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

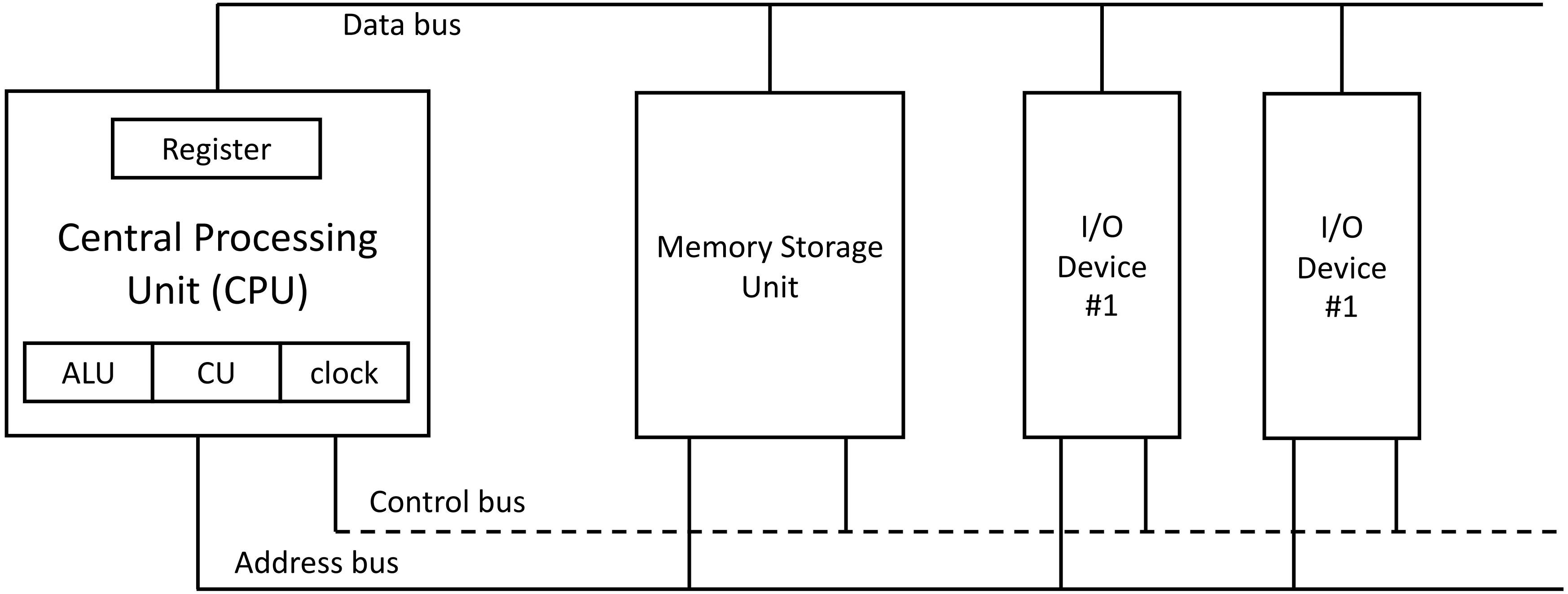
- **General Concepts**
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

General Concepts

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

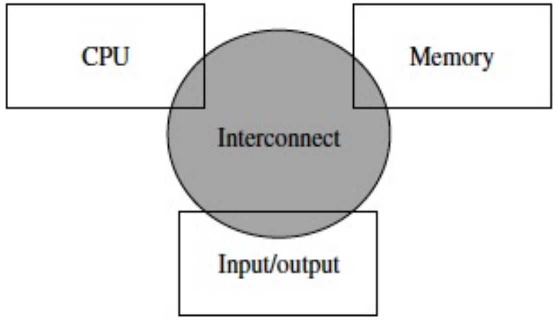
General Concepts: Basic Microcomputer Design

