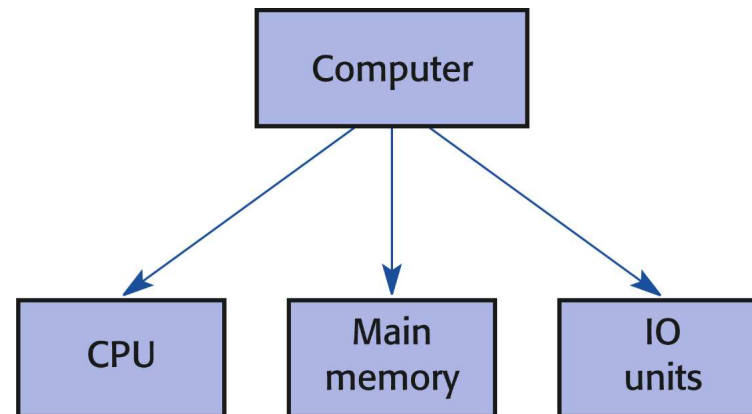


2 CPU Architecture: Fetch-Execute Cycle

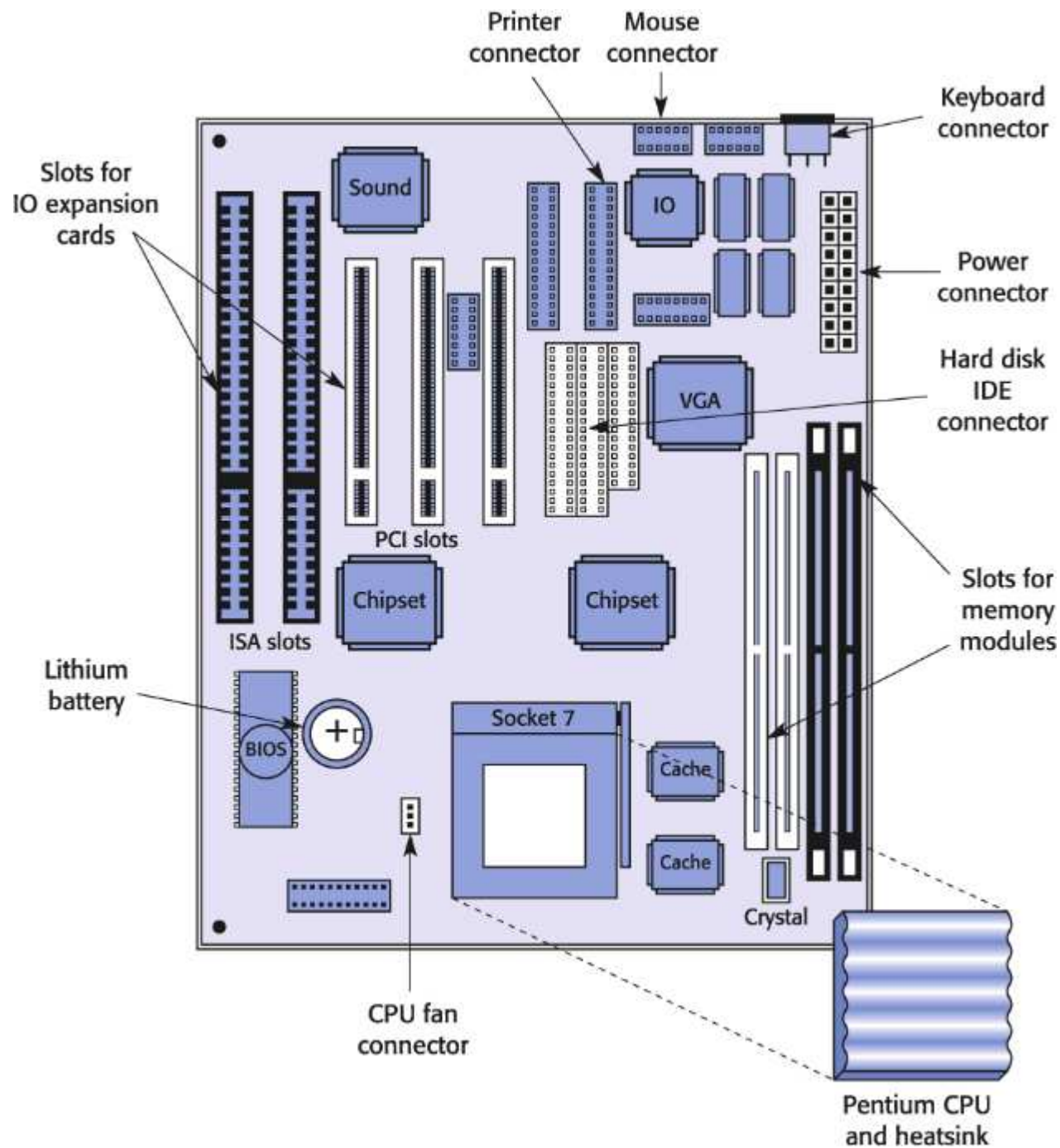
2.1 CPU, Main memory, I/O units

Any modern digital computer can be described as having only three major **functional hardware units**:



(For now, we will ignore *secondary memory*, like hard disk drives.)

