

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

EE-2003

Computer Organization & Assembly Language

Engr. M Kariz Kamal

Lecturer, Department of Cyber Security

Office: 1st Floor New building

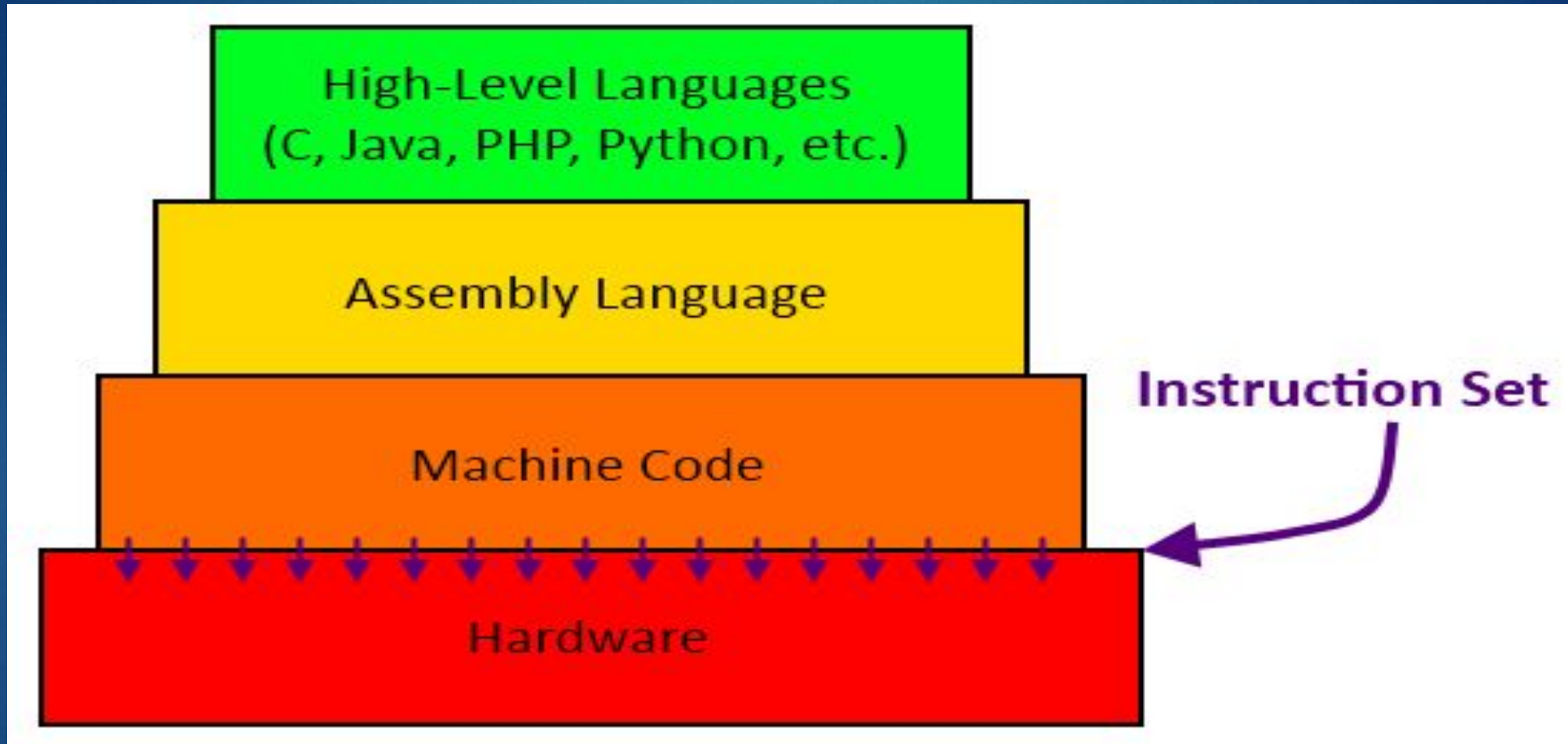
FAST NUCES (Karachi)

Kariz.kamal@nu.edu.pk

Computer Organization

- ▶ **Computer architecture** is concerned with the way hardware components to operate and the way they are connected together to form the computer system.
- ▶ **Computer organization** is concerned with the structure and behavior of a computer system as seen by the user.
- ▶ The computer organization is concerned with the **structure** and **behavior** of digital computers.
- ▶ Working of Internal Parts of Computer. i.e. RAM, CACHE etc.
- ▶ Computer organization describes how a task is done by the computer.

Hierarchy of Languages



High Level Languages

- ▶ A high-level language (HLL) is a programming language such as C, JAVA, or PYTHON.
- ▶ It enables a programmer to write programs that are more or less independent of a particular type of computer. (**Machine Independent**)
- ▶ Such languages are considered high-level because they are closer to human languages and further from machine languages.
- ▶ A single statement in C++ expands into multiple assembly language or machine instructions. (*one-to-many* relationship)

Assembly Language

- ▶ An assembly language is a low-level **programming language** designed for a specific type of **processor**. (*Machine Specific*)
- ▶ Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL.
- ▶ Each assembly language instruction corresponds to a single machine-language instruction. (*one-to-one relationship*)

Translating Languages

English: D is assigned the sum of A times B plus 10.



High-Level Language: $D = A * B + 10$



A statement in a high-level language is translated typically into several machine-level instructions

Intel Assembly Language:

```
mov  eax, A
mul   B
add   eax, 10
mov   D, eax
```



Intel Machine Language:

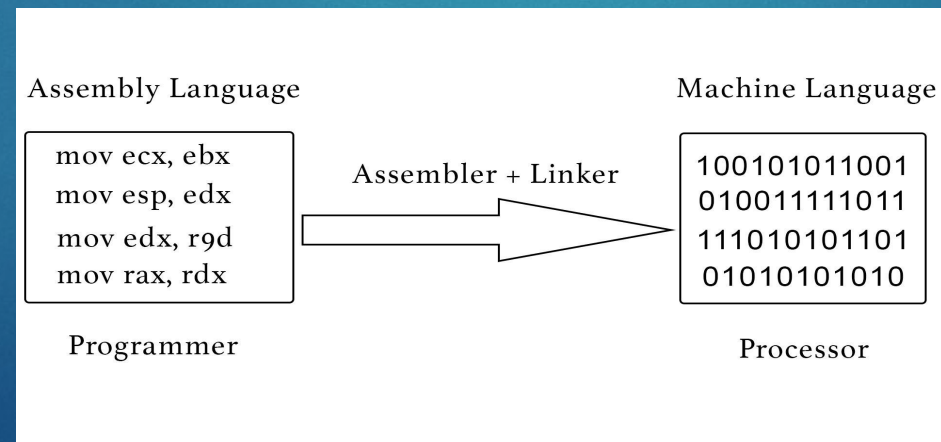
```
A1 00404000
F7 25 00404004
83 C0 0A
A3 00404008
```

Compiler and Assembler

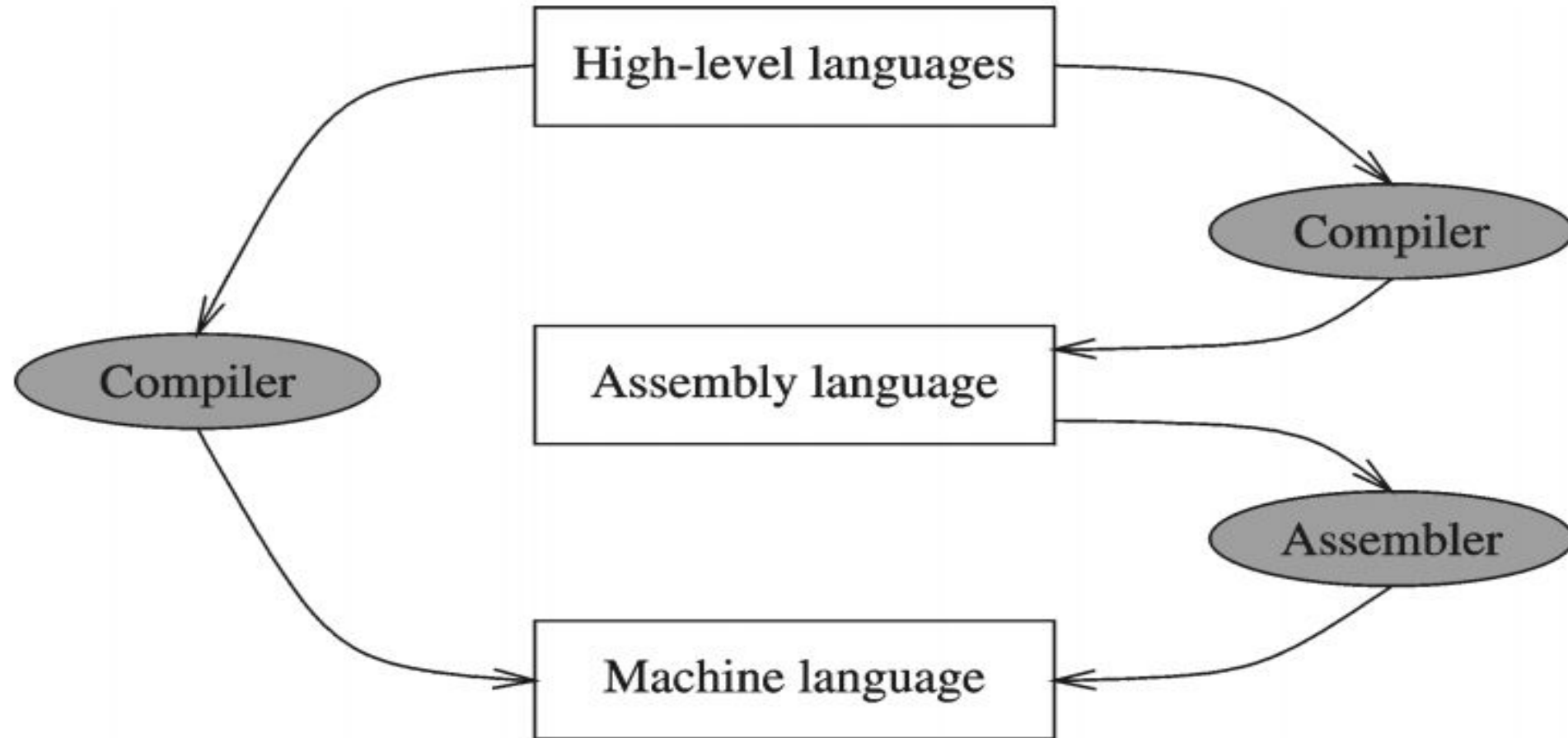
- Compilers translate high-level programs to machine code.(Directly/Indirectly).



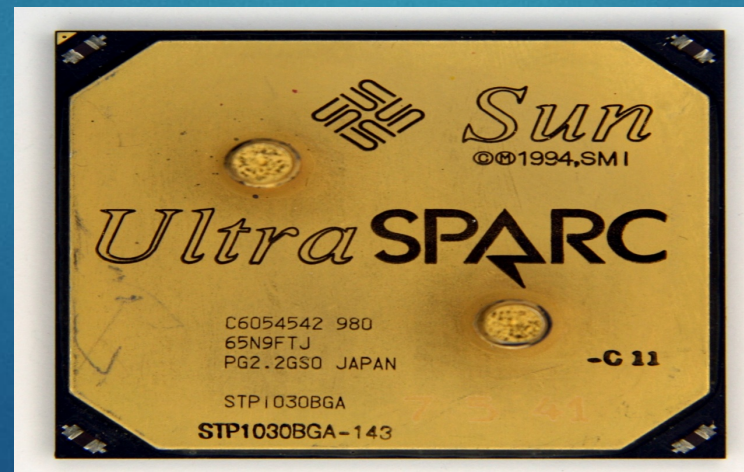
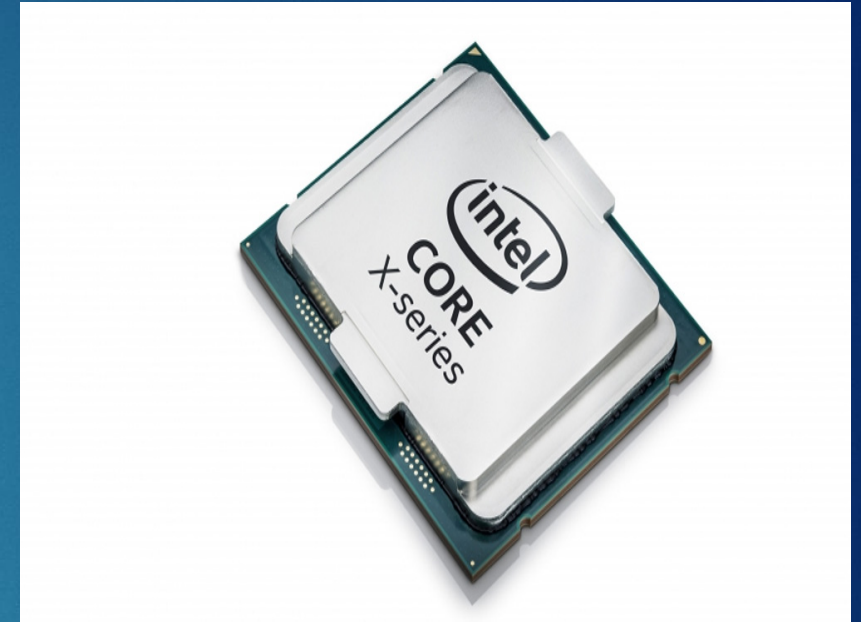
- Assemblers translate assembly language to machine code.



