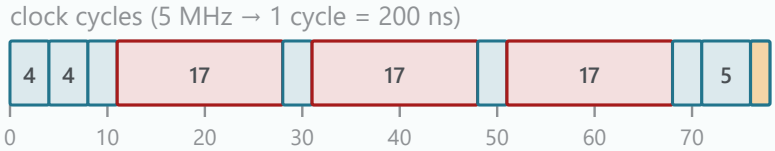
VARIABLES

CX = 0 | AX = 10 | cycles = 76 | time = 15.2 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Code Trace: How Many Clock Cycles Does a Loop Take?

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



so far: **78 cycles × 200 ns = 15.6 µs**

this one: HLT → 2 cycles

CX = 0 · AX = 10

`hlt`: 2 cycles. Total **78 cycles × 200 ns = 15.6 µs**; more than half the cycles (51) went to taken LOOPs.

Step 11 / 11

VARIABLES

CX = 0 | AX = 10 | cycles = 78 | time = 15.6 µs

Cycle counts are best-case values from the 8086 data sheet (full queue, no wait states). Time = cycles × T.

Clock Speeds of 16-, 32- and 64-Bit Processors

16-bit

- 8086: 5 · 8 · 10 MHz
- 8088: 5 · 8 MHz
  (4.77 MHz in the IBM PC)
- 80186/80188: from 6 MHz
- 80286: 6 · 8 · 10 · 12.5 MHz

32-bit

- 80386: 16 – 33 MHz
- 80486: 16 – 100 MHz
  (DX2/DX4 clock multiplier)
- Pentium: started at 60 · 66 MHz

64-bit

- Core 2: 1.2 – 3 GHz
- Core i7 (1st gen): 2.66 – 3.33 GHz
- Core i5 (1st gen): 2.4 – 3.6 GHz
- Core i3 (1st gen): 2.93 – 3.33 GHz

In 30 years clock speed rose about **600×** (5 MHz → ~3 GHz). Around 2005 it hit the power and heat wall; since then performance has come mainly from **multiple cores** and more instructions per cycle.

4.77 MHz is the clock speed of the 8088 in the IBM PC.

MICROPROCESSOR STRUCTURE

What Does “n-Bit Processor” Mean?

The word size the processor handles at once: the register and ALU width, and usually the data bus width. n bits represent the range 0 … $2^n - 1$.

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

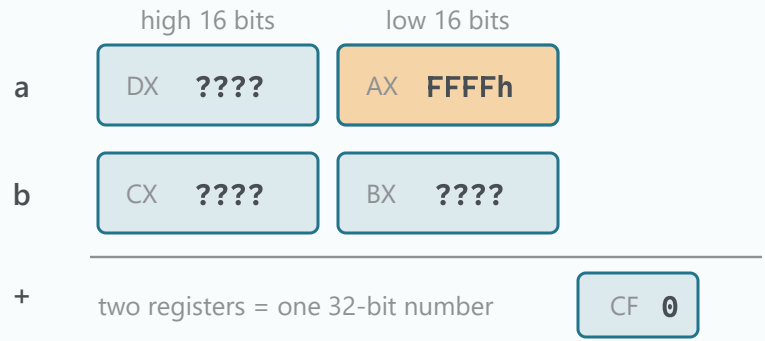
The **address bus** width, in turn, sets how much memory can be reached: n lines → 2^n bytes.

Address bus	Memory	Example
16 bits	64 KB	8080, 8085
20 bits	1 MB	8086, 8088
24 bits	16 MB	80286
32 bits	4 GB	80386, 80486
36 bits	64 GB	Pentium Pro (PAE)

The 8086 is a **16-bit** processor, yet it has a **20-bit** address bus. The 20-bit address is formed by combining two 16-bit values (segment and offset).

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
▶ mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
8 ret
```



a = 0001FFFFh (131,071) does not fit in a 16-bit register, so it is split in two. Low half **AX = FFFFh**.

Step 1 / 7

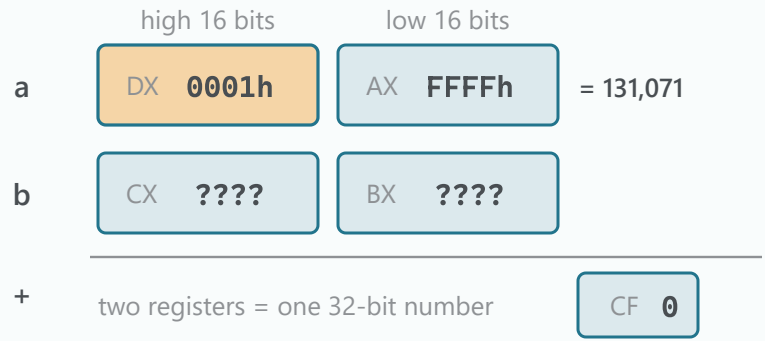
VARIABLES

AX = **FFFFh** DX = **0000h** BX = **0000h** CX = **0000h** CF = **0**

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
▶ mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
8 ret
```



High half **DX = 0001h**. The 32-bit a now sits in the **DX:AX** pair.

Step 2 / 7

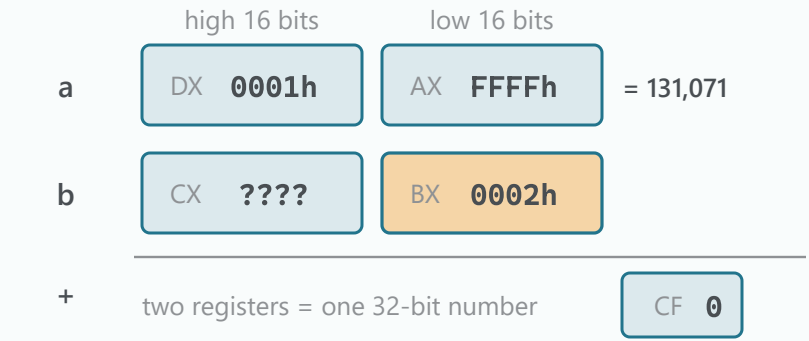
VARIABLES

`AX = FFFFh` `DX = 0001h` `BX = 0000h` `CX = 0000h` `CF = 0`

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
▶ mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
8 ret
```



b = 00030002h (196,610): low half **BX = 0002h**.

Step 3 / 7

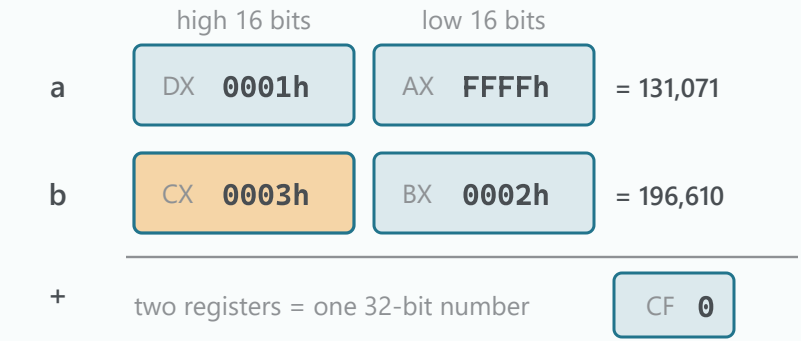
VARIABLES

`AX = FFFFh` `DX = 0001h` `BX = 0002h` `CX = 0000h` `CF = 0`

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
▶ mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
8 ret
```



High half **CX = 0003h**; b sits in the **CX:BX** pair.

Step 4 / 7

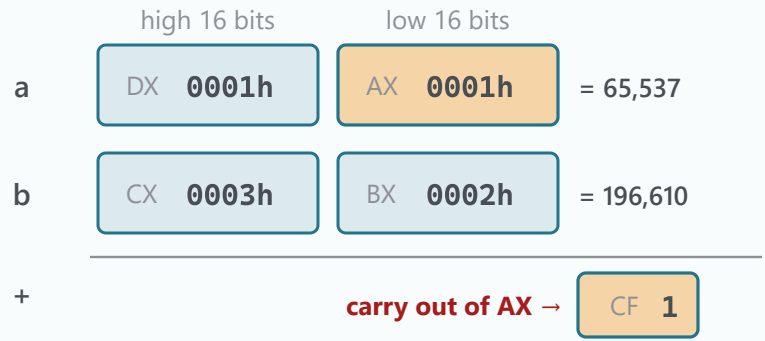
VARIABLES

AX = FFFFh | DX = 0001h | BX = 0002h | CX = 0003h | CF = 0

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
▶ add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
8 ret
```



`add ax, bx` : FFFFh + 0002h = 1 0001h. Bit 17 does not fit in 16-bit AX: AX = **0001h**, and the carry goes to **CF = 1**.

Step 5 / 7

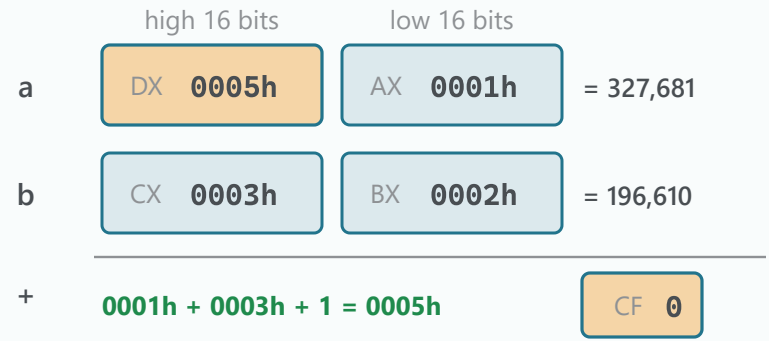
VARIABLES

AX = 0001h DX = 0001h BX = 0002h CX = 0003h CF = 1

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
▶ adc dx, cx     ; high + CF
8 ret
```



`adc dx, cx` : 0001h + 0003h + CF(1) = **0005h**. Result DX:AX = 00050001h = **327,681**.