PC motherboard (Intel Pentium)



Bus

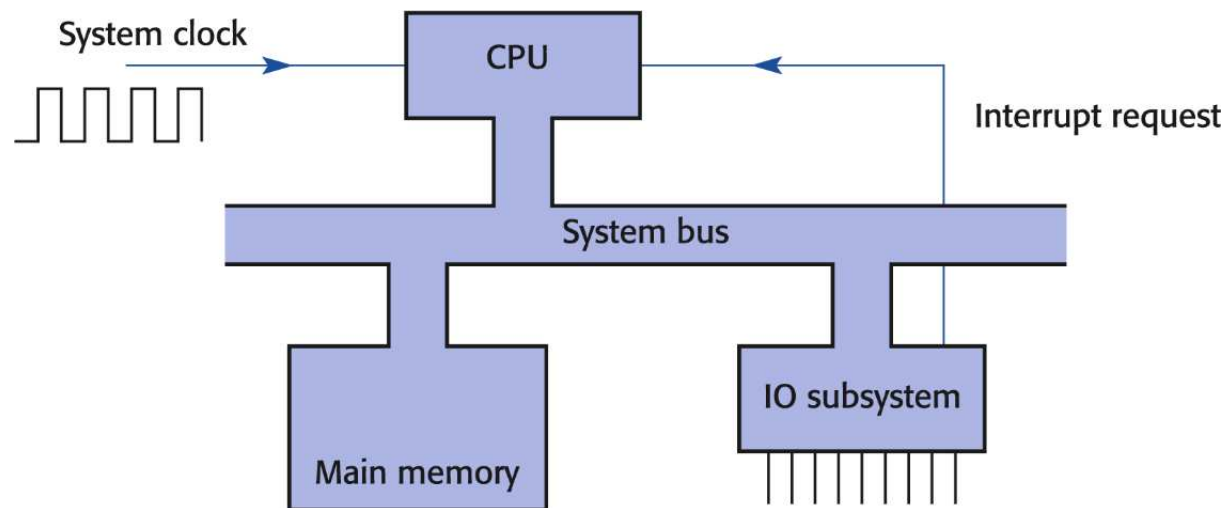
The functional units are interconnected to enable data transport (e.g., write CPU register **data** content to a certain **address** in memory)

- The unit \longleftrightarrow unit interconnections are referred to as a **busses**
 - **Bus**: a bundle of conductors (wires/tracks) layed out on the motherboard (**bus width** = number of conductors in bundle)
- Data and addresses are communicated on separate **data** and **address busses** (history: e.g., Intel 8086 used shared data/address bus)
- **Examples**

CPU	Address bus width	Data bus width
Intel Pentium IV	32	64
Intel Itanium	44	64

Bus

- Bus types:
 - ① **data bus** (move data)
 - ② **address bus** (select address in memory, select port in I/O unit)
 - ③ **control bus** (synchronize units, request action from unit, unit state)
- The busses provide an infrastructure that is **shared** by all units. There are *no* individual unit \longleftrightarrow unit (point-to-point) interconnections (exception: interrupt request lines)



Bus vs. point-to-point

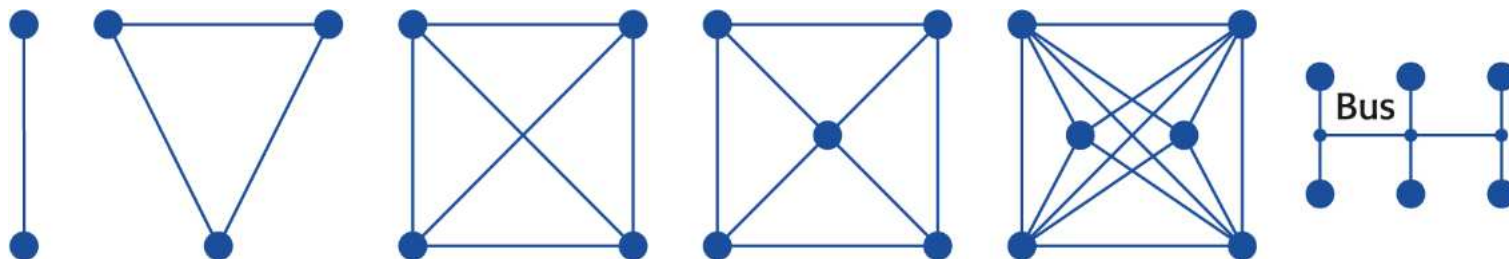
- Point-to-point (p2p) interconnections may seem a viable alternative to the system bus concept

A fully wired motherboard with n units, however, would need $\frac{n \times (n - 1)}{2}$ interconnections. Unmanageable.

Example: $n = 30$ units, data bus width 32, control bus width 6:

$$\# \text{wires} = \underbrace{\frac{30 \times 29}{2}}_{\text{interconnections}} \times \underbrace{(32 + 6)}_{\text{wires}} = 16530$$

Example: p2p interconnections for $n = 2 \dots 6$ units (and bus with 6 units):



The bus bottleneck

- **Note:** the bus architecture comes with a serious disadvantage:



An electronic bus can transfer only **one item at a time** (e.g., one data word, one address)

- The bus transmission speed thus poses a limit on the overall performance of the system (this phenomenon is known as the **bus bottleneck**)
 - Bus transmission speed is limited by physical characteristics (the capacitance and inductance of the bus wires)
 - Note that advances in CPU speed do not help here: the faster the CPU operates internally, the slower the bus transfers appear to be
 - While the CPU waits for a bus transfer to complete, the CPU is **stalled**