Compiler and Assembler



Is Assembly Language Portable?



Is Assembly Language Portable?

- ▶ A language whose source programs can be compiled and run on a wide variety of computer systems is said to be portable.
- ▶ A C++ program, for example, should compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system.
- ▶ Assembly language is ***not portable*** because it is designed for a specific processor family.

Assembly Language for x86 Processors

- ▶ Intel stands for “**I**ntegrated **E**lectronics”
- ▶ AMD stands for “**A**dvanced **M**icro **D**evelopments”
- ▶ Assembly Language for x86 Processors focuses on programming microprocessors compatible with the **Intel IA-32** and **AMD x86** processors running under Microsoft Windows.

Advantages of High-Level Languages

- ▶ Program development is faster
 - High-level statements: fewer instructions to code
- ▶ Program maintenance is easier
 - For the same above reasons
- ▶ Programs are portable
 - Contain few machine-dependent details
 - Can be used with little or no modifications on different machines

Why Learn Assembly Language?

- ▶ **Complete control over a system's resources**

- gateway to optimization in speed, offering great efficiency and performance.

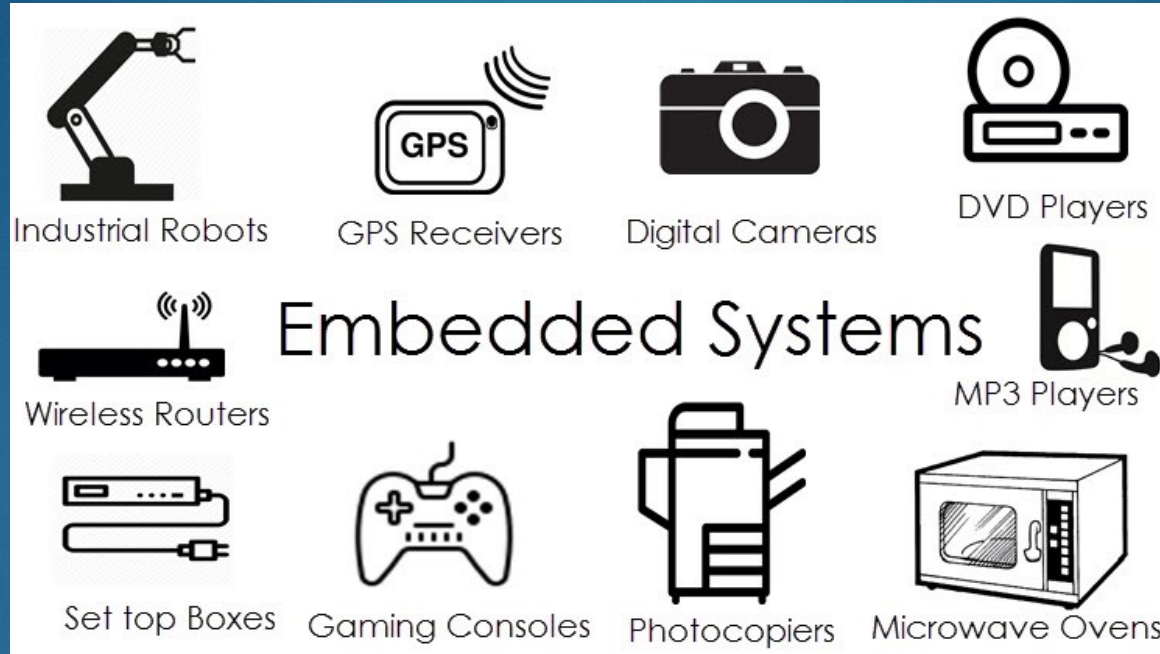
- ▶ **Assembly language is transparent**

- It has a small number of operations, but it is helpful in understanding the algorithms and other flow of controls. It makes the code less complex and easy debugging as well.

- ▶ For writing the compilers or device drivers, write some code in assembly language.

Applications of Assembly Language

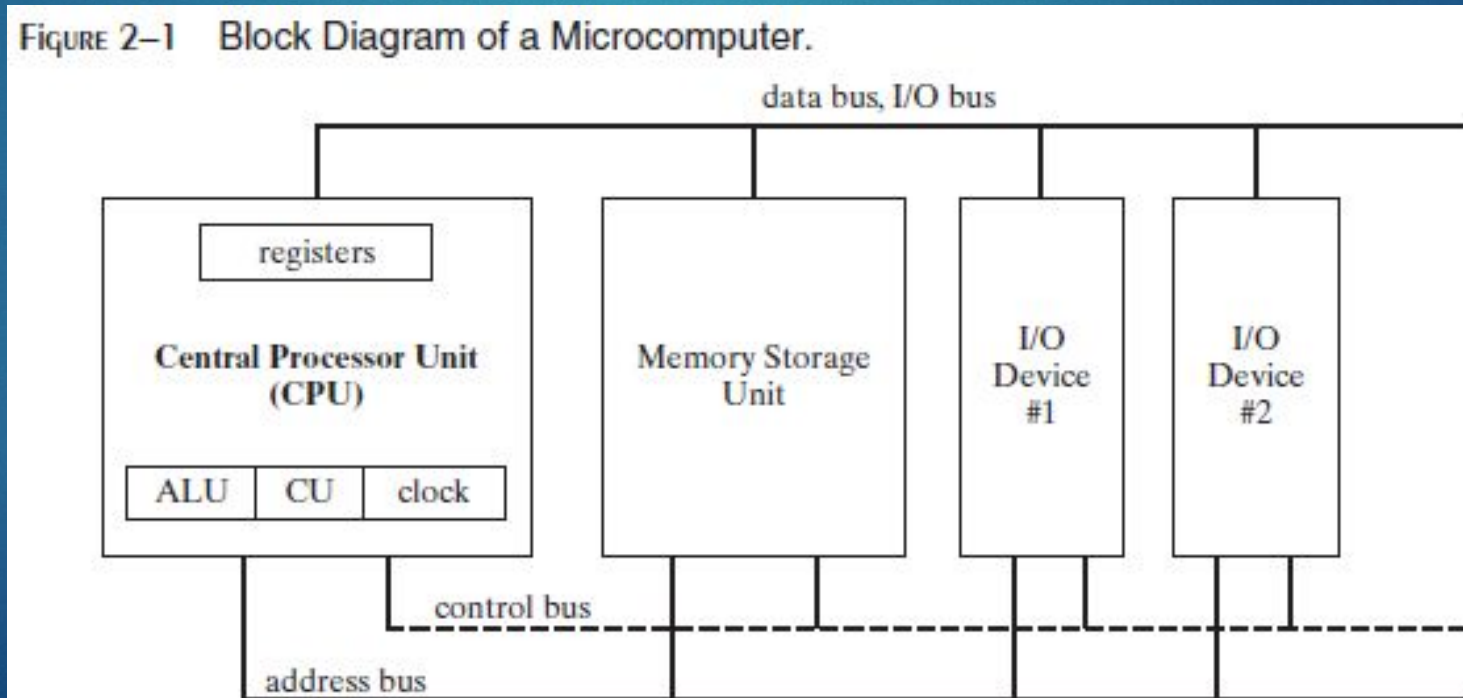
► Embedded System Programming



- **Game Programmers** □ programs to be highly optimized for both space and runtime speed.

Basic Microcomputer Design

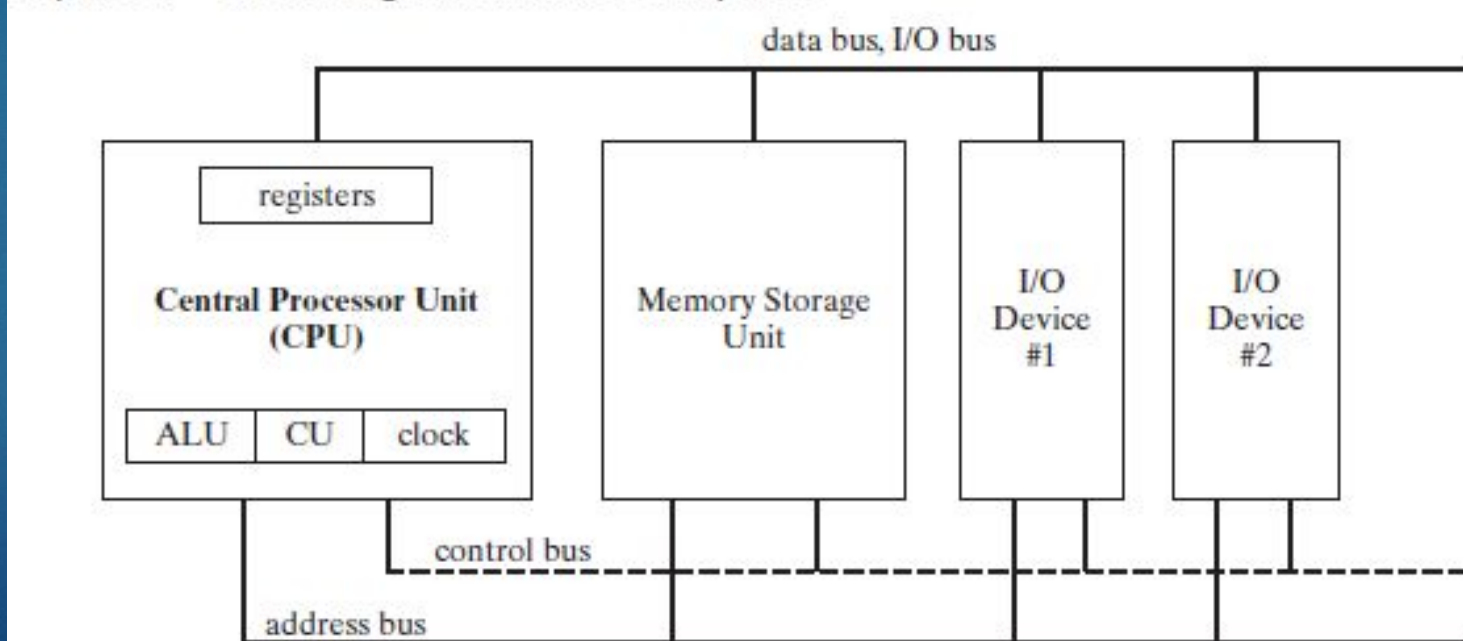
- ▶ The *central processor unit* (CPU), where calculations and logic operations take place, contains a limited number of storage locations named registers, a high-frequency clock, a control unit, and an arithmetic logic unit.
- ▶ The memory storage unit is where instructions and data are held while a computer program is running.



Basic Microcomputer Design

- ▶ The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.
- ▶ All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute.

FIGURE 2-1 Block Diagram of a Microcomputer.



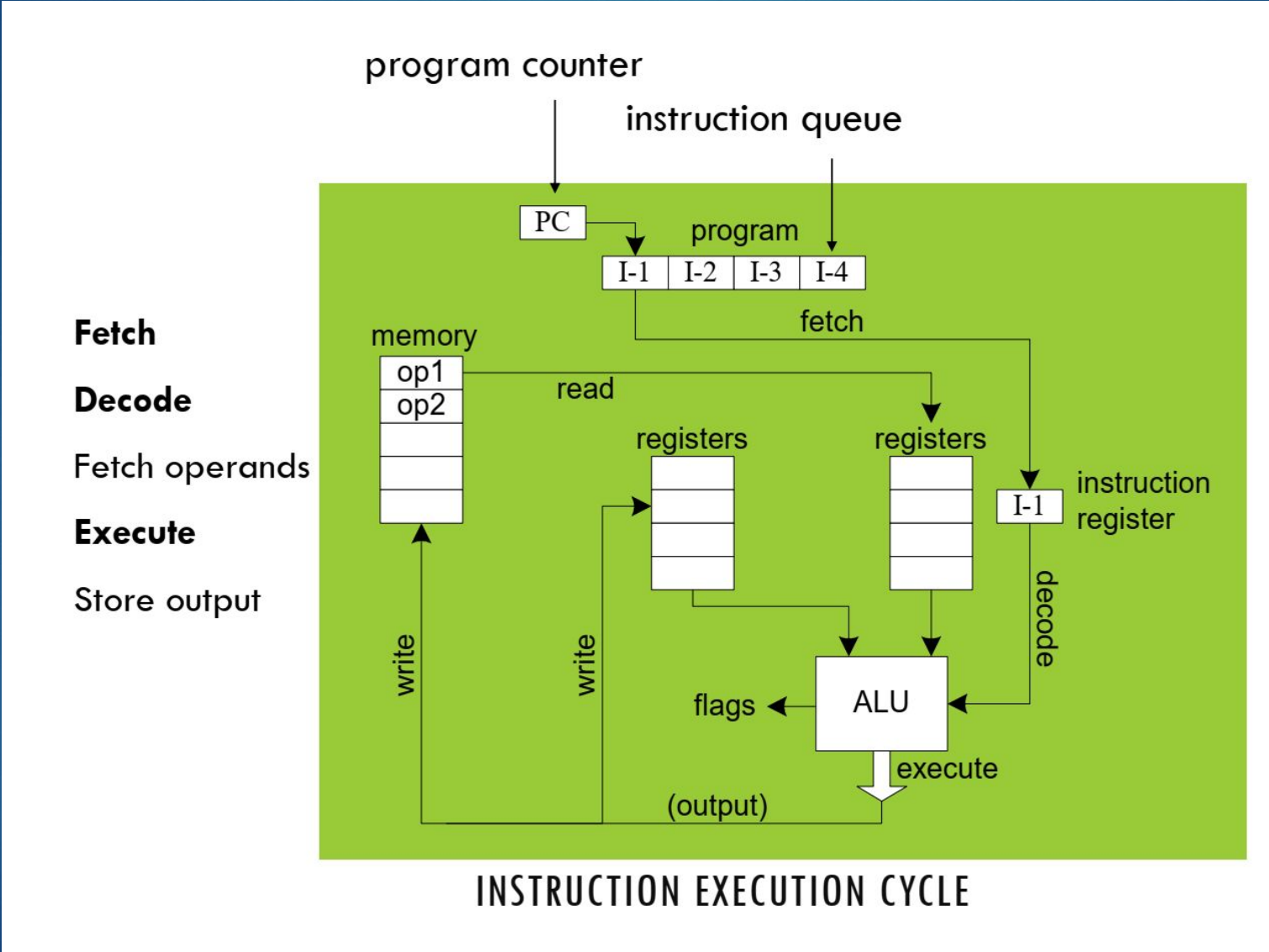
Buses

- ▶ A *bus* is a group of parallel wires that transfer data from one part of the computer to another.
- ▶ A computer system usually contains *four bus types*: **data, I/O, control, and address**.
- ▶ The data bus transfers instructions and data between the CPU and memory.
- ▶ The I/O bus transfers data between the CPU and the system input/output devices.
- ▶ The control bus uses binary signals to synchronize actions of all devices attached to the system bus.
- ▶ The *address bus* holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

