Computer Organization and Assembly Language

Week 1 to 3

Shoaib Rauf

Introduction

Definition 1:

- Computer organization is concerned with the way hardware components operate and the way they are connected together to form a computer system.

Definition 2:

- Computer organization is concerned with the structure and behavior of a computer system.

Definition 3:

- *Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL.

Introduction

- The main objective of this course is to:
  - Understand the basic structure of a computer system
  - How to write programs by understanding the hardware structure

Assembly Language Programming

- An assembly language is a low-level programming language designed for a specific type of processor
- Assembly language is used to write the program using alphanumeric symbols (or mnemonic), e.g. ADD, MOV, PUSH etc.
- Assembly Language depends on the machine code instructions
  - Every assembler has its own assembly language
- Assembly language has a one-to-one relationship with machine language:
  - Each assembly language instruction corresponds to a single machine-language instruction.
  - A single machine-language instruction can take up one or more bytes of code

Assembly language programming

- The instruction must specify which operation (opcode) is to be performed and the operands
- E.g. ADD AX, BX
  - ADD is the operation
  - AX is called the destination operand
  - BX is called the source operand
  - The result is AX = AX + BX
- When writing assembly language program, you need to think in the instruction level

Assembly language programming: Example

- MOV AL, 00H
  - The native language of a computer is machine language (using 0,1 to represent the operation)
  - The machine language code for the above instruction is B4 00 (2 bytes)
  - After assembled, and linked into an executable program, the value B400 will be stored in the memory
  - When the program is executed, then the value B400 is read from memory, decoded and carry out the task
    - The executable program could be .com, .exe, or .bin files

Assembly language programming

- Learning assembly language programming will help understanding the operations of a microprocessor
- To learn:
  - Need to know the functions of various registers
  - Need to know how external memory is organized and how it is addressed to obtain instructions and data (different addressing modes)
  - Need to know what operations (or the ***instruction set***) are supported by the CPU

High Level Languages

- A high-level language (HLL) is a programming language that enables a programmer to write programs that are more or less independent of a particular type of computer

- Such languages are considered high-level because they are closer to human languages and further from machine languages.

- Examples are C, C++, Java, FORTRAN, or Pascal

High Level Languages

- High-level languages such as C++ and Java have a *one-to-many* relationship with assembly language and machine language.

- A single statement in C++ expands into multiple assembly language or machine instructions.

- The following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

```
int Y;
int X = (Y + 4) * 3;
```

Example:

```
int Y;
int X = (Y + 4) * 3;
```

- Following is the equivalent translation to assembly language.

- The translation requires multiple statements because assembly language works at a detailed level:

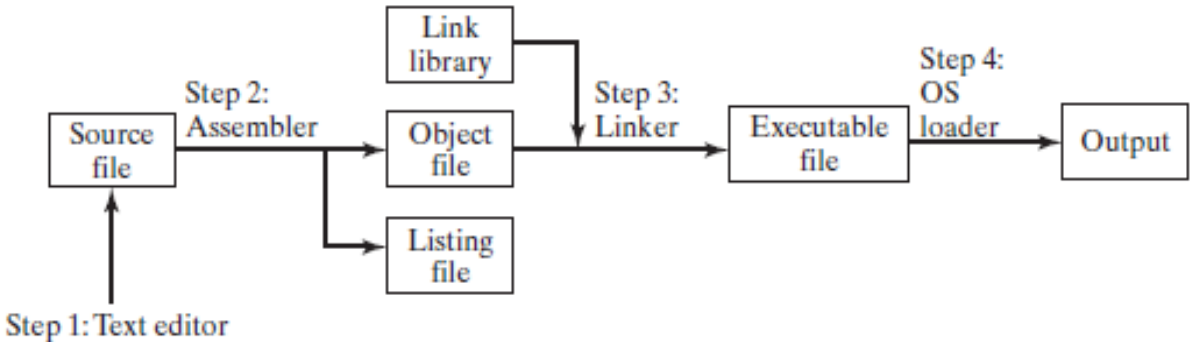
```
mov eax,Y ; move Y to the EAX register
add eax,4 ; add 4 to the EAX register
mov ebx,3 ; move 3 to the EBX register
imul ebx ; multiply EAX by EBX
mov X,eax ; move EAX to X
```

Registers are named storage locations in the CPU that hold intermediate results of operations

What Are Assemblers and Linkers?

- **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

FIGURE 3–7 Assemble-Link-Execute cycle.



MASM provides CodeView, TASM provides Turbo Debugger and msdev.exe for 32-bit Window console programs.

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.

FIGURE 3–8 Excerpt from the AddTwo source listing file.

```
 1:    ; AddTwo.asm - adds two 32-bit integers.
 2:    ; Chapter 3 example
 3:
 4:    .386
 5:    .model flat,stdcall
 6:    .stack 4096
 7:    ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                        .code
10:     00000000                        main PROC
11:     00000000  B8 00000005              mov  eax,5
12:     00000005  83 C0 06                 add  eax,6
13:
14:                                        invoke ExitProcess,0
15:     00000008  6A 00                    push   +000000000h
16:     0000000A  E8 00000000 E            call   ExitProcess
17:     0000000F                        main ENDP
18:                                     END main
```

Assembly Language for x86 Processors

- *Assembly Language for x86 Processors* focuses on programming microprocessors compatible with the Intel IA-32 and AMD x86 processors running under Microsoft Windows.

- Assembly language bears the closest resemblance to native machine language.

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.

- A major feature of the Java language is that compiled programs run on nearly any computer system.

Is Assembly Language Portable?

- Assembly language is not portable because it is designed for a specific processor family.

- There are a number of different assembly languages widely used today, each based on a processor family.

  – Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370.

- 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*.

What you'll learn from Assembly Language Programming

You will learn:

- Some basic principles of computer architecture, as applied to the Intel IA-32 processor family.
- How IA-32 processors manage memory, using real mode, protected mode, and virtual mode.
- How high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code.
- How high-level languages implement arithmetic expressions, loops, and logical structures at the machine level.

What you'll learn from Assembly Language Programming

- You will improve your machine-level debugging skills.
  - Even in C++, when your programs have errors due to pointers or memory allocation, you can dive to the machine level and find out what really went wrong.
- High-level languages purposely hide machine-specific details, but sometimes these details are important when tracking down errors.
- Assembly language will help you understanding the interaction between the computer hardware, operating system, and application programs.

Applications of Assembly Language

- Embedded systems programs are written in C, Java, or assembly language, and downloaded into computer chips and installed in dedicated devices.
  - Some examples are automobile fuel and ignition systems, air-conditioning control systems, security systems, flight control systems, hand-held computers, modems, printers, and other intelligent computer peripherals.

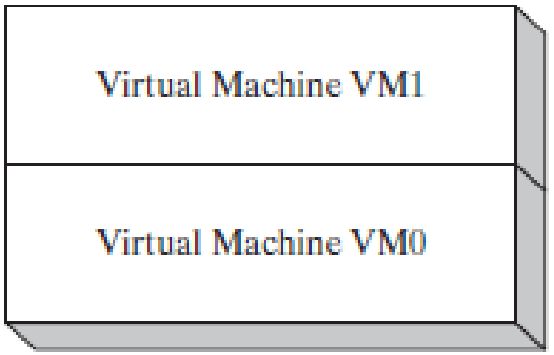
- Many dedicated computer game machines have stringent memory restrictions, requiring programs to be highly optimized for both space and runtime speed.
  - Game programmers use assembly language to take full advantage of specific hardware features in a target system.