The CPU fetch–execute cycle

2.2 The CPU fetch–execute cycle

- A (non-parallel) CPU executes **one instruction at a time** (sequentially); a sequence of instructions is also referred to as a **program**
- On today's computers, data as well as programs reside in memory (this is known as the **von Neumann architecture**), the CPU thus iterates the following steps over and over:
 - ① Compute the **address of the next instruction**, advise main memory unit to read the data (= instruction) at the selected address (address/control bus)
 - ② **Fetch the instruction** into the CPU (data bus)
 - ③ **Decode and execute** the instruction inside the CPU

Instruction decoding

- In a von Neumann computer, instructions are like any other data items stored in memory³

Example (decoding a Pentium instruction):

- Memory contents:

10111000 00000000 00000001

- This encodes the Pentium instruction MOV AX, 0x100:

$$\underbrace{10111000}_{\text{MOV AX}} \underbrace{000000000000000001}_{0x100}$$

(**move** value 0x100 into CPU register AX)

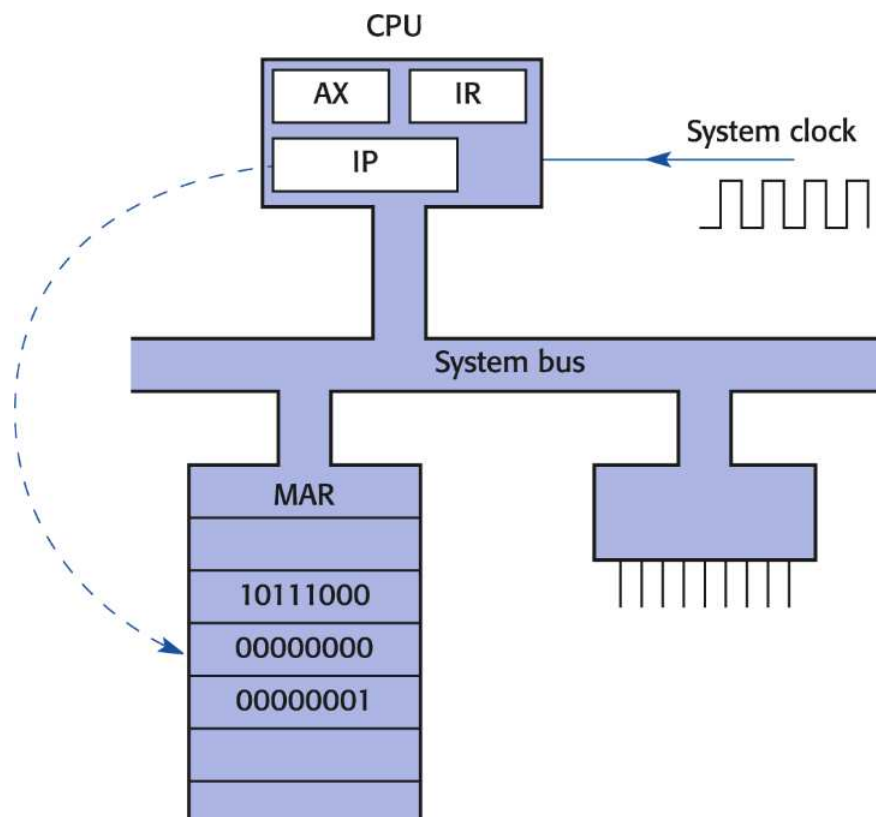
- One equivalent instruction in C:

i = 256;

³**Question:** This is potential hazard and beneficial opportunity at the same time. How?

Fetch–execute cycle in detail

- The following units interact during the fetch–execute cycle:



- CPU registers:
 - accumulator AX, instruction pointer IP, instruction register IR
- Main memory:
 - memory address register MAR, main memory stores the instruction(s)

Fetch phase (identical for all instructions)

- ① The address in the CPU register IP is transmitted via the address bus to the memory unit's MAR:

$$IP \rightarrow MAR$$

- ② IP is incremented to point at the next program instruction, ready for the next cycle (IP points at individual **bytes** [= 8 bits] in memory):

