CSC 3210

Computer Organization and Programming

Chapter 2: x86 Processor Architecture

Dr. Zulkar Nine

mnine@gsu.edu

Georgia State University

Spring 2021

X86 Processor Architecture

- One step **before using assembly language**
  - What is the selected processor **Internal architecture and capabilities**.
- What **is the underline hardware** associated with X86?
- Assembly language is **a great tool** for learning **how a computer works**.
  - It require you to have working knowledge of **computer hardware**

You should have some basic knowledge about **the processor** and the **system architecture** in order to effectively program in **the assembly language**.

Outline

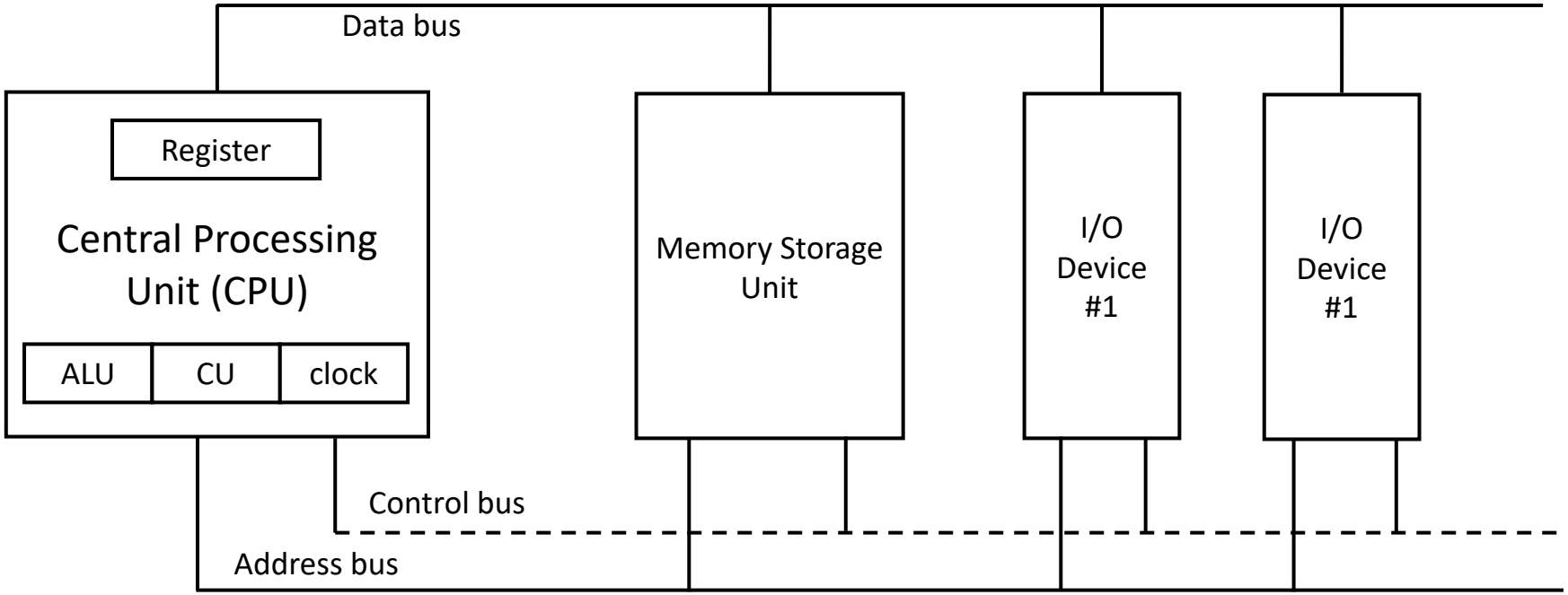
- **General Concepts**
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

General Concepts

- **Basic microcomputer design**
- Instruction execution cycle
- Reading from memory
- How programs run

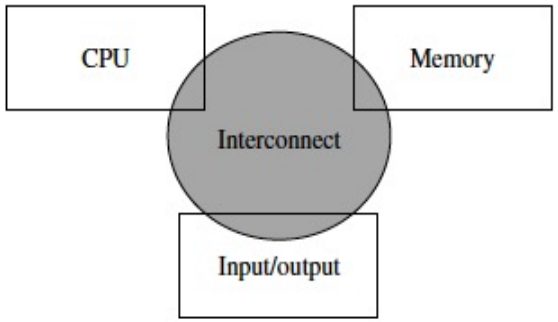
General Concepts: Basic Microcomputer Design

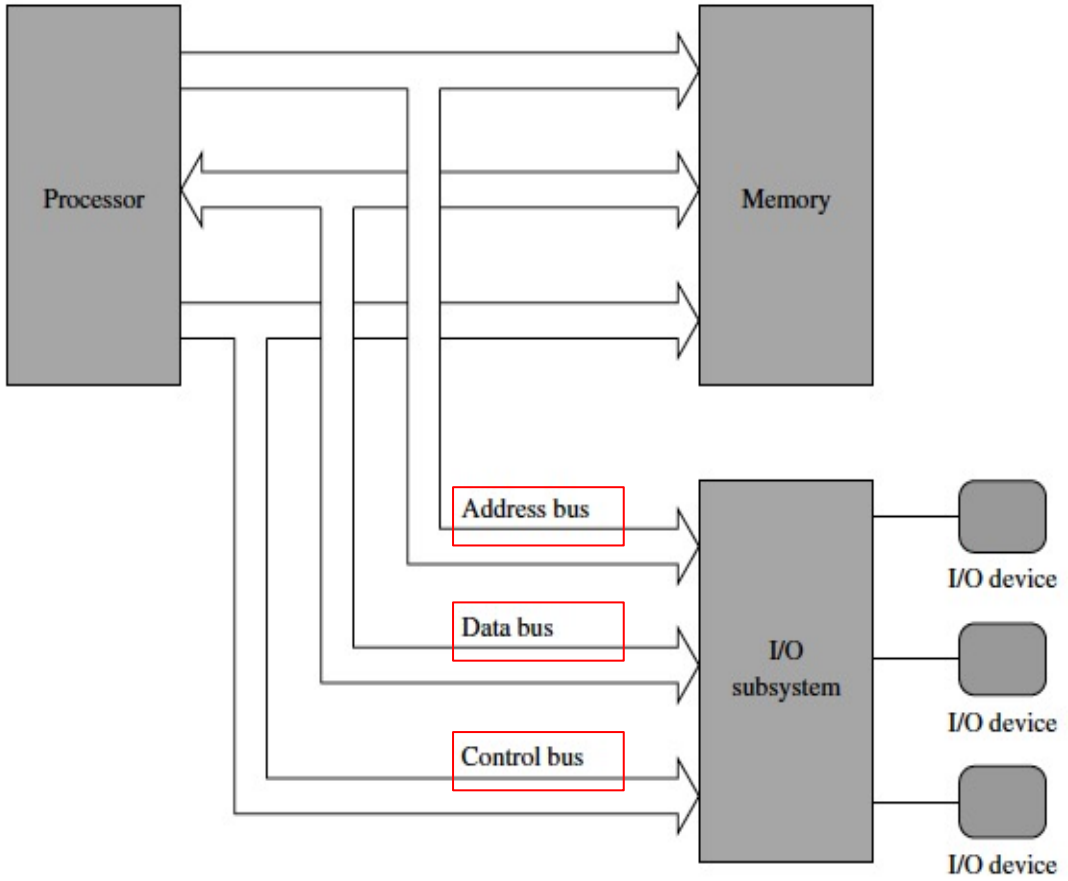
- **ALU** performs **arithmetic** and **logical** (bitwise) operations
- **Control unit (CU)** **coordinates** sequence of **execution steps**
- **Clock** synchronizes CPU **operations** with other **system components**



General Concepts: Basic Microcomputer Design

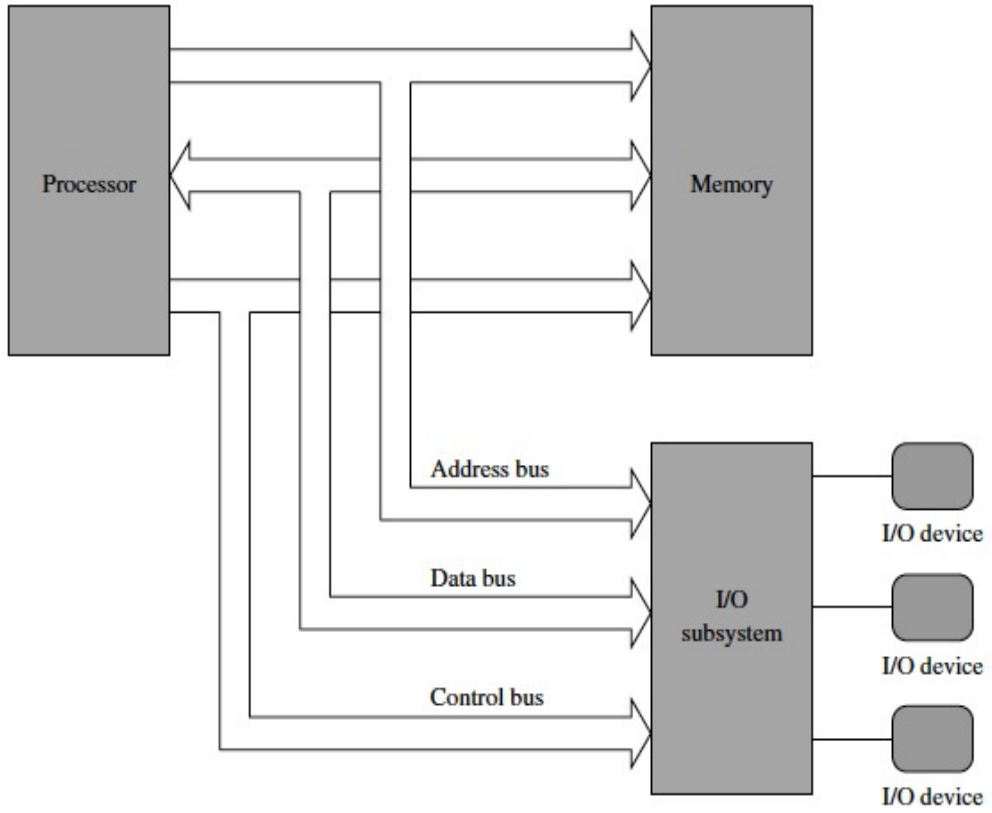
- A **bus**: a group of parallel wires that **transfer data**
  - bus types:
    - address
    - data
    - control





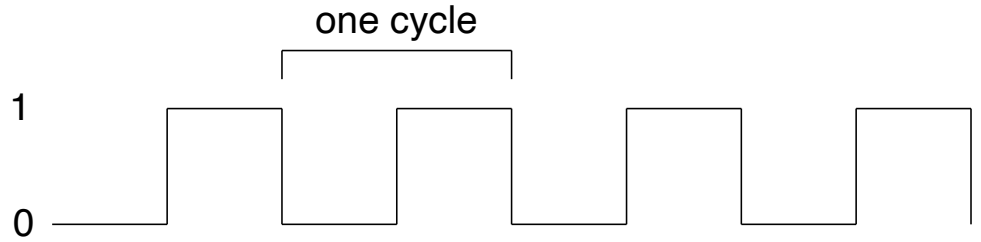
General Concepts: Basic Microcomputer Design

- The **Address bus** **holds the addresses** of instructions and data, when the currently executing instruction transfers data between the CPU and memory.
- The **Data bus** **transfers** **instructions** and **data** between the CPU and memory.
- The **Control bus** uses binary signals to **synchronize actions** of all devices attached to the system bus.



General Concepts: Clock

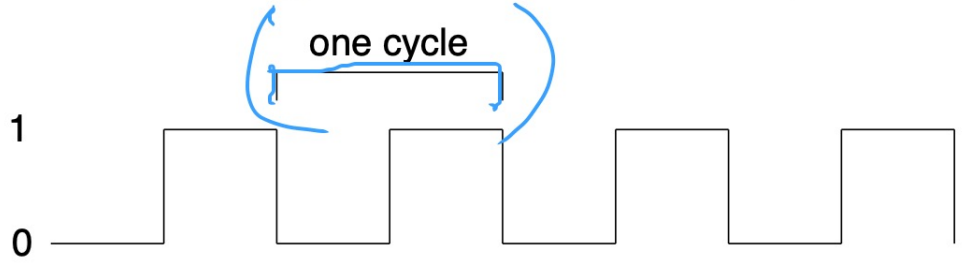
- The system clock provides **a timing signal** to synchronize the **operations** of the system.
  - Synchronizes all CPU and BUS **operations**
- A **clock** is a sequence of **1's** and **0's**



The frequency:
is the number of cycles that happens each second

General Concepts: Clock

- The clock frequency is measured in the number of cycles per second.
- This number is referred to as **Hertz** (Hz: the unit of frequency, defined as one cycle per second).
  - **MHz** and **GHz** represent 10^6 and 10^9 cycles per second
- The **system clock** defines ~~**the speed**~~ at which the system operates.



10^6 cycle → 1 sec.