Virtual Machine Concept

- Virtual Machine Concept:
  - An effective way to explain how a computer's hardware and software are related is called the *virtual machine concept.*
- Specific Machine Levels

Virtual Machines

- Virtual machine is a software program that emulates the functions of some other physical or virtual computer.
- 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 Machines (Continue…)

- Programs written in L1 can run in two different ways:
  - Translation – L1 program is completely translated into an L0 program (which then runs on the computer hardware)
  - Interpretation – L0 program interprets and executes L1 instructions one by one

Translating Languages

English: Display the sum of A times B plus C.

C++: cout << (A * B + C);

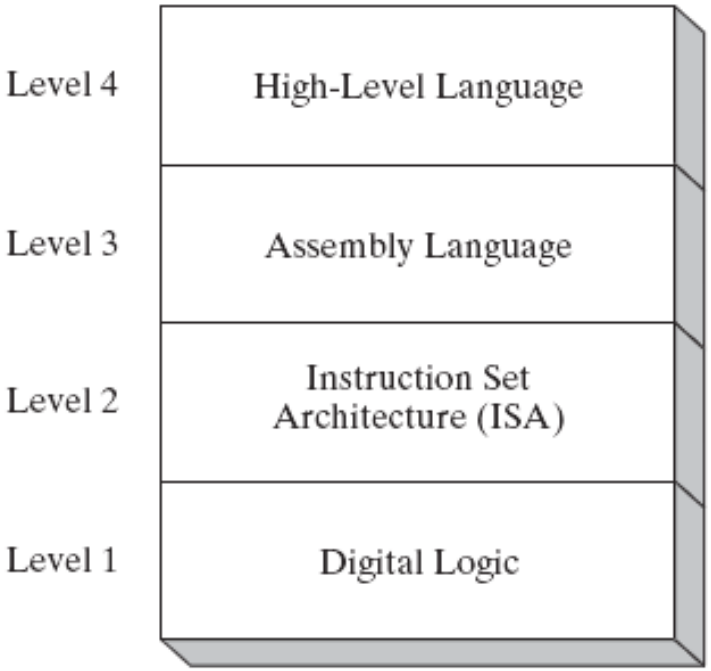
Assembly Language:

```
mov eax,A
mul B
add eax,C
call WriteInt
```


Intel Machine Language:

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Specific Machine Levels

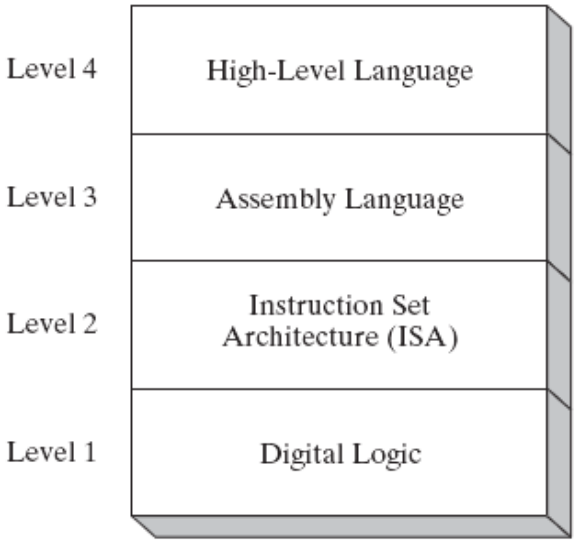


(descriptions of individual levels follow . . .)

High-Level Language

Level 4

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



High-Level Language

- 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 code.
- Java byte code is executed at runtime by a program known as a *Java virtual machine (JVM)*.

MyProgram.java - Notepad

File Edit Format View Help

```
public class MyProgram {

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

}
```

MyProgram.class - Notepad

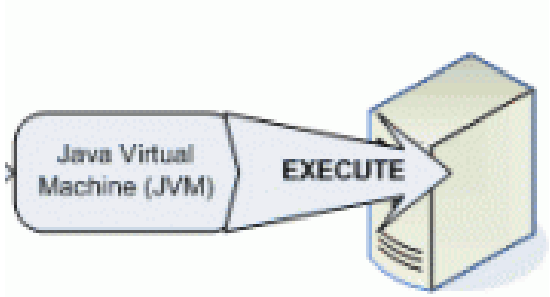
File Edit Format View Help

```
Êþº¾   . - ¸     + ¼ ¡ ¢ ¦ ª ±  -<init>  ()V  Code
LineNumberTable  main  ([Ljava/lang/String;)V
SourceFile  MyProgram.java ¸ ¦ ¹ ¡  Hello, World ª ¬
      MyProgram  ¡java/lang/Object  ¡java/lang/System
out  ¬Ljava/io/PrintStream;  ¡java/io/PrintStream
println  ¬(Ljava/lang/String;)V ! | -     ± ² ¦ ¶ ²
 ±¬¶ ¬±    ²   ± ¶ | ²   ± ¶
```

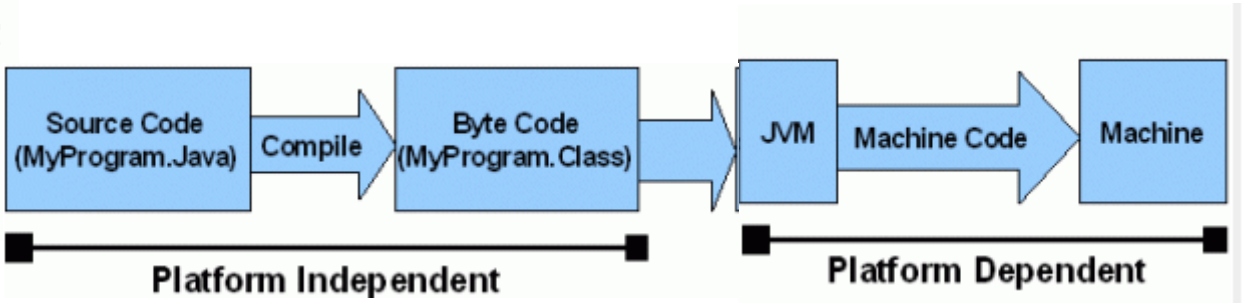
Source code is first written in plain text files ending with the .java extension.

Interpret

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



JVM reads bytecode and converts it into machine specific instructions.

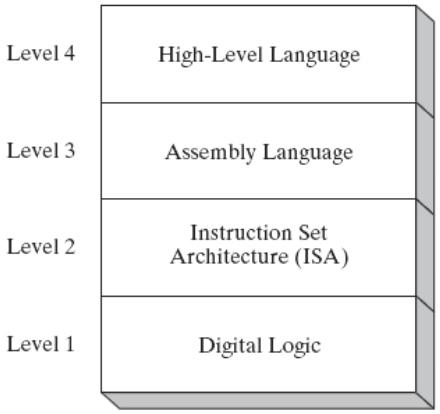


Java code / Bytecode is always the same on different OS.

- That makes java program as platform independent.

JVM is platform dependent that means there are different implementation of JVM on different OS.