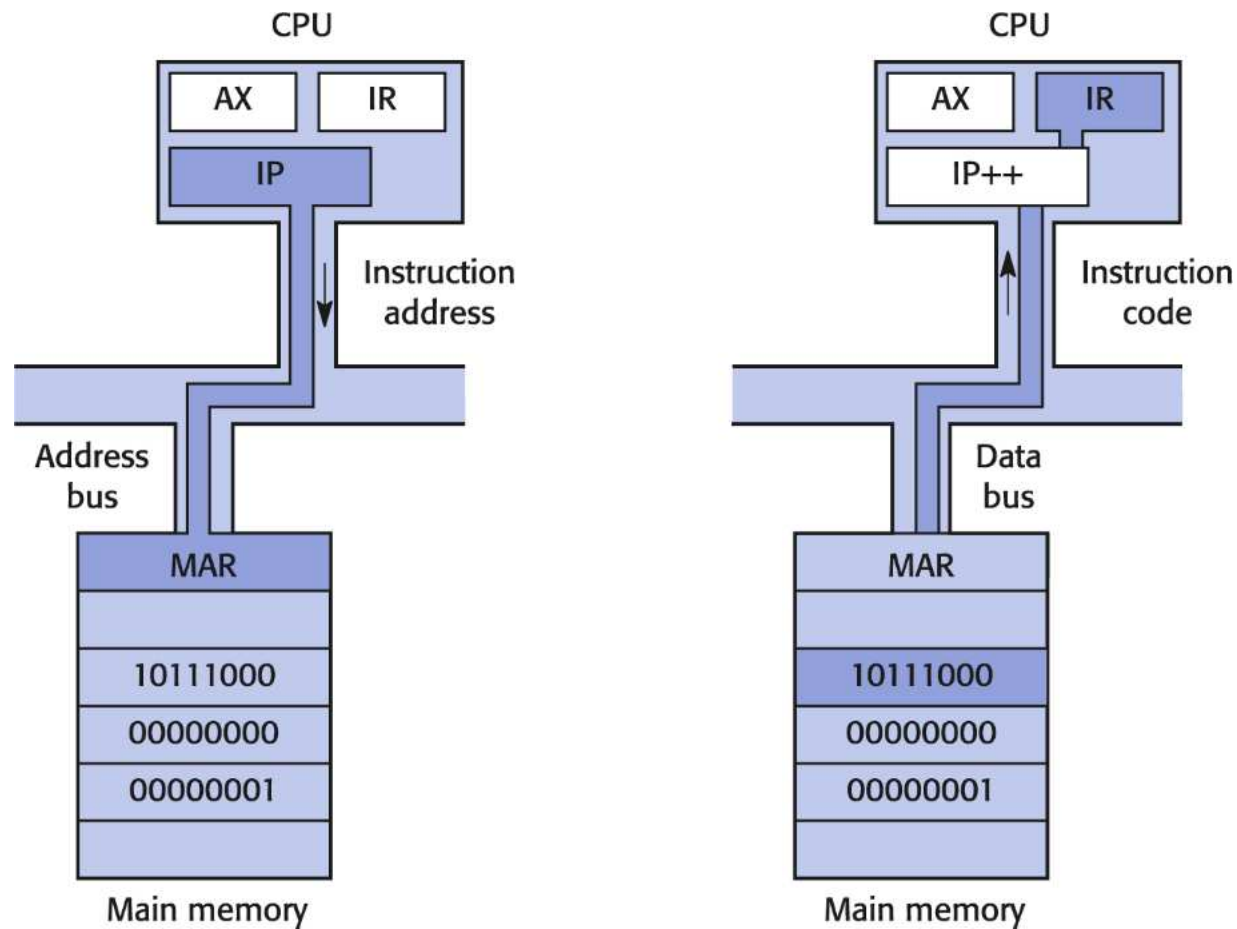
$$IP ++$$

- ③ Memory selects addressed location and copies *its contents* onto the data bus; CPU loads received data into IR:

$$(MAR) \rightarrow IR$$

- ④ CPU starts decoding the instruction in IR

Fetch phase (identical for all instructions)



- **Note:**

- only the the instruction code (here: 1 byte) is copied into IR
- The actions on the control bus are not shown here

Execution phase (varies with instruction type)

32

This illustrates the execution phase of the `MOV AX, c` (*move constant value c into register AX*) instruction:

- ① IP is transferred via address bus into MAR of main memory

$IP \rightarrow MAR$

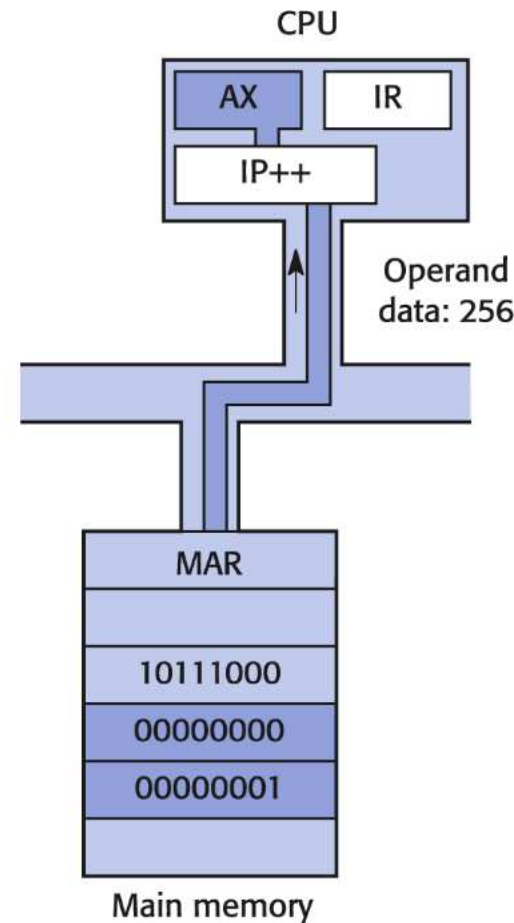
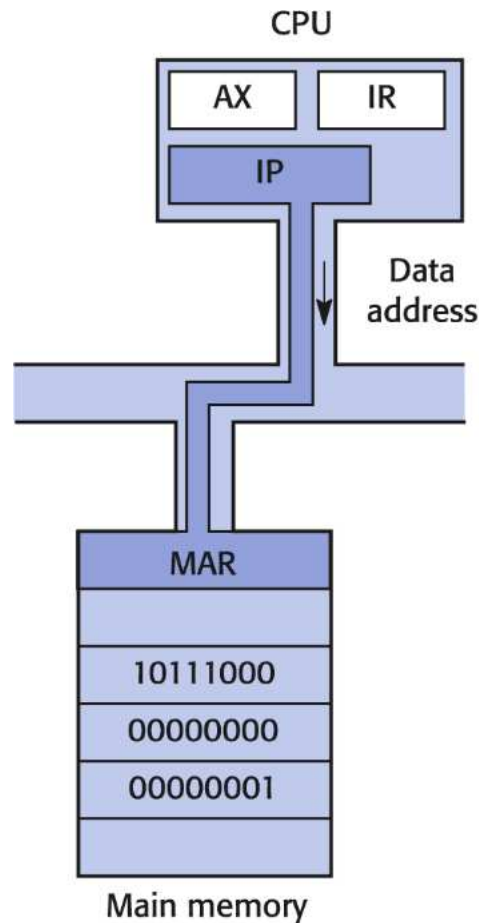
- ② IP is incremented