1 cycle → $\frac{1}{10^6}$ sec. (clock period)

General Concepts: Clock

- **Ex: transfer of data** from a memory location to X86 (Pentium) takes **three clock cycles**.
- The **clock period** is defined as the length of time taken by one clock cycle .

$$\text{Clock period} = \frac{1}{\text{Clock frequency}}$$

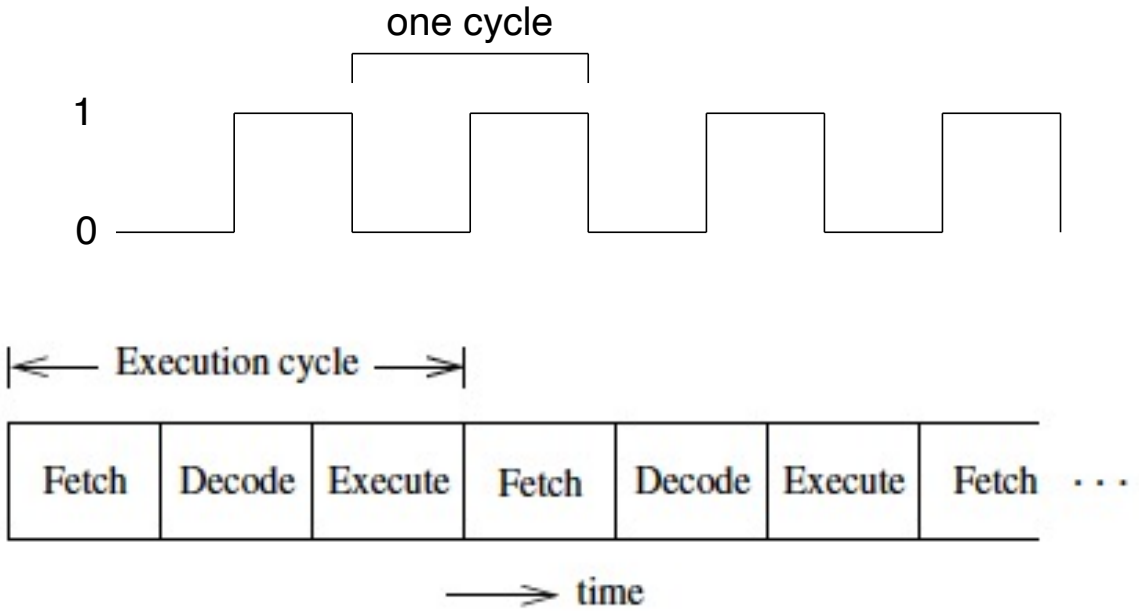
For example, a clock frequency of 1 GHz yields a clock period of

$$\frac{1}{1 \times 10^9} = 1 \text{ ns}$$

- If it takes three clock cycles to execute an instruction, it takes 3 X 1 ns = 3 ns.
- Machine **(clock) cycle** measures time of a single operation
- Clock is used to trigger events

General Concepts: Clock

- A **machine instruction** requires one clock cycle to execute, few require 50 clocks
- Instructions require memory access: Empty clock cycle, **wait states, Why?**
  - CPU, system bus, and memory circuits



Clock per Instruction (CPI)

- Is an effective average.
- It is the average number of clocks required by the instructions in a program.
- In a program 60% instructions takes 4 clock cycles and the rest of the instructions takes 1 clock cycles.
- CPI = 0.6 * 4 + 0.4 * 1 = 2.8 clocks per instruction.

Million Instructions Per Second

- **Step 1:** Perform Divide operation between no. of instructions and Execution time.
- **Step 2:** Perform Divide operation between that variable and 1 million for finding millions of instructions per second.
- For example,
  - if a computer completed 2 million instructions in 0.10 seconds
  - 2 million/0.10 = 20 million.
  - No of MISP=20 million/1 million
  - =20

An Example

- An instruction on average takes 4 clock cycles to execute. A program with these instructions take 5 seconds to run on a 1.2 GHz processor. How many instructions the program have?

General Concepts

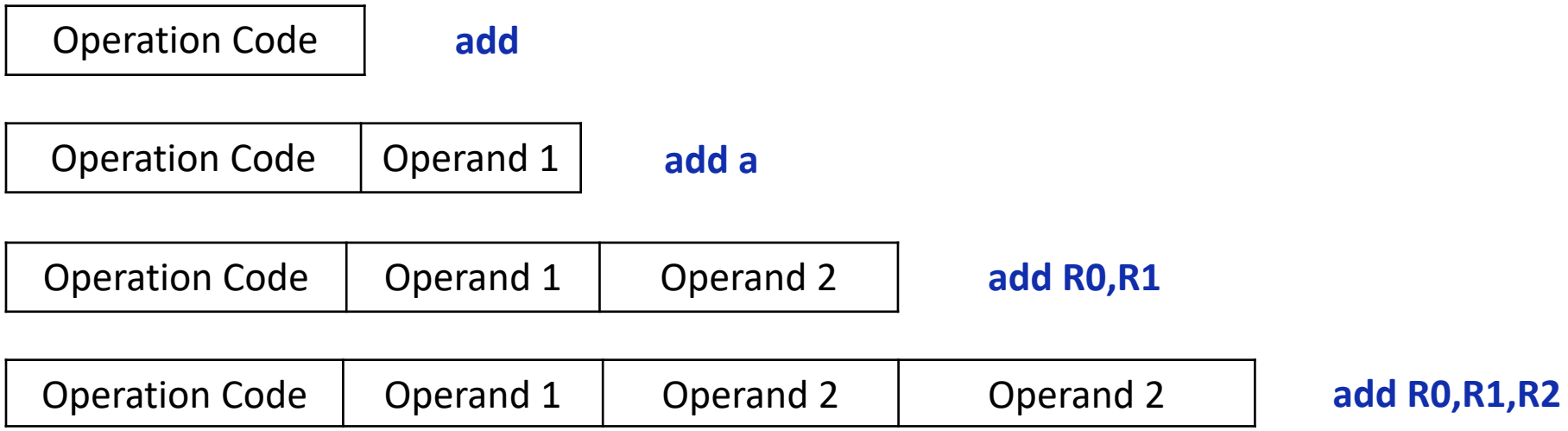
- Basic microcomputer design
- **Instruction execution cycle**
- Reading from memory
- How programs run

Instruction Execution Cycle

- An instruction is a **binary pattern designed inside a microprocessor** to perform a specific function.
- The entire group of instructions that a microprocessor supports is called **Instruction Set**.
- 8086 has more than **20,000** instructions.
- **Classification of Instruction Set**
  - Data Transfer Instructions: mov, push, pop,…
  - Arithmetic Instructions: add, sub, inc …
  - Bit Manipulation Instructions: and, or, xor, ….
  - Program Execution Transfer Instructions: jmp, call, ret, ….
  - String Instructions: cmps, movs, rep, …
  - Processor Control Instructions: stc, clc, wait…

Instruction Execution Cycle

- **Instruction format**
  - An instruction consists of an **opcode**, usually with some additional information like where operands come from, and where results go.

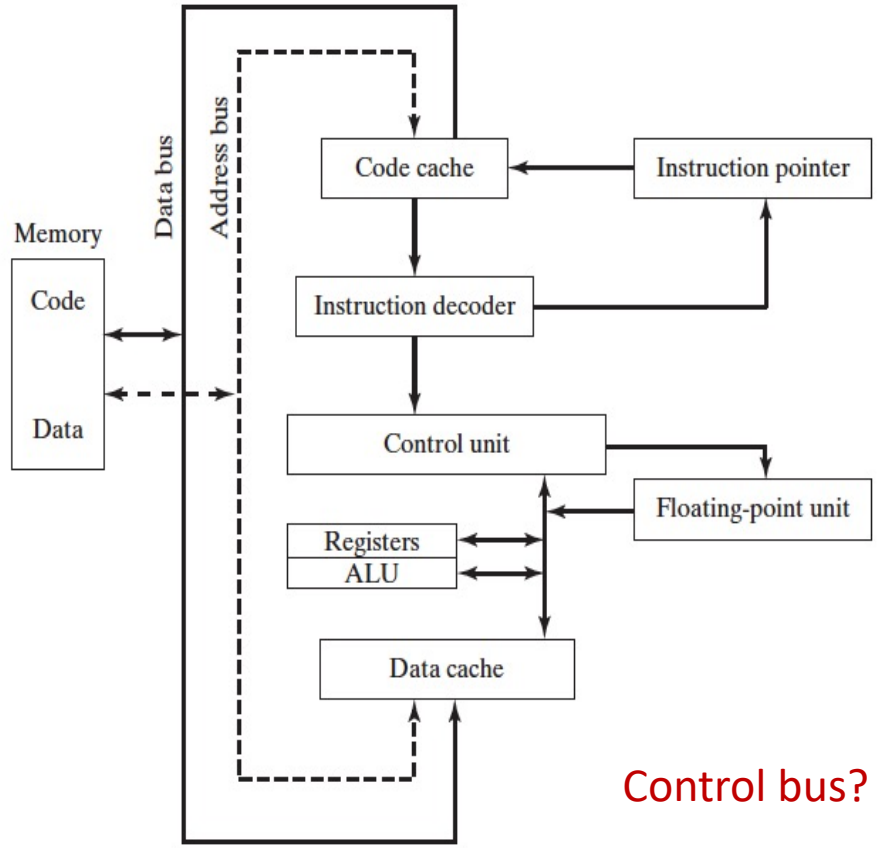


  - An **operand** can be register, memory location or immediate (ex. mov 5,R0) operand

Instruction Execution Cycle

- Predefined sequence of steps to execute a machine instruction
- Simple IEC: **Fetch**, **Decode**, **Execute**
  - **Fetch**
  - **Decode**
  - **Fetch operands** (not always needed?)
    - Address calculation?
  - **Execute**
    - Update few status flags: zero, carr overflow
  - **Store output** (not always needed?)

FIGURE 2–2 Simplified CPU block diagram.



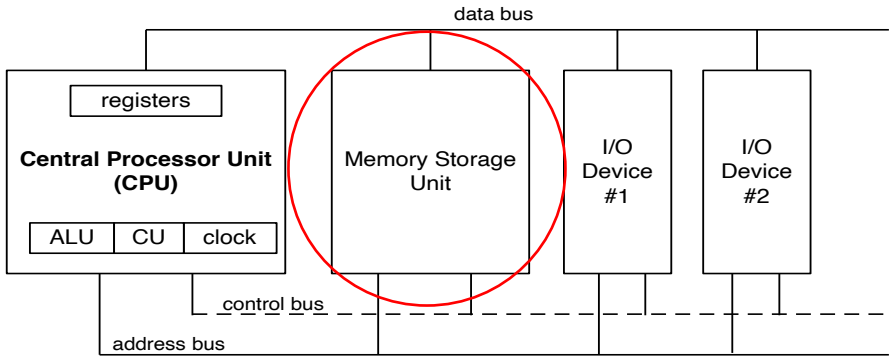
Control bus?

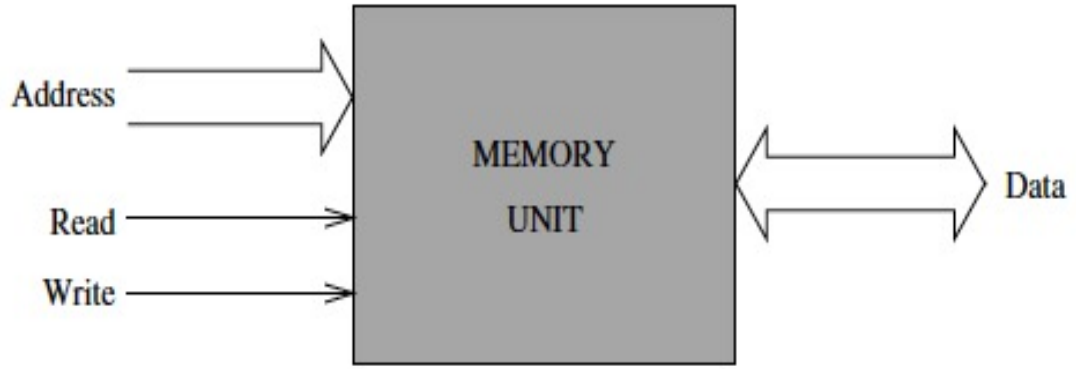
General Concepts

- Basic microcomputer design
- Instruction execution cycle
- **Reading from memory**
- How programs run

Memory

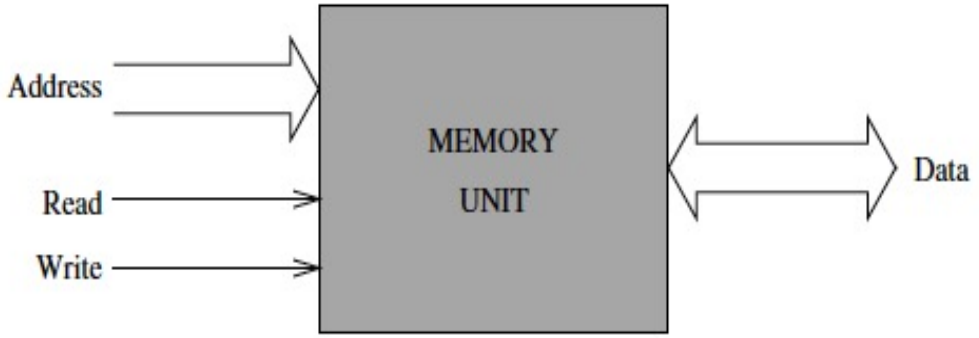
- The memory unit supports <u>two fundamental operations</u>: **read and write** .
  - The **read operation** reads a previously stored data
  - The **write operation** stores a value in memory.