Step 6 / 7

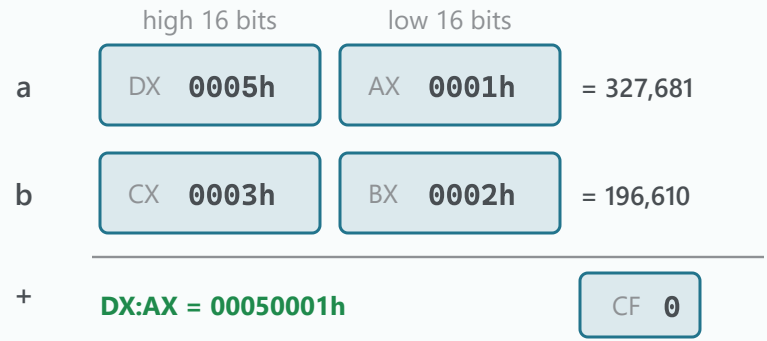
VARIABLES

AX = 0001h DX = 0005h BX = 0002h CX = 0003h CF = 0

`adc` (add with carry) also adds the carry (CF) from the previous addition.

32-Bit Addition on a 16-Bit Processor

```
1 org 100h
2 mov ax, 0FFFFh ; a: low half
3 mov dx, 0001h  ; a: high half
4 mov bx, 0002h  ; b: low half
5 mov cx, 0003h  ; b: high half
6 add ax, bx     ; low halves
7 adc dx, cx     ; high + CF
▶ ret
```



The 16-bit processor needed **two instructions** for a 32-bit add; a 32-bit processor (80386) does it with a single `add`.

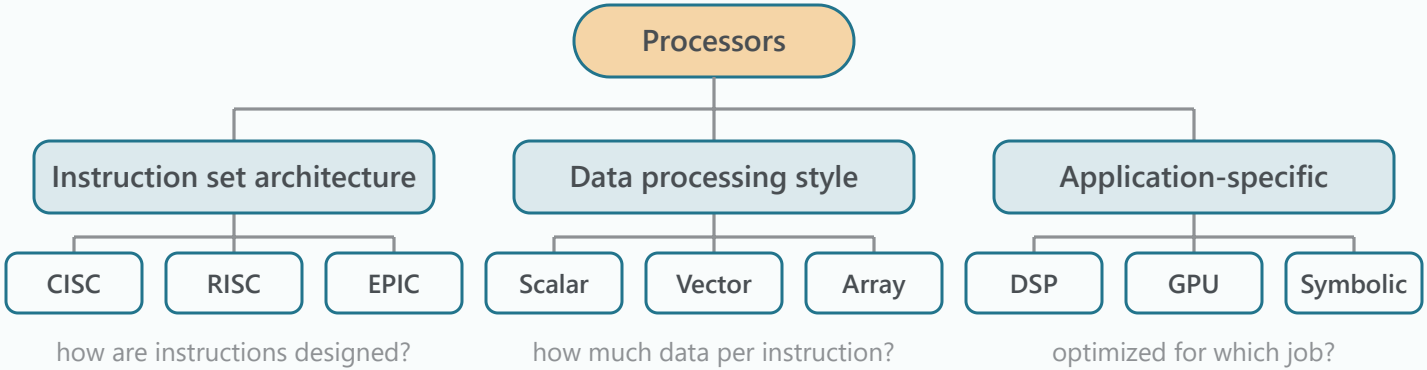
Step 7 / 7

VARIABLES

`AX = 0001h` `DX = 0005h` `BX = 0002h` `CX = 0003h` `CF = 0`

`adc` (add with carry) also adds the carry (CF) from the previous addition.

Processor Types: An Overview



CISC
Complex Instruction Set Computer

RISC
Reduced Instruction Set Computer

EPIC
Explicitly Parallel Instruction Computing

Complex Instruction Set Computer (CISC)

- Performs complex operations in **a single instruction**.
- One instruction can bundle **several low-level operations**, such as a load from memory, an arithmetic operation and a store to memory.
- Instructions offer many **addressing modes** and direct memory operands.
- Instruction lengths **vary** (1–6 bytes on the 8086); decoding is complex.
- Has relatively **few registers**.
- Usually a **microprogrammed** control unit.

Examples

- Intel 8086, 80386, 80486
- Pentium, Pentium Pro, II, III
- Motorola 68000, 68020, 68040
- DEC VAX, IBM System/360

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

One instruction: read + add + write

Reduced Instruction Set Computer (RISC)

- A small number of **simple, fixed-length** instructions (e.g. all 32 bits).
- Fast execution through streamlined instructions and a **pipeline**.
- Goal: ideally **complete one instruction every clock cycle** (CPI ≈ 1).
- Only **load/store** instructions access memory; operations work between registers.
- **Many registers** (32 or more) to cut memory accesses.
- Hardwired control unit; the complexity moves to the compiler.

Examples

- IBM RS/6000 (POWER)
- Motorola MC88100
- DEC Alpha 21064, 21164, 21264
- MIPS, SPARC
- **ARM** (phones, Apple M series)
- **RISC-V** (open-source instruction set)

An instruction does *not finish* in one cycle; it passes through several stages. Thanks to the pipeline, *one instruction completes* every cycle (throughput).

Pipelining: Overlapping the Stages

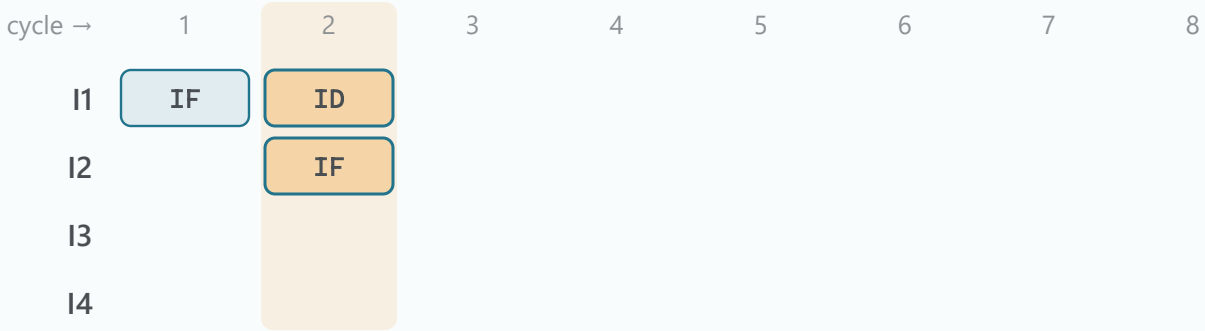


Cycle 1: I1 is **fetched** (IF). The other stages are empty.

Step 1 / 8

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Pipelining: Overlapping the Stages



Completed: 0 / 4

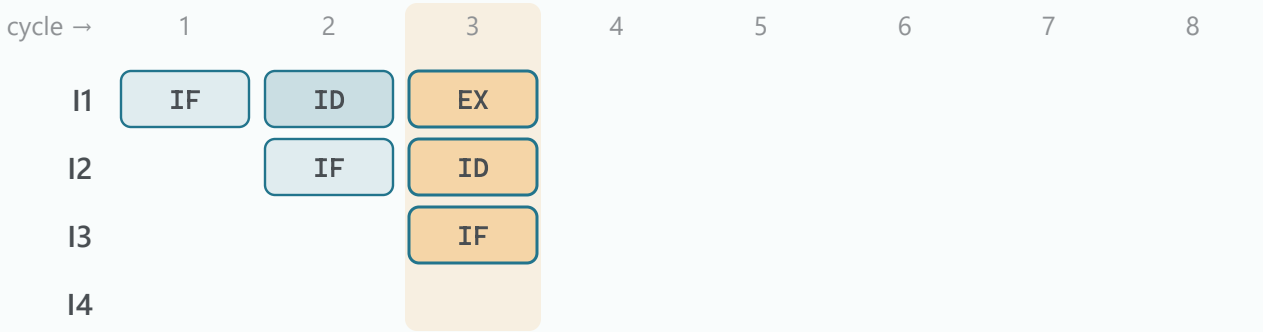
IF fetch · ID decode · EX execute · MEM memory · WB write back

Cycle 2: while I1 is **decoded** (ID), I2 is fetched. Two instructions are in different stages at once.

Step 2 / 8

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

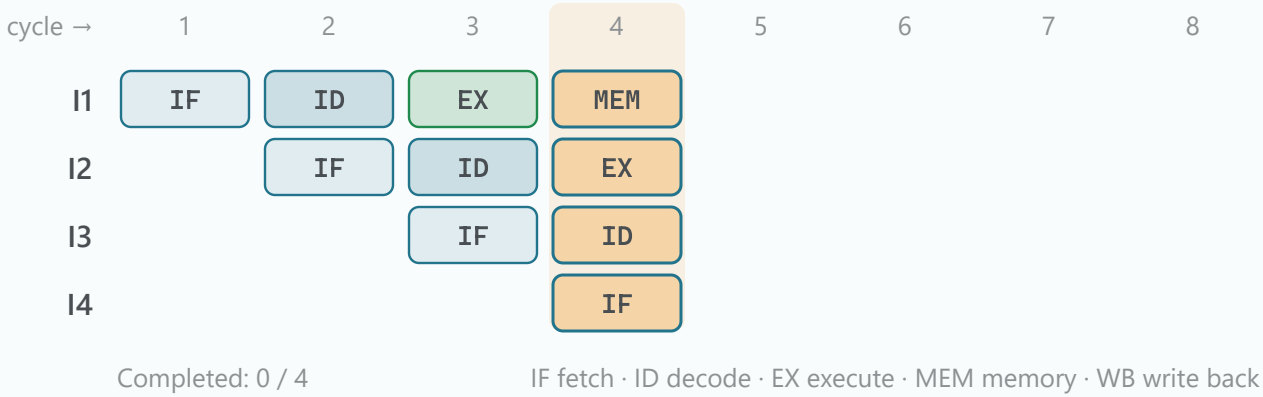
Pipelining: Overlapping the Stages



Cycle 3: three instructions in three different stages.