$IP ++; IP ++;$

- ③ The contents of the selected memory location is copied into CPU register AX (Pentium: actually EAX)

$(MAR) \rightarrow AX$


Execution phase (varies with instruction type)



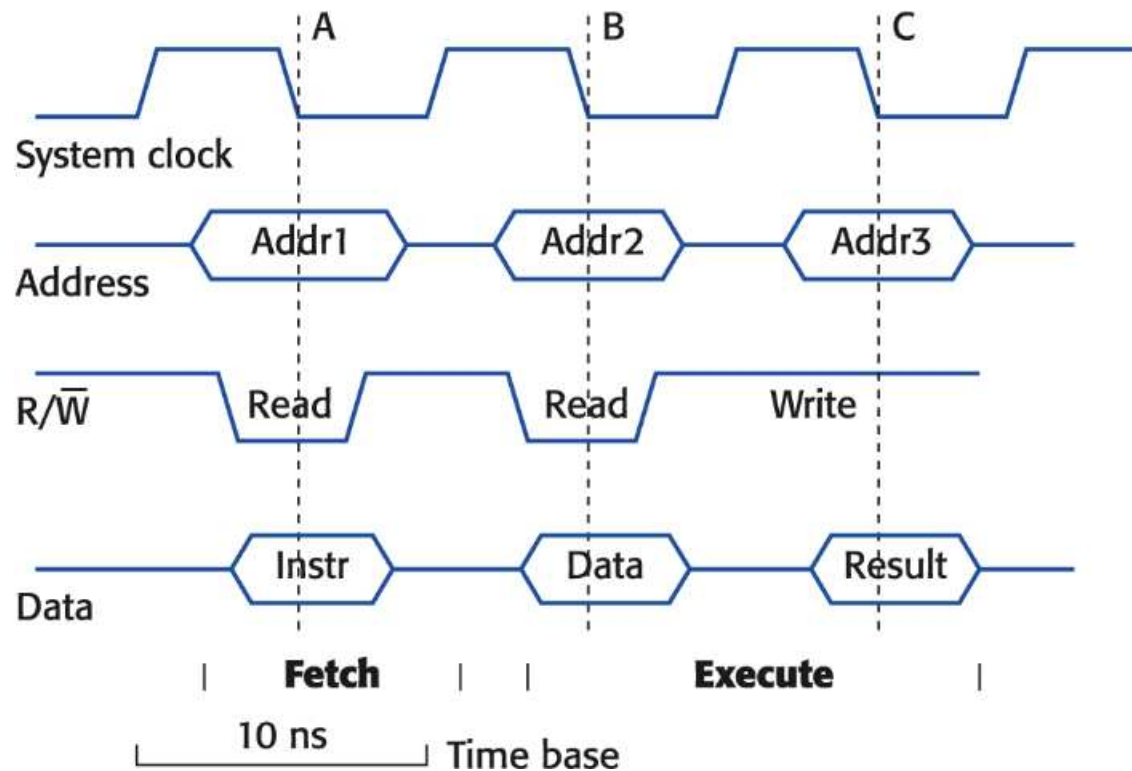
• Note:

- This time 16 bits (2 bytes) are copied via the data bus (the byte width of the data to be copied is encoded in the MOV instruction)
- Accordingly, IP is incremented twice to point after the data

Synchronous system bus

- One of the control bus lines is the **system clock**, whose signal is generated by a high-frequency crystal oscillator
- Normally, the CPU starts an operation by sending signals to other units (e.g., main memory) over the control bus
- The triggered unit may then take control of the system bus for a while (e.g., main memory transmits requested data over the data bus)
- Obviously, bus usage must follow a precise timing pattern (otherwise two units may want to use the bus simultaneously )
- If this timing pattern is completely locked into the system clock, the system is operating with a **synchronous bus**