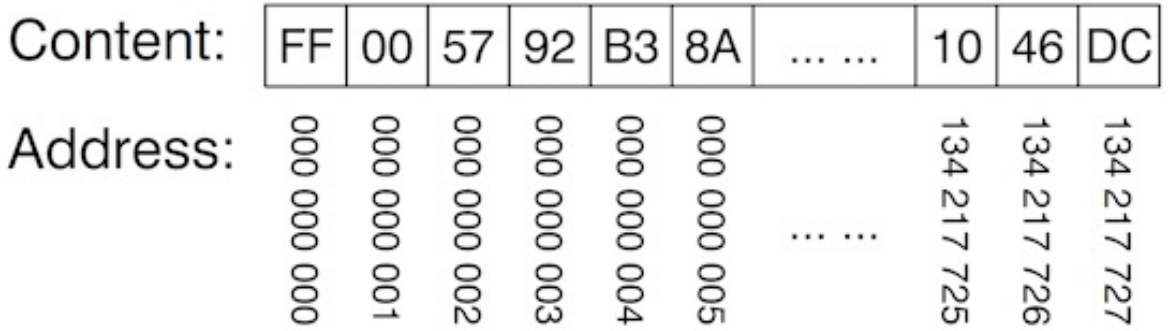


- Both of these operations require an **address** in memory from which to read a value or to which to write a value.

Memory

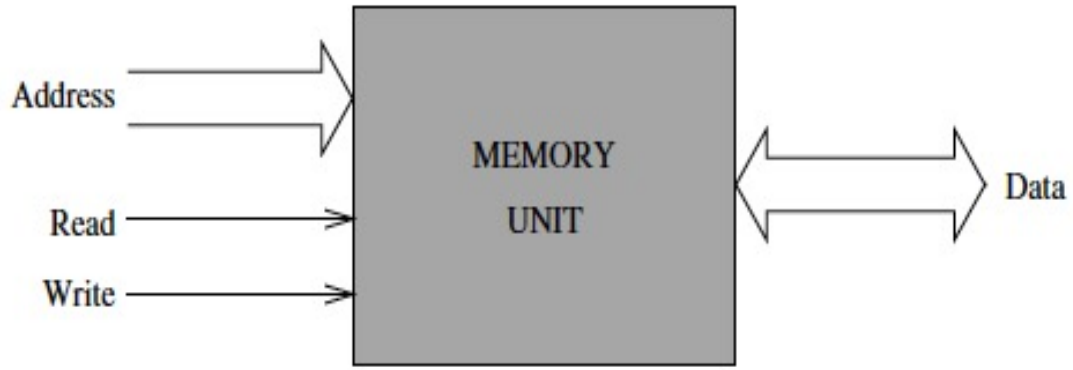


Memory (RAM) as an array of bytes



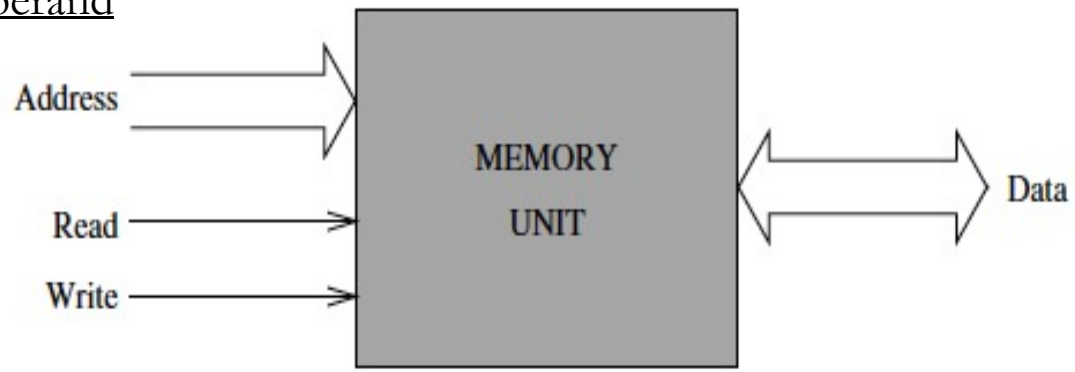
Memory

- write operation requires **specification** of the data to be written.
- The read and write signals come from the control bus.



Memory: **Reading from Memory**

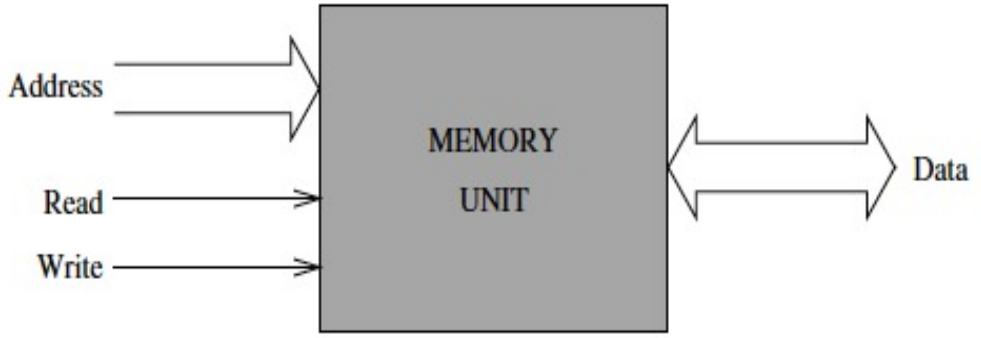
- **Multiple machine cycle** are required when **reading** from memory, Why?
  - Because **it** responds much more slowly than the CPU.
- Steps in a typical **read cycle**
  1. **Place** the address of the value you want to read on the address bus.
  2. **Assert** (changing the value of) the processor's **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



Memory: **Writing to Memory**

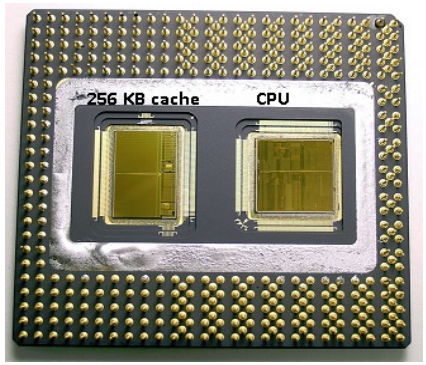
- Steps in a typical **write cycle:**

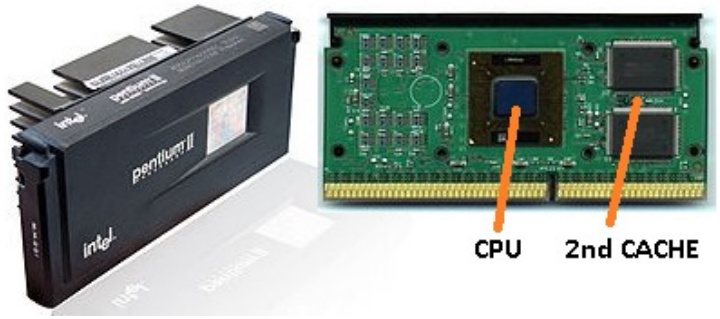
1. **Place** the address of the location to be written on the address bus,
2. **Place** the data to be written on the data bus,
3. **Assert** (changing the value of) the processor's **WR** (write) pin.
4. **Wait** for the memory to store the data at the addressed location

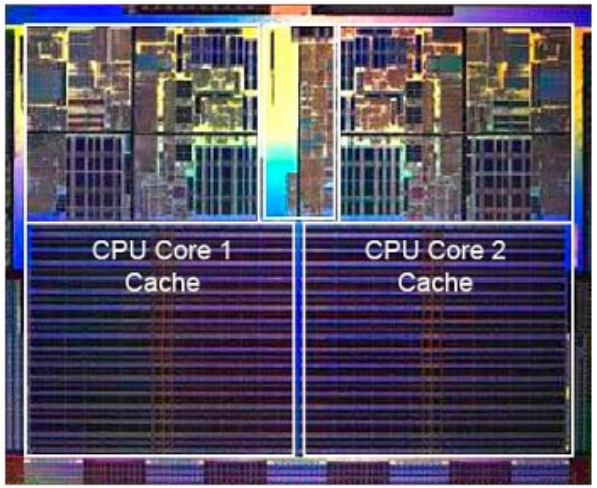


Reading from Memory: Cache Memory

- In practice, **instructions and data** are **not fetched**, most of the time, from the **main memory**.
- There is a **high-speed cache memory** that provides
  - faster access to **instructions** and **data** than the main memory.

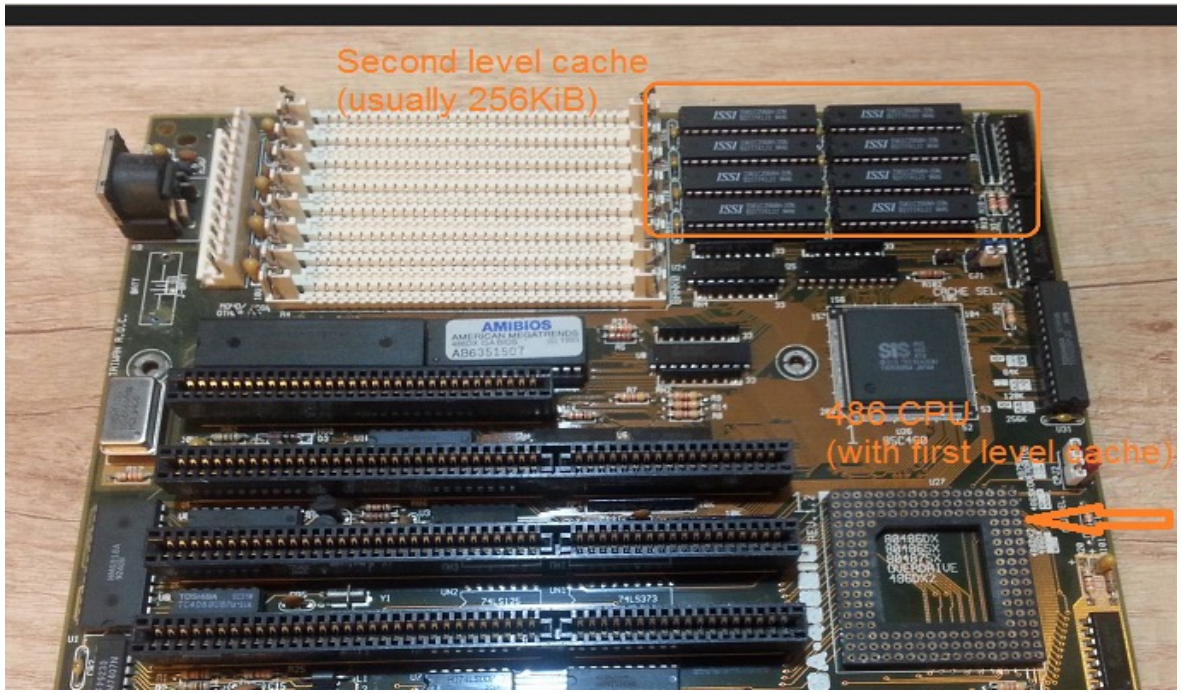






Reading from Memory: Cache Memory

- **Level-1 cache**: inside the CPU



- **Level-2 cache**: outside the CPU (attached to CPU by high speed data bus)

Reading from Memory: Cache Memory

- **Cache hit**: when data to be read is already in cache memory
- **Cache miss**: when data to be read is not in cache memory.

General Concepts

- Basic microcomputer design
- Instruction execution cycle
- Reading from memory
- **How programs run**

How a Program Runs

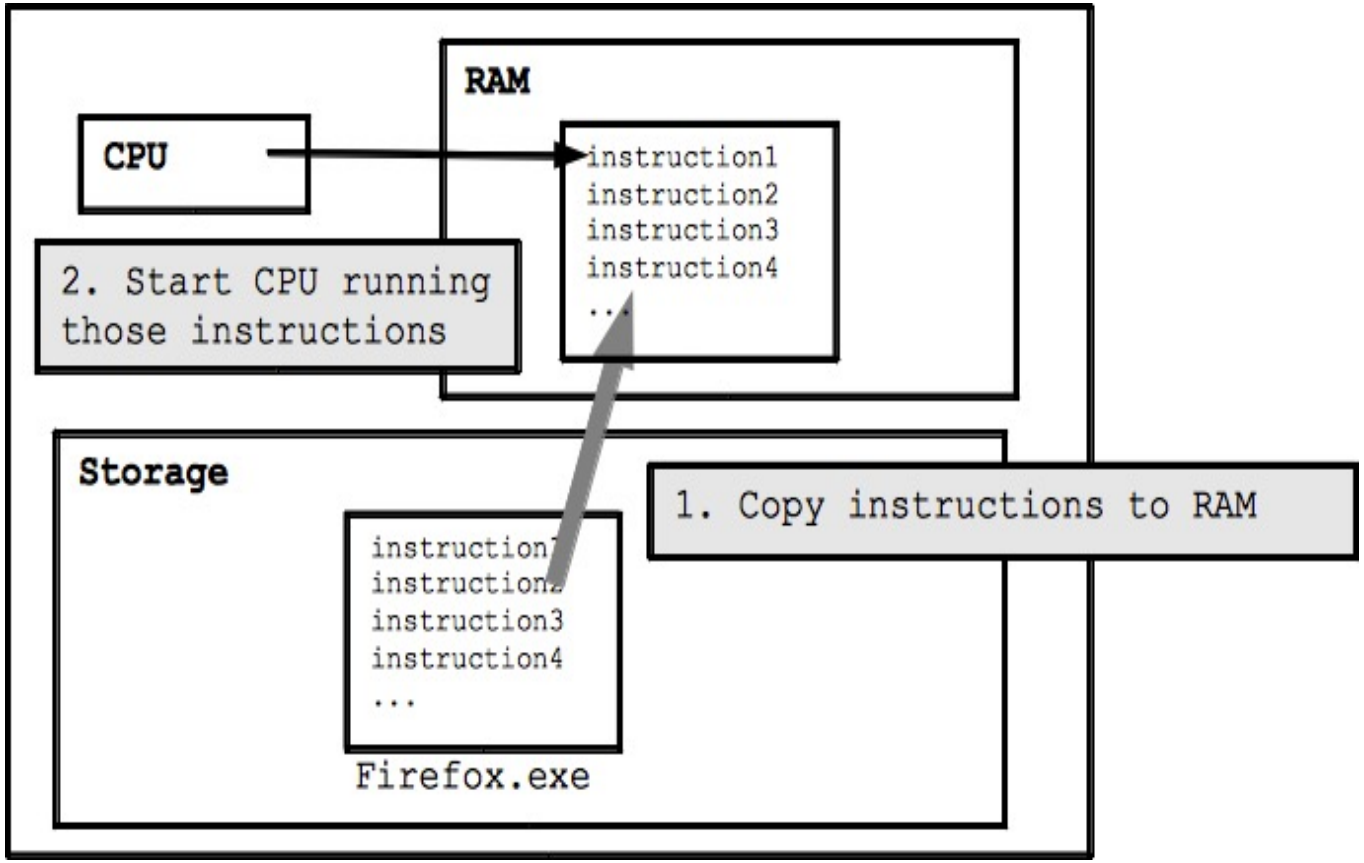
Program loaded: Hard disk drive

loader

RAM

OS points to program **entry point**

CPU



Think Again?