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Pipelining: Overlapping the Stages

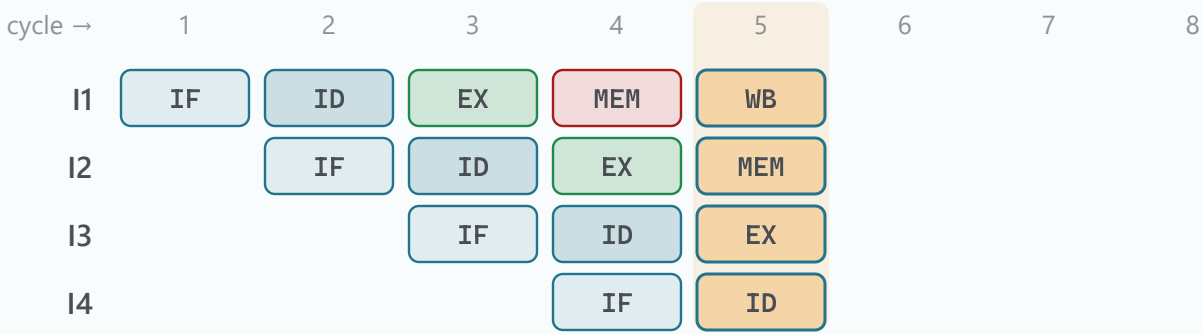


Cycle 4: four instructions in the pipeline at once.

Step 4 / 8

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Pipelining: Overlapping the Stages



Completed: 1 / 4

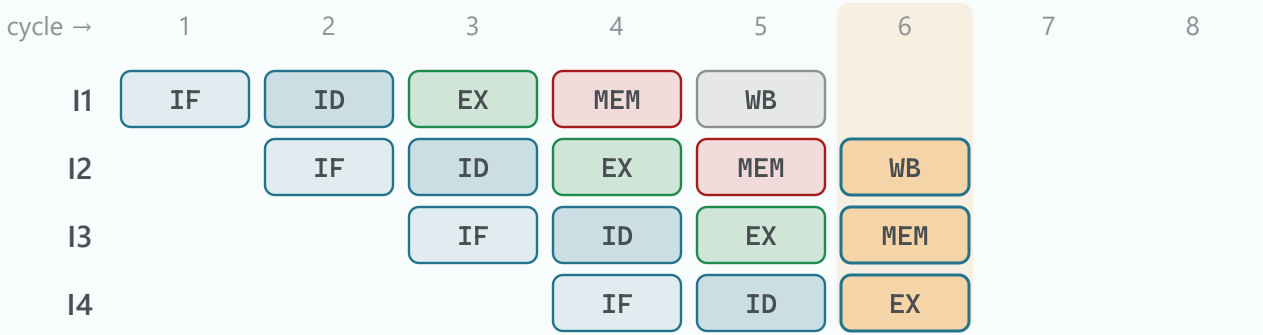
IF fetch · ID decode · EX execute · MEM memory · WB write back

Cycle 5: the pipeline is **full**; I1 completes (WB). From now on one instruction finishes every cycle.

Step 5 / 8

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

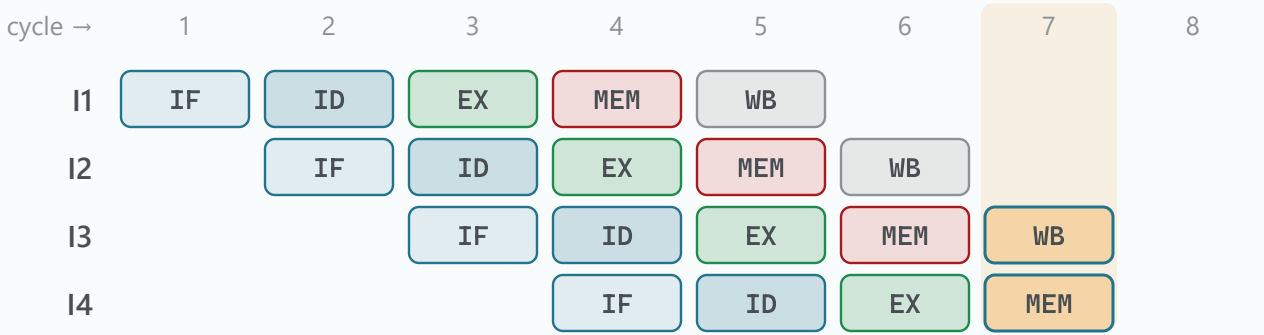
Pipelining: Overlapping the Stages



Cycle 6: I2 completes.

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Pipelining: Overlapping the Stages



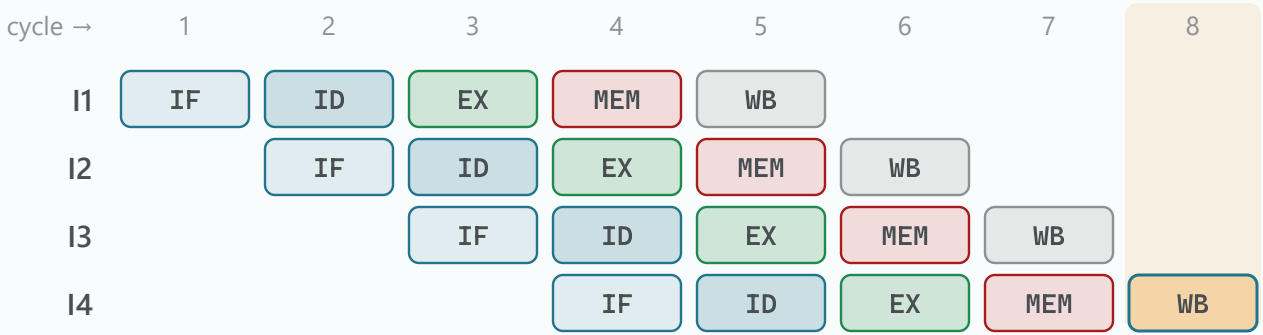
Completed: 3 / 4

IF fetch · ID decode · EX execute · MEM memory · WB write back

Cycle 7: I3 completes.

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Pipelining: Overlapping the Stages



Completed: 4 / 4

IF fetch · ID decode · EX execute · MEM memory · WB write back

Cycle 8: I4 completes. 4 instructions finish in **k + n − 1 = 5 + 4 − 1 = 8** cycles; without a pipeline it would take **4 × 5 = 20**.

Like a factory assembly line: each stage is its own hardware unit and moves on to the next instruction every cycle. Dependences and branches can cause pipeline **stalls**.

Same Job, Two Approaches

Task: add a register's value to a number in memory (`mem[addr] += x`).

CISC (8086)

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

- Short code, small memory footprint.
- The instruction takes many cycles.

RISC (ARM)

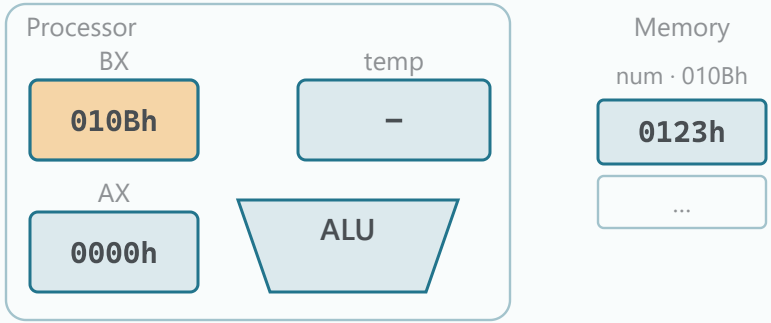
```
LDR r2, [r1]      @ load
ADD r2, r2, r0    @ add
STR r2, [r1]      @ store
```

- 3 simple instructions; each flows smoothly through the pipeline.
- If the value stays in a register, later operations skip memory.

Modern x86 processors (since the Pentium Pro) translate complex instructions internally into RISC-like **micro-operations (µops)**: CISC outside, RISC inside.

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
▶ mov bx, offset num  ; address
3 mov ax, 7           ; value
4 add [bx], ax        ; read, add,
  write
5 add [bx], ax        ; again
6 ret
7 num dw 0123h
```



BX = 010Bh: the address of the variable `num`. In memory num = **0123h**.

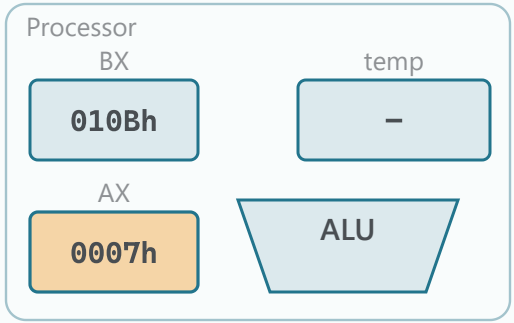
Step 1 / 7

VARIABLES

`BX = 010Bh` `AX = 0000h` `temp = –` `num = 0123h`

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
2 mov bx, offset num  ; address
▶ mov ax, 7           ; value
4 add [bx], ax        ; read, add,
  write
5 add [bx], ax        ; again
6 ret
7 num dw 0123h
```



AX = **0007h**, the value to add.