Synchronous bus trace

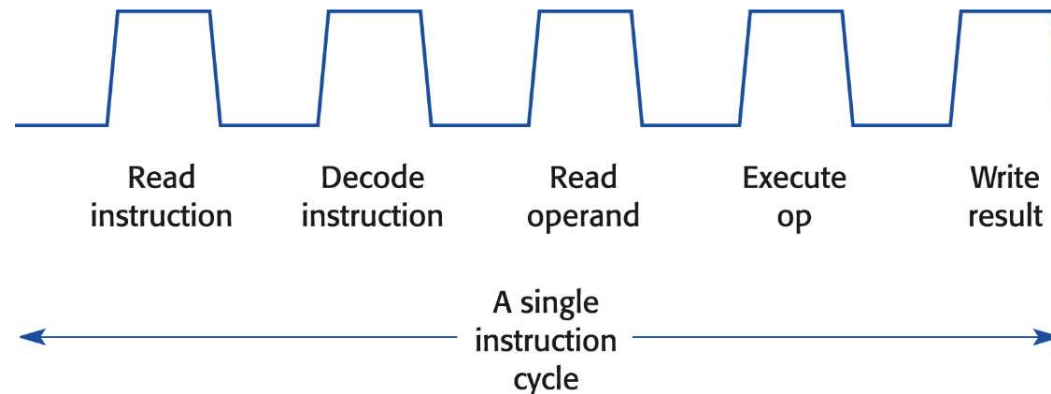


• Notes:

- Clock cycle 10 ns: **all** units run at 100 MHz, three cycles shown here
- The R/ \bar{W} (read/write) signal is part of the control bus
- The bus contents are considered valid at times A, B, C (falling edges)
- The trace shows a *memory-to-memory* move instruction
- The CPU can only run as fast as the **slowest** unit on the bus

Instruction cycle timing

- As we have seen, a single instruction cycle (or fetch-execute cycle) itself is comprised of a number of distinct phases, the **micro-cycles**:



- Example:** A current motherboard with a 100 MHz clock runs with a micro-cycle period of 10 ns (period = $\frac{1}{\text{frequency}}$):

$$t_{\text{micro}} = \frac{1}{100 \times 10^6} \text{ s} = \frac{1}{100} \mu\text{s} = \frac{1000}{100} \text{ ns} = 10 \text{ ns}$$

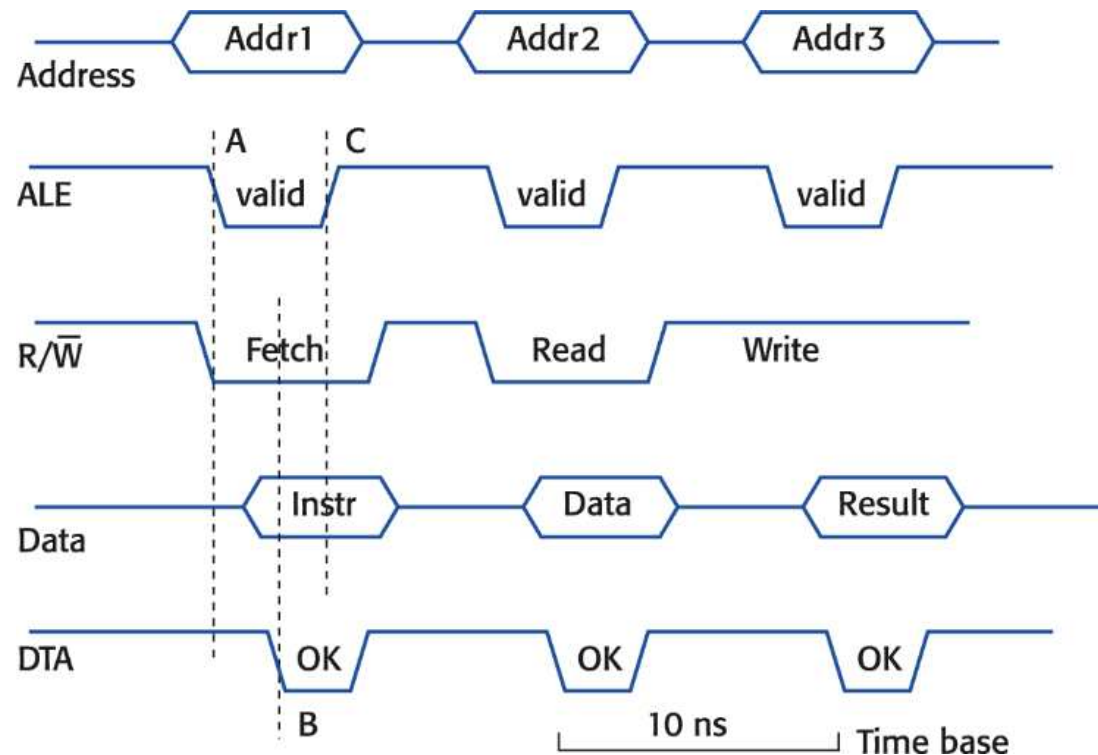
- Internally, Pentiums quadruple the motherboard system clock; the resulting micro-cycle period of 2.5 ns exceeds main memory speed (memory access time ≈ 50 ns)



Asynchronous system bus

- The operation of an **asynchronous bus** is a bit more complicated but also more flexible
- The system uses two new control bus lines to **adapt the CPU to the speed of the unit** it is actively communicating with:
 - ALE (Address Latch Enable): controlled by CPU, set when valid address signal is on address bus
 - DTA (Data Transfer Acknowledge): controlled by unit, indicates that data has been written/read successfully
- Together, the ALE and DTA signals are said to implement a **handshake protocol**
(this is a widely used “trick” in computer science in general, *e.g.*, in networking)