- Why does memory access take more machine cycles than register access?
- What are the three basic steps in the instruction execution cycle?
- Which two additional steps are required in the instruction execution cycle when a memory operand is used?

Outline

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

Basic Execution Environment

- **Addressable memory**
- General-purpose registers
- Index and base registers
- Specialized register uses
- Status flags
- Floating-point, MMX, XMM registers

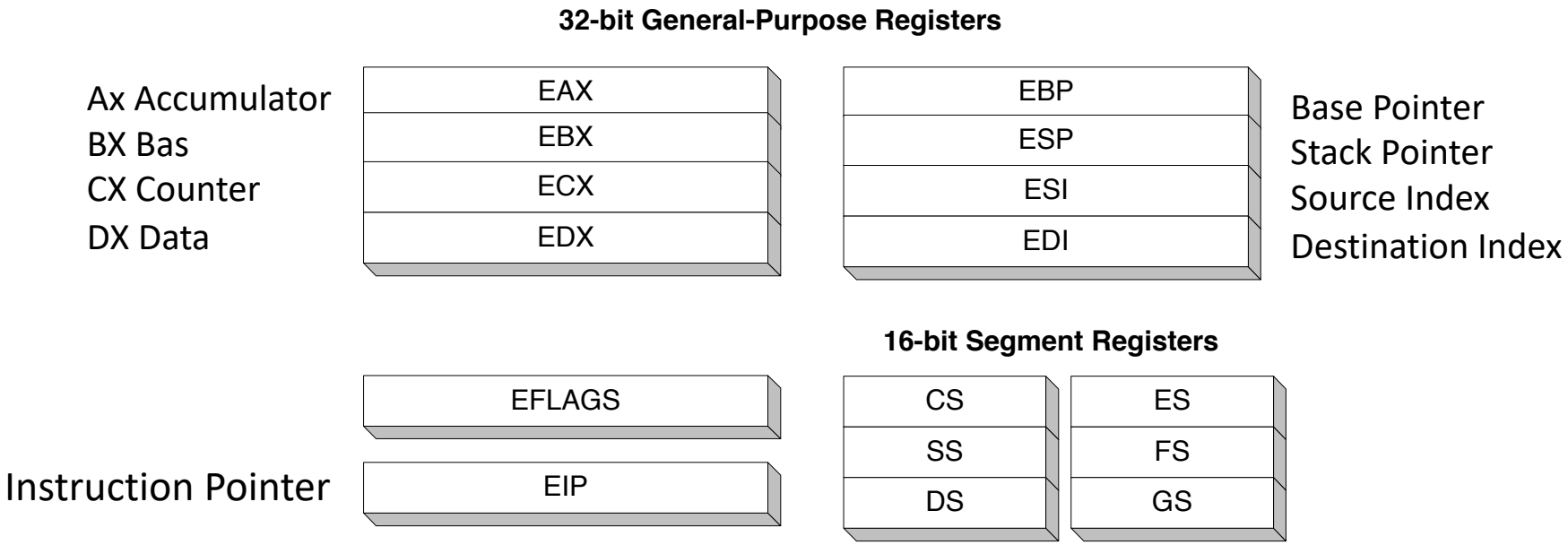
Basic Execution Environment: Addressable Memory

- **Address Space**
  - **Protected mode**
    - 4 GB space
    - 32-bit address
  - **Real-address** and **Virtual-8086 modes**
    - 1 MB space
    - 20-bit address

Basic Execution Environment: **General-Purpose** Registers

- Registers are high speed **storage locations** inside the CPU.

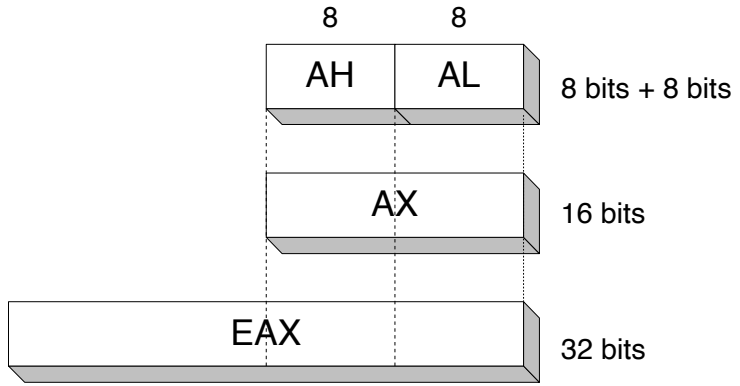
E for Extended?



Basic Execution Environment: **Accessing** Parts of Registers

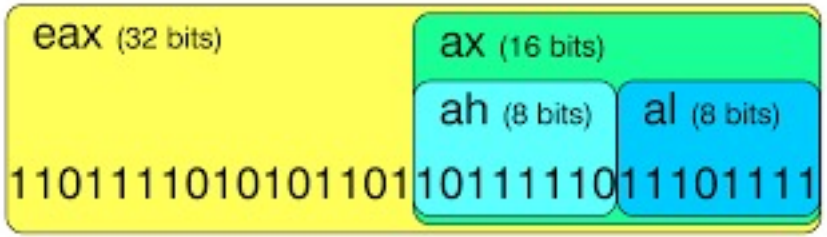
- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to **EAX**, **EBX**, **ECX**, and **EDX**

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



Ex:

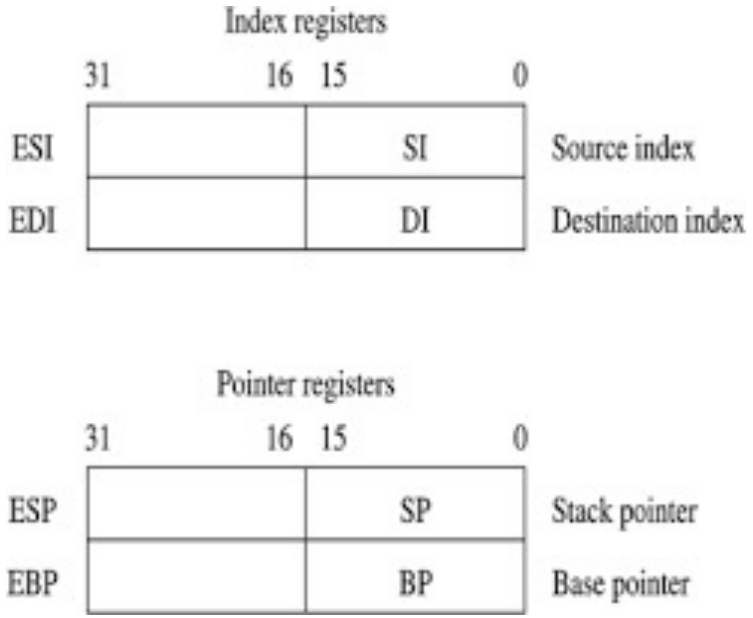
Register aliasing / sub-registers



Basic Execution Environment: **Index** and **Base** Registers

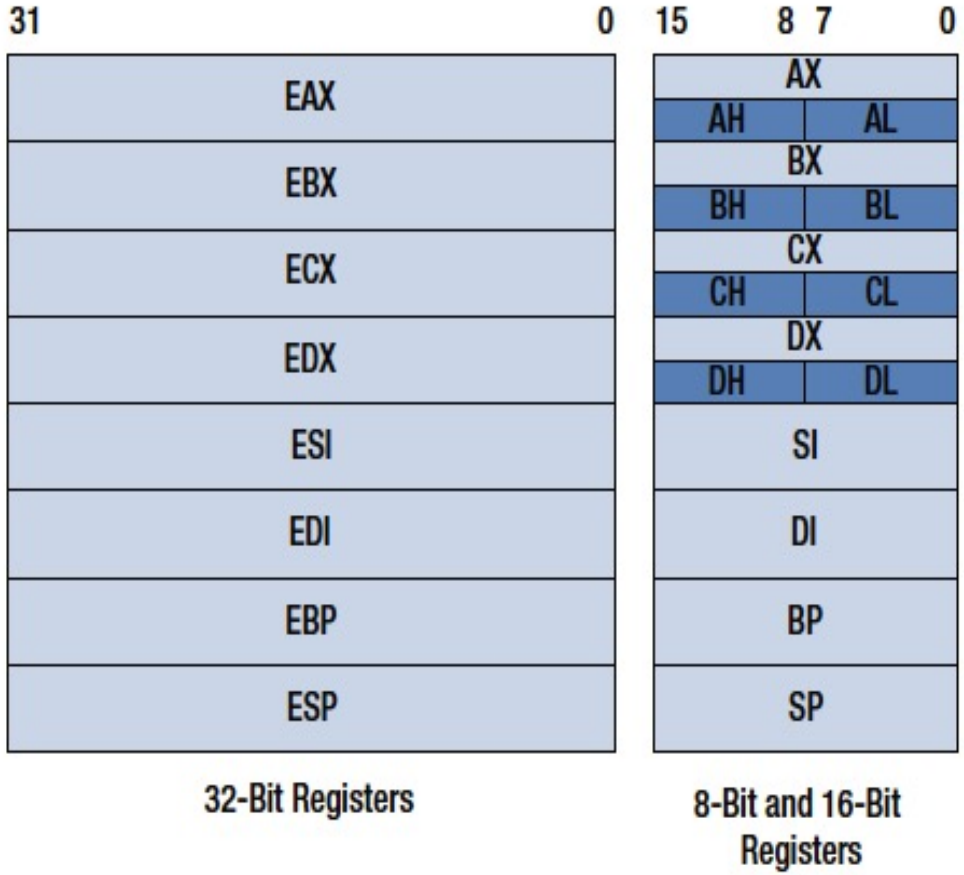
- Some registers have only a 16-bit name for their lower half:

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



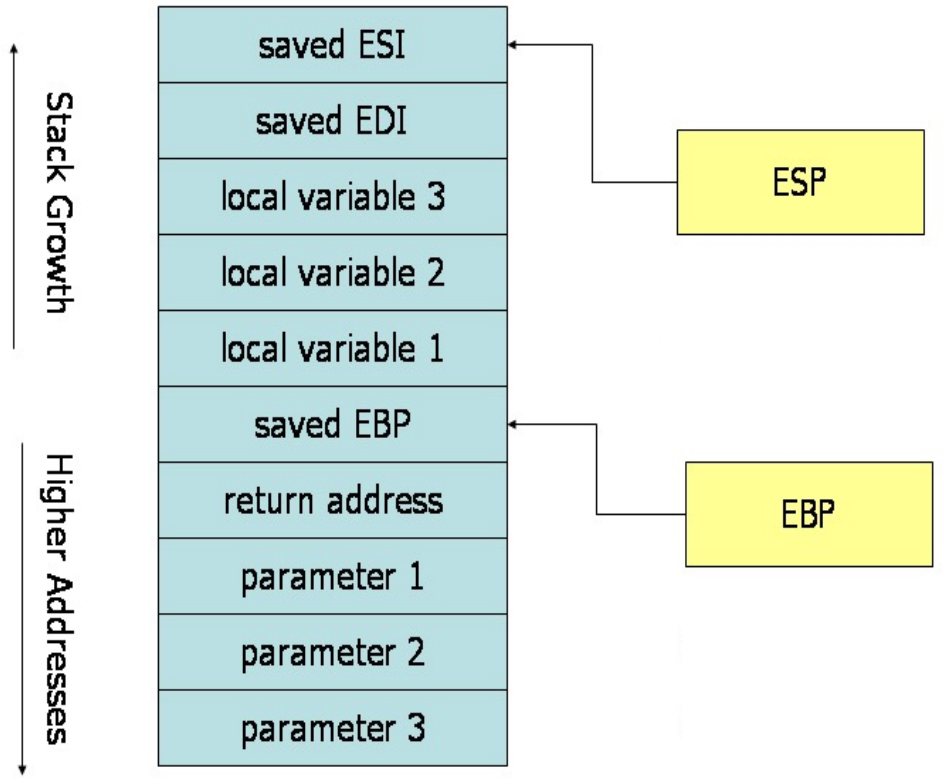
Basic Execution Environment: Some **Specialized** Register Uses

- **General-Purpose**
  - **EAX – accumulator:** automatically used by multiplication and division instructions
  - **ECX – loop counter:** contain the loop count value for iterative instructions
  - **ESI, EDI – index registers:** used by high-speed memory transfer instructions.