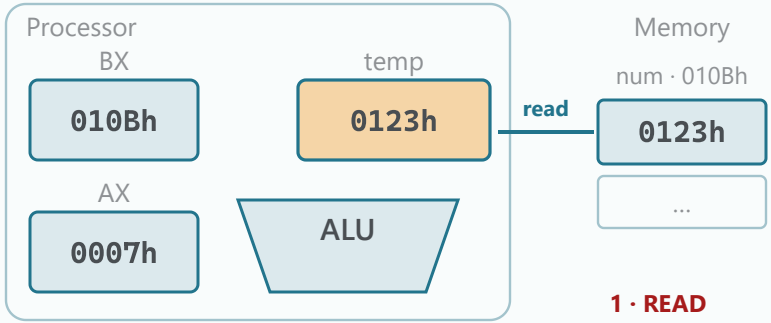
Step 2 / 7

VARIABLES

BX = **010Bh** | AX = **0007h** | temp = **–** | num = **0123h**

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
2 mov bx, offset num   ; address
3 mov ax, 7            ; value
▶ add [bx], ax         ; read, add,
  write
5 add [bx], ax         ; again
6 ret
7 num dw 0123h
```



`add [bx], ax` is one instruction but does three jobs inside. **1 · Read:** the BIU brings the word at 010Bh (0123h) into a temporary register with a memory read cycle.

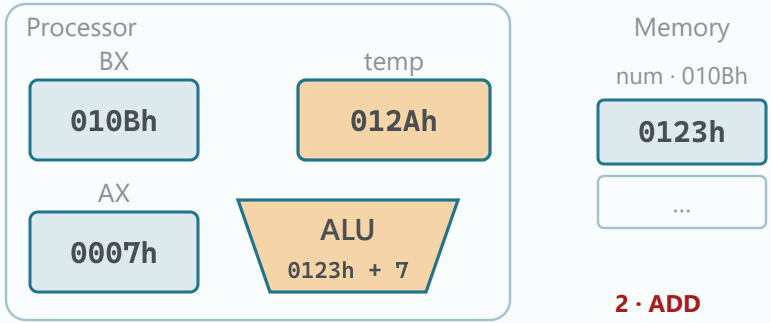
Step 3 / 7

VARIABLES

`BX = 010Bh` `AX = 0007h` `temp = 0123h` `num = 0123h`

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
2 mov bx, offset num   ; address
3 mov ax, 7            ; value
▶ add [bx], ax         ; read, add,
  write
5 add [bx], ax         ; again
6 ret
7 num dw 0123h
```



2 · Add: the ALU computes 0123h + 0007h = **012Ah** and updates the flags.

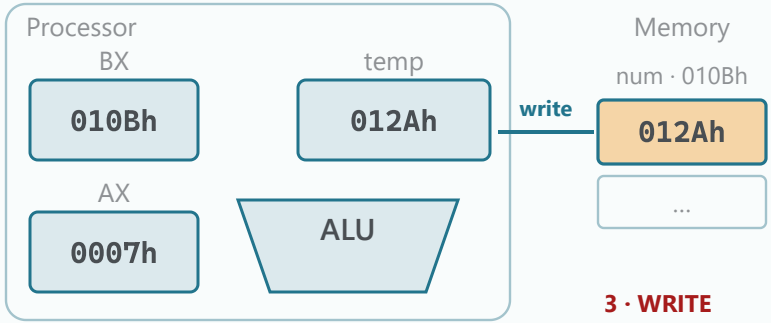
Step 4 / 7

VARIABLES

BX = **010Bh** AX = **0007h** temp = **012Ah** num = **0123h**

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
2 mov bx, offset num   ; address
3 mov ax, 7            ; value
▶ add [bx], ax         ; read, add,
  write
5 add [bx], ax         ; again
6 ret
7 num dw 0123h
```



3 · Write: a second bus cycle writes the result back to memory: num = **012Ah**. One instruction = 2 memory accesses.

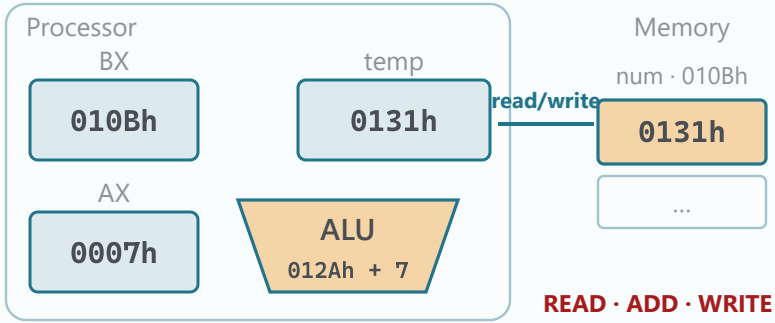
Step 5 / 7

VARIABLES

`BX = 010Bh` `AX = 0007h` `temp = 012Ah` `num = 012Ah`

Code Trace: One CISC Instruction, Three Jobs

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



The same instruction again: read (012Ah), add (+7), write → num = **0131h**.

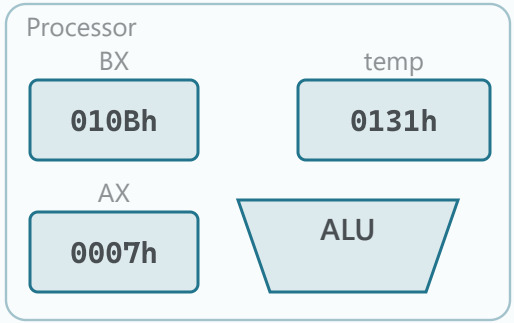
Step 6 / 7

VARIABLES

BX = **010Bh** AX = **0007h** temp = **0131h** num = **0131h**

Code Trace: One CISC Instruction, Three Jobs