Assembly Language



Level 3

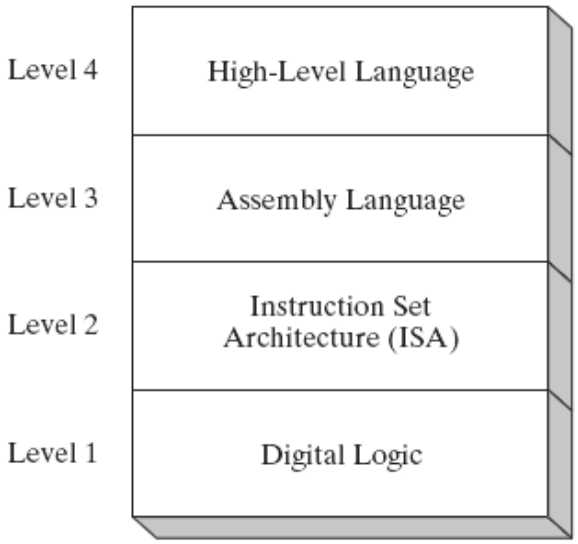
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Programs are translated into Instruction Set Architecture Level - machine language (Level 2)

- 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

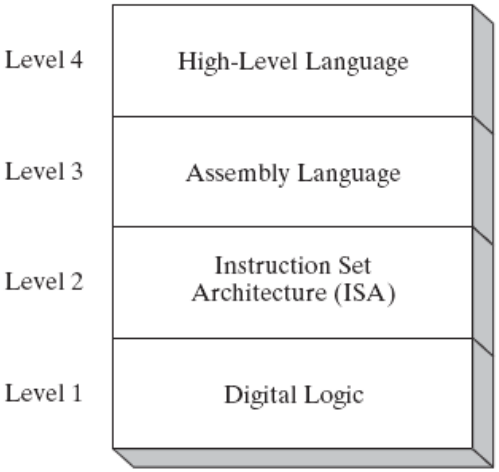
- Also known as conventional machine language
- Executed by Level 1 (Digital Logic)



Digital Logic

Level 1

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



Basic Microcomputer Design

FIGURE 2–1 Block Diagram of a Microcomputer.

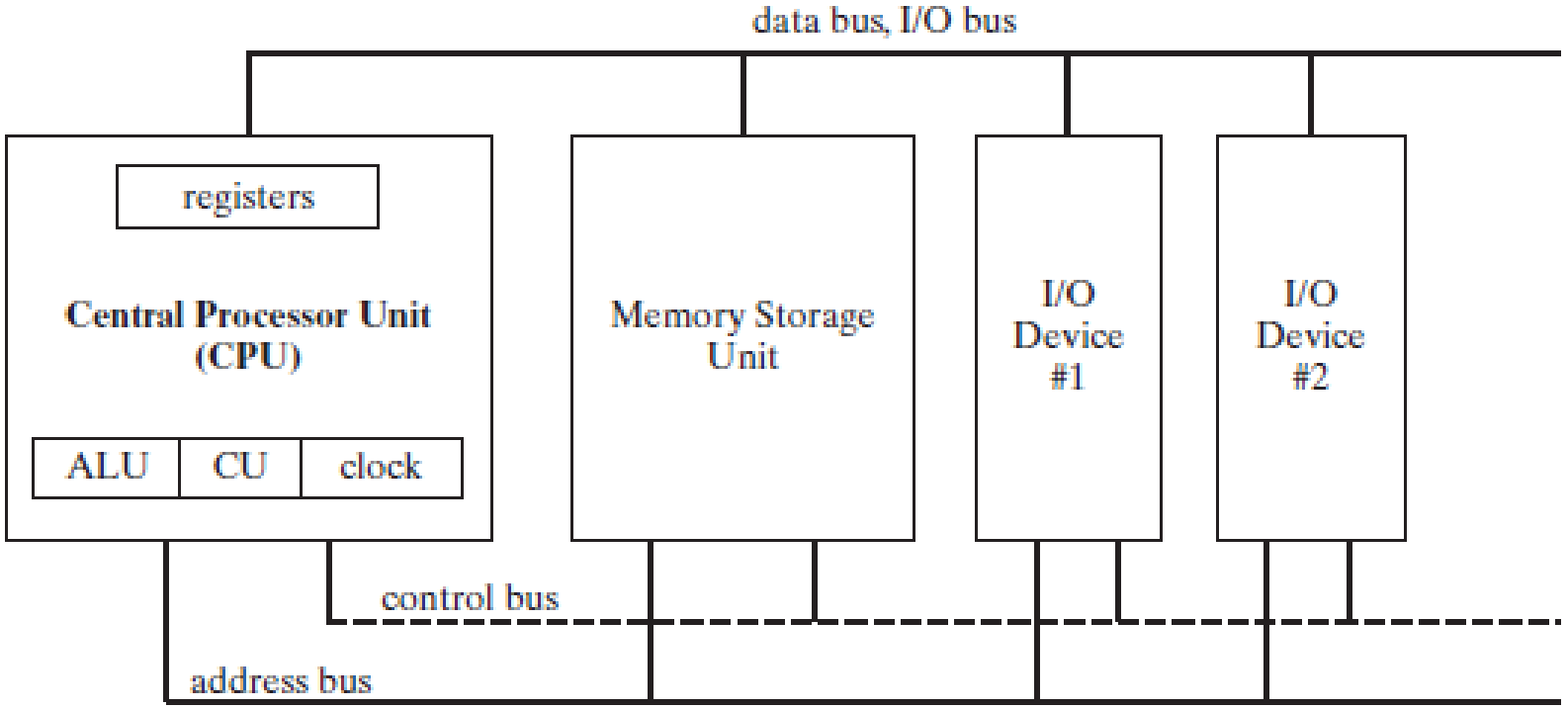
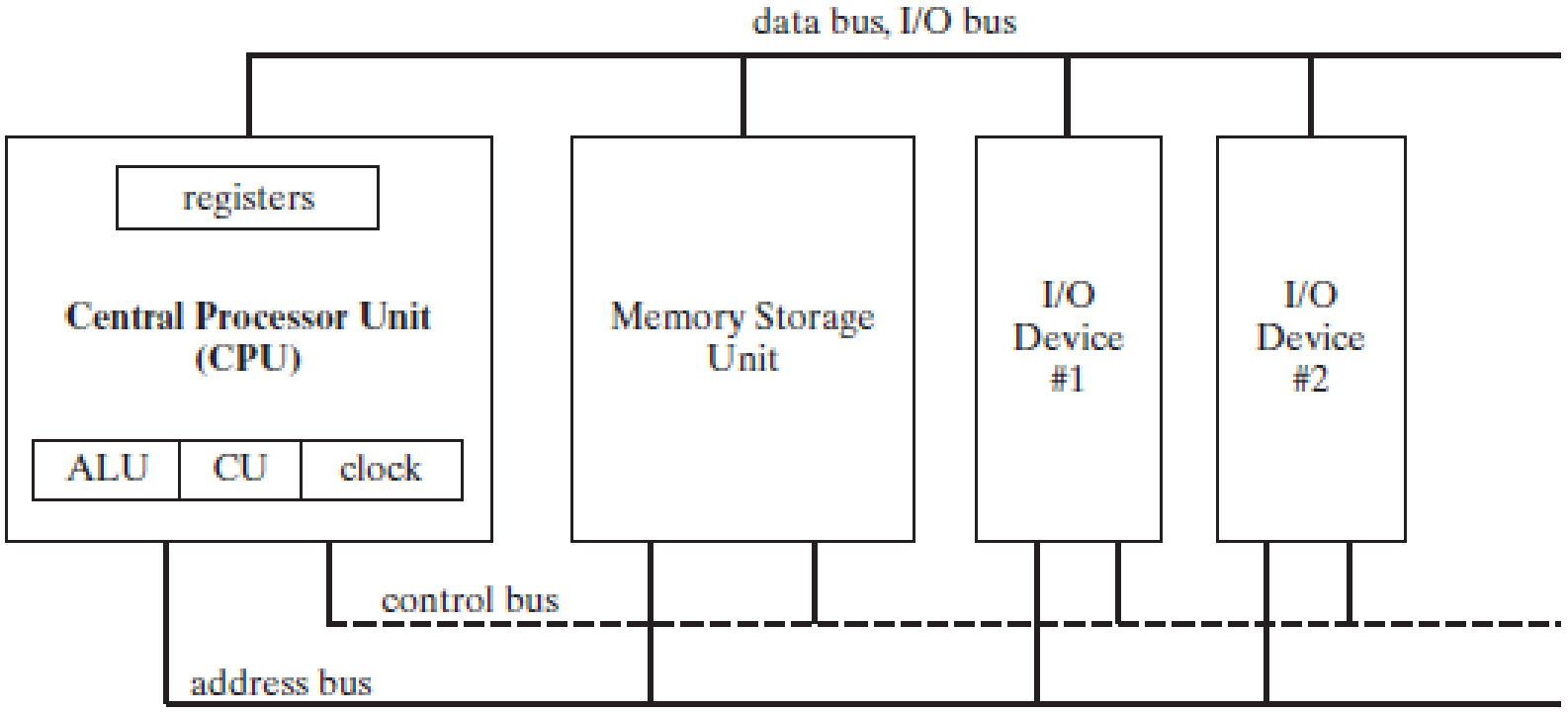


FIGURE 2–1 Block Diagram of a Microcomputer.