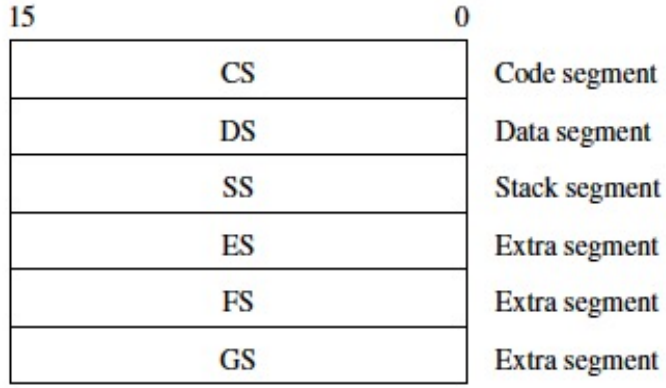
Basic Execution Environment: Some **Specialized** Register Uses

- **General-Purpose**
  - **ESP – stack pointer:** addresses data on the stack, rarely used for ordinary arithmetic or data transfer.
  - **EBP – frame pointer (stack):** used by high-level languages to reference function parameters and local variables on the stack.



Basic Execution Environment: Some **Specialized** Register Uses

- **Segment**:
  - Indicate **base addresses** of preassigned memory areas.
  - The six segment registers point to where these segments are located in the memory.
  - **CS – code segment:** hold program instructions
  - **DS – data segment:** hold variables
  - **SS – stack segment:** holds local function variables and function parameters.
  - **ES, FS, GS - additional segments:** can be used in a similar way as the other segment registers.

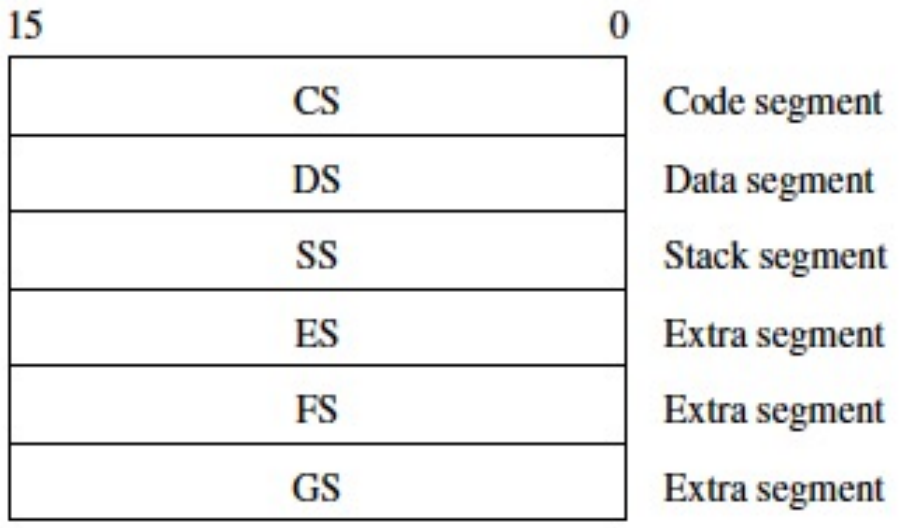


support the segmented memory organization

Basic Execution Environment: Some **Specialized** Register Uses

- **ES, FS, GS - additional segments**

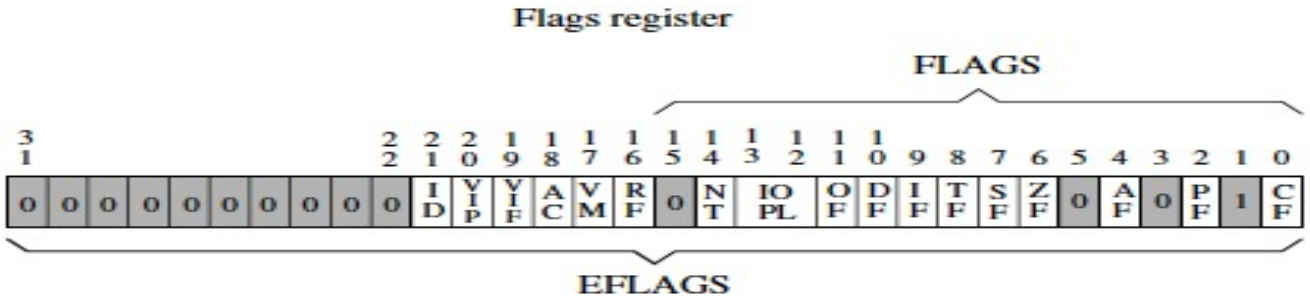
- **For example**,
  If a program's data **could not fit** into a single **data segment**, we could use two segment registers to point to the two data segments.



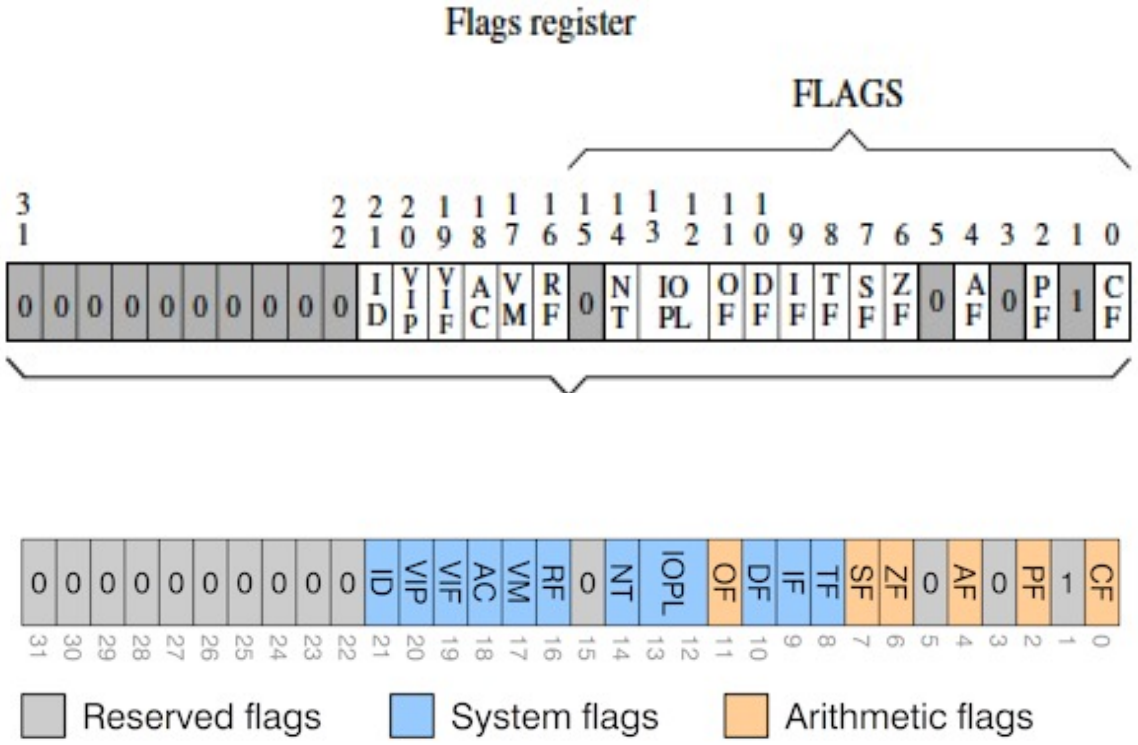
Basic Execution Environment: Some **Specialized** Register Uses

- **EIP –** instruction pointer (also called **program counter**)**:** contains the address of the **next instruction to be executed** .
- **EFLAGS-** a register consists of **individual binary bits** that control the **operation** of the CPU or reflect **the outcome** of **some CPU operation.**
  - **status** and **control** flags
  - Each flag is a single binary bit

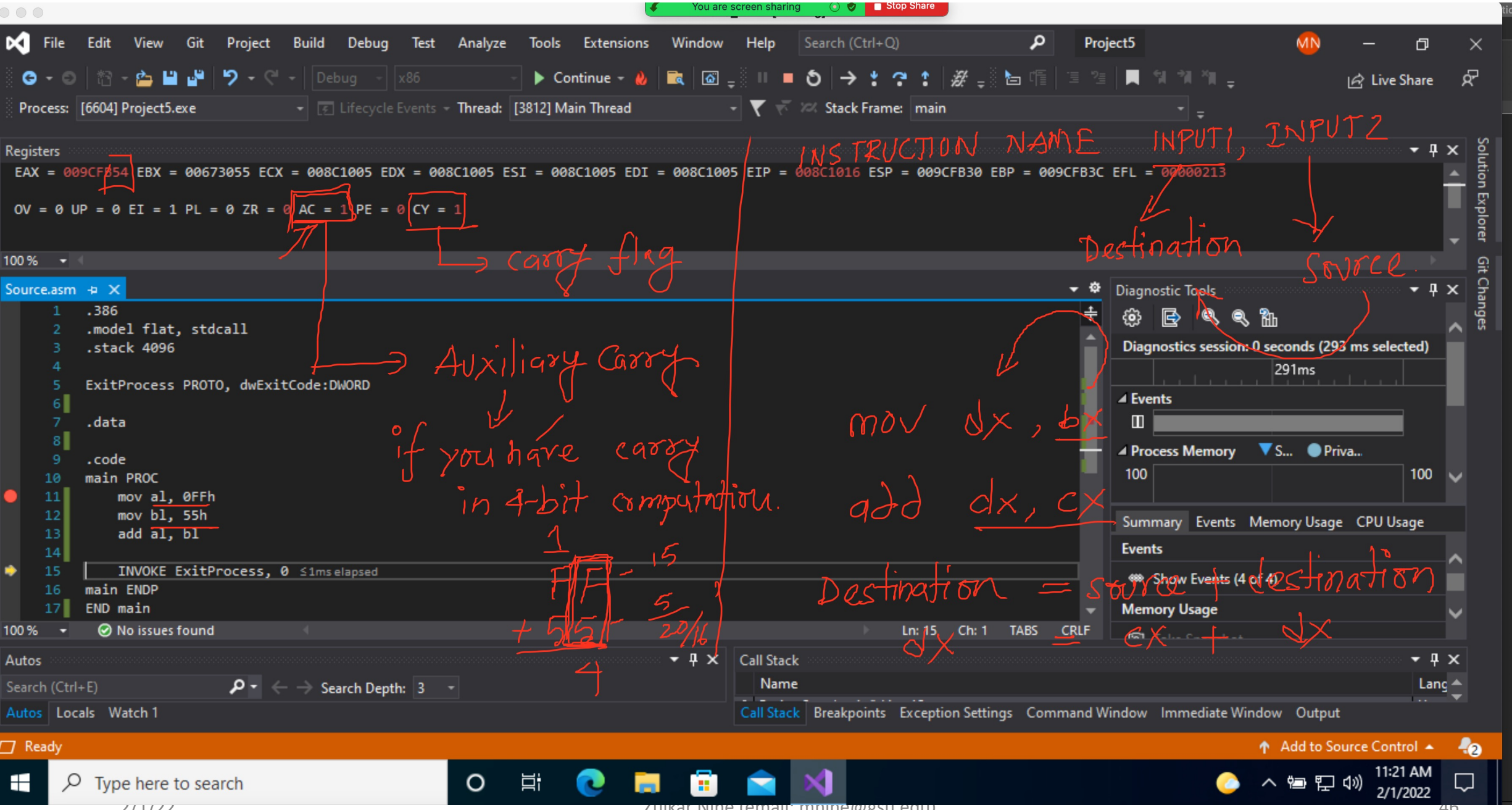
A flag is **set** when it equals 1; it is **clear** (or reset) when it equals 0.