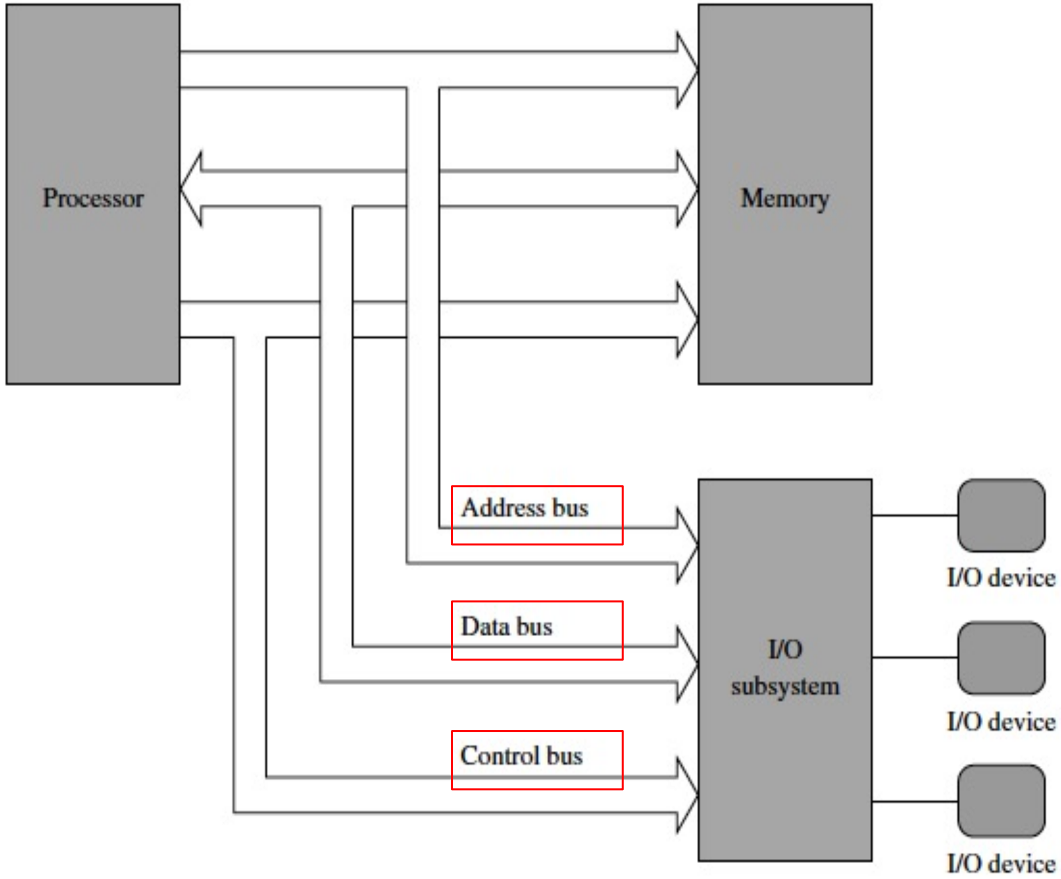
- **ALU** performs **arithmetic** and **logical** (bitwise) operations
- **Control unit (CU) coordinates** sequence of **execution steps**
- **Clock** synchronizes CPU **operations** with other **system components**