Clock Cycles

- ▶ A machine instruction requires at least one clock cycle to execute.
 - ▶ Few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example).
- ▶ Instructions requiring memory access often have empty clock cycles called **wait states**.
 - ▶ Because of the differences in the speeds of the CPU, the system bus, and memory circuits.

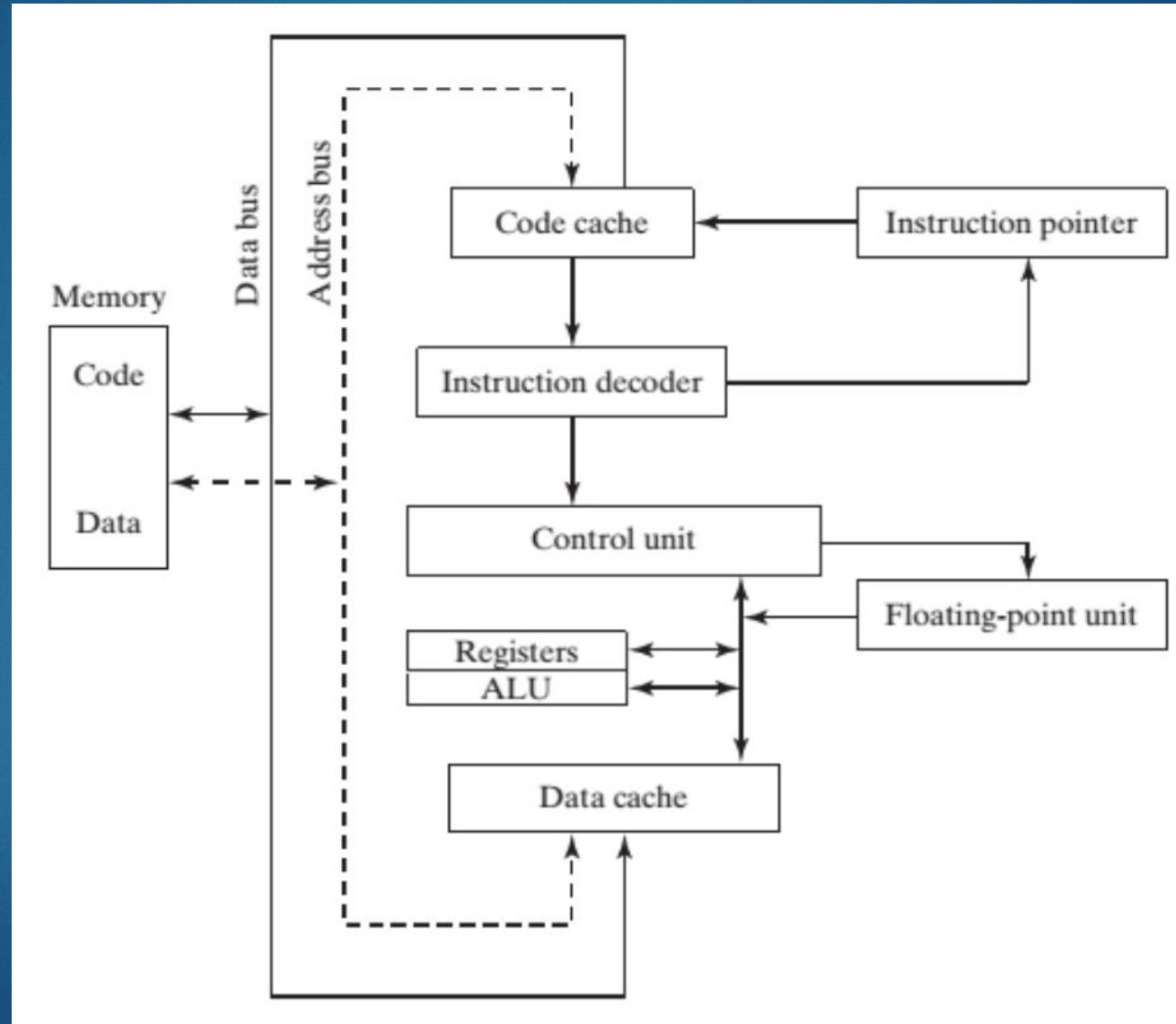
100



Instruction Execution Cycle

- ▶ The CPU go through a predefined sequence of steps to execute a machine instruction, called the instruction **execution cycle**.
- ▶ The **instruction pointer** (IP) register holds the address of the instruction we want to execute.
- ▶ Here are the steps to execute it:
 1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction queue*. It then increments the instruction pointer.
 2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern.
 - ▶ This bit pattern might reveal that the instruction has operands (input values).
 3. If operands are involved, the CPU **fetches the operands** from registers and memory.
 - ▶ Sometimes, this involves address calculations.
 4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step. It also *updates a few status flags*, such as Zero, Carry, and Overflow.
 5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its execution in the operand.

Instruction Execution Cycle



Instruction Execution Cycle

- ▶ An **operand** is a value that is either an input or an output to an operation.
- ▶ For example, the expression $Z = X + Y$ has **two input operands** (X and Y) and a single **output operand** (Z).
- ▶ In order to **read program instructions from memory**, an address is placed on the **address bus**.
- ▶ Next, the **memory controller** places the requested code on the data bus, making the code available inside the code cache.
- ▶ The **instruction pointer's** value determines which instruction will be executed next.
- ▶ The instruction is analyzed by the **instruction decoder**, causing the appropriate digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit.
- ▶ **Control bus** carries signals that use the system clock to coordinate the transfer of data between different CPU components.

Reading from Memory

- ▶ As a rule, computers read memory much more slowly than they access internal registers.
- ▶ Reading a single value from memory involves four separate steps:
 1. Place the address of the value you want to read on the address bus.
 2. Assert (change the value of) the processor's RD (read) pin.
 3. Wait one clock cycle for the memory chips to respond.
 4. Copy the data from the data bus into the destination operand.
- ▶ Each of these steps generally requires a single clock cycle.

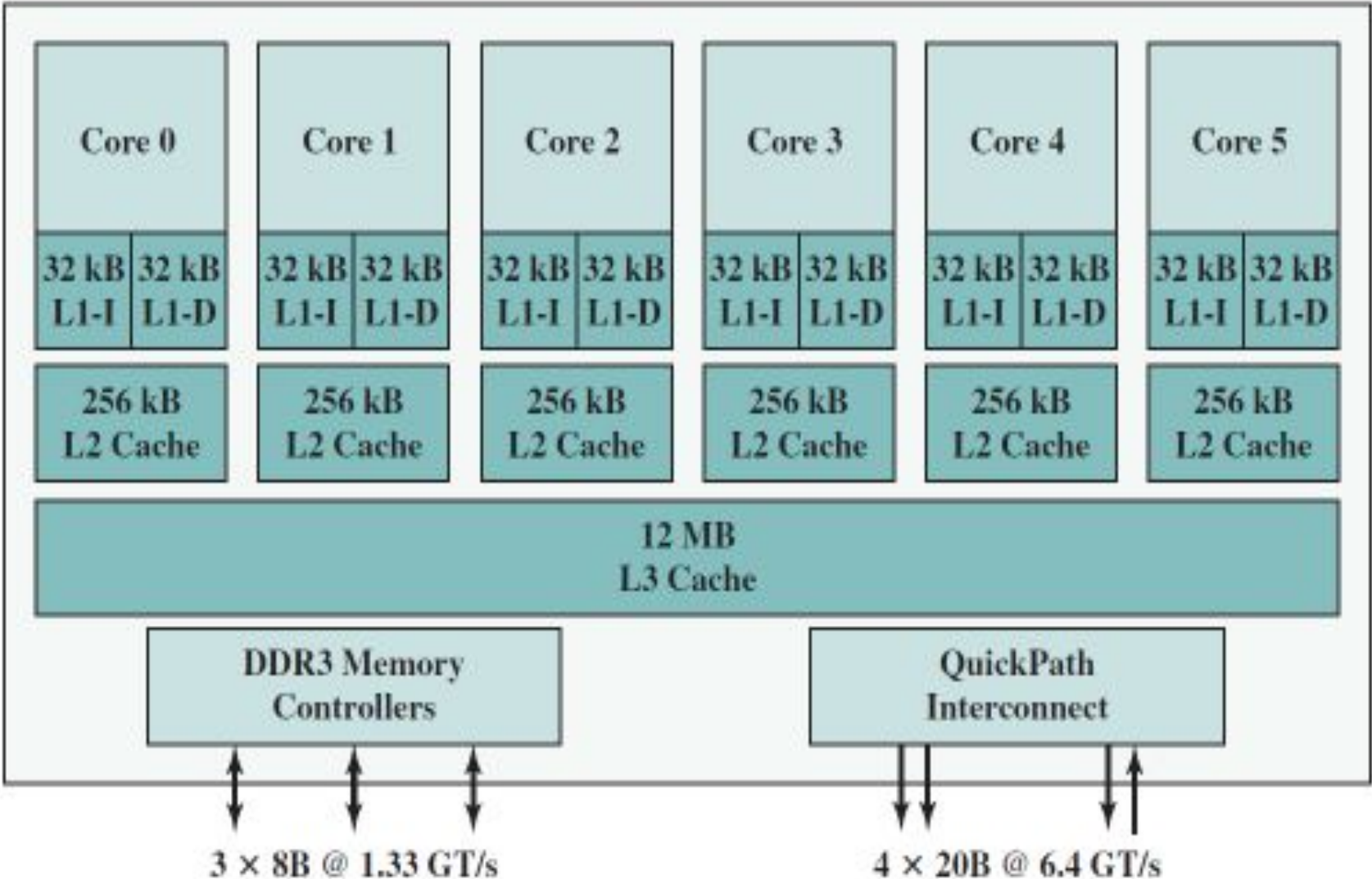
Cache

- ▶ CPU designers figured out that **computer memory creates a speed bottleneck** because most programs have to **access variables**.
- ▶ To reduce the amount of time spent in reading and writing memory the **most recently used instructions and data** are stored in high-speed memory called **cache**.
- ▶ The idea is that a program is more likely to want **to access the same memory and instructions repeatedly**, so cache keeps these values where they can be accessed quickly.
- ▶ When the CPU begins to execute a program, it **loads the next thousand instructions** (for example) into cache, on the assumption that these instructions will be needed in the near future.
- ▶ If there happens to be a **loop** in that block of code, the same instructions will be in cache.
- ▶ When the processor is able to find its data in cache memory, we call that a **cache hit**.
- ▶ On the other hand, if the CPU tries to find something in cache and it's not there, we call that a **cache miss**.

X86 family Cache types

- ▶ Cache memory for the x86 family comes in two types.
 1. Level-1 cache (or primary cache) is stored right on the CPU.
 2. Level-2 cache (or secondary cache) is a little bit slower, and attached to the CPU by a high-speed data bus.