Flags Table

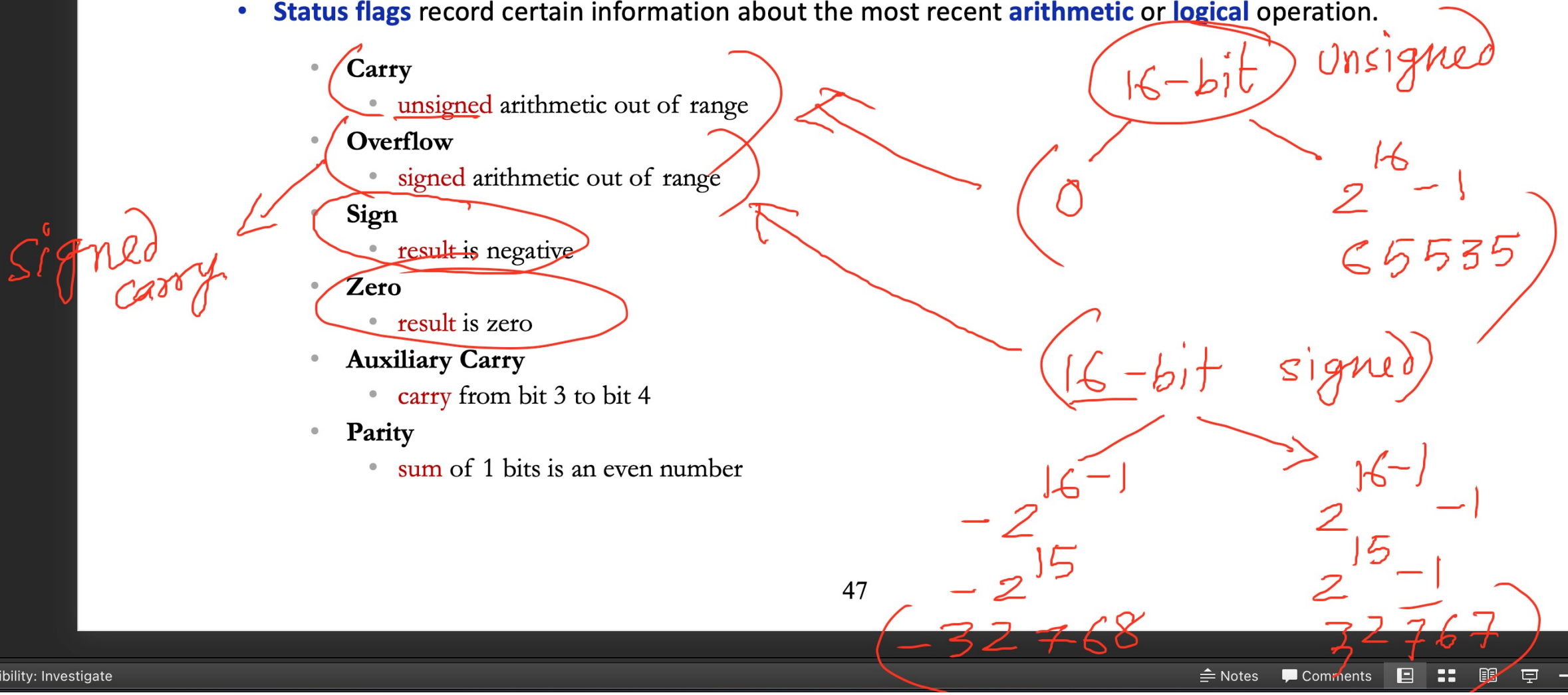


Bit	Name	Symbol	Use
0	Carry Flag	CF	Status
1	Reserved		1
2	Parity Flag	PF	Status
3	Reserved		0
4	Auxiliary Carry Flag	AF	Status
5	Reserved		0
6	Zero Flag	ZF	Status
7	Sign Flag	SF	Status
8	Trap Flag	TF	System
9	Interrupt Enable Flag	IF	System
10	Direction Flag	DF	Control
11	Overflow Flag	OF	Status
12	I/O Privilege Level Bit 0	IOPL	System
13	I/O Privilege Level Bit 1	IOPL	System
14	Nested Task	NT	System
15	Reserved		0
16	Resume Flag	RF	System
17	Virtual 8086 Mode	VM	System
18	Alignment Check	AC	System
19	Virtual Interrupt Flag	VIF	System
20	Virtual Interrupt Pending	VIP	System
21	ID Flag	ID	System
22 - 31	Reserved		0


You are screen sharing
Stop Share
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project5
Debug
x86
Continue
Live Share
Process: [6604] Project5.exe
Lifecycle Events
Thread: [3812] Main Thread
Stack Frame: main
Registers
EAX = 009CFB54 EBX = 00673055 ECX = 008C1005 EDX = 008C1005 ESI = 008C1005 EDI = 008C1005 EIP = 008C1016 ESP = 009CFB30 EBP = 009CFB3C EFL = 00000213
OV = 0 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 1 PE = 0 CY = 1
100 %
Source.asm
1 .386
2 .model flat, stdcall
3 .stack 4096
4
5 ExitProcess PROTO, dwExitCode:DWORD
6
7 .data
8
9 .code
10 main PROC
11 mov al, 0FFh
12 mov bl, 55h
13 add al, bl
14
15 INVOKE ExitProcess, 0 ≤1ms elapsed
16 main ENDP
17 END main
100 %
No issues found
Ln: 15 Ch: 1 TABS CRLF
Diagnostic Tools
Diagnostics session: 0 seconds (293 ms selected)
291ms
Events
Process Memory
100
100
Summary Events Memory Usage CPU Usage
Events
Show Events (4 of 4)
Memory Usage
Autos
Search (Ctrl+E)
Search Depth: 3
Autos Locals Watch 1
Call Stack
Name
Call Stack Breakpoints Exception Settings Command Window Immediate Window Output
Ready
Add to Source Control
Type here to search
11:21 AM
2/1/2022
carry flag
Auxiliary Carry
if you have carry in 4-bit computation.
INSTRUCTION NAME
INPUT1, INPUT2
Destination
Source
mov dx, bx
add dx, cx
Destination = source + destination
dx
cx + dx

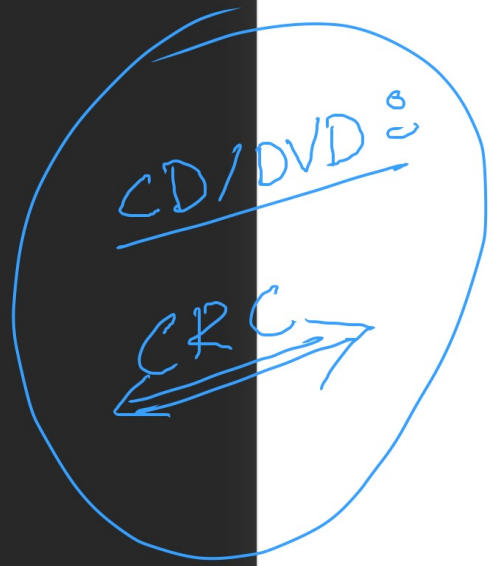
Basic Execution Environment: Status Flags

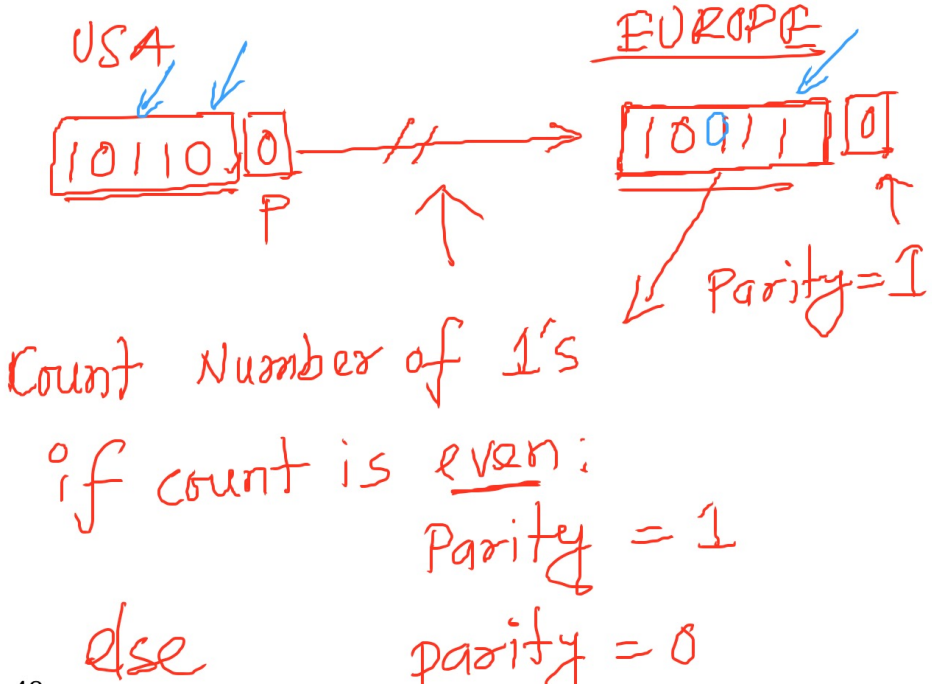
- **Status flags** record certain information about the most recent **arithmetic** or **logical** operation.
  - **Carry**
    - unsigned arithmetic out of range
  - **Overflow**
    - signed arithmetic out of range
  - **Sign**
    - result is negative
  - **Zero**
    - result is zero
  - **Auxiliary Carry**
    - carry from bit 3 to bit 4
  - **Parity**
    - sum of 1 bits is an even number



Basic Execution Environment: Status Flags

- **Status flags** record certain information about the most recent **arithmetic** or **logical** operation.
  - **Carry**
    - unsigned arithmetic out of range
  - **Overflow**
    - signed arithmetic out of range
  - **Sign**
    - result is negative
  - **Zero**
    - result is zero
  - **Auxiliary Carry**
    - carry from bit 3 to bit 4
  - **Parity**
    - sum of 1 bits is an even number





Basic Execution Environment: Some **Specialized** Register Uses

General-Purpose (Note)

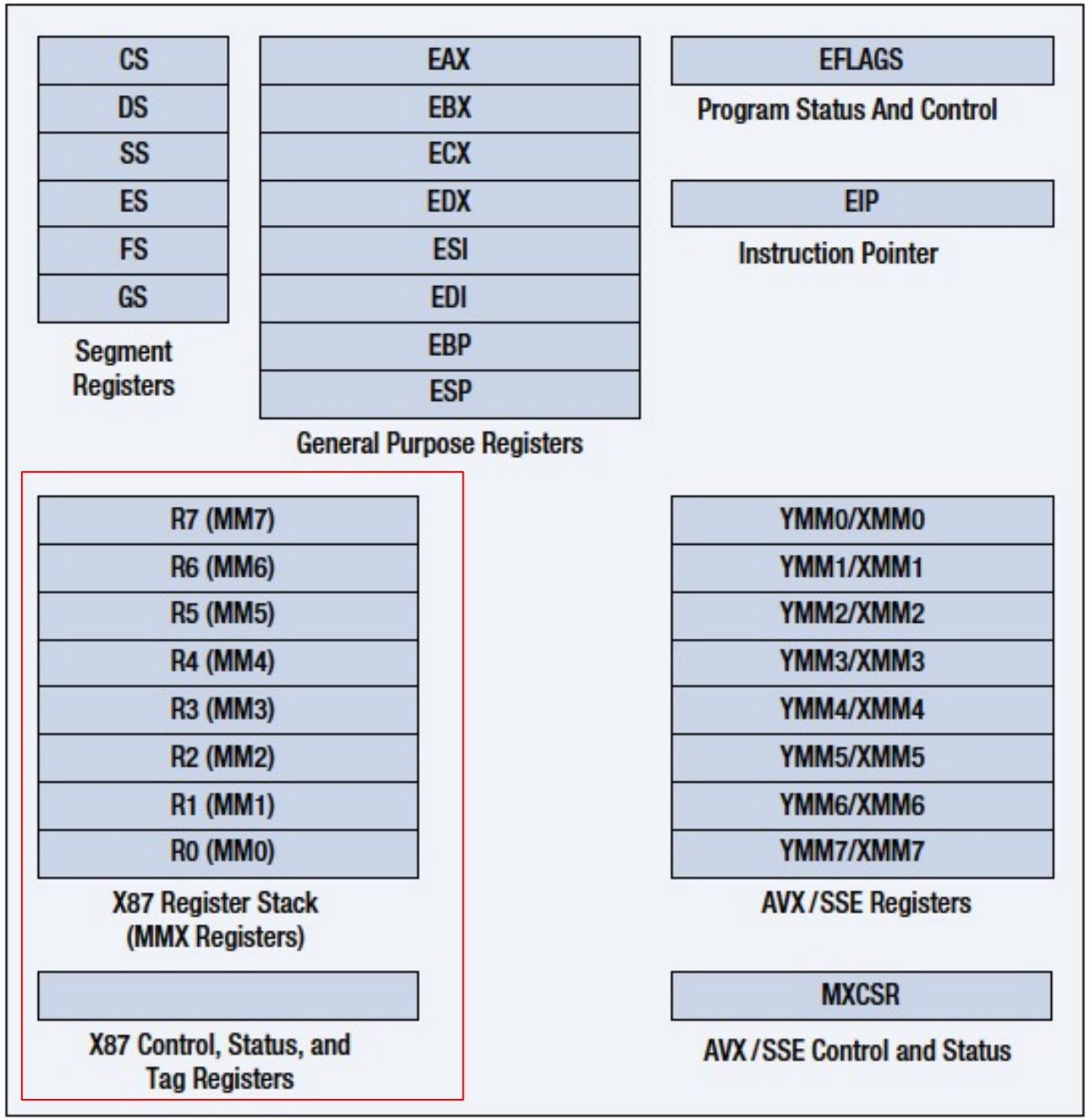
- Despite their designation as general-purpose registers, there restrictions on how they can be used.
- Many instructions either **require** or **implicitly** use specific registers as operands.
- **Ex**:
  - Some variations of the **imul** (**Signed** Multiply) and **idiv** (**Signed** Divide) instructions use the **EDX** register to hold the high-order **doubleword** of a product or dividend.
  - The **string instructions** require that the addresses of the source and destination operands be placed in the **ESI** and **EDI** registers, respectively.