```
1 org 100h
2 mov bx, offset num  ; address
3 mov ax, 7           ; value
4 add [bx], ax        ; read, add,
  write
5 add [bx], ax        ; again
▶ ret
7 num dw 0123h
```



CISC code is short: one instruction. On a RISC processor the same job takes three (`LDR` , `ADD` , `STR`), and the intermediate value is visible in a register.

Step 7 / 7

VARIABLES

`BX = 010Bh` `AX = 0007h` `temp = 0131h` `num = 0131h`

PROCESSOR TYPES

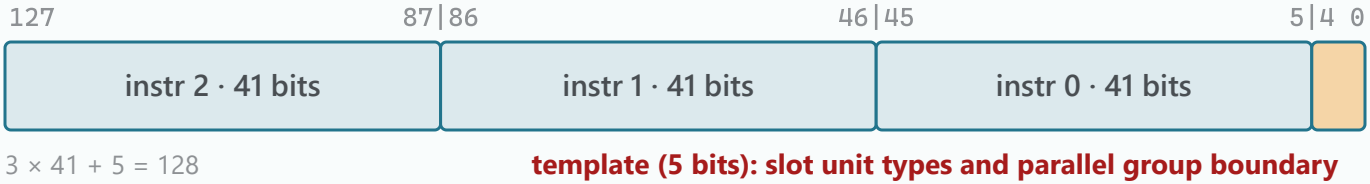
RISC and CISC Processors

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

RISC runs **a few kinds of instructions fast**: high performance, low power. CISC offers **many kinds of instructions**, but each one is slower.

Explicitly Parallel Instruction Computing (EPIC)

- The **compiler**, not the hardware, decides which instructions can run in **parallel** and states it explicitly in the code.
- Many instructions can run at once without a high clock frequency.
- Instructions are encoded in **128-bit bundles**: **three 41-bit instructions** + a **5-bit template**.



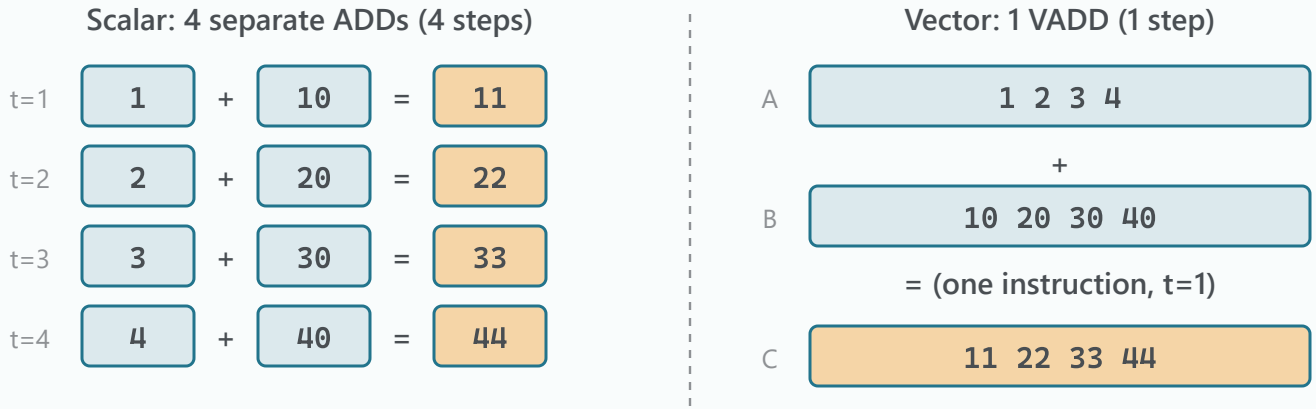
Example

Intel and HP's joint architecture **IA-64**; the Itanium processors (2001–2021).

Outcome

It burdened the compiler and had weak x86 compatibility; it lost to x86-64 in the market.

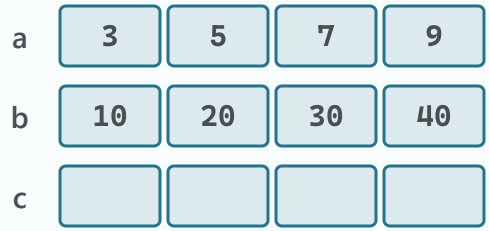
Scalar vs. Vector Processing



A scalar processor handles **one data item** per instruction; a vector processor handles **a whole array of data** (a vector) with one instruction. Instruction fetch/decode and loop overhead are paid once.

Code Trace: c = a + b on a Scalar Processor

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



scalar

2 instr.

vector (4 items)

1 add

SI = 0 (index), CX = 4 (item count). A scalar processor works on a **single** data item per instruction.

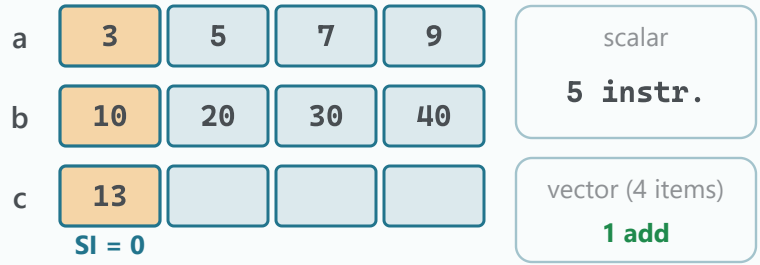
Step 1 / 9

VARIABLES

`SI = 0` `CX = 4` `AL = 0` `instr = 2`

Code Trace: c = a + b on a Scalar Processor

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



Item 0: a[0] + b[0] = 3 + 10 = **13** → c[0]. Three instructions fetched and decoded for one item.

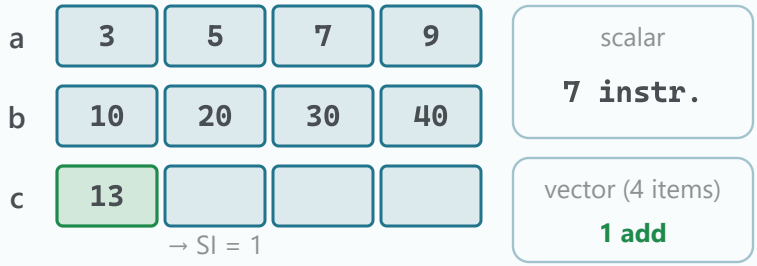
Step 2 / 9

VARIABLES

`SI = 0` `CX = 4` `AL = 13` `instr = 5`

Code Trace: c = a + b on a Scalar Processor

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



SI = 1, CX = 3 ≠ 0 → back to the top. The loop overhead (INC, LOOP) is paid again for every item.

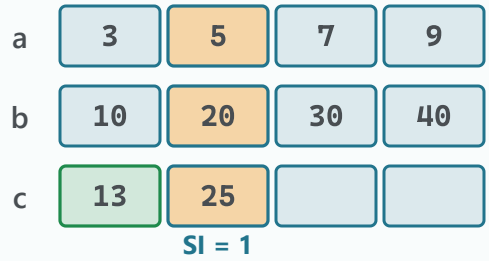
Step 3 / 9

VARIABLES

SI = 1 | CX = 3 | AL = 13 | instr = 7

Code Trace: c = a + b on a Scalar Processor

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



scalar

10 instr.

vector (4 items)

1 add

Item 1: a[1] + b[1] = 5 + 20 = **25** → c[1]. Three instructions fetched and decoded for one item.

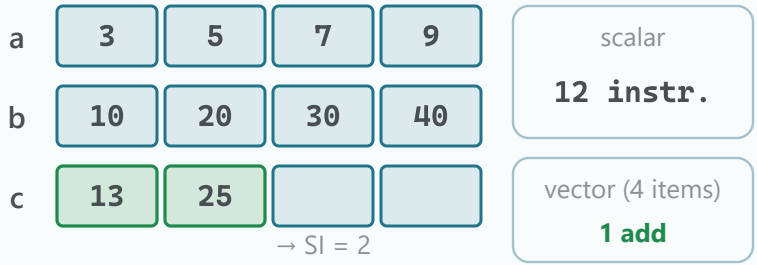
Step 4 / 9

VARIABLES

SI = **1** CX = **3** AL = **25** instr = **10**

Code Trace: c = a + b on a Scalar Processor

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



SI = 2, CX = 2 ≠ 0 → back to the top. The loop overhead (INC, LOOP) is paid again for every item.

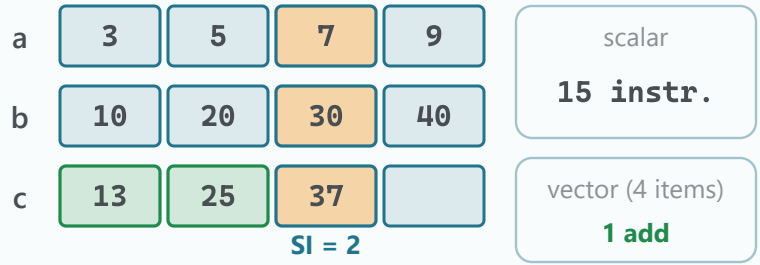
Step 5 / 9

VARIABLES

SI = 2 | CX = 2 | AL = 25 | instr = 12

Code Trace: c = a + b on a Scalar Processor

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



Item 2: a[2] + b[2] = 7 + 30 = **37** → c[2]. Three instructions fetched and decoded for one item.

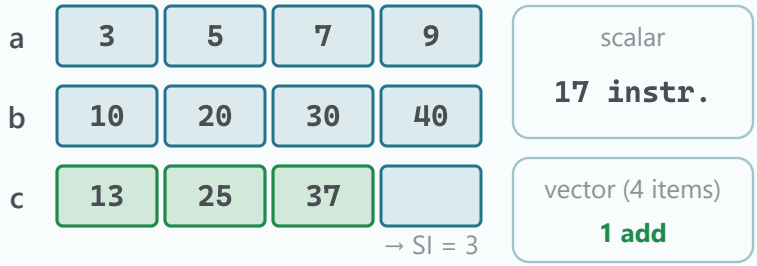
Step 6 / 9

VARIABLES

`SI = 2` `CX = 2` `AL = 37` `instr = 15`

Code Trace: c = a + b on a Scalar Processor

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



SI = 3, CX = 1 ≠ 0 → back to the top. The loop overhead (INC, LOOP) is paid again for every item.

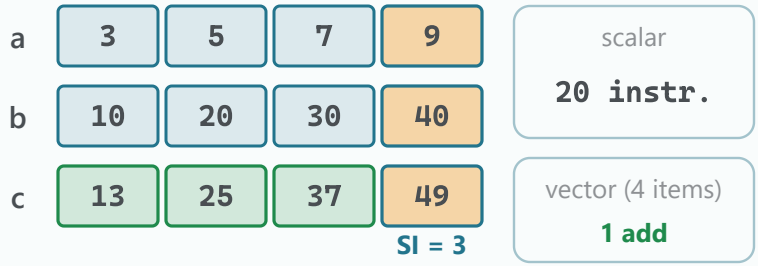
Step 7 / 9

VARIABLES

`SI = 3` `CX = 1` `AL = 37` `instr = 17`

Code Trace: c = a + b on a Scalar Processor

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



Item 3: a[3] + b[3] = 9 + 40 = **49** → c[3]. Three instructions fetched and decoded for one item.

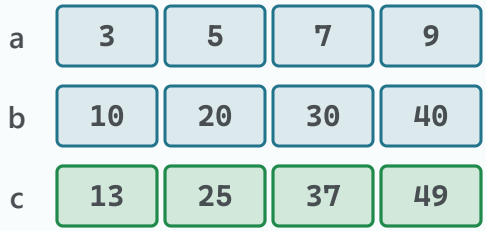
Step 8 / 9

VARIABLES

`SI = 3` `CX = 1` `AL = 49` `instr = 20`

Code Trace: c = a + b on a Scalar Processor

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



scalar

22 instr.

vector (4 items)

1 add

CX = 0: the loop is done. Four additions took **22 instructions**; a vector processor handles all four items with a single vector add.

Step 9 / 9

VARIABLES

SI = 4 | CX = 0 | AL = 49 | instr = 22

Vector Processors

- Perform mathematical operations on arrays of data called **vectors**.
- For such work they are much faster than **scalar** processors, which handle one data item at a time.
- They process many data items at once (or back to back in a deep pipeline).
- They have long **vector registers** and vector load/store units.
- Used in high-performance systems such as supercomputers, servers and GPUs.



History
the Cray-1 (1976) supercomputer

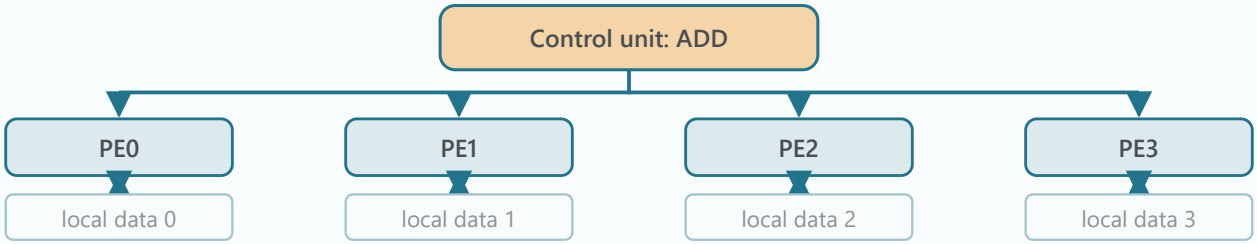
Today
x86 SSE/AVX, ARM NEON/SVE, the RISC-V V extension

Uses
weather forecasting, physics simulation, AI

Array Processors

- Also called **SIMD** (single instruction, multiple data).
- One control unit broadcasts the same instruction to many **processing elements** (PEs); each PE works on its own data.
- Optimized for parallel computation and vector operations.
- Scalar processors, by contrast, handle one data item.
- Used in supercomputers, servers and GPUs; rare in general-purpose computing.

Flynn's taxonomy	
SISD	single instruction, single data (classic 8086)
SIMD	single instruction, multiple data (array/vector, GPU)
MISD	multiple instruction, single data (rare; fault tolerance)
MIMD	multiple instruction, multiple data (multicore, clusters)

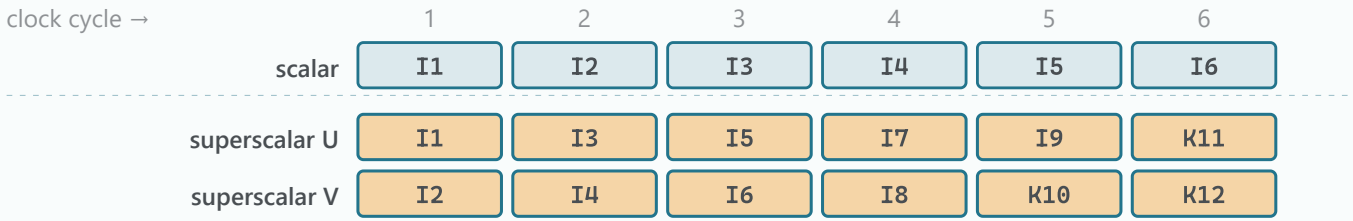


Scalar Processors

- Process scalar (non-vector) data, **one data item** per instruction.
- Simple scalar processors handle integers only.
- More powerful ones also handle floating point: the integer unit (**ALU**) and the floating-point unit (**FPU**) share the chip.
- They can be CISC or RISC.
- The most common type in general-purpose computing.

Superscalar

A scalar processor with several **pipelines**/execution units that can issue **more than one instruction per clock cycle**. The first x86 example: the **Pentium** (U and V pipes).



Digital Signal Processors (DSP)

- Optimized for processing digital signals.
- A digital signal is a **sequence of samples** representing sensory information such as sound or images.
- An instruction set and hardware tailored to signal processing: single-cycle **multiply-accumulate (MAC)**, circular buffer addressing, zero-overhead loops.
- Applications:
  - audio processing and filtering
  - image processing and enhancement
  - telecom infrastructure (modems, base stations)

FIR filter

$$y[n] = \Sigma\, h[k] \cdot x[n - k]$$

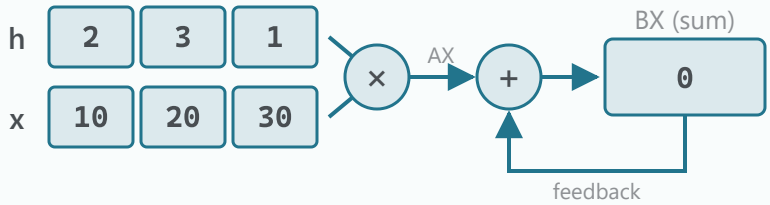
N multiplies + N adds per output sample. A DSP does each term with **one MAC instruction**:

```
A ← A + h[k] × x[n−k]
```

E.g. TI TMS320, Analog Devices SHARC

Code Trace: An FIR Sum on the 8086

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



AX = 0 y = 2·10 + 3·20 + 1·30 = 110

BX = 0 is the accumulator, CX = 3 terms. Filter output y = h[0]·x[0] + h[1]·x[1] + h[2]·x[2].

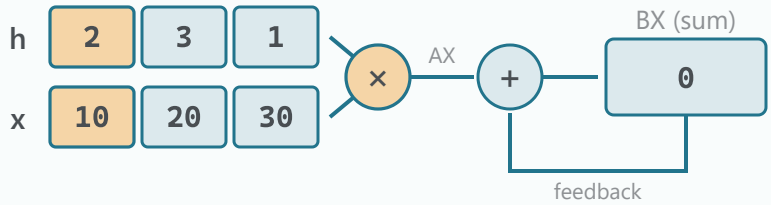
Step 1 / 7

VARIABLES

SI = 0 | CX = 3 | AX = 0 | BX = 0

Code Trace: An FIR Sum on the 8086

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



AX = 20 y = 2·10 + 3·20 + 1·30 = 110

Term 0: AL = h[0] = 2, `mul` → AX = 2 × 10 = **20**. On the 8086 a MUL alone takes more than 70 cycles.

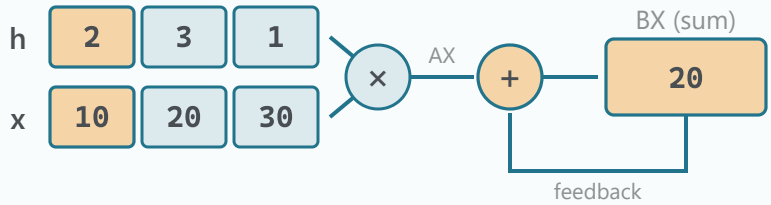
Step 2 / 7

VARIABLES

SI = **0** CX = **3** AX = **20** BX = **0**

Code Trace: An FIR Sum on the 8086

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



AX = 20 y = 2·10 + 3·20 + 1·30 = 110

`add bx, ax` : sum = **20**. Multiply and add are two separate instructions; on a DSP they are one MAC.

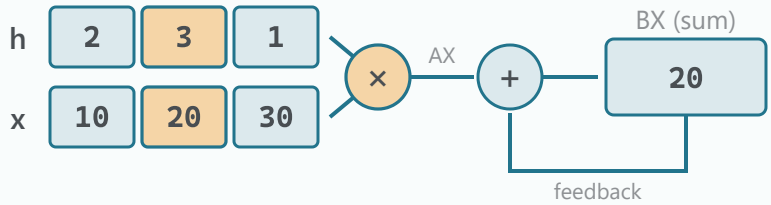
Step 3 / 7

VARIABLES

SI = **0** CX = **3** AX = **20** BX = **20**

Code Trace: An FIR Sum on the 8086

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



AX = 60 y = 2·10 + 3·20 + 1·30 = 110

Term 1: AL = h[1] = 3, `mul` → AX = 3 × 20 = **60**. On the 8086 a MUL alone takes more than 70 cycles.

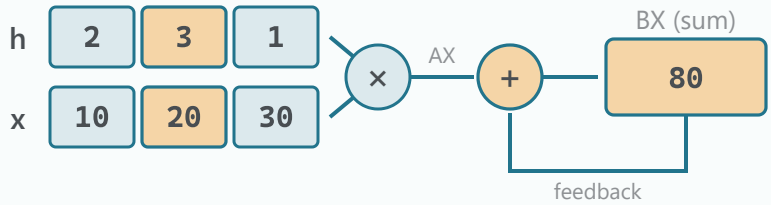
Step 4 / 7

VARIABLES

SI = **1** CX = **2** AX = **60** BX = **20**

Code Trace: An FIR Sum on the 8086

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



AX = 60 y = 2·10 + 3·20 + 1·30 = 110

`add bx, ax` : sum = **80**. Multiply and add are two separate instructions; on a DSP they are one MAC.

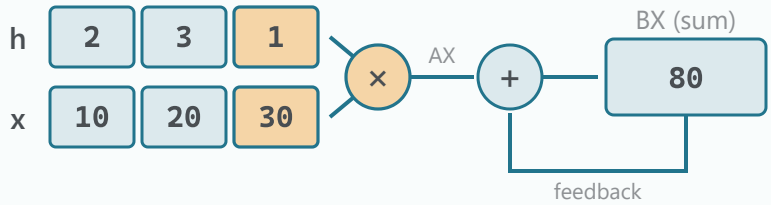
Step 5 / 7

VARIABLES

SI = **1** CX = **2** AX = **60** BX = **80**

Code Trace: An FIR Sum on the 8086

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



AX = 30 y = 2·10 + 3·20 + 1·30 = 110

Term 2: AL = h[2] = 1, `mul` → AX = 1 × 30 = **30**. On the 8086 a MUL alone takes more than 70 cycles.

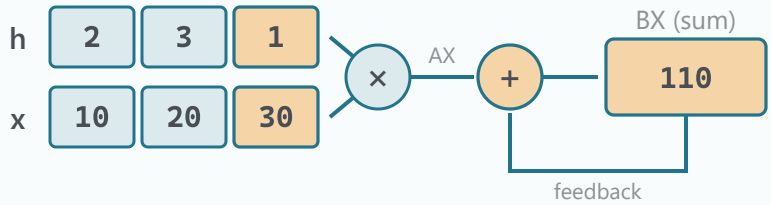
Step 6 / 7

VARIABLES

SI = **2** CX = **1** AX = **30** BX = **80**

Code Trace: An FIR Sum on the 8086

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



AX = 30 y = 2·10 + 3·20 + 1·30 = 110

`add bx, ax` : sum = **110**. Three terms done: y = **110**. Each term needed its own MUL + ADD (+ loop overhead); a DSP does it with one **MAC** instruction in one cycle.

Step 7 / 7

VARIABLES

SI = **2** CX = **1** AX = **30** BX = **110**

Symbolic Processors

- Built for **symbolic computation**: it manipulates symbols, lists and rules instead of numbers.
- Also known as **PROLOG** (and **LISP**) processors.
- Focuses on semantic computation and rules of symbolic logic.
- Manipulates mathematical expressions and symbols; can simplify complex expressions and take derivatives.

Application areas

- Expert systems
- Machine intelligence
- Rule-based inference

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

A Prolog rule: symbolic inference

The LISP machines of the 1980s (Symbolics) and Japan's Fifth Generation Computer project were of this kind; today this work is done in software on general-purpose processors.

Graphics Processing Units (GPU)

- Designed to **accelerate** 2D and 3D graphics.
- Renders high-quality graphics in real time.
- Efficient at rendering tasks: **shading**, **texturing**, rasterization.
- Thousands of simple cores run the same program on different pixels (SIMD/SIMT).
- Found in dedicated hardware such as graphics cards and accelerators.

CPU

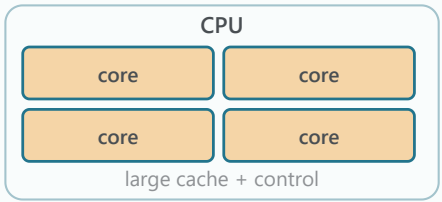
a few powerful cores · cuts latency · complex control flow

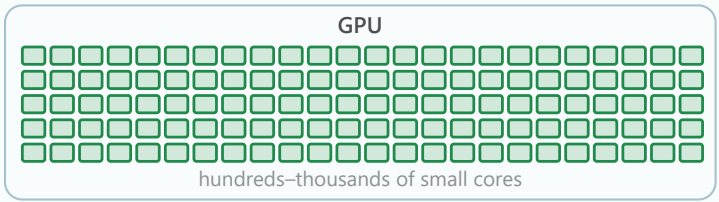
GPU

thousands of simple cores · raises throughput · data parallel

GPGPU

scientific computing and deep learning with CUDA/OpenCL





Processor Types: Summary Table

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

The Evolution of Microprocessors

1947 · The transistor

Invented at Bell Labs (Bardeen, Brattain, Shockley) and announced in 1948; the microelectronics revolution began.

The Evolution of Microprocessors

1947 · The transistor

Invented at Bell Labs (Bardeen, Brattain, Shockley) and announced in 1948; the microelectronics revolution began.

1958 · The integrated circuit (IC)

Jack Kilby demonstrated the first integrated circuit at Texas Instruments. Many transistors on one chip made complex circuits possible.

The Evolution of Microprocessors

1947 · The transistor

Invented at Bell Labs (Bardeen, Brattain, Shockley) and announced in 1948; the microelectronics revolution began.

1958 · The integrated circuit (IC)

Jack Kilby demonstrated the first integrated circuit at Texas Instruments. Many transistors on one chip made complex circuits possible.

1971 · The first microprocessor

Intel 4004: a whole CPU on one chip. The microprocessor became the computer's central processing unit.

The Evolution of Microprocessors

1947 · The transistor

Invented at Bell Labs (Bardeen, Brattain, Shockley) and announced in 1948; the microelectronics revolution began.

1958 · The integrated circuit (IC)

Jack Kilby demonstrated the first integrated circuit at Texas Instruments. Many transistors on one chip made complex circuits possible.

1971 · The first microprocessor

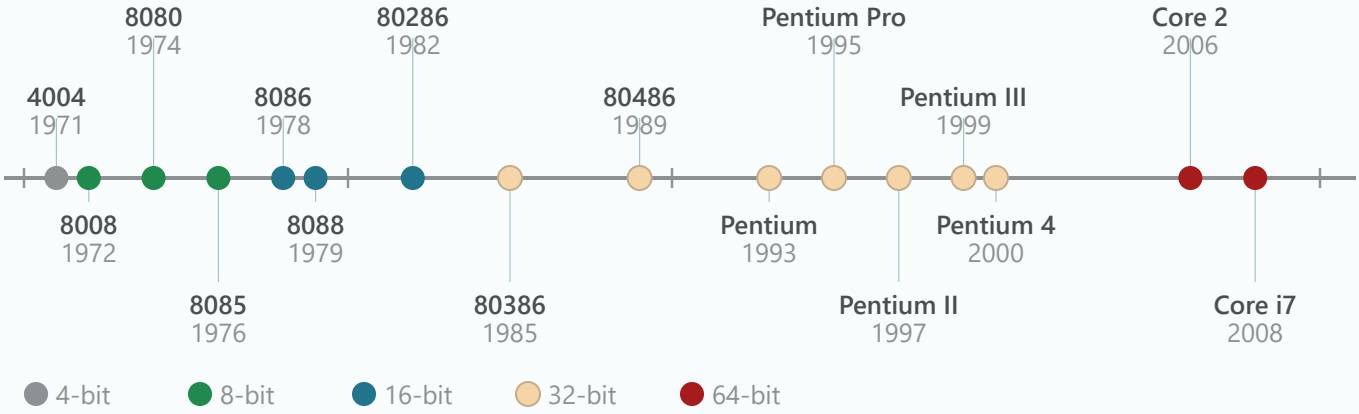
Intel 4004: a whole CPU on one chip. The microprocessor became the computer's central processing unit.

Result

Computers shrank from room size to the desktop and then to the pocket: **smaller**, more **powerful**, **faster** and **cheaper**.

EVOLUTION OF INTEL X86

Intel Processors: A Timeline