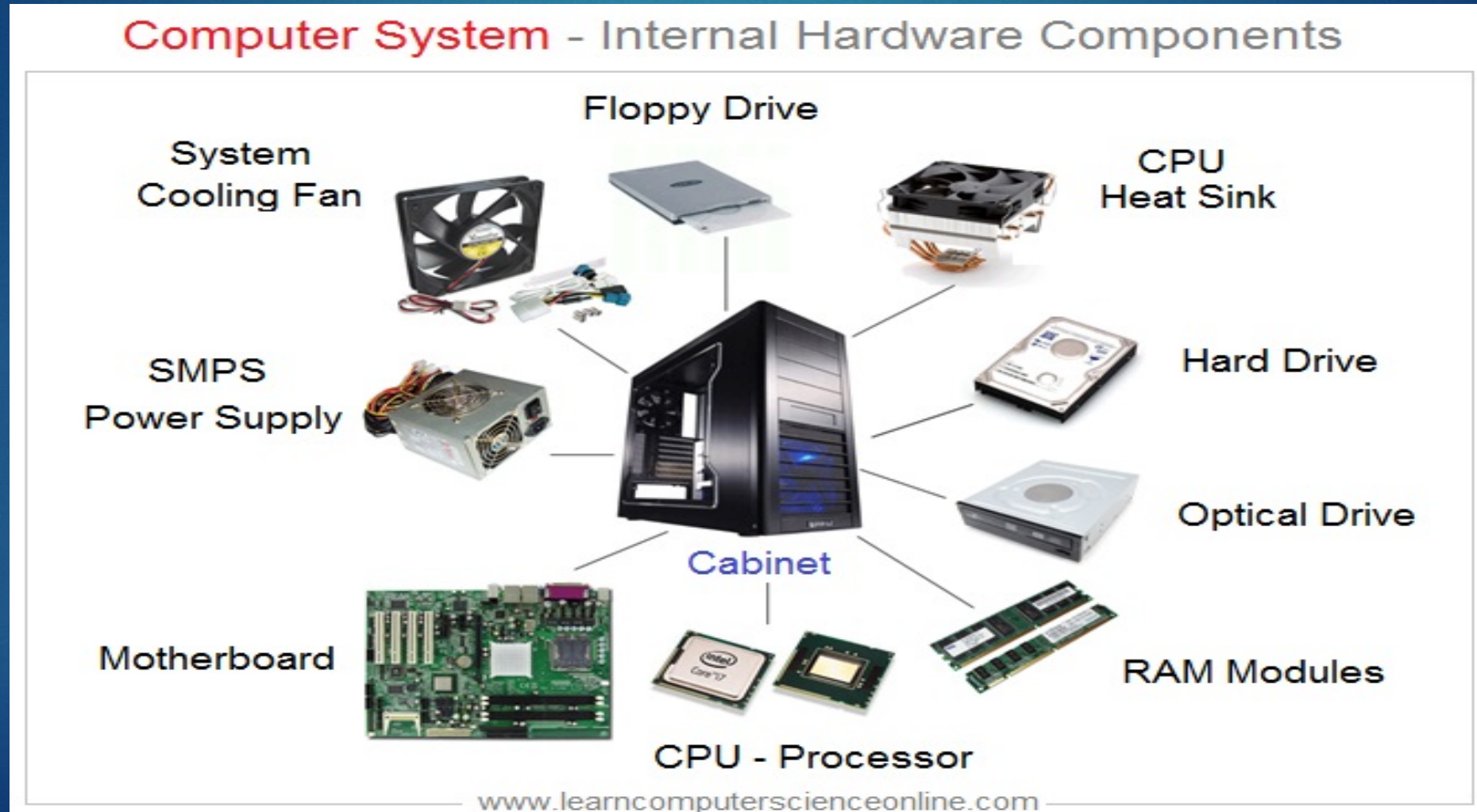
The block diagram of Intel Core i7-990X is shown below. Explain the working of processor and how cache coherency is maintained? Also compute the aggregate speed of DDR3 Memory Controller and Quick Path Interconnect?



Intel Core i7-990X Block Diagram

Virtual Machine Concept

- Computers are Build with Physical Parts i.e. Hardware



Virtual Machine Concept

- ▶ SOFTWARES □ Makes Computer easy to use.



Virtual Machine Concept

- ▶ **OPERATING SYSTEM SOFTWARES**
- ▶ **They are able to control Physical Components of Computer i.e. HARDWARE**

Virtual Machine Concept

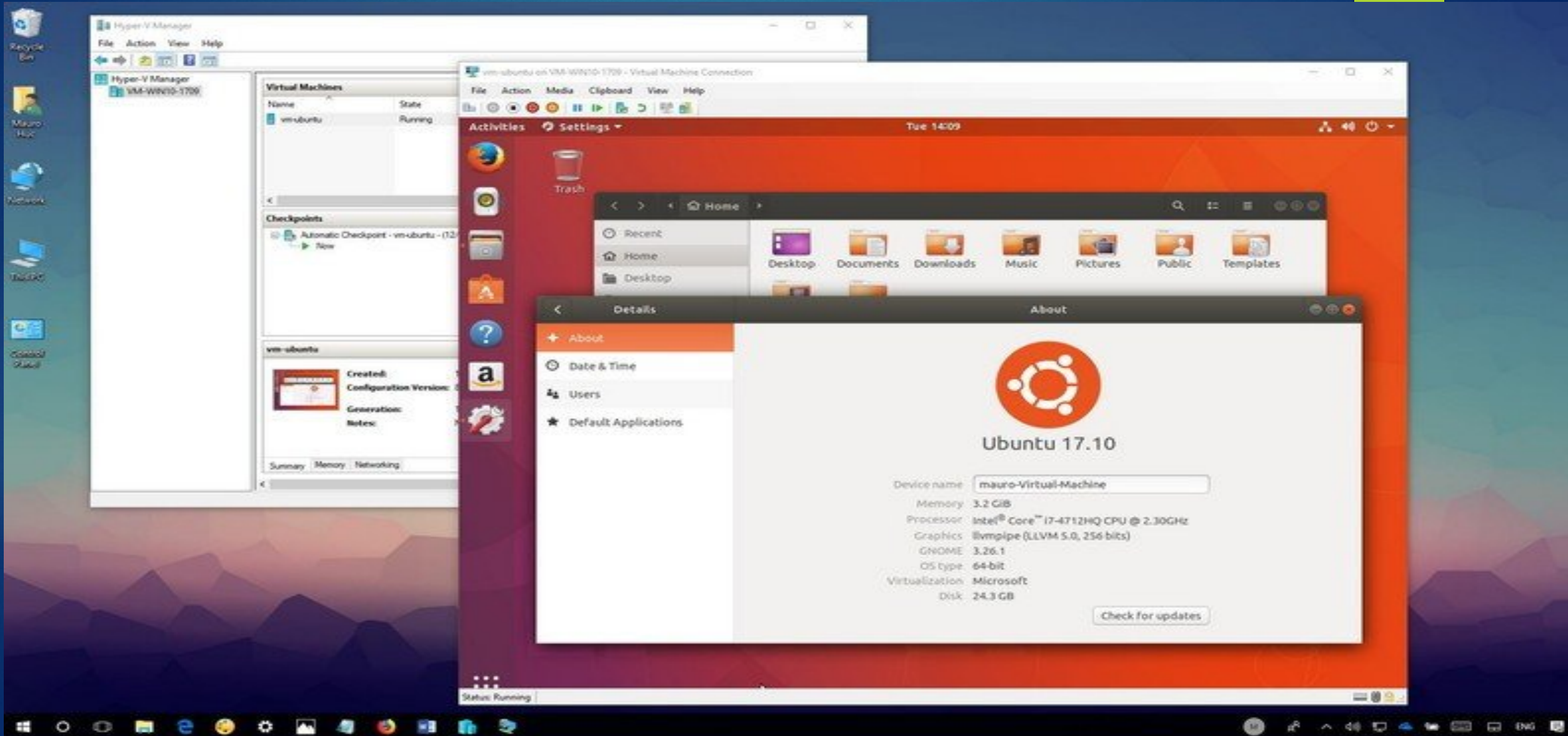
- ▶ **VIRTUAL MACHINE MANAGER (HYPERVISOR)**
- ▶ Software □ allows to run □ an Operating System □ Within another Operating System.