Basic Execution Environment: Some **Specialized** Register Uses

General-Purpose

- The processor uses the **ESP** register to support stack-related operations such as **function calls and returns**.
- Register **EBP** is typically used as a base pointer to access data items that are stored on the stack

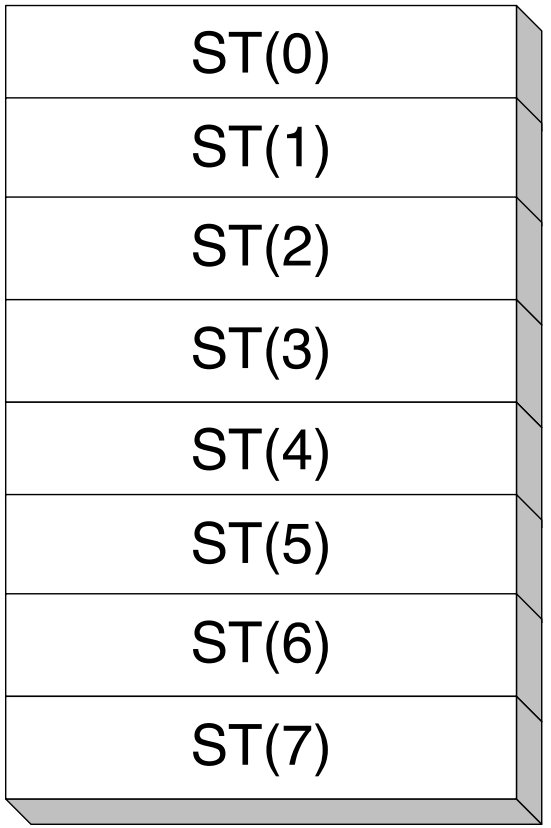
- Given the limited number general-purpose registers available in x86,
  - It is frequently necessary to **use** a general-purpose register in a **non-conventional manner**.
  - x86 **assemblers** do not enforce these usage conventions.


CS
DS
SS
ES
FS
GS
Segment Registers
EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP
General Purpose Registers
EFLAGS
Program Status And Control
EIP
Instruction Pointer
R7 (MM7)
R6 (MM6)
R5 (MM5)
R4 (MM4)
R3 (MM3)
R2 (MM2)
R1 (MM1)
R0 (MM0)
X87 Register Stack (MMX Registers)
X87 Control, Status, and Tag Registers
YMM0/XMM0
YMM1/XMM1
YMM2/XMM2
YMM3/XMM3
YMM4/XMM4
YMM5/XMM5
YMM6/XMM6
YMM7/XMM7
AVX/SSE Registers
MXCSR
AVX/SSE Control and Status

Attendance!

Basic Execution Environment: MMX, XMM, Floating-Point Registers

- Eight 64-bit MMX registers: MMX instructions operates on parallel
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations
- **Floating-Point**: Eight **80-bit** floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - arranged in a stack
  - used for all **floating-point arithmetic**



Outline

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

IA-32 Memory Management

- **Real-address mode**
- **Protected mode**
- **Virtual-8086**

Real-address Mode

- Only **1 M Byte of memory can be addressed**,
- The processor **can run only one program at a time**,
  - but it **can momentarily interrupt that program to process requests** (called **interrupts**) from peripherals.
- **Application programs** are permitted to access **any memory location**,
  - including **addresses that are linked directly to system hardware**.
- The **MS-DOS operating system runs in real-address mode**, and Windows 95 and 98 can be booted into this mode.

Protected Mode

- The processor can **run multiple programs at the same time**.
  - It assigns **each process** (running program) **a total of 4 GByte of memory**.
  - Each program can be assigned **its own reserved memory area**,
    - and **programs are prevented** from accidentally accessing each other's code and data.
- **MS-Windows** and **Linux** run in protected mode.

Virtual-8086 Mode

- The computer **runs in protected mode** and **creates a virtual-8086** machine with **1-MByte** address space
  - **simulates** an 80x86 computer running in real address mode.
  - **Windows NT** and **2000**, create a virtual-8086 machine when you open a Command window.
  - You can run many such windows at the same time, and **each is protected from the actions of the others.**
- Some **MS-DOS programs** that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP.

Think Again?

2. Name all eight 32-bit general-purpose registers.

3. Name all six segment registers.

4. What special purpose does the ECX register serve?

Outline

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

64-Bit Processors

- **64-Bit** Operation Modes
  1. Compatibility mode – can run existing 16-bit and 32-bit applications (Windows supports only 32-bit apps in this mode)
  2. 64-bit mode – Windows 64 uses this
- Basic Execution Environment
  - addresses can be 64 bits (48 bits, in practice)
  - **16** 64-bit general purpose registers
  - 64-bit instruction pointer named **RIP**

64-Bit Processors: 64-Bit General Purpose Registers

- 32-bit general purpose registers:
  - EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 64-bit general purpose registers:
  - RAX, RBX, RCX, RDX, R

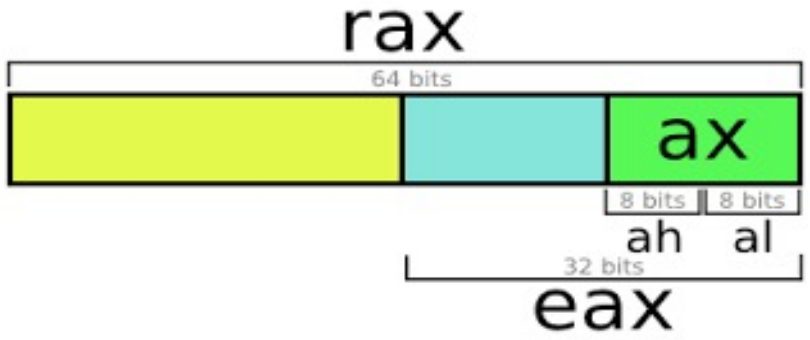


Table 2-1 Operand Sizes in 64-Bit Mode When REX Is Enabled.

Operand Size	Available Registers
8 bits	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
16 bits	AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32 bits	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64 bits	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

Outline

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

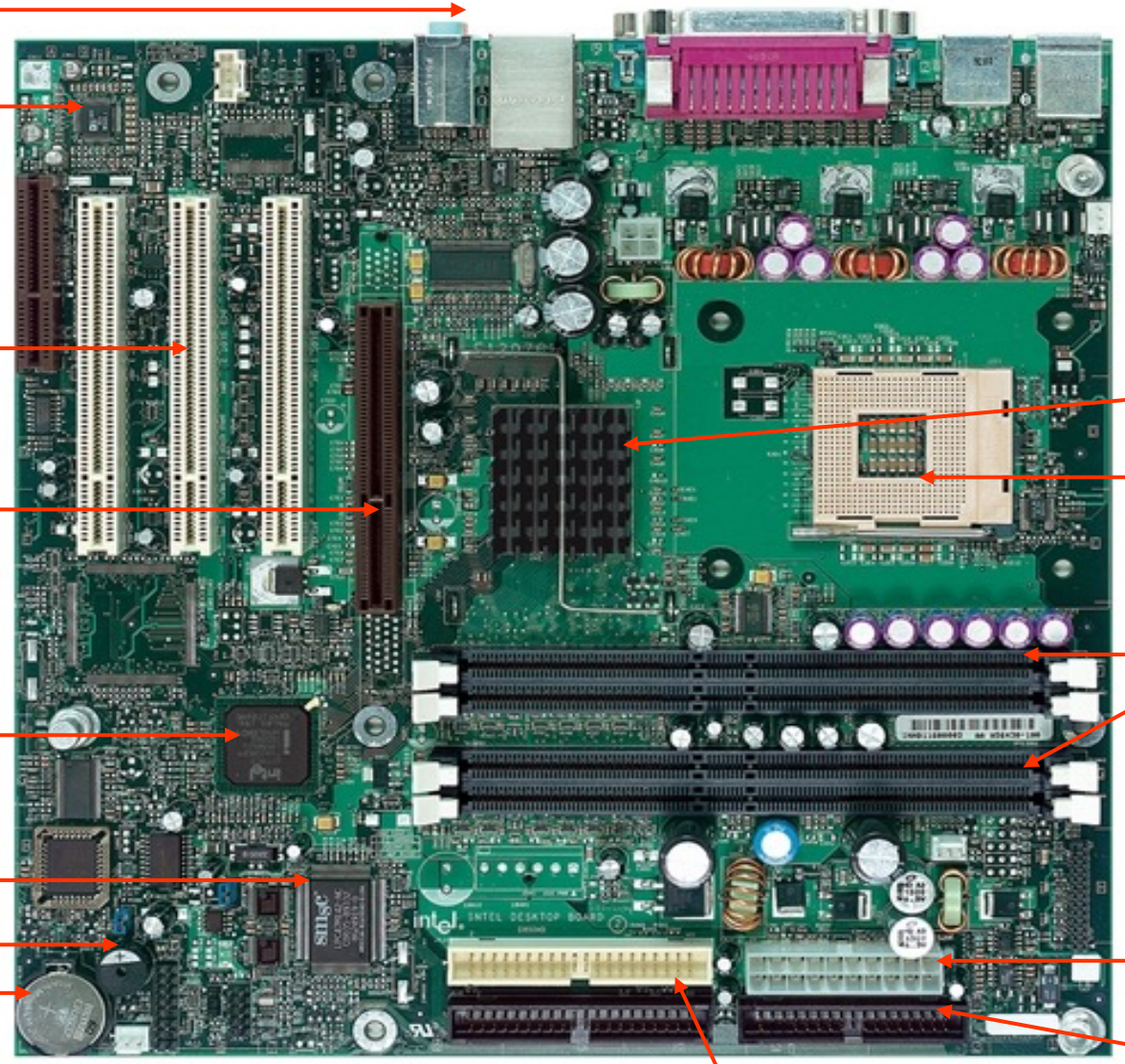
Components of an IA-32 Microcomputer

- Motherboard
- Video output
- Memory
- Input-output ports

Motherboard

- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips
- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)