The name x86 comes from the common “86” suffix of the 80186, 80286, 80386 and 80486 that followed the 8086. All of them run 8086 code: **backward compatibility**.

4- and 8-Bit Microprocessors

Processor	Year	Clock	Transistors	Instruction rate / note
4004 (4-bit)	1971	740 kHz	2,300	≈ 60,000 instr/s; Ted Hoff, Federico Faggin, Stanley Mazor
4040 (4-bit)	1974	740 kHz	≈ 3,000	improved 4004 (interrupts, more instructions)
8008 (8-bit)	1972	500 kHz	3,500	≈ 50,000 instr/s; the first 8-bit
8080 (8-bit)	1974	2 MHz	6,000	≈10× the 8008's speed; 16-bit address (64 KB); first personal computers (Altair 8800)
8085 (8-bit)	1976	3 MHz	6,500	single +5 V supply; 16-bit address bus; ≈ 769,000 instr/s (shortest instruction 1.3 µs)

16-Bit Microprocessors

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

8086 instructions take more than 10 cycles on average, so at 5 MHz you get about **0.33 MIPS**.

32-Bit Microprocessors

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

64-Bit Microprocessors

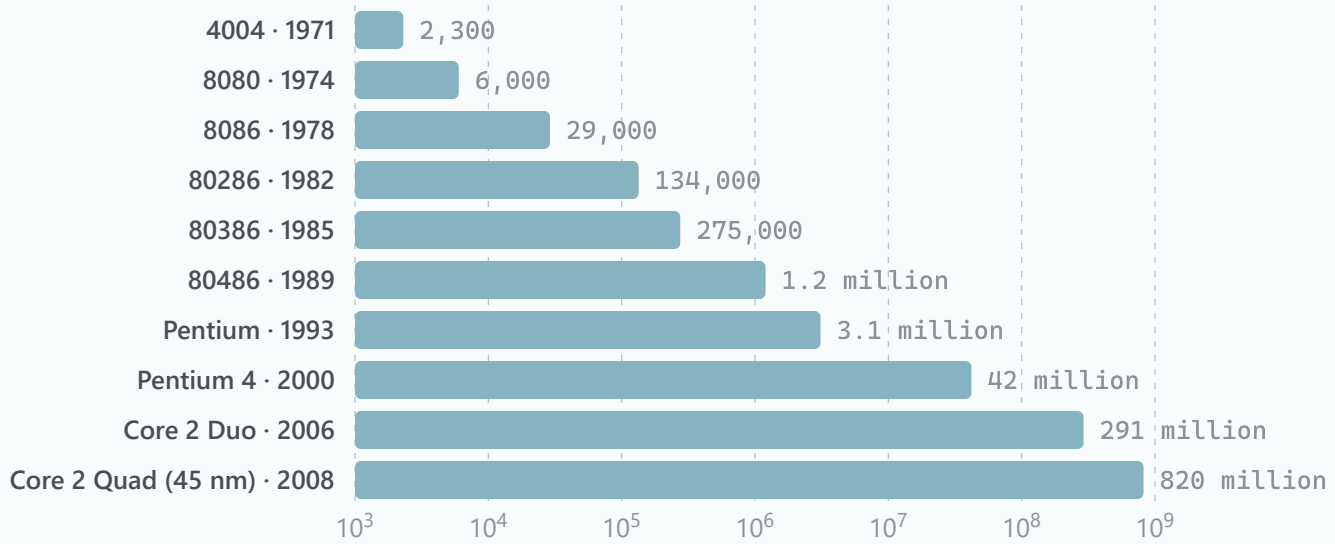
Processor	Year	Clock	Transistors	Feature
Core 2	2006	1.2 – 3 GHz	291 million (Duo)	64 KB L1 per core (32 KB instruction + 32 KB data), 2–4 MB shared L2; versions: Duo, Quad, Extreme
Core i7	2008	2.66 – 3.33 GHz	731 million	4 cores, Hyper-Threading, integrated memory controller, 8 MB L3
Core i5	2009	2.4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

64-bit x86 (x86-64) was introduced by **AMD** (AMD64, Opteron/Athlon 64, 2003); Intel adopted it in 2004 with the Pentium 4.

Clock ranges are for 1st-generation desktop/mobile models.

EVOLUTION OF INTEL X86

Transistor Count and Moore's Law



Moore's law (1965/1975): the number of transistors per chip roughly **doubles every two years**. From the 4004 to the Core 2 Quad, 37 years gave $\approx 2^{18}\times$.

8080 and 8086

The structure, organization and technology of microprocessors changed dramatically: from the 8080, the first general-purpose microprocessor, to the 29,000-transistor 8086 and the 820-million-transistor quad-core Core 2.

8080 (1974)

- Known as the first **general-purpose** microprocessor.
- 8-bit data bus, 16-bit address bus (64 KB).
- Used in the first personal computers (Altair 8800).

8086 (1978)

- 16-bit data bus and more registers, all 16 bits wide.
- 20-bit address bus: **1 MB** of addressable memory.
- **Real mode** only.
- **Prefetches** instructions into a **6-byte instruction queue** before running them → fetch and execute overlap.

The 8086 has no cache; the instruction queue provides the speedup. On-chip cache came to x86 with the 80486.

80286, 80386 and 80486

80286 (1982)

- **16 MB** of addressable memory (24-bit address).
- Real mode + **16-bit protected mode**.
- 16-bit data bus and registers.

80386 (1985)

- Intel's **first 32-bit** microprocessor.
- Supports multitasking in hardware.
- 32-bit protected mode, **paging**, virtual 8086 mode.
- **4 GB** of addressable memory.

80486 (1989)

- On-chip **cache** (8 KB) and an instruction **pipeline**.
- Page write protection (the CR0.WP bit).
- An **integrated math coprocessor** (FPU) for complex arithmetic; previously the separate 80387 chip.

Protected mode: each program's memory is isolated from the others; one program's bug cannot crash the operating system. This is the basis of modern operating systems.

Pentium and Pentium Pro

Pentium (1993)

- **Superscalar:** two pipelines (U, V) let instructions run in parallel.
- Branch prediction (a first for x86).
- **Page size extension** (PSE): 4 MB pages alongside 4 KB → better paging.
- Separate instruction and data caches.

Pentium Pro (1995)

- Advanced **branch prediction**.
- Out-of-order execution through **data flow analysis**.
- **Speculative execution**.
- A **level 2 (L2) cache** in the same package.
- PAE: 32-bit virtual addresses map to **36-bit physical** addresses (64 GB).

Intel called the first three Pentium Pro techniques together “dynamic execution”.

Pentium II, III and 4

Pentium II (1997)

- Combined **MMX** technology (integer SIMD instructions for multimedia) with the Pentium Pro core.
- Handles video, audio and graphics data efficiently.

Pentium III (1999)

- **SSE** (Streaming SIMD Extensions): **70 new instructions**, floating-point SIMD.
- Supports 3D graphics software.
- Top clock speed 1.4 GHz.

Pentium 4 (2000)

- Floating-point improvements for multimedia (**SSE2**, later SSE3).
- A very deep pipeline (NetBurst), up to 3.8 GHz.
- Hyper-Threading.

MMX first appeared in 1997 in the “Pentium MMX”; the Pentium II inherited it.

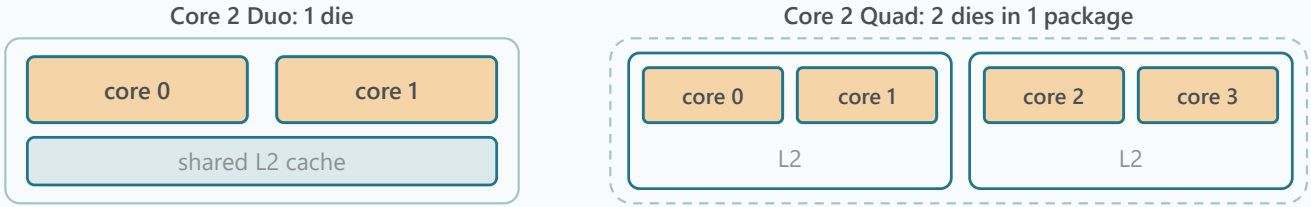
Core and Core 2

Core (Core Duo, 2006)

- The first Intel processor in which two cores on the **same die** share the L2 cache.
- Added **virtualization technology** (VT-x).
- Low power (a mobile-derived design).

Core 2 (2006)

- **64-bit** architecture (Intel 64).
- 64-bit register set and addressing modes.
- **Core 2 Quad**: four cores in one package (two dual-core dies).



Intel's first dual-core desktop processor was the Pentium D in 2005 (its cores had separate L2 caches).

Advantages of the x86 Architecture

Compatibility

Old software runs unchanged on new x86 processors: code written for the 8086 in 1978 can still run today.

Performance

Each new processor generation was faster and more efficient than the last.

Versatility

A wide range of uses, from personal computers to servers, embedded systems and mobile devices.

Broad industry support

Backed by a large ecosystem of hardware and software vendors; this support has driven innovation.

Disadvantages of the x86 Architecture

Complex instruction set

Thousands of instructions and variable-length encoding; the decoder gets complex and code is harder to optimize.

Power consumption