General Concepts: Basic Microcomputer Design

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



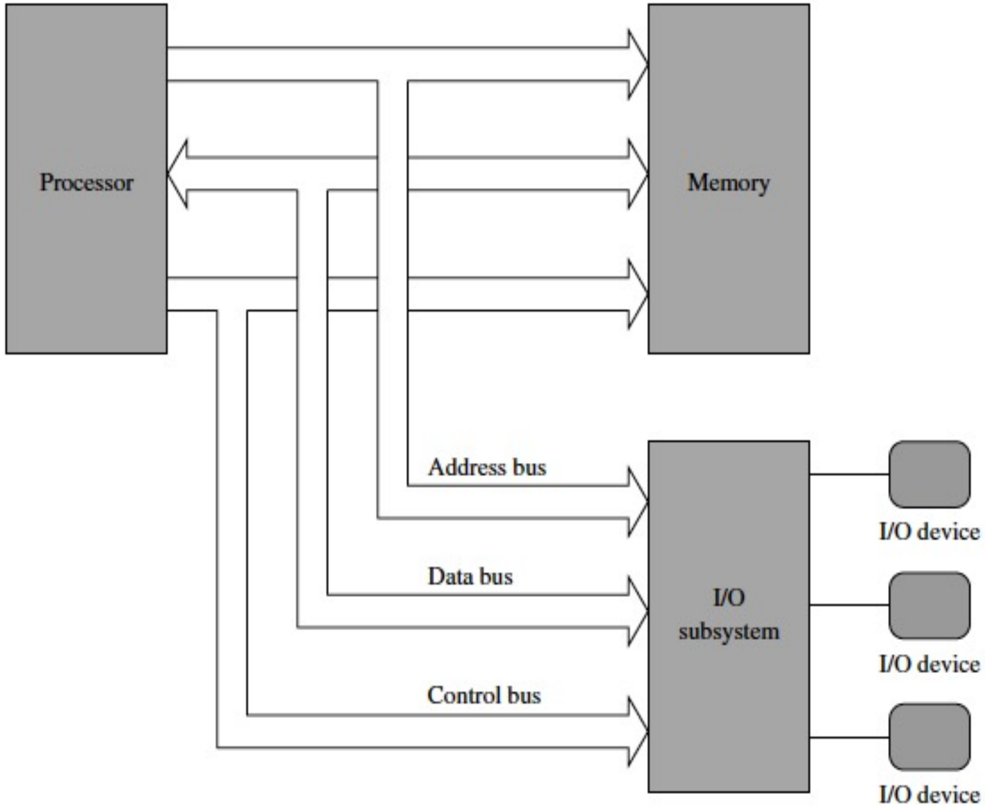


General Concepts: Basic Microcomputer Design

- The **Address bus** **holds the addresses** of instructions and data, when the currently executing instruction transfers data between the CPU and memory.

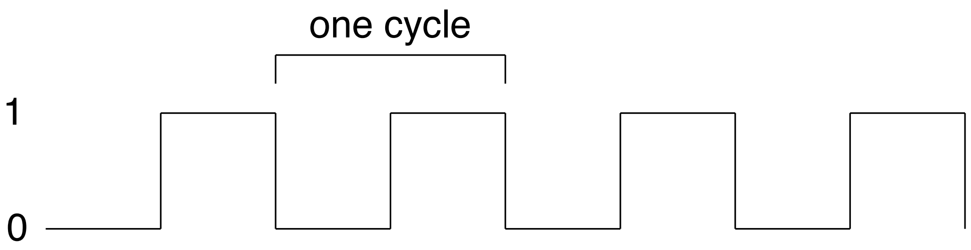
- The **Data bus** **transfers** **instructions** and **data** between the CPU and memory.

- The **Control bus** uses binary signals to **synchronize actions** of all devices attached to the system bus.



General Concepts: Clock

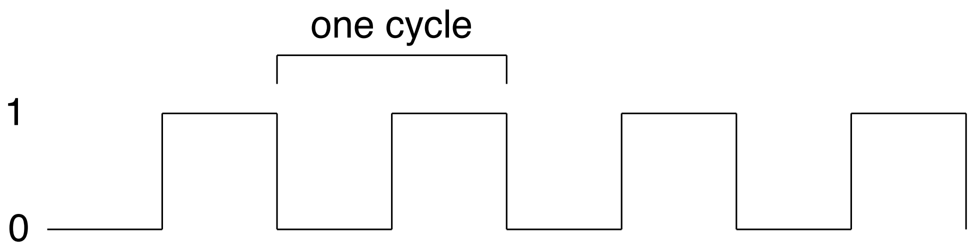
- The system clock provides **a timing signal** to synchronize the **operations** of the system.
  - ○ Synchronizes all CPU and BUS **operations**
- A **clock** is a sequence of **1's** and **0's**



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

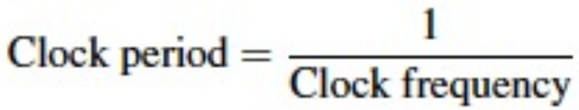
General Concepts: Clock

- 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.

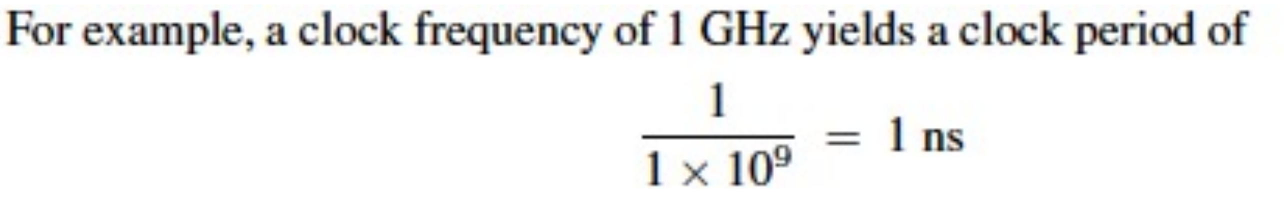


General Concepts: Clock

- **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 .