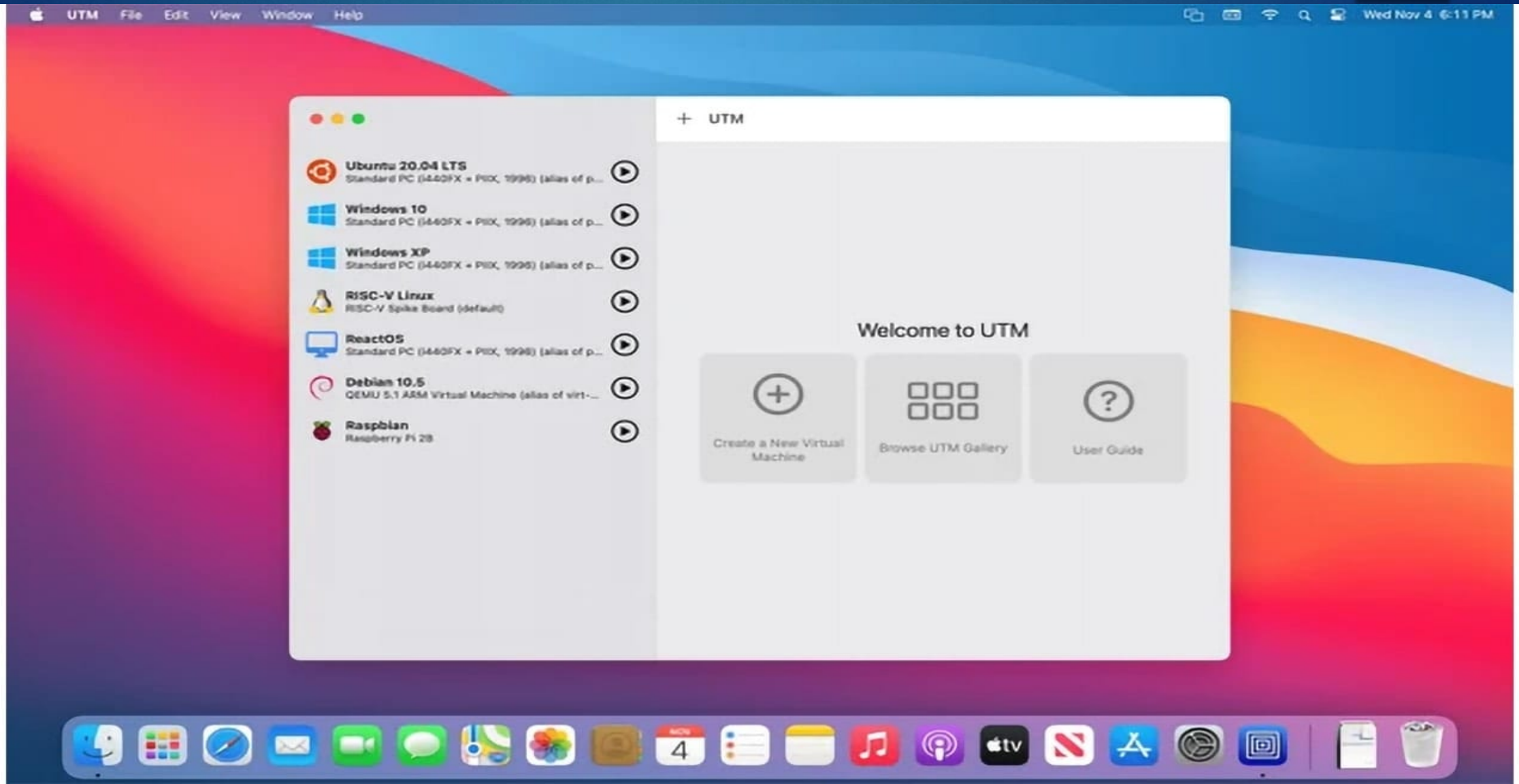
VirtualBox



Virtual Machine Concept



Virtual Machine Concept



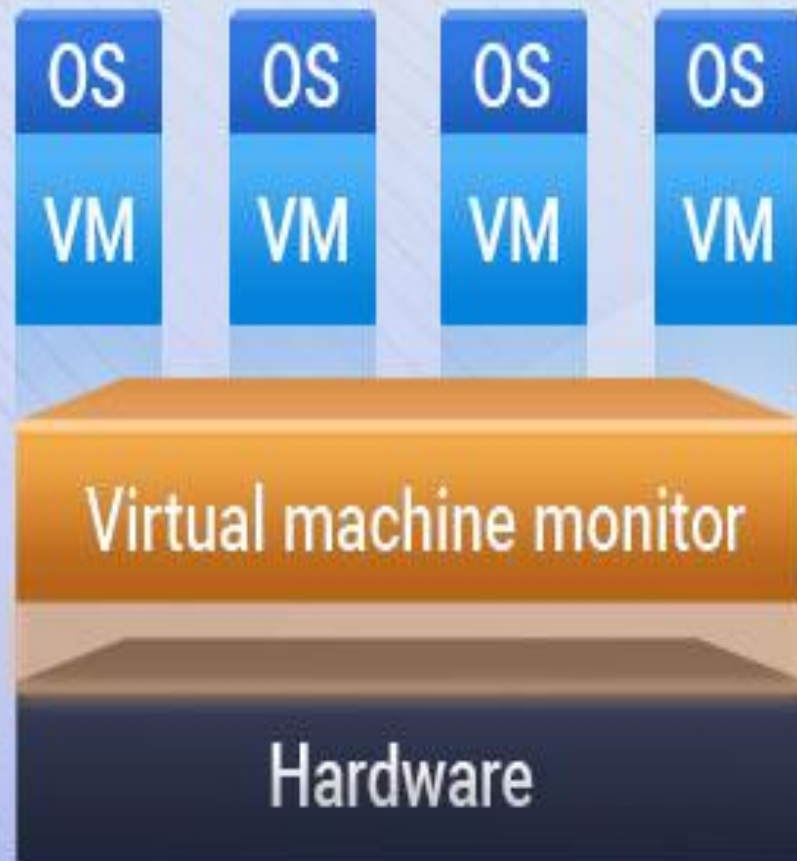
Virtual Machine Concept



Virtual Machine Concept

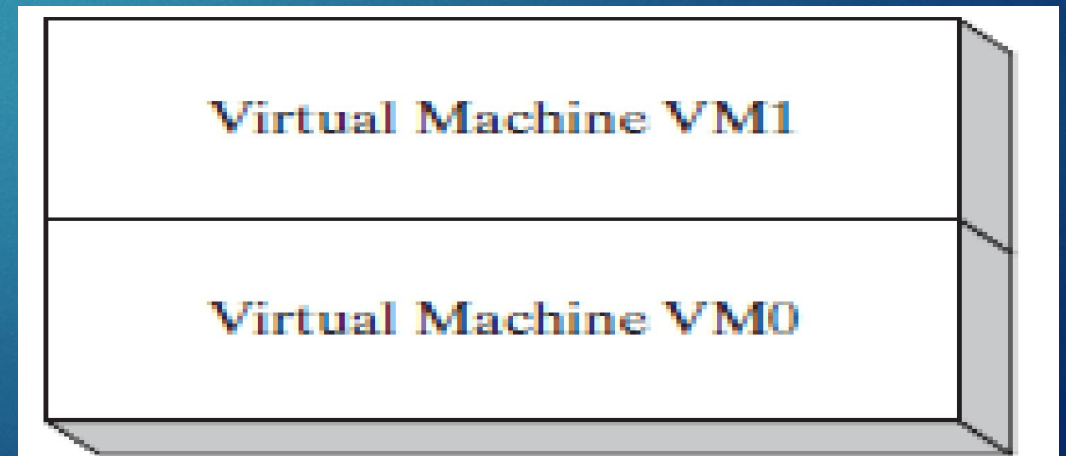
- ▶ A virtual machine (VM) is an image file managed by the hypervisor that exhibits the behavior of a separate computer, capable of performing tasks such as running applications and programs like a separate computer.
- ▶ In other words, a VM is a software application that performs most functions of a physical computer, actually behaving as a separate computer system.
- ▶ A virtual machine, usually known as a guest, is created within another computing environment referred as a "host." Multiple virtual machines can exist within a single host at one time.

Virtual Machine Concept



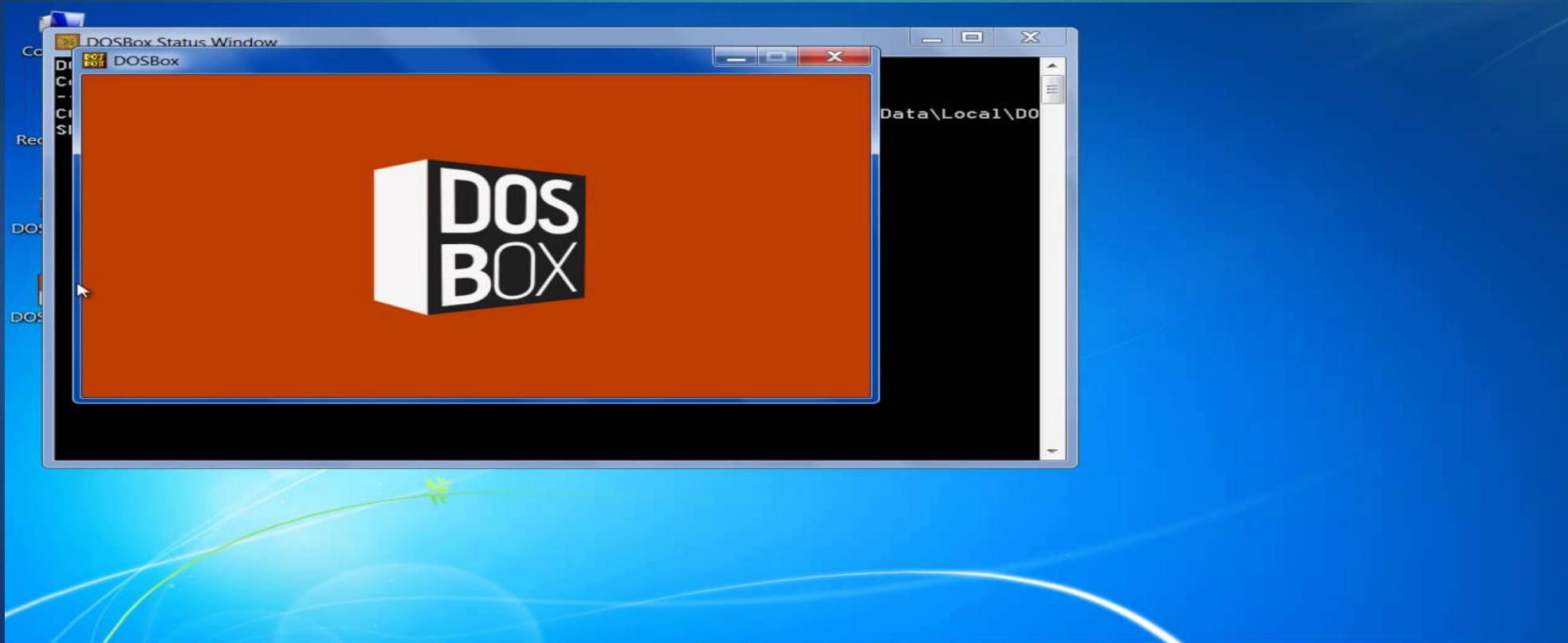
Virtual Machine Concept

- ▶ Programming Language analogy:
 - ▶ Each computer has a native machine language (language L0) that runs directly on its hardware
 - ▶ A more human-friendly language is usually constructed above machine language, called Language L1
- ▶ The virtual machine **VM1**, can execute commands written in language L1.
- ▶ The virtual machine **VM0** can execute commands written in language L0



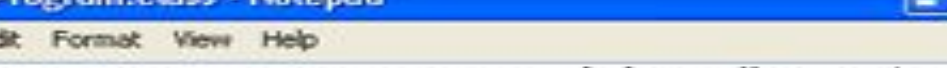
Virtual Machine Concept

- ▶ The virtual machine **VM0** can execute commands written in language L0



Virtual Machine Concept

- ▶ The **Java programming language** is based on the virtual machine concept.
- ▶ A program written in the Java language is translated by a Java compiler into *Java byte code* - a low-level language quickly executed at runtime by a program known as a *Java virtual machine (JVM)*.



MyProgram.class - Notepad

File Edit Format View Help

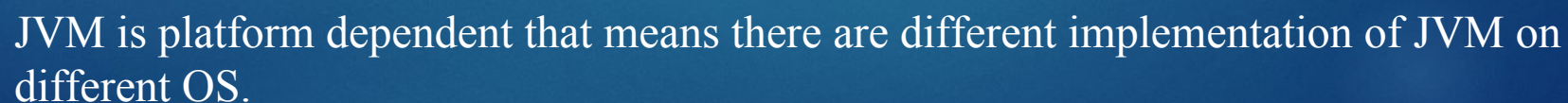
```

package java.lang;
import java.io.*;
import java.util.*;

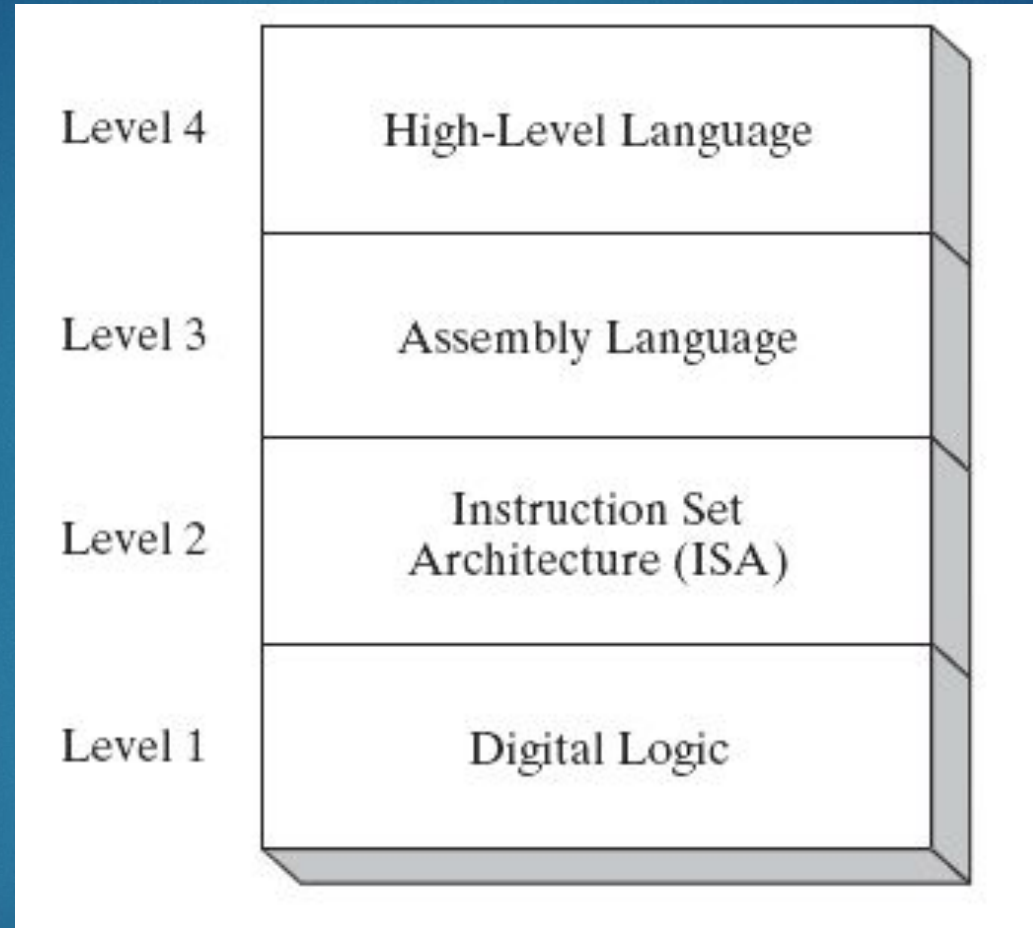
class MyProgram {
    public static void main (String[] args) {
        PrintStream out = new PrintStream(System.out);
        out.println("Hello, World.");
    }
}

```

After the compilation is successful, java compiler will generate an intermediate ".class" file that contains the bytecode.



Specific Machine Levels

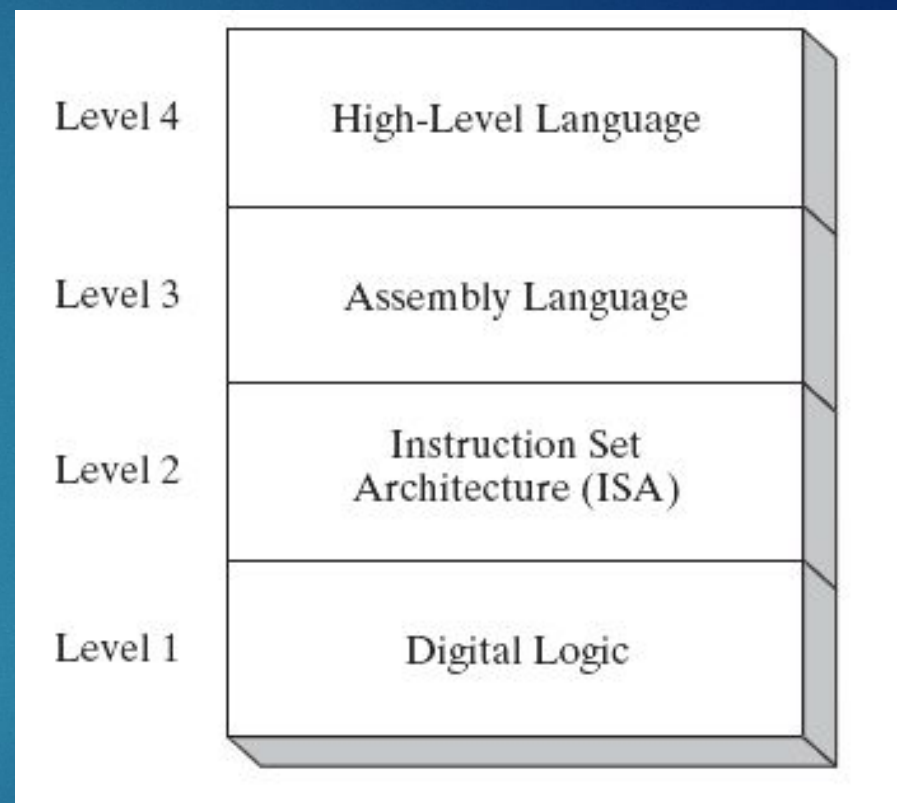


(descriptions of individual levels follow . . .)