The *central processor unit* (CPU): Where calculations and logic operations are performed:

- 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.

FIGURE 2–1 Block Diagram of a Microcomputer.

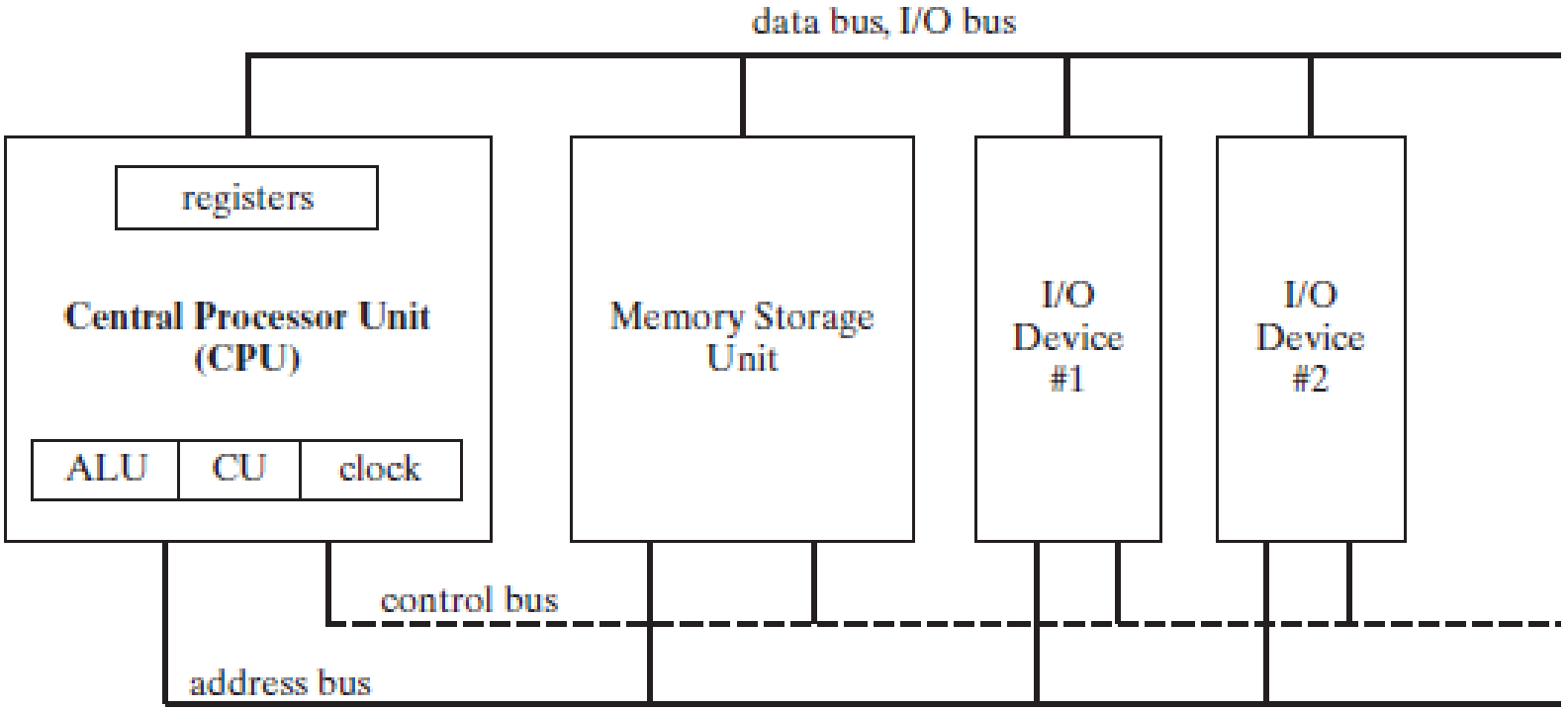
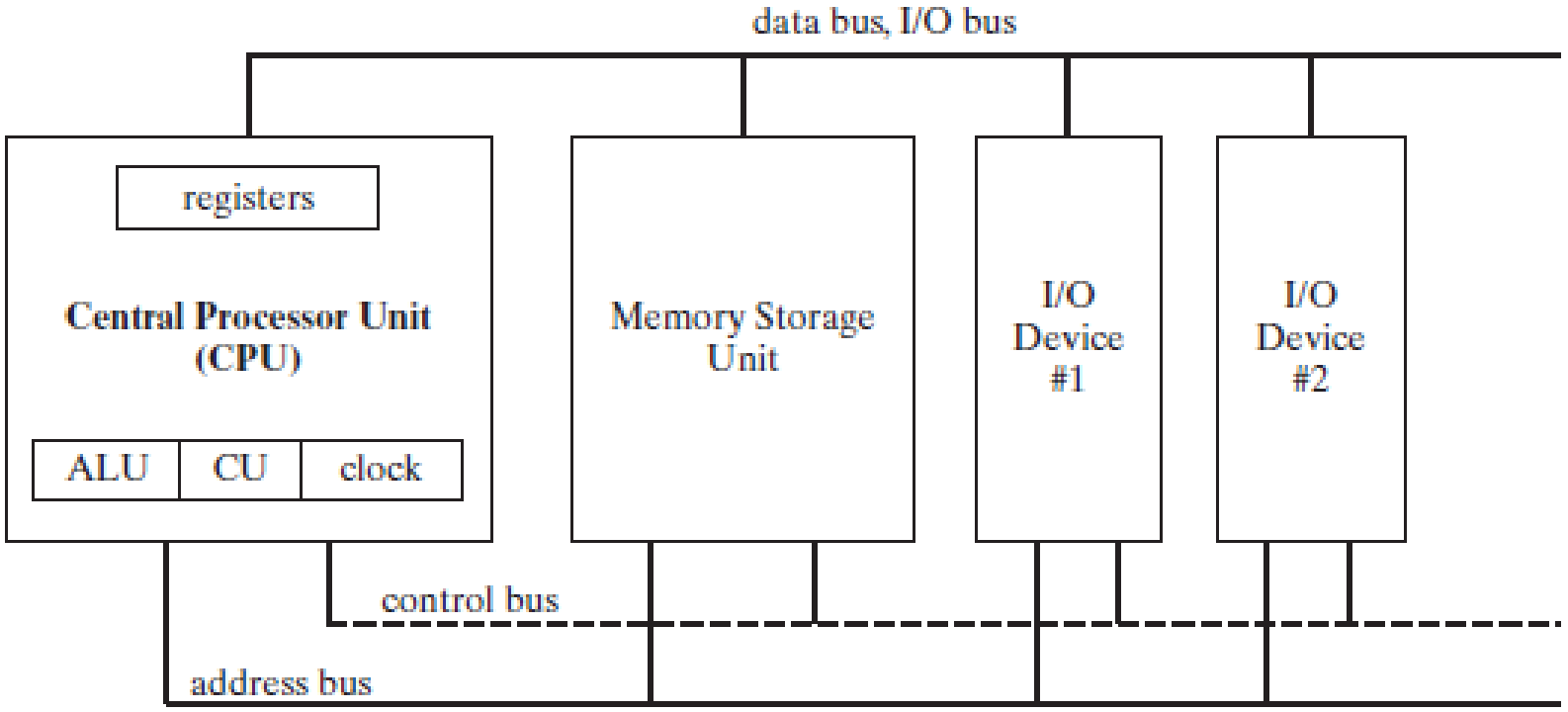


FIGURE 2–1 Block Diagram of a Microcomputer.