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



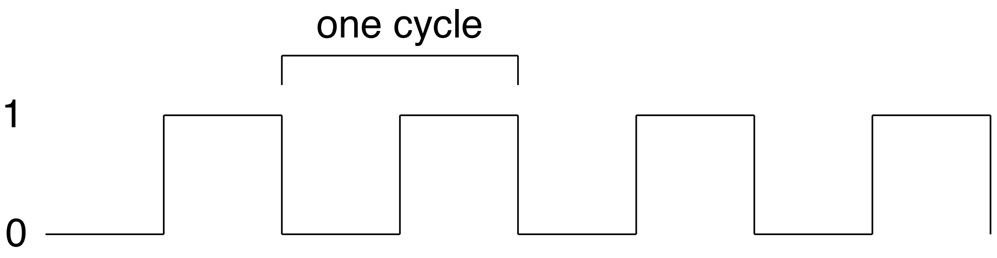
For example, a clock frequency of 1 GHz yields a clock period of

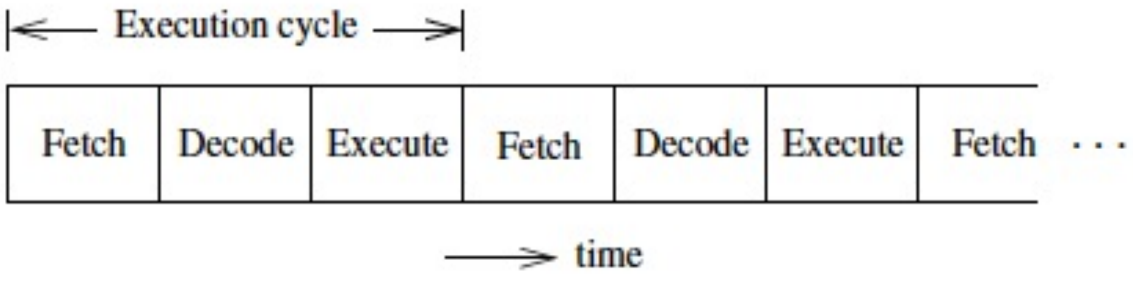
$$\frac{1}{1 \times 10^9} = 1 \text{ ns}$$

- 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

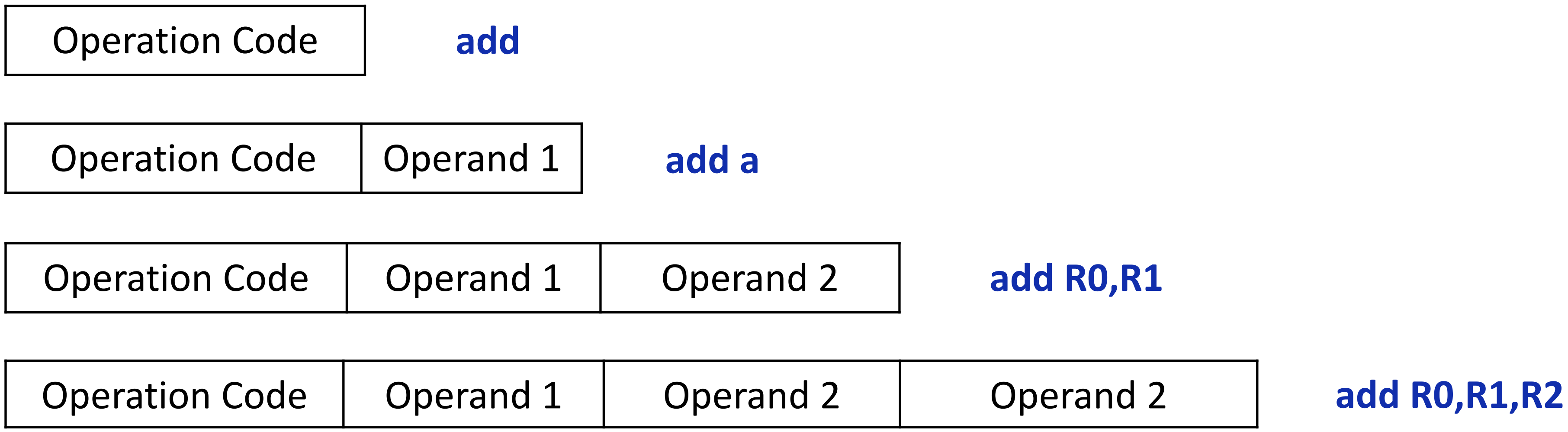
- An instruction is a **binary pattern designed** **inside a microprocessor** to perform a specific function.
- The entire group of instructions that a microprocessor supports is called **Instruction Set**.
- 8086 has more than **20,000** instructions.