Intel D850MD Motherboard



Video

Audio chip

mouse, keyboard, parallel, serial, and USB connectors

4

PCI slots

memory controller hub

Pentium 4 socket

2

AGP slot

dynamic RAM

3

1

Firmware hub

I/O Controller

Speaker

Battery

Power connector

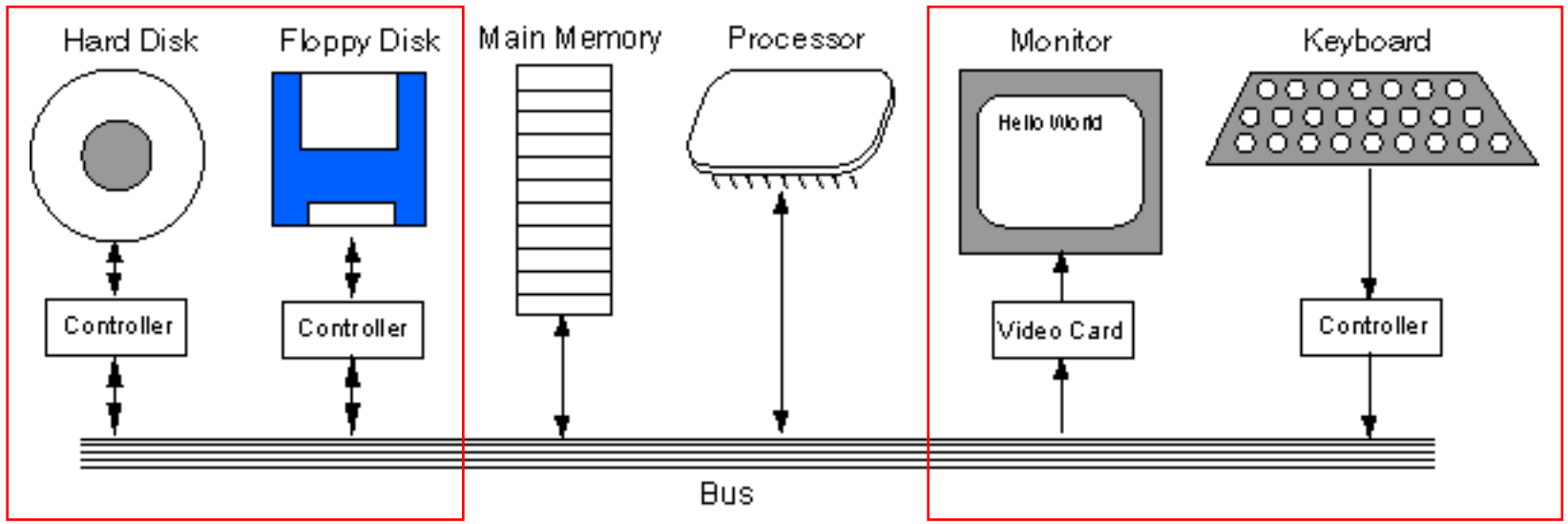
Diskette connector

IDE drive connectors

Outline

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

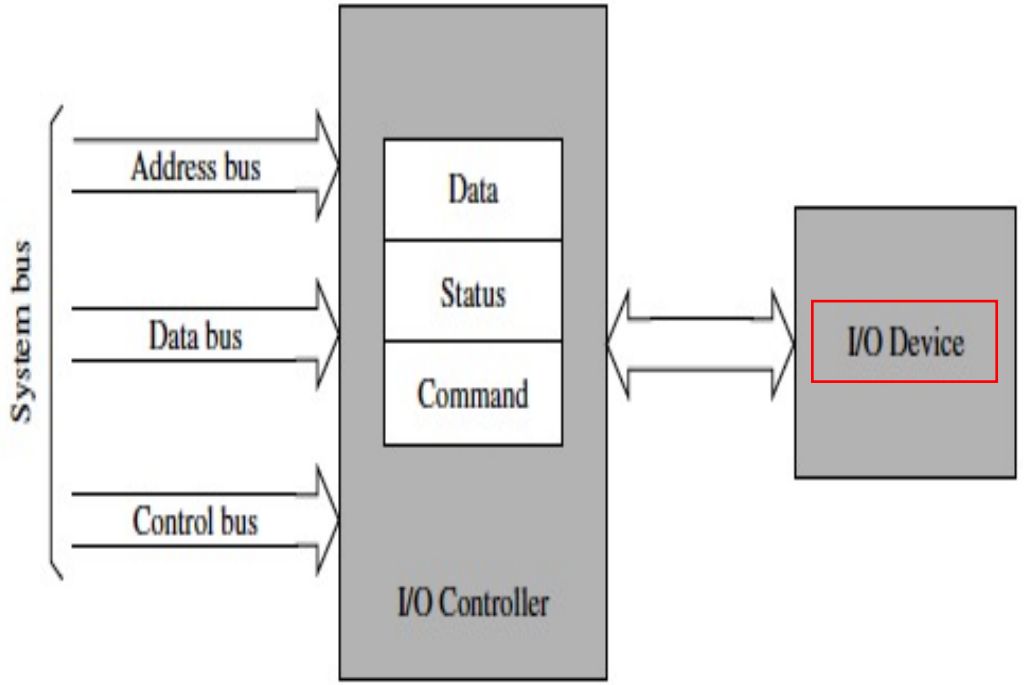
Input-Output Systems



Main Components of a Computer System

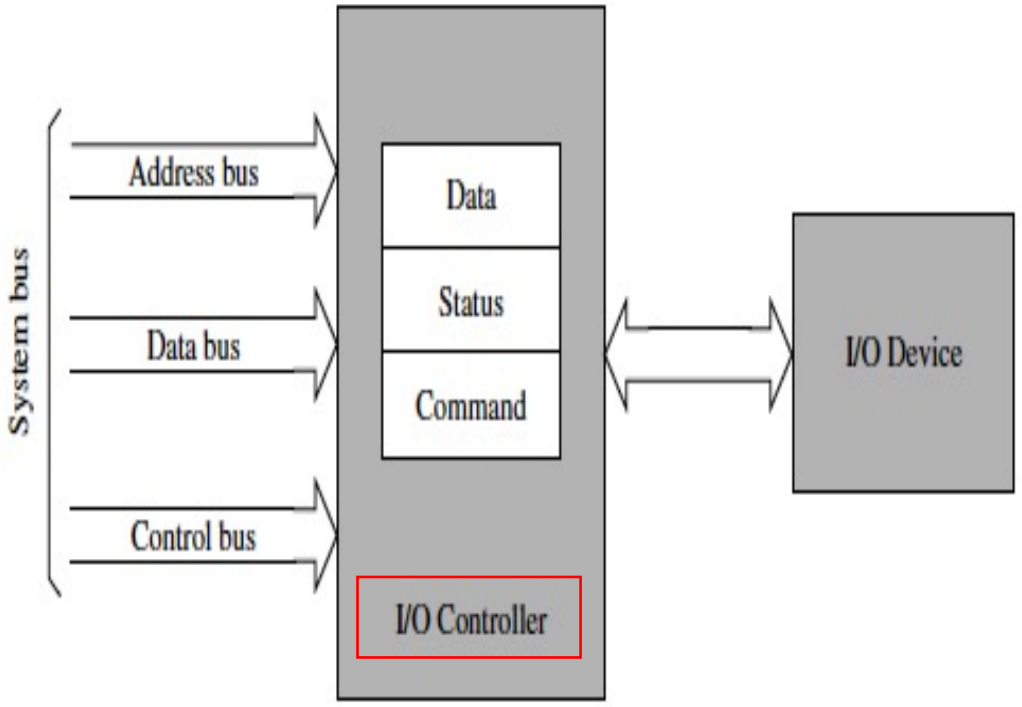
Input-Output Systems

- Input/output (I/O) **devices** provide the means by which a computer system can interact with the outside world
- An I/O device can be:
  - an input device (e.g., keyboard, mouse), or
  - an output device (e.g., printer, display screen), or
  - both an input and output device (e.g., disks)



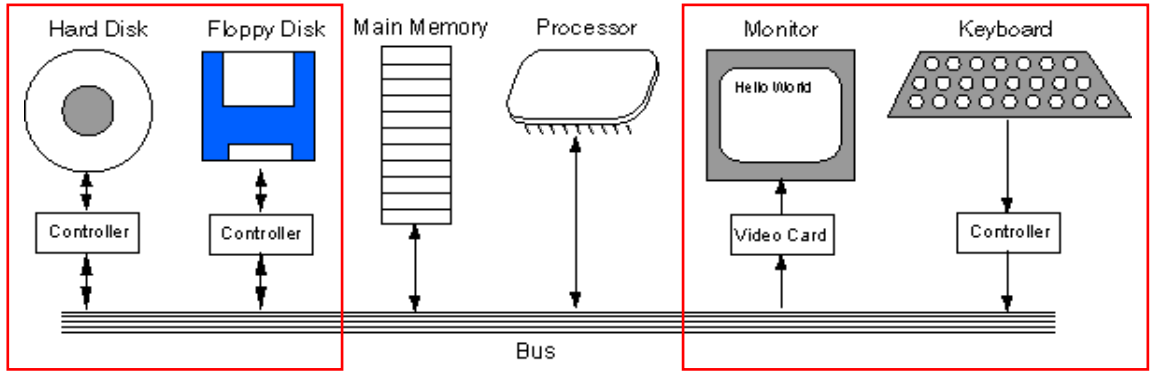
Input-Output Systems

- I/O devices are not directly connected to the system bus
  - **I/O controller** acts as an interface between the system and the I/O device.
  - I/O controller receives input and output requests from the central processor, and then send device-specific control signals to the device they control.
- I/O controllers typically have three types of internal registers a data register, a status register, and a command register



Input-Output Systems: **Level of I/O Access**

- **Application programs** read **input** from keyboard and disk files and write **output** to the screen and to files.



Main Components of a Computer System

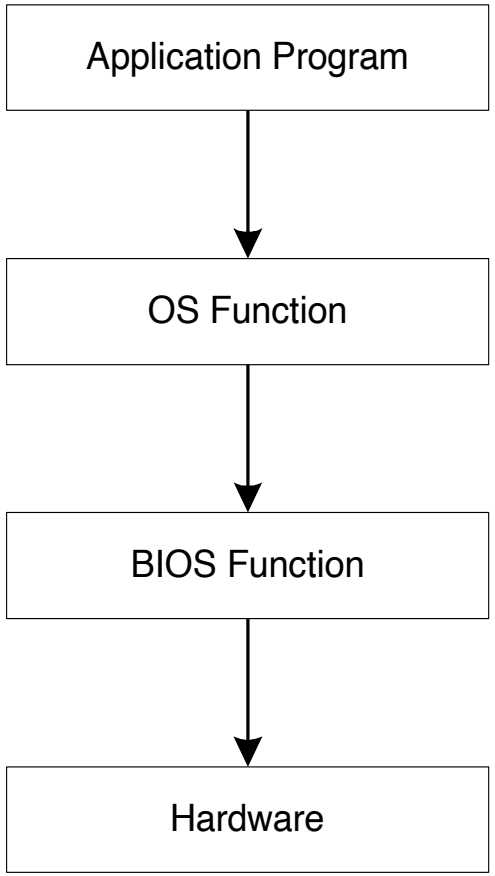
- **I/O need not** be accomplished by directly accessing hardware
  - You can **call functions** provided by the operating system.
  - **I/O is available** at different access levels, **High-level language functions**, **Operating system**, **BIOS**

Input-Output Systems: **Level of I/O Access**

- Level 3: **High-level language function**
  - examples: C++, Java
  - contains **functions** to perform input–output
  - portable, convenient, not always the fastest
- Level 2: **Operating system**
  - Application Programming Interface (**API**) (**writing** string to files, **read** string from keyboard, **allocating** block of memory)
  - extended capabilities, lots of details to master
- Level 1: **BIOS**
  - Collection of low-level **subroutines** that **communicate directly with hardware devices**.
  - Installed by computer's manufacturer
  - OS security may prevent **application-level** code from working at this level

Input-Output Systems: **Displaying a String of Characters**

- When a **HLL** program displays a string of characters, the following steps take place:



- Statement calls library function
- Calls an operating system function
- Pass a string pointer

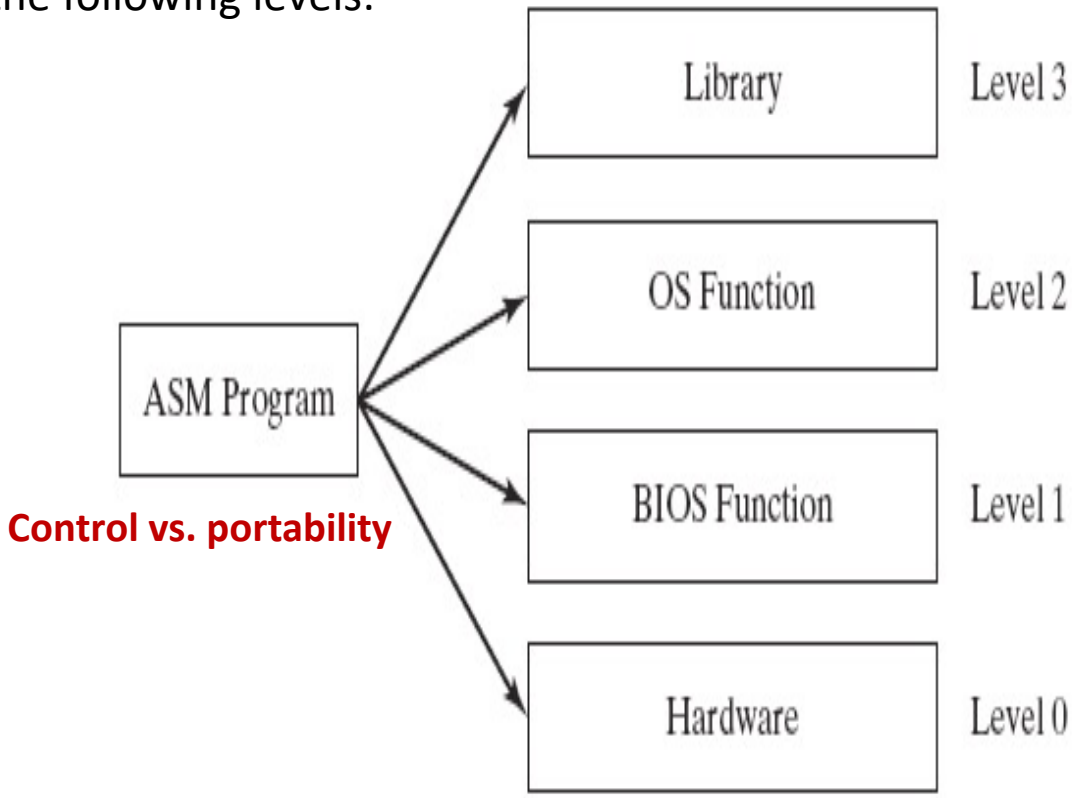
- Calls BIOS subroutine
- Passing ASCI and color for each
- Call another subroutine to move cursor to next position

- Receives a character, maps it to a particular system font,
- Sends the character to a hardware port

- The Video controller card generates hardware signals to the video display

Input-Output Systems: Programming levels

- **Assembly language** programs can perform input-output at each of the following levels:



Control vs. portability

- Call library functions to perform generic text I/O and file-based I/O

- Call operating system functions to perform generic text I/O and file-based I/O.

- Call BIOS functions to control device-specific features such as keyboard input, low-level disk I/O. etc.

- Send and receive data from hardware ports, having absolute control over specific devices.

Summary

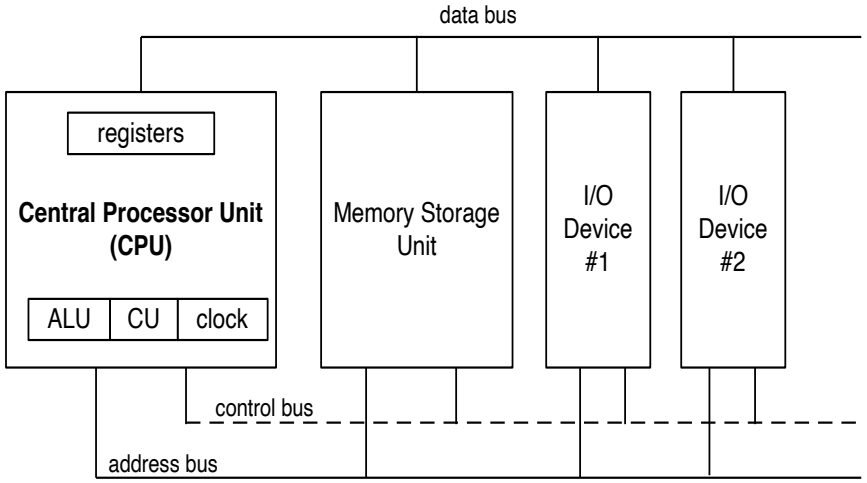


FIGURE 2–2 Simplified CPU block diagram.