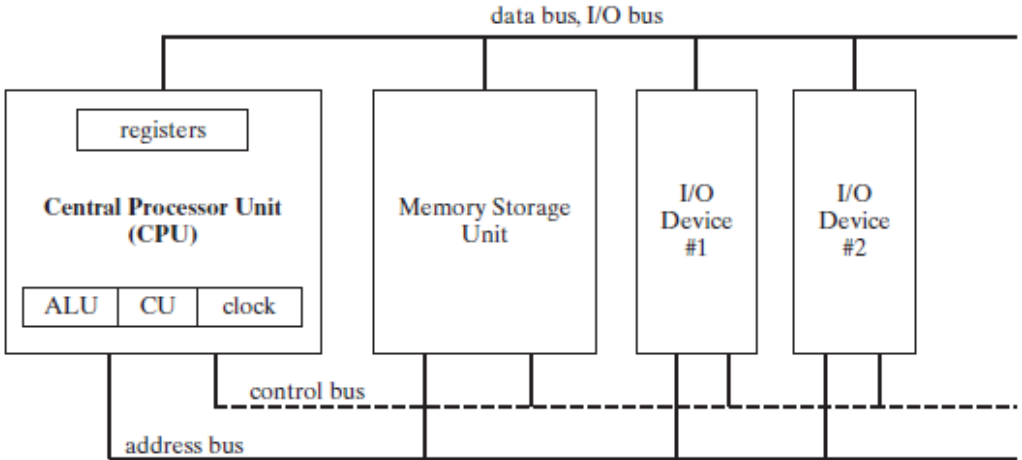
The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.

All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute.

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.

FIGURE 2–1 Block Diagram of a Microcomputer.



Clock and Clock Cycles

- Clock synchronizes all CPU and BUS operations
  - Clock is used to trigger events
  - Clock cycles measure time of a single operation
- 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

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.

Instruction Execution Cycle

Here are the steps to execute it:

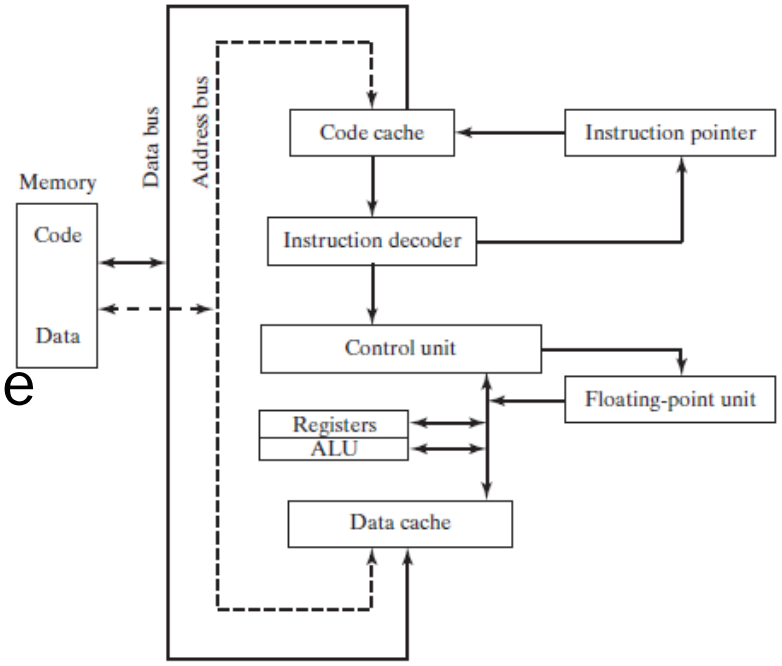
1. First, the CPU **fetch the instruction** from the *instruction queue*
   - It then increments the instruction pointer
2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern
   - This bit pattern might reveal that the instruction has operands (input values)
3. If operands 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

- An *operand* is a value that is either an input or an output to an operation
- For example, the expression
(X and Y) and a single outpu
- In order to read program ins
is placed on the address bus

- The memory controller places the requested code on the data bus
  - Making the code available inside the code cache

Figure 2–2 Simplified CPU block diagram.

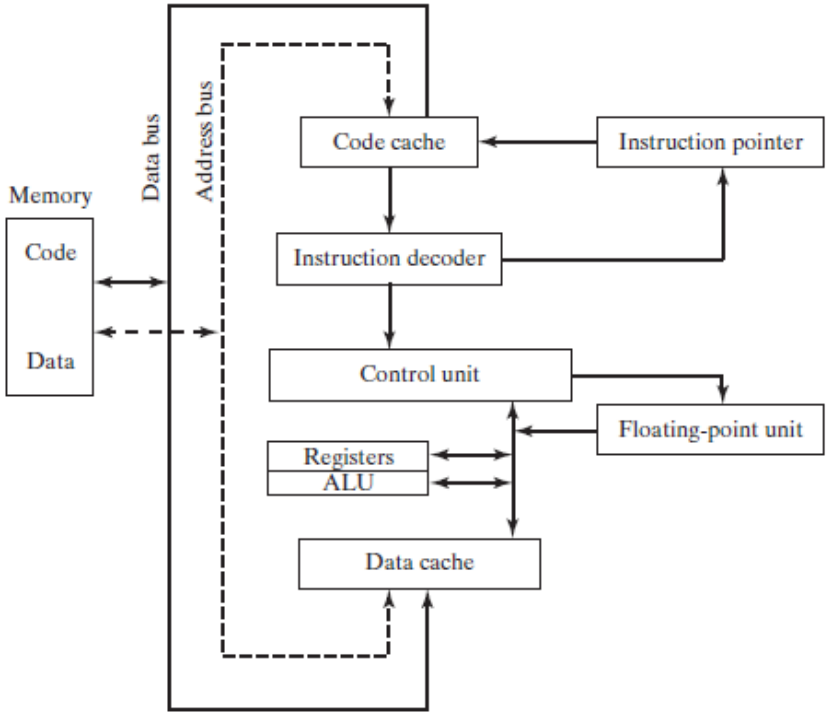


Instruction Execution Cycle

- The instruction pointer's value determines which instruction will be executed next.
- The instruction is analyzed by the **instruction decoder**
  - Causing appropriate digital signals to be sent to the Control Unit
  - Control Unit coordinates with 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.

FIGURE 2–2 Simplified CPU block diagram.



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 (1 of 2)

- 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 want to access the same memory and instructions repeatedly
  - so cache keeps these values where they can be accessed quickly

Cache (2 of 2)

- When the CPU begins to execute a program, it loads the next thousand instructions (for example) into cache
  - The assumption is that these instructions will be needed in the near future.
- If it 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.
  - *Level-1 cache* (or *primary cache*) is stored right on the CPU.
  - *Level-2 cache* (or *secondary cache*) is a little bit slower, and attached to the CPU by a high-speed data bus.