- **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.



Operation Code | add

Operation Code | Operand 1 | add a

Operation Code | Operand 1 | Operand 2 | add R0,R1

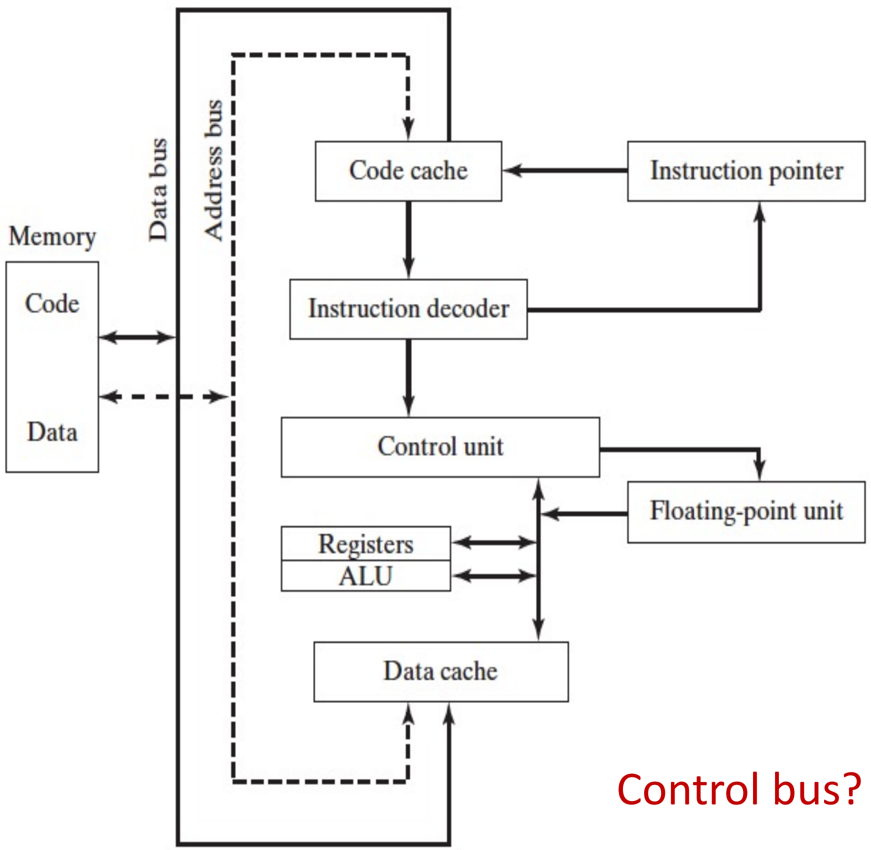
Operation Code | Operand 1 | Operand 2 | Operand 2 | add R0,R1,R2

  - 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.



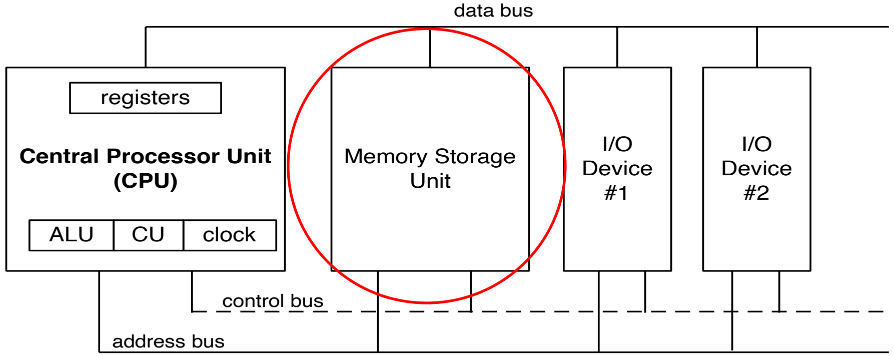
Control bus?