High-Level Language

Level 4

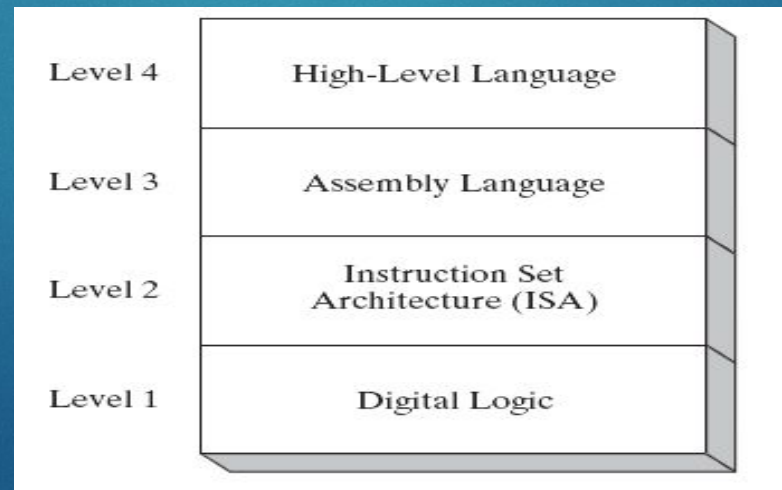
- ▶ Application-oriented languages
 - ▶ C++, Java, Pascal, Visual Basic . . .
- ▶ Programs compile into assembly language (Level 3)



Assembly Language

Level 3

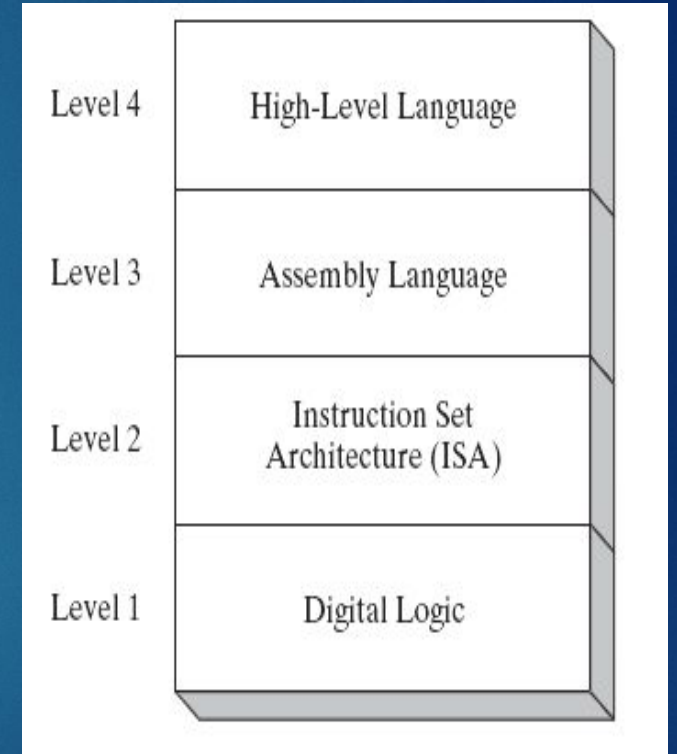
- ▶ Instruction mnemonics that have a one-to-one correspondence to machine language
- ▶ Programs are translated into Instruction Set Architecture Level - machine language (Level 2)
- ▶ The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.



Instruction Set Architecture (ISA)

Level 2

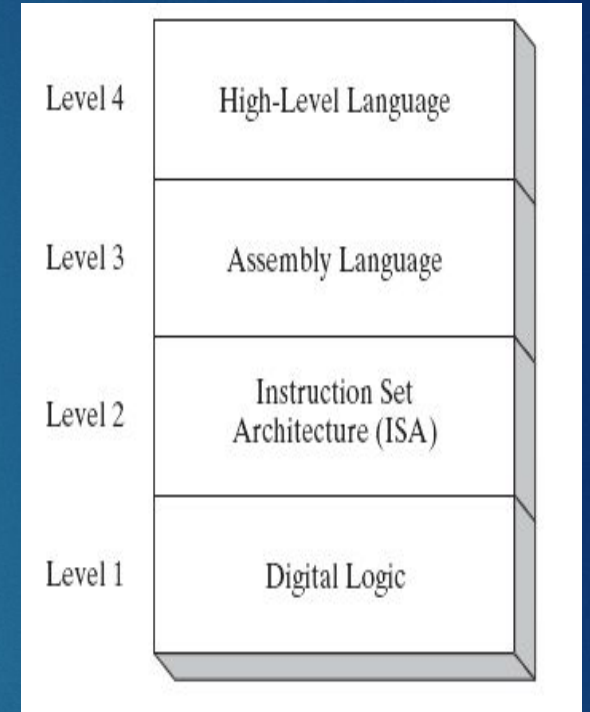
- ▶ Also known as conventional machine language.
- ▶ Executed by Level 1 (Digital Logic).



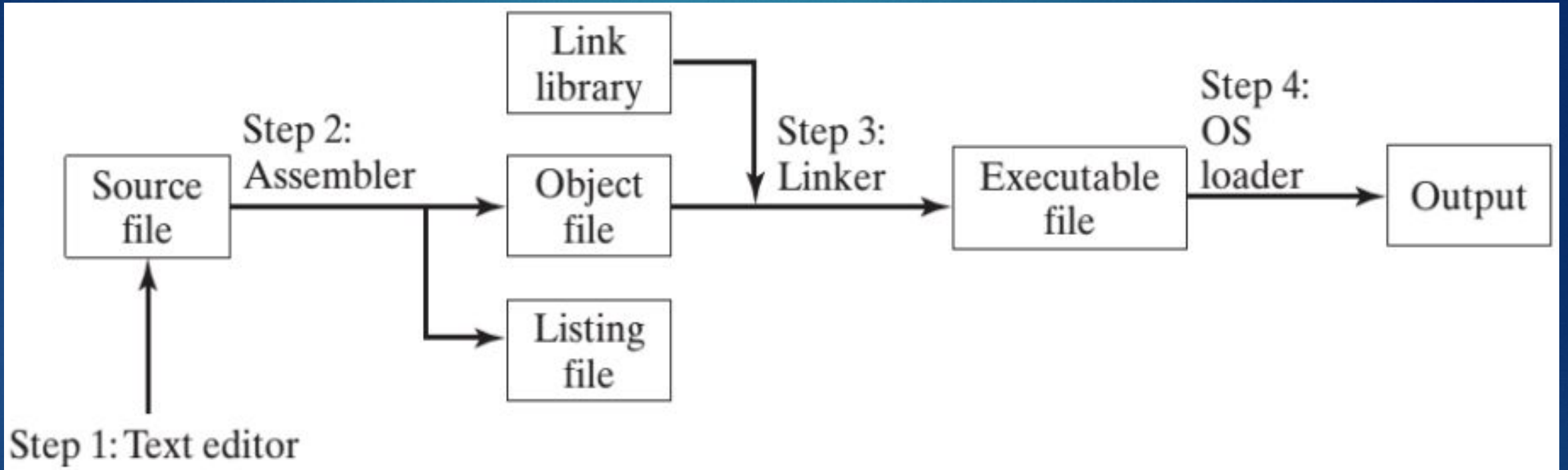
Digital Logic

Level 1

- ▶ CPU, constructed from digital logic gates
- ▶ System bus
- ▶ Memory
- ▶ Implemented using bipolar transistors



ASSEMBLING, LINKING, AND RUNNING PROGRAMS



ASSEMBLING, LINKING, AND RUNNING PROGRAMS

- ▶ **Assembler** is a utility program that converts source code programs from assembly language into an object file, a machine language translation of the program. Optionally a Listing file is also produced. We'll use MASM as our assembler.
- ▶ The linker reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the executable file. Microsoft 16-bit linker is LINK.EXE and 32-bit is Linker LINK32.EXE.
- ▶ **OS Loader:** A program that loads executable files into memory, and branches the CPU to the program's starting address, (may initialize some registers (e.g. IP)) and the program begins to execute.
- ▶ **Debugger** is a utility program, that lets you step through a program while it's running and examine registers and memory
- ▶ Is called Assemble-Link-Execute Cycle.

Listing File

A listing file contains:

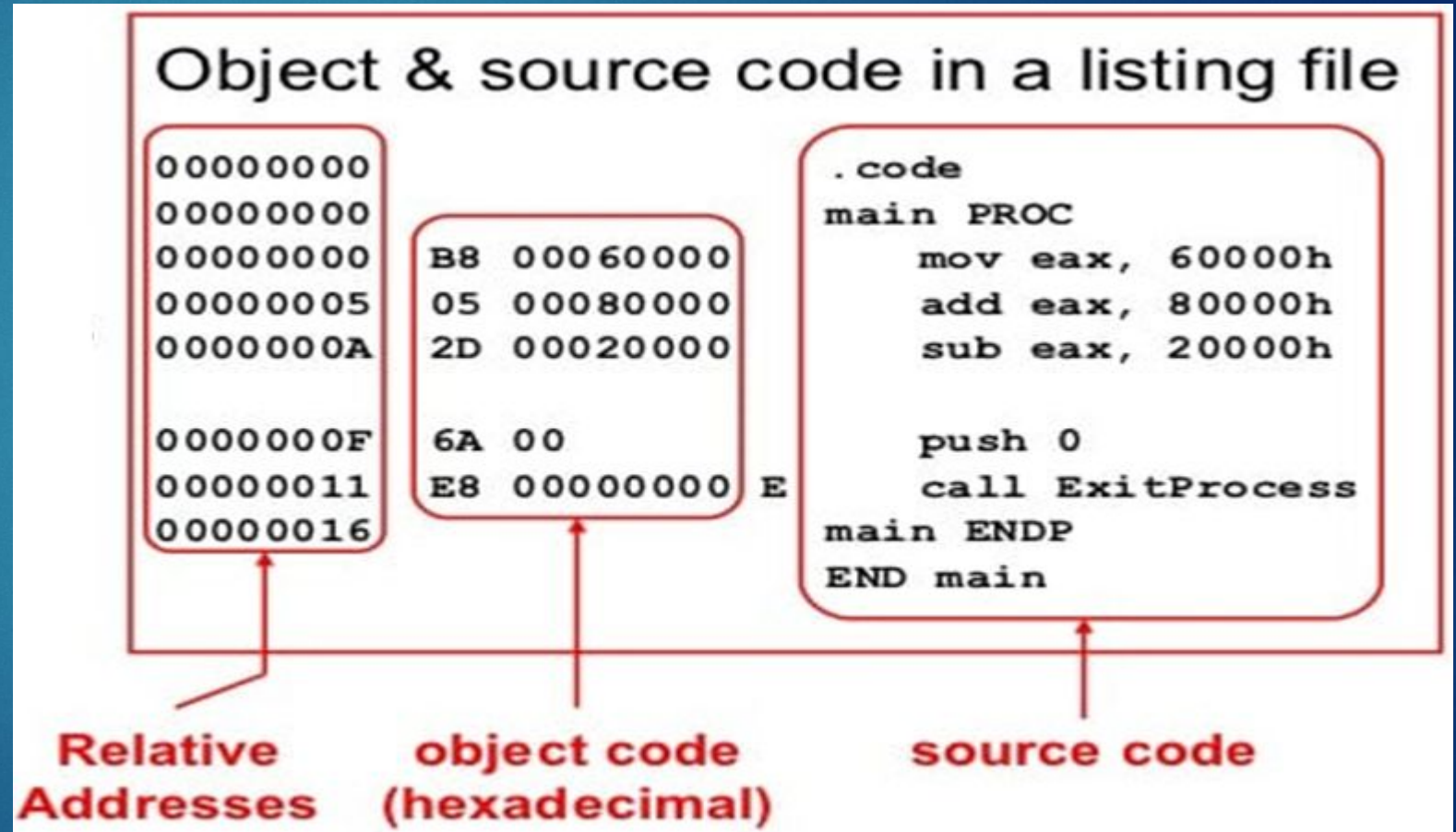
- ▶ a copy of the program's source code,
 - ▶ with line numbers,
 - ▶ the numeric address of each instruction,
 - ▶ the machine code bytes of each instruction (in hexadecimal), and
 - ▶ a symbol table.
-
- ▶ The symbol table contains the names of all program identifiers, segments, and related information.

Listing File

Use it to see how your program is assembled

Contains:

- ▶ source code
- ▶ Object Code
- ▶ Relative addresses
- ▶ Segment names
- ▶ Symbols
 - ▶ Variables
 - ▶ Procedure
 - ▶ Constants



Listing File

```
1  TITLE My First Program (Test.asm)
2  INCLUDE Irvine32.inc
3
4  .code
5  main PROC
6  mov eax, 10h
7  mov ebx, 25h
8  call DumpRegs
9  exit
10 main ENDP
11 END main
12
13
```

```
25  00000000      .code
26  00000000      main PROC
27  00000000 B8 00000010      mov eax, 10h
28  00000005 BB 00000025      mov ebx, 25h
29  0000000A E8 00000000 E      call DumpRegs
30          exit
31  00000016      main ENDP
32          END main
33
34  Microsoft (R) Macro Assembler Version 14.29.30133.0      09/12/21 20:28:23
35  My First Program (Test.asm      Symbols 2 - 1
36
```

Loading and Executing a Program

- ▶ The operating system (OS) searches for the program's filename in the current disk directory.
 - ▶ If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename.
 - ▶ If the OS fails to find the program filename, it issues an error message.
- ▶ If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- ▶ The OS determines the next available location in memory and loads the program file into memory.
 - ▶ It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table*).
 - ▶ Additionally, the OS adjust the values of pointers within the program so they contain addresses of program data.