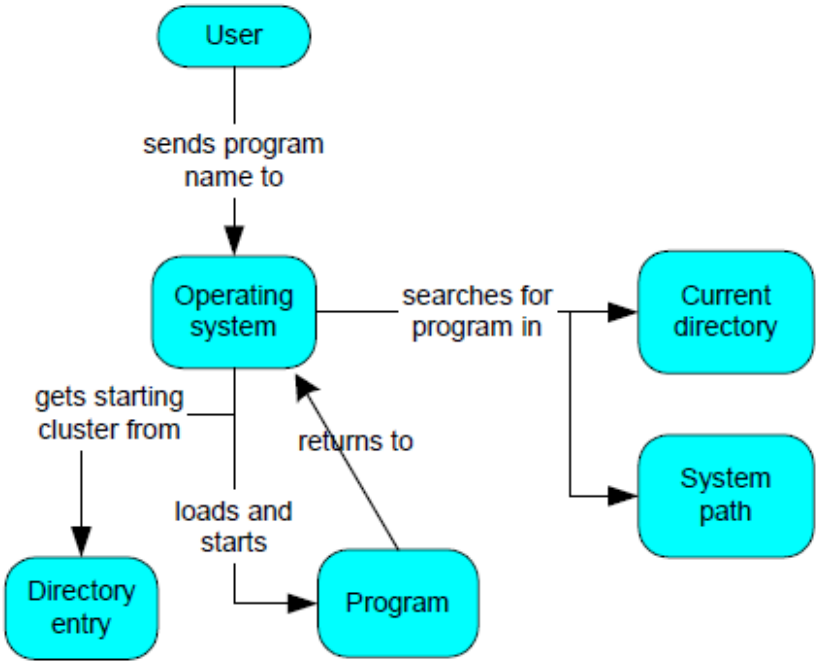
Why cache memory is faster than conventional RAM?

- It's because cache memory is constructed from a special type of memory chip called *static RAM*
  - It's expensive, but it does not have to be constantly refreshed in order to keep its contents

- Conventional memory, known as *dynamic RAM*, refreshed constantly
  - It's much slower, and cheaper

Loading and Executing a Program (1 of 3)

- 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



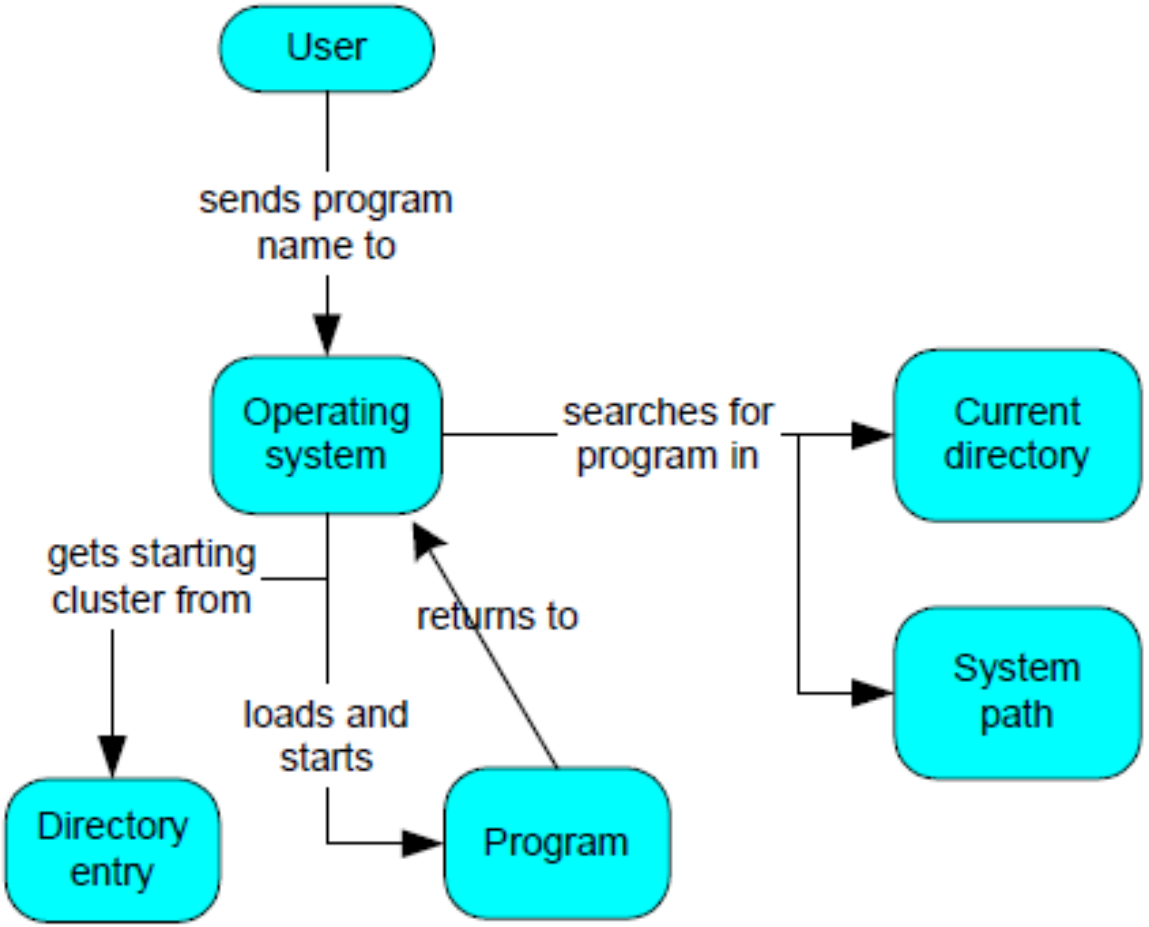
Loading and Executing a Program (2 of 3)

- 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*).
  - The OS also adjust the values of pointers within the program so they contain addresses of program data.

Loading and Executing a Program (3 of 3)

- 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.

How a Program Runs



Mode of Operations

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

- 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, and can be used to run an 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)

Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Mode of Operations

- Protected mode (introduced with the 80386 processor)
  - 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

Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

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

Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

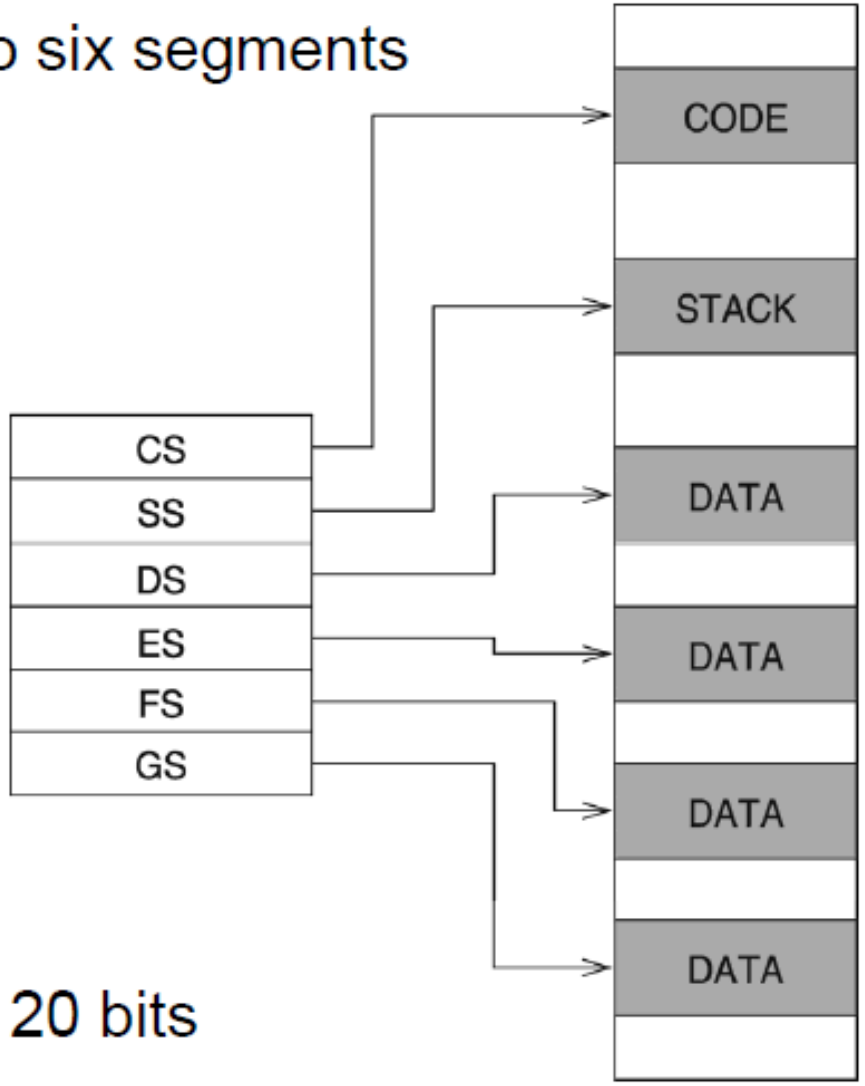
Mode of Operations

❖ 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 (1)

- 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



Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Real Address Mode

Program Segments and Segment 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 (2)

Linear address = Segment × 10 (hex) + Offset

Example:

segment = A1F0 (hex)

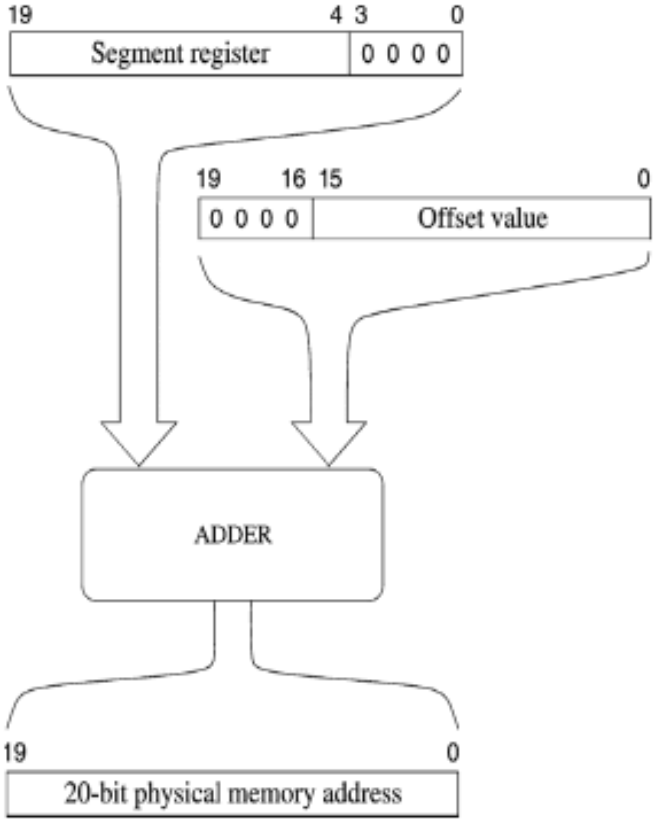
offset = 04C0 (hex)

logical address = A1F0:04C0 (hex)

what is the linear address?

Solution:

```
  A1F00  (add 0 to segment in hex)
+ 04C0   (offset in hex)
-------
  A23C0  (20-bit linear address in hex)
```



Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Real Address Mode (3)

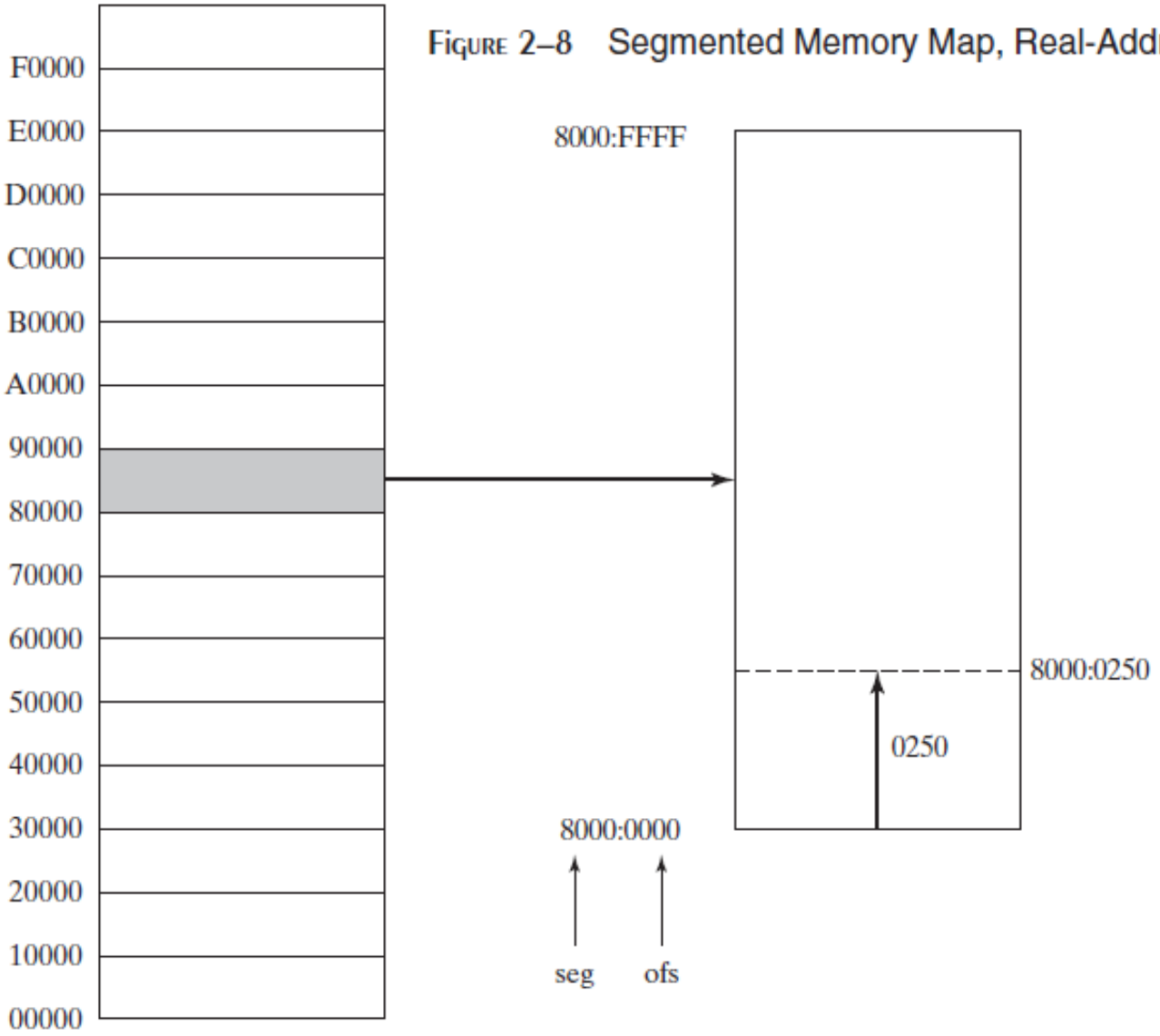


FIGURE 2–8 Segmented Memory Map, Real-Address Mode.