Asynchronous bus trace

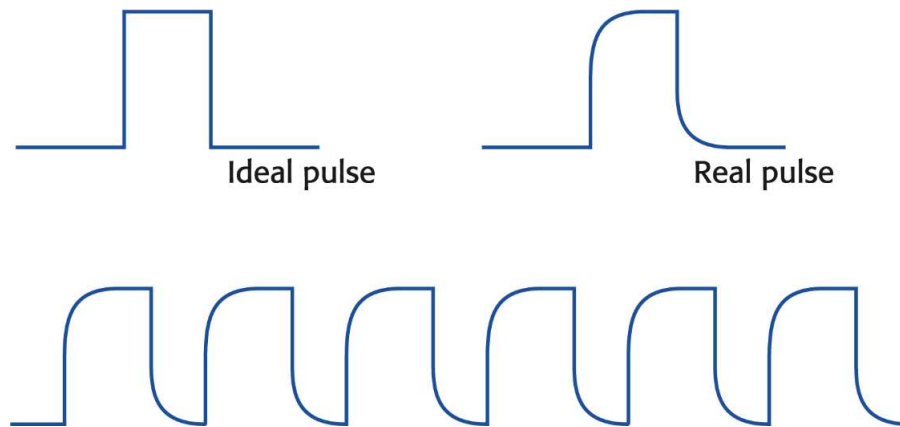


• Notes:

- ① CPU puts address value on address bus, then activates ALE (time A)
 - ② Memory unit copies requested data (here: instruction code) onto data bus, then activates DTA (time B)
 - ③ CPU recognizes DTA and copies instruction code into IR, then deactivates ALE (time C)
- In the time span A–B, the CPU is **stalled** (waiting for DTA)

High clock frequencies

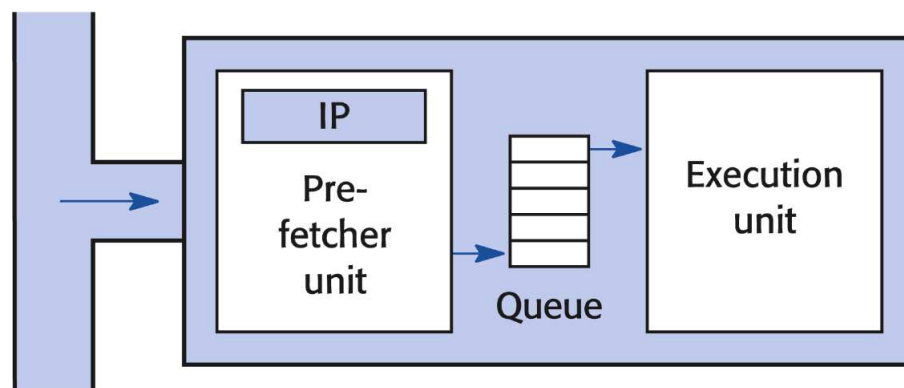
- Electrical effects like the resistance and capacitance of the bus wiring **pose limits** on the achievable system clock frequency
- The edges of a real pulse are “round”, pulse tails may not interfere with the rising of the next pulse:



- Power supply of current motherboards: 3.5 V
 - Voltage $< 0.8\text{ V} \Rightarrow$ logic 0
 - Voltage $> 2.5\text{ V} \Rightarrow$ logic 1
- Other limits: CPU heat problems (\Rightarrow cooling equipment, *e.g.*, fans)

Pre-fetching by the CPU

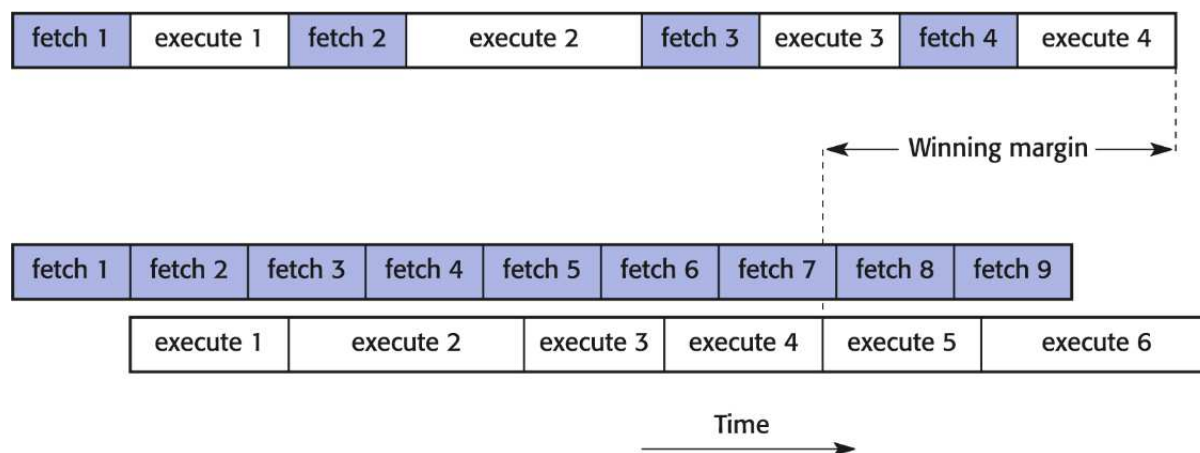
- The faster CPUs are clocked, the slower main memory accesses appear to be (the CPU spends more micro-cycles waiting for the DTA signal)
- There are, however, phases during the fetch–execute cycle which do *not* involve any bus activity:
 - ① decode instruction
 - ② execute instruction
- **Pre-fetching CPUs** use these periods of “bus silence” to *read instructions ahead* from main memory and store them for later use:



- Instructions are stored in a **queue**, a tabular structure that is written to/read from using two independent circulating pointers

Parallelism in the CPU

- In pre-prefetching CPUs, the fetch and execute phases for two instructions may overlap:

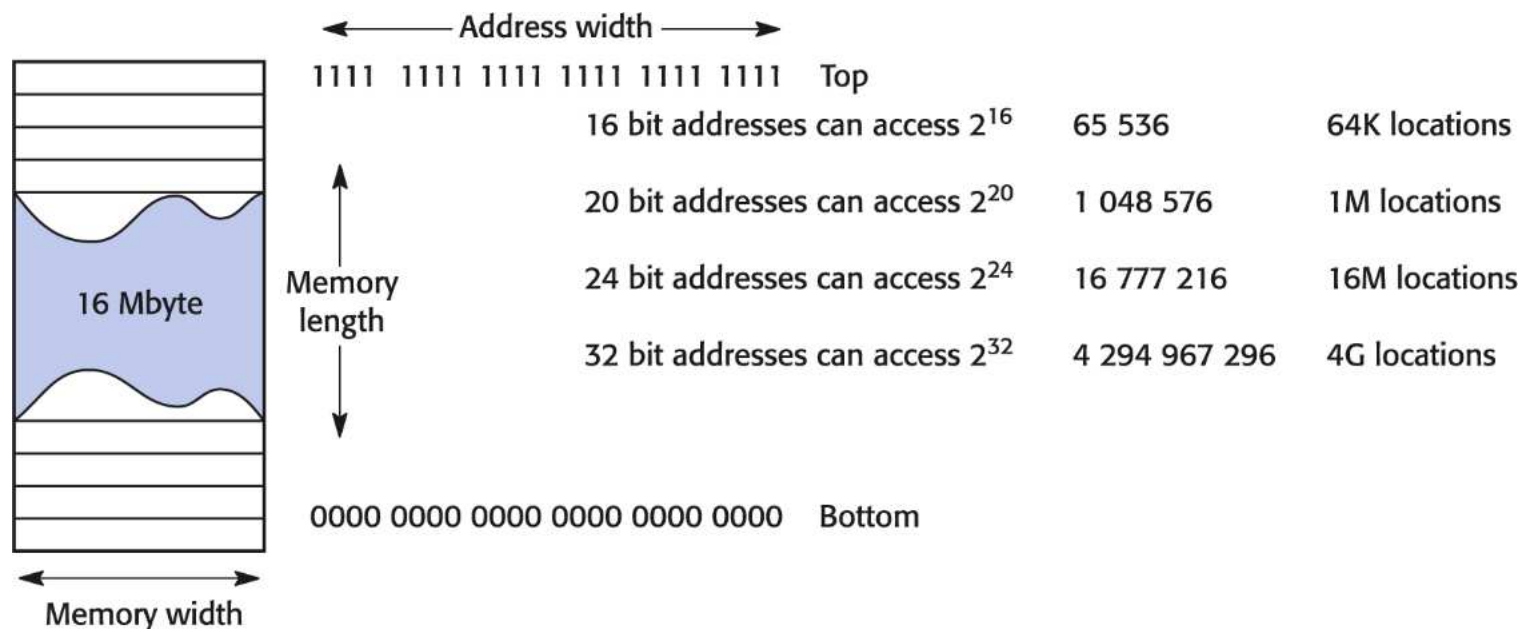


- Whenever the program contains a conditional jump instruction, the pre-fetcher unit has to decide to follow *one of the two possible* branches
 - The contents of the pre-fetch queue may be useless if the CPU follows the other branch
 - Modern CPUs try to “guess” which branch will be followed (observations from previous outcomes of the same conditional branch)

Memory size and address width

2.3 Memory size and address width

- Note that the address bus width determines how much memory a CPU can use: each address bus wire represents a single bit of the address value
- Similarly, CPU address registers (e.g., IP) are of a specific **register width** (= number of bits in the register):⁴



⁴1 K = 1024, 1 M = 1024 K, 1 G = 1024 M

Bottom and top memory addresses

- **Example** (memory addressing with address width of 8 bits):

Address bits	Address (decimal)	
00000000	0	bottom
00000001	1	
00000010	2	
00000011	3	
00000100	4	
00000101	5	
⋮	⋮	
10000000	128	
10000001	129	
⋮	⋮	
11111110	254	
11111111	255	top

- With an address width of n bits, a CPU can access 2^n memory locations
- The bottom address is 0, the top address is $2^n - 1 = \underbrace{1111 \cdots 111}_n_2$

Hexadecimal address notation

- Especially for large address widths (e.g., 32 bits or soon 64 bits), handling memory addresses in binary notation is error-prone (at least for humans)
- It is common practice to “pack” 4 binary digits (bits) into a single hexadecimal digit
 - **Example** (a 32 bit address in binary and hexadecimal notation):

$\underbrace{1010}_A$
 $\underbrace{0011}_3$
 $\underbrace{1101}_D$
 $\underbrace{0010}_2$
 $\underbrace{0101}_5$
 $\underbrace{1111}_F$
 $\underbrace{1110}_E$
 $\underbrace{0001}_1$

Dec	Bin	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dec	Bin	Hex
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Hexadecimal address notation