Loading and Executing a Program

- ▶ The OS begins execution of the program's first machine instruction (its entry point).
- ▶ As soon as the program begins running, it is called a *process*.
- ▶ The OS assigns the process an *identification number* (*process ID*), which is used to keep track of it while running.
- ▶ It is the *OS's job to track the execution of the process* and *to respond to requests* for system resources.
 - ▶ Examples of resources are memory, disk files, and input-output devices.
- ▶ When the process ends, it is removed from memory.

Basic Program Execution Registers

- ▶ **Registers** are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.
 - ▶ There are eight general-purpose registers (32-bit).
 - ▶ Six segment registers (16-bits)
 - ▶ A processor status flags register (EFLAGS), and an instruction pointer (EIP)

Basic Program Execution Registers

FIGURE 2-5 Basic Program Execution Registers.

32-bit General-Purpose Registers

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

16-bit Segment Registers

EFLAGS
EIP

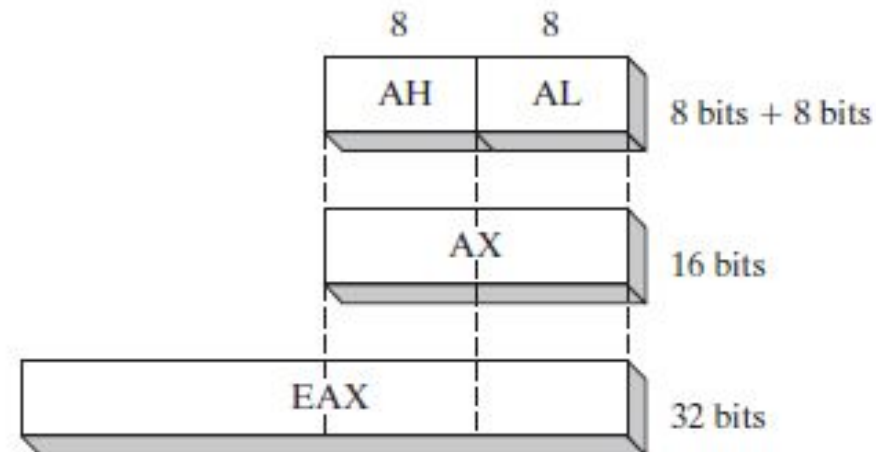
CS	ES
SS	FS
DS	GS

Basic Program Execution Registers

The general-purpose registers are primarily used for arithmetic and data movement.

- ▶ As shown in Figure 2–6, the lower 16 bits of the EAX register can be referenced by the name AX
- ▶ Portions of some registers can be addressed as 8-bit values
 - ▶ For example, the AX register, has an 8-bit upper half named AH and an 8-bit lower half named AL

FIGURE 2–6 General-Purpose Registers.



Basic Program Execution Registers

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

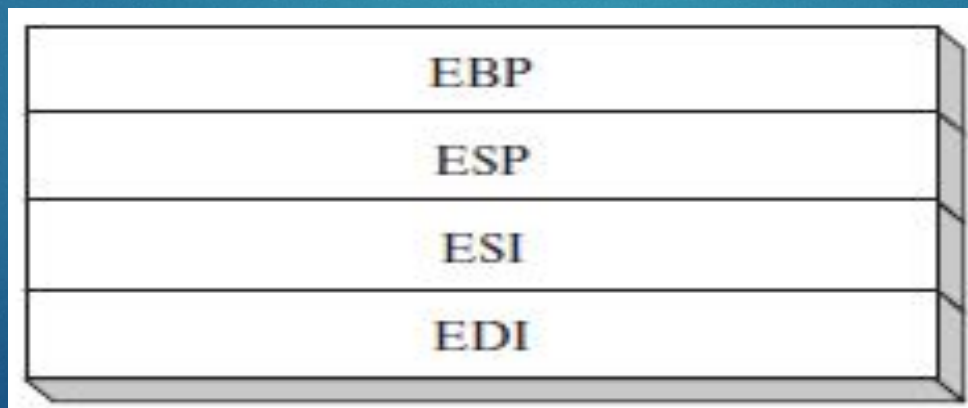
32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Specialized Uses of general-purpose registers

- ▶ Data Registers (EAX, EBX, ECX, EDX): These four registers are available to the programmer for general data manipulation.
- ▶ The high and low bytes of the data registers can be accessed separately.
- ▶ EAX (**Extended Accumulator Register**) is preferred to use in arithmetic, logic and control instructions.
- ▶ EBX (**Extended Base Register**) is used to serve as an address register.
- ▶ ECX (**Extended Counter Register**) serves as loop counter.
- ▶ EDX (**Extended Data Register**) is used in multiplication and division.

Index Registers

- ▶ Index Registers contain the offsets for data and instructions.
- ▶ **Offset**- distance (in bytes) from the base address of the segment.
- ▶ **ESP** (extended stack pointer register) contains the offset for the top of the stack to addresses data on the **stack** (a system memory structure)
- ▶ **ESI** and **EDI** (extended source index and extended destination index) points to the source and destination string respectively in the string move instructions
- ▶ **EBP** is used to reference function parameters and local variables on the stack



Segment Registers

- ▶ In real-address mode, 16-bit segment registers indicate base addresses of pre-assigned memory areas named segments.
- ▶ In protected mode, segment registers hold pointers to segment descriptor tables (The descriptor describes the location, length and access rights of the memory segment).
- ▶ Some segments hold program instructions (code), others hold variables (data), and another segment named the stack segment holds local function variables and function parameters.



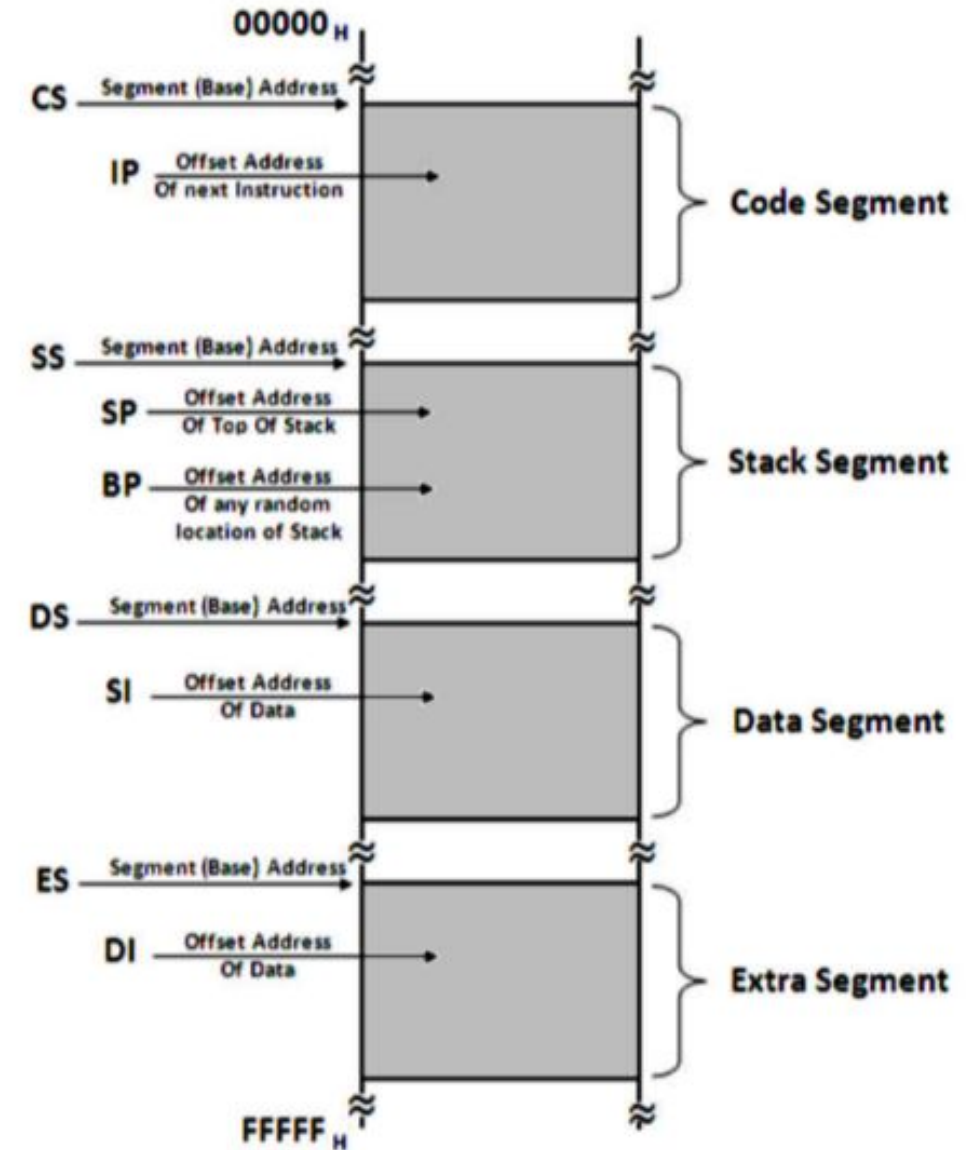
Memory

A
d
d
r
e
s
s

0xFFFFFFFF	1000 0000

0x00000008	0100 1001
0x00000007	1100 1100
0x00000006	0110 1110
0x00000005	0110 1110
0x00000004	0000 0000
0x00000003	0110 1011
0x00000002	0101 0001
0x00000001	1100 1001
0x00000000	0100 1111

Main Memory



Segment Registers

- ▶ Memory Segment: A memory segment is a block of consecutive memory bytes. Each segment is identified by a segment number, starting with 0.
- ▶ Within a segment, a memory location is specified by giving an offset. This is the number of bytes from the beginning of the segment.
- ▶ A memory location may be specified by providing a segment number and an offset, written in the format segment:offset.
- ▶ E.g. A4FB:4872h means offset 4872h within segment A4FBh.

Segment Registers

- ▶ The program's code, data, and stack are loaded into different memory segments, we call them the code segment, data segment, and stack segment.
- ▶ Stack segment holds local function variables and function parameters.
- ▶ To keep track of the various program segments, segment register are used.
- ▶ The ECS (Extended Code Segment), EDS (Extended Data Segment), and ESS (Extended Stack Segment) registers contain the code, data, and segment numbers respectively.
- ▶ If a program needs to access a second data segment, it can use the EES (Extended Extra segment) register.

Instruction Pointer

- ▶ The EIP, or instruction pointer, register contains the address of the next instruction to be executed.
- ▶ Certain machine instructions manipulate EIP, causing the program to branch to a new location.

EFLAGS Register

- ▶ The EFLAGS register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation.
 - ▶ A flag is set when it equals 1; it is clear (or reset) when it equals 0.
- ▶ Programs can **set individual bits in the EFLAGS** register to control the CPU's operation
- ▶ For example: **Interrupt when arithmetic overflow is detected**

x	x	x	x	O	D	I	T	S	Z	x	A	x	P	x	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

O = Overflow

S = Sign

D = Direction

Z = Zero

I = Interrupt

A = Auxiliary Carry

T = Trap

P = Parity

x = undefined

C = Carry

EFLAGS Register

- ▶ There are two types of flags:
 - ▶ **Control flags**: which determine how instructions are carried out
 - ▶ **Status flag**: which report on the result operation
- ▶ Control flags include:
 - ▶ **Direction Flag (DF)**: affects the direction of block data transfers (like long character string) 1=up; 0= down.
 - ▶ **Interrupt Flag (IF)**: determines whether interrupts can occur (whether devices like keyboard, disk drives and system clock can get the CPU's attention to get their needs attended to).
 - ▶ **Trap Flag (TF)**: determines whether the CPU is halted after every instruction. Used for debugging purposes

EFLAGS Register

- ▶ Status flags include:
 - ▶ **Carry Flag (CF)**: set when the result of unsigned arithmetic is too large to fit in the destination, 1=carry; 0=no carry.
 - ▶ **Overflow Flag (OF)**: set when the result of signed arithmetic is too large to fit in the destination, 1=overflow; 0=no overflow.
 - ▶ **Sign Flag (SF)**: set when an arithmetic or logical operation generates a negative result. 1=negative, 0=positive.
 - ▶ **Zero Flag (ZF)**: set when an arithmetic or logical operation generates a result of zero. Used primarily in jump and loop operations, 1=zero; 0=not zero.
 - ▶ **Auxiliary-carry Flag (AF)**: set when an operation causes a carry from bit3 to 4 or borrow (from bit 4 to 3), 1=carry; 0=no carry.
 - ▶ **Parity Flag (PF)**: is set if the least significant byte in the result contains an even number of 1 bits. It is used to verify memory integrity.

Mode of Operations

- ▶ x86 processors have three primary modes of operation:
 - ▶ protected mode,
 - ▶ real-address mode, and
 - ▶ system management mode.