Address Space

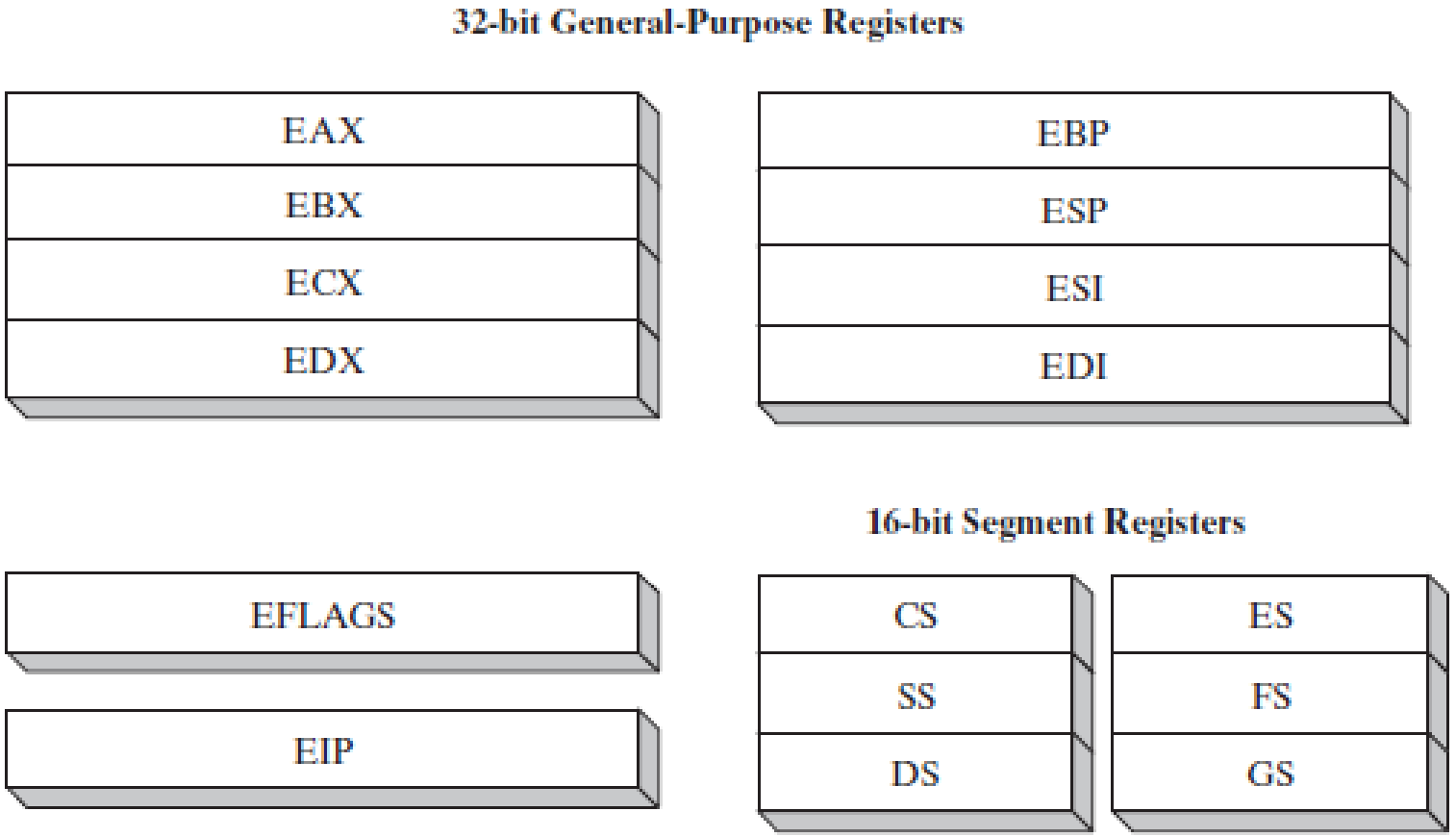
- In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes
  - Extended Physical Addressing allows a total of 64 GBytes of physical memory to be addressed
- Real-address mode programs, on the other hand, can only address a range of 1 MByte
- If the processor is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

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, six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP)

FIGURE 2–5 Basic Program Execution Registers.

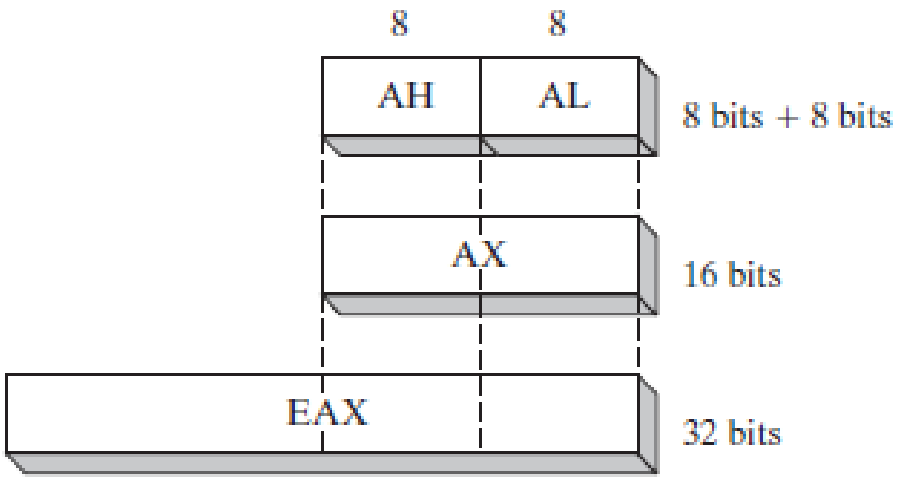


General-Purpose Registers:

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.



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

Some general-purpose registers have specialized uses:

- EAX (*accumulator) -* favored for arithmetic operations.
  - It is automatically used by multiplication and division instructions

- EBX (Base) - Holds base address for procedures and variables

- ECX - The CPU automatically uses ECX as a counter for looping operations

- EDX (Data) - Used in multiplication and division operations

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

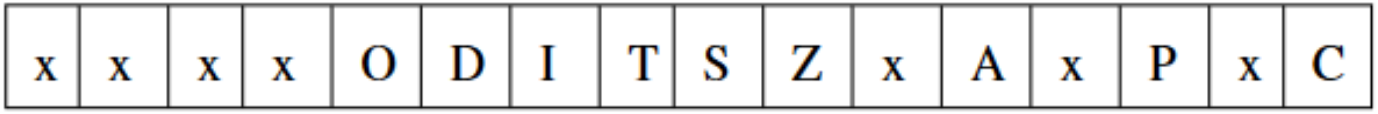
- EBP is used to reference function parameters and local variables on the stack

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



O = Overflow

D = Direction

I = Interrupt

T = Trap

x = undefined

S = Sign

Z = Zero

A = Auxiliary Carry

P = Parity

C = Carry

Flags

There are two types of flags: control flags (which determine how instructions are carried out) and status flags (which report on the results of operations.

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 hardware devices like the 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.

Status Flags

The Status flags reflect the outcomes of arithmetic and logical operations performed by the CPU.

- 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 - set when an operation causes a carry from bit 3 to 4 or borrow (frombit 4 to 3). 1 = carry, 0 = no carry.
  - ***Parity-*** is set if the least-significant byte in the result contains an even number of 1 bits. It used to verify memory integrity.