Power draw rose sharply as x86 evolved, a big problem where battery life is critical.

Heat

More powerful processors run hotter; cooling cost and noise go up.

Cost and licensing

Intel owns the x86 architecture; only licensed firms (e.g. AMD) can build it, which limits competition.

That is why **ARM** (RISC) dominates phones and tablets, while x86 is strong on desktops and servers.

Clock Speed Alone Is Not Performance

$$\textbf{CPU time} = N \times CPI \times T = \textbf{N} \times \textbf{CPI} / \textbf{f}$$

Definitions

N: instructions executed · CPI: average clock cycles per instruction · f: clock frequency (T = 1/f).

Clock Speed Alone Is Not Performance

$$\textbf{CPU time} = N \times CPI \times T = \textbf{N} \times \textbf{CPI} / f$$

Definitions

N: instructions executed · CPI: average clock cycles per instruction · f: clock frequency (T = 1/f).

Processor A: f = 2 GHz, CPI = 2

N = 10^9 → time = $10^9 \times 2 / (2 \times 10^9)$ = **1 s**

Clock Speed Alone Is Not Performance

$$\textbf{CPU time} = N \times CPI \times T = \textbf{N} \times \textbf{CPI} / f$$

Definitions

N: instructions executed · CPI: average clock cycles per instruction · f: clock frequency (T = 1/f).

Processor A: f = 2 GHz, CPI = 2

N = 10^9 → time = $10^9 \times 2 / (2 \times 10^9)$ = **1 s**

Processor B: f = 1.5 GHz, CPI = 1

time = $10^9 \times 1 / (1.5 \times 10^9)$ ≈ **0.67 s** → B, with the slower clock, is **50% faster**!

Clock Speed Alone Is Not Performance

$$\textbf{CPU time} = N \times CPI \times T = \textbf{N} \times \textbf{CPI} / f$$

Definitions

N: instructions executed · CPI: average clock cycles per instruction · f: clock frequency (T = 1/f).

Processor A: f = 2 GHz, CPI = 2

N = 10^9 → time = $10^9 \times 2 / (2 \times 10^9)$ = **1 s**

Processor B: f = 1.5 GHz, CPI = 1

time = $10^9 \times 1 / (1.5 \times 10^9)$ ≈ **0.67 s** → B, with the slower clock, is **50% faster**!

Takeaway

Clock speed counts cycles per second, not instructions. Instructions/s = f / CPI; architecture (pipelines, caches, superscalar) lowers CPI.

Modern Processor Features (1)

Clock speed

Clock cycles per second (GHz). Instruction rate also depends on CPI.

Instruction set architecture (ISA)

Architectures such as CISC and RISC affect efficiency and complexity; the ISA is the contract between software and hardware.

Cache

Small, very fast memory that keeps frequently used data close to the processor (L1, L2, L3).

Multicore

Runs several tasks truly in parallel, raising overall performance.

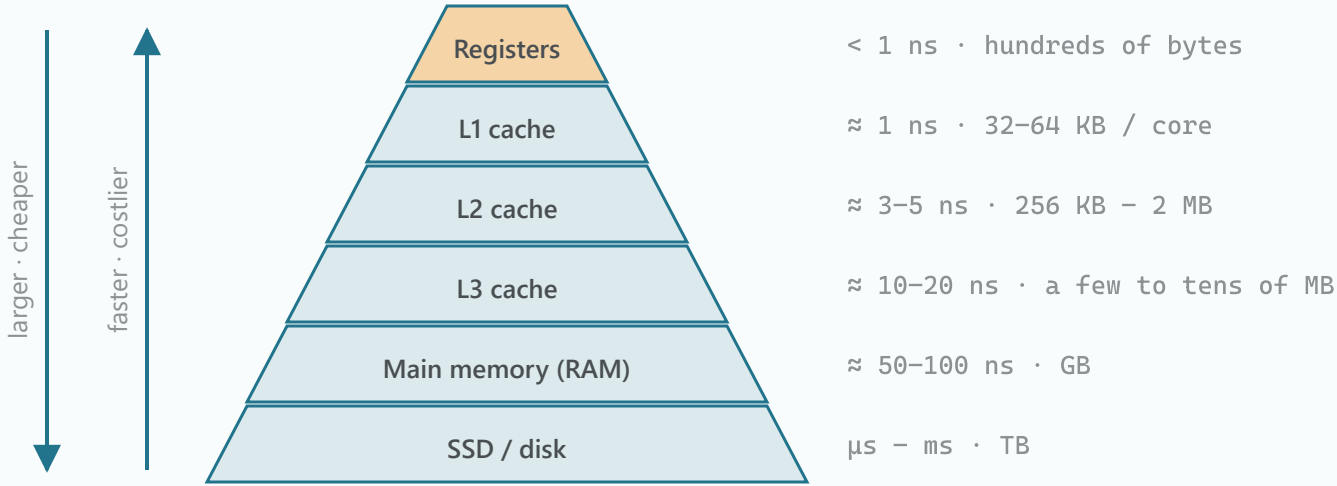
Virtualization

Lets several operating systems (virtual machines) run isolated on the same hardware (Intel VT-x, AMD-V).

Power management

Clock and voltage drop when the load falls (DVFS); idle cores are put to sleep.

Cache and the Memory Hierarchy



Programs access the same and nearby data again and again (**the principle of locality**). By keeping that data close, the cache brings the average access time far below RAM's. Times are approximate.

Modern Processor Features (2)

Power management

Modern processors cut power use and raise energy efficiency: Turbo/SpeedStep, sleep states (C-states).

Graphics

An integrated graphics unit (iGPU) handles graphics-heavy tasks faster and more efficiently.

Security features

Hardware encryption (AES-NI), secure boot, isolated secure enclaves (TPM, SGX/TrustZone).

Connectivity

Built-in network units such as Wi-Fi, Bluetooth and Ethernet, especially in mobile and embedded chips (SoCs).

Machine learning

Dedicated AI units: NPUs, matrix accelerators (Intel AMX), vector dot-product instructions.

System on a chip (SoC)

CPU, GPU, NPU, memory controller and I/O on one chip; a return to the microcontroller idea.

Summary

- A microprocessor is a programmable CPU chip that runs the instructions in memory through the **fetch–decode–execute** cycle; a microcontroller also carries memory and peripherals on the same chip.
- A CPU consists of the **ALU**, the **control unit**, **registers** and the **bus interface**; it talks to the outside world over the address, data and control buses.
- Period T = 1/f; execution time = N × CPI / f. Clock speed alone is not a measure of performance.
- **CISC** offers rich instructions, **RISC** simple instructions + pipelining, **EPIC** compiler-driven parallelism; vector, SIMD, DSP, GPU and symbolic processors target special jobs.
- Intel 4004 (1971) → 8086 (1978, 16 bits, 1 MB) → 80386 (32 bits) → Core 2 (64 bits): backward compatibility is x86's greatest strength.
- Modern processors gain performance through cache, multiple cores, virtualization, power management and special accelerators.

Exercises

1. On a 5 MHz 8086, what is the clock period, and how long does a 4-T bus cycle take?
2. How many bytes can a 24-bit address bus reach? Which processor has one?
3. How many cycles do 100 instructions take in a 5-stage pipeline? How many without a pipeline?
4. How many seconds does a program of $N = 2 \times 10^9$ instructions take on a 3 GHz processor with CPI = 1.5?
5. For a washing machine controller and for a laptop, would you choose a microprocessor or a microcontroller? Why?
6. How are the 128 bits of an EPIC bundle divided up?

▶ **Hints / answers**

References

- Tanenbaum and Bos, *Modern Operating Systems*, 3rd ed., Pearson.
- Brey, *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium …*, 8th ed., Pearson.
- Stallings, *Computer Organization and Architecture*, 10th ed., Pearson.
- Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.
- Intel Corporation, *8086 16-Bit HMOS Microprocessor* data sheet and the Intel 64 and IA-32 Architectures Software Developer's Manual.

The End

Chapter 1.1: Introduction

Next: Chapter 1.2: Number Systems