Mode of Operations

- ▶ Real Address mode (original mode provided by 8086)
 - ▶ Only 1 MB of memory can be addressed from 0 to FFFFF (hex)
 - ▶ Programs can access any part of main memory
 - ▶ MS-DOS runs in real address mode
- ▶ Implements the programming environment of the Intel 8086 processor
- ▶ This mode is available in Windows 98, can be used to run MS-DOS program that requires direct access to system memory and hardware devices
- ▶ Programs running in real-address mode can cause the operating system to crash (stop responding to commands)

Mode of Operations

- ▶ Protected mode (introduced with the 80386)
 - ▶ Each Program can address a maximum of 4 GB of memory
 - ▶ The operating system assigns memory to each running program
 - ▶ Programs are prevented from accessing each other's memory (segments)
 - ▶ Native mode used by Windows NT, 2000, XP and Linux

Mode of Operations

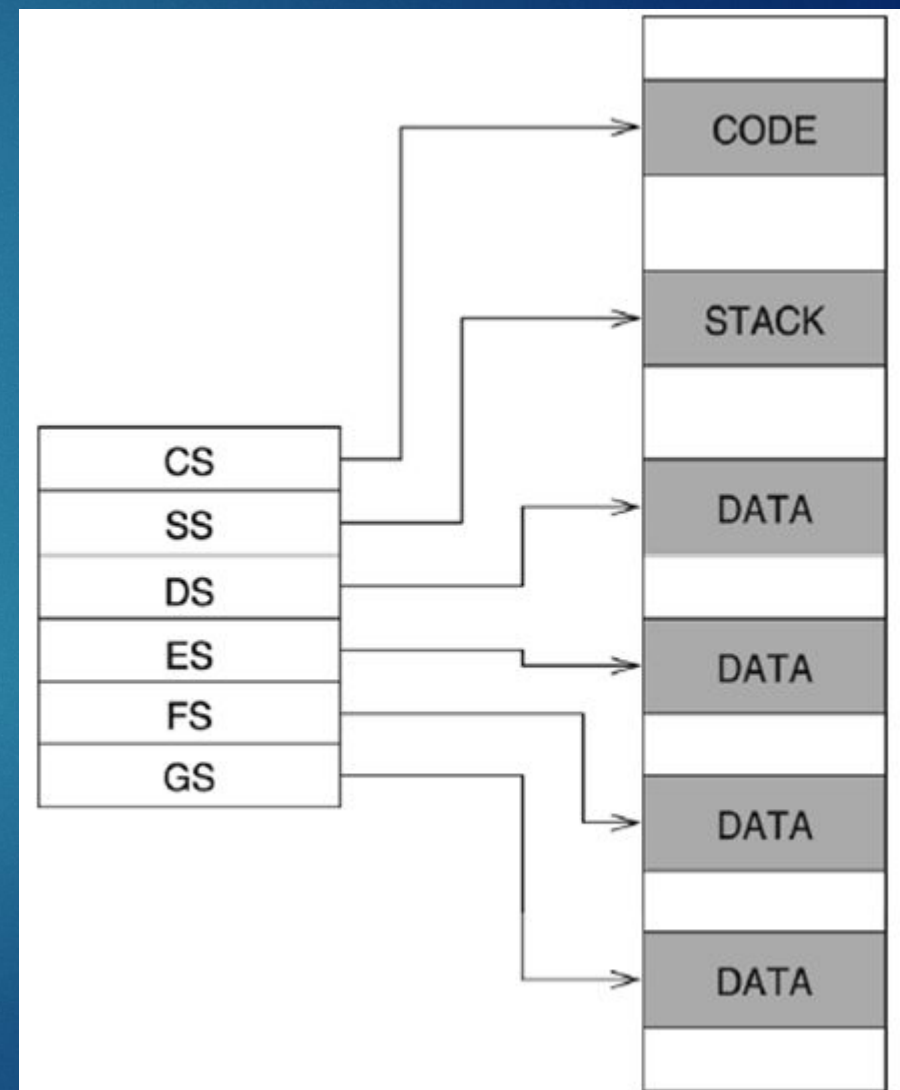
- ▶ Virtual 8086 mode: A sub-mode, virtual-8086, is a special case of protected mode
- ▶ Processor runs in protected mode, and creates a virtual 8086 machine with 1 MB of address space for each running Program, such as: MS-DOS
 - ▶ If an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time
- ▶ Windows XP can execute multiple separate virtual-8086 sessions at the same time

Mode of Operations

- ▶ System Management Mode:
 - ▶ Provides a mechanism for implementation power management and system security
 - ▶ Manage system safety functions, such as shutdown on high CPU temperature and turning the fans on and off
 - ▶ Handle system events like memory or chipset errors

Real Address Mode

- ▶ A Program can access up to six segments at any time
 - ▶ Code segment
 - ▶ Stack segment
 - ▶ Data Segment
 - ▶ Extra segments (up to 3)
- ▶ Each segment is 64 KB
- ▶ Logical address
 - ▶ Segment = 16 bits
 - ▶ Offset = 16 bits
- ▶ Linear (physical) address = 20 bits



Real Address Mode

- ▶ Program Segments and Segments Registers: Segment registers are used to hold base addresses for the program code, data and stack.
 - ▶ The **code segment** holds the base address for all executable instructions in the program
 - ▶ The **data segment** holds the base address for variables. This segment stores data for the program
 - ▶ The **extra segment** is an extra data segment (often used for shared data)
 - ▶ The **stack segment** holds the base address for the stack. The segment is also to store interrupt and subroutine return addresses

Real Address Mode

- ▶ Linear address = Segments x 10 (hex) + Offset

Example:

- ▶ Given:

Segment = A1F0 (hex)

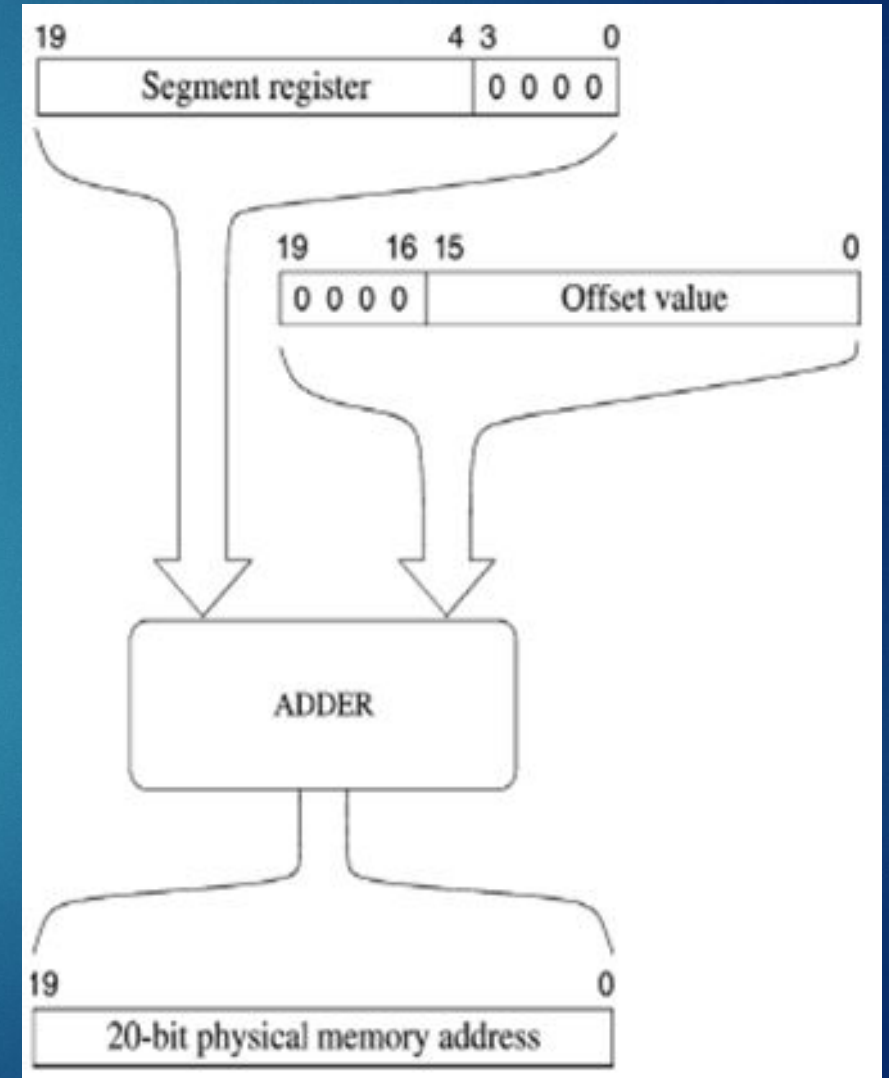
Offset = 04C0 (hex)

Logical Address = A1F0:04C0 (hex)

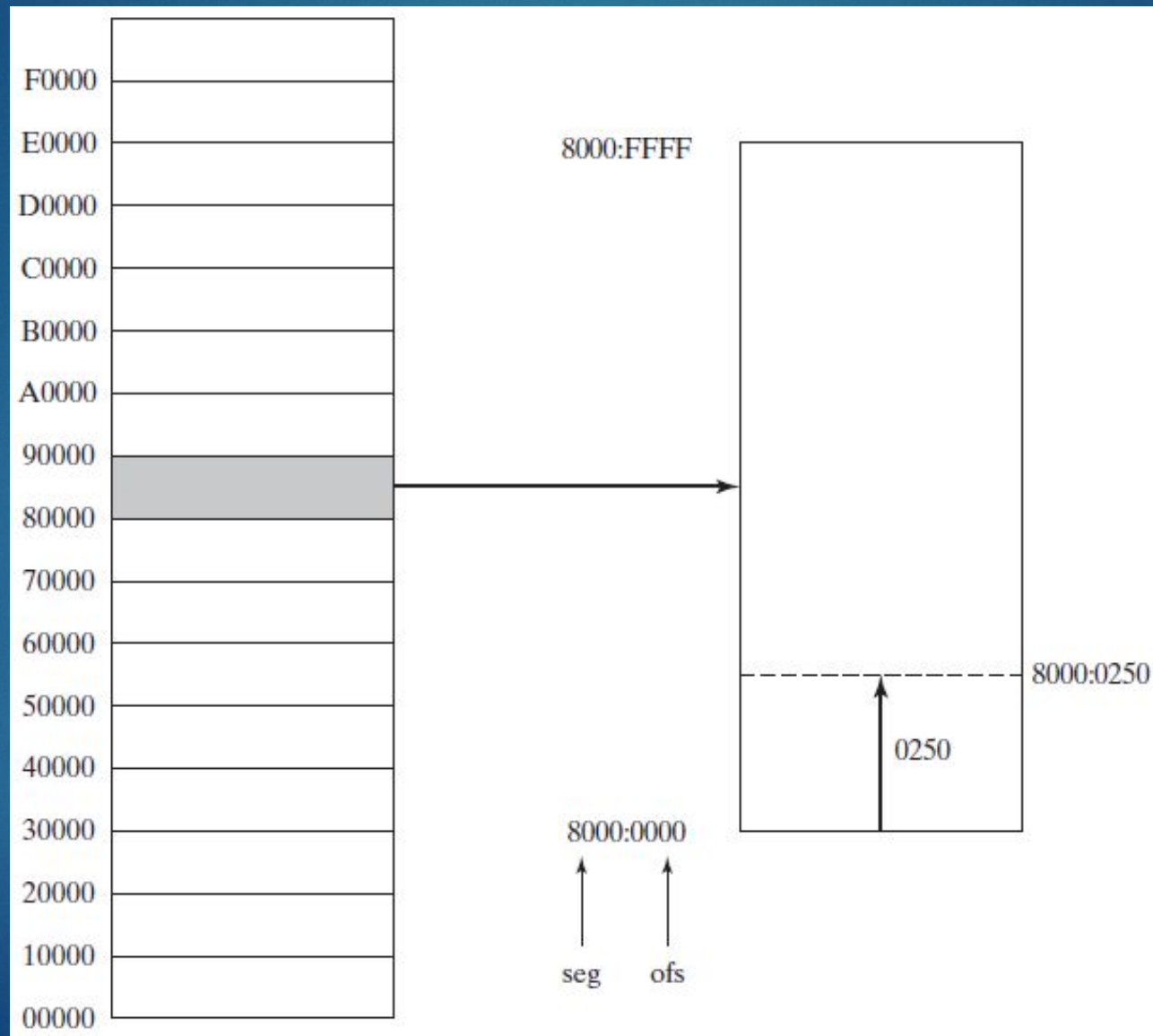
What is linear address?

- ▶ Solution:

A1F0	0	(add 0 to segment in hex)
+	04C0	(offset in hex)
<hr/>		
A23C0		(20-bit linear address in hex)



Real Address Mode



Address Space

- ▶ In **32-bit protected mode**, a task or program can address a linear address space of up to 4 GB
 - ▶ Extended Physical Addressing allows a total of 64 GB of physical memory to be addressed
- ▶ **Real-address mode** programs, on the other hand, can only address a range of 1 MB
- ▶ If the processor is in protected mode and running multiple programs in **virtual-8086 mode**, each program has its own 1-MByte memory area