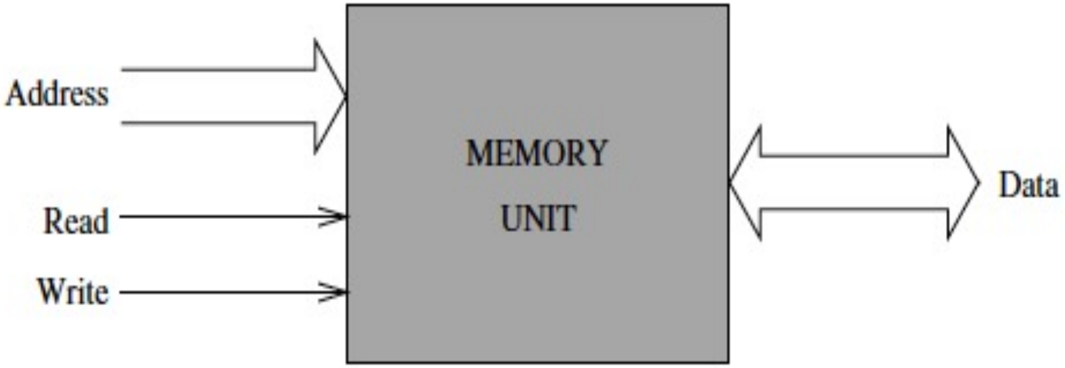
General Concepts

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

Memory

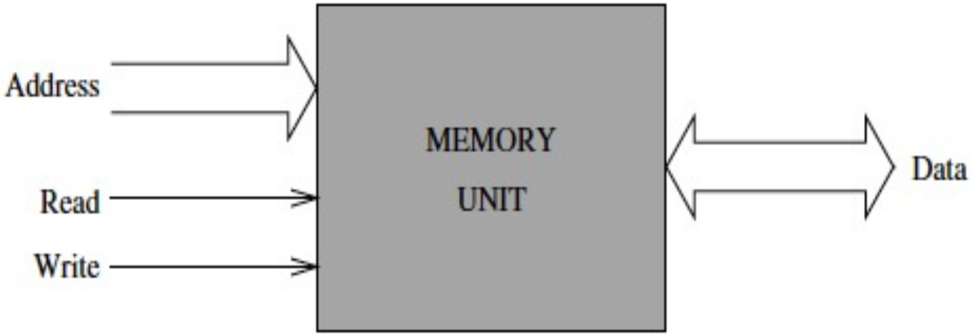
- The memory unit supports two fundamental operations: **read and write** .
  - The **read operation** reads a previously stored data
  - The **write operation** stores a value in memory.





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

Memory

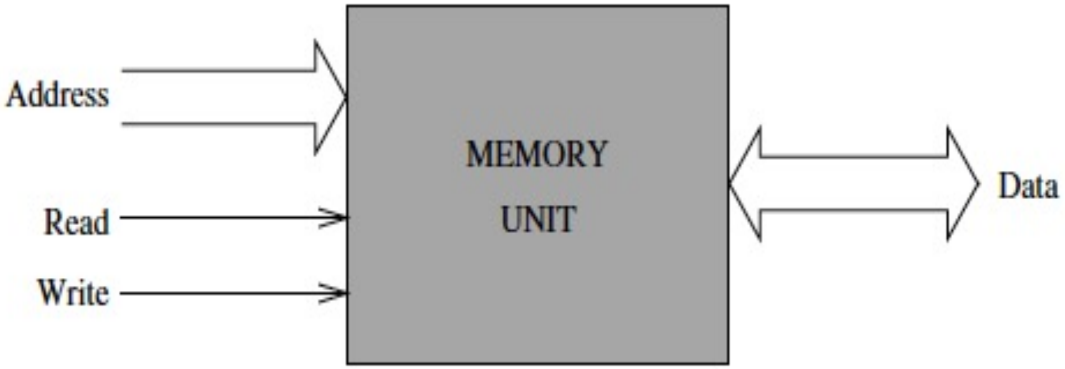


Memory (RAM) as an array of bytes

Content:	FF	00	57	92	B3	8A	… …	10	46	DC
Address:	000 000 000	000 000 001	000 000 002	000 000 003	000 000 004	000 000 005	… …	134 217 725	134 217 726	134 217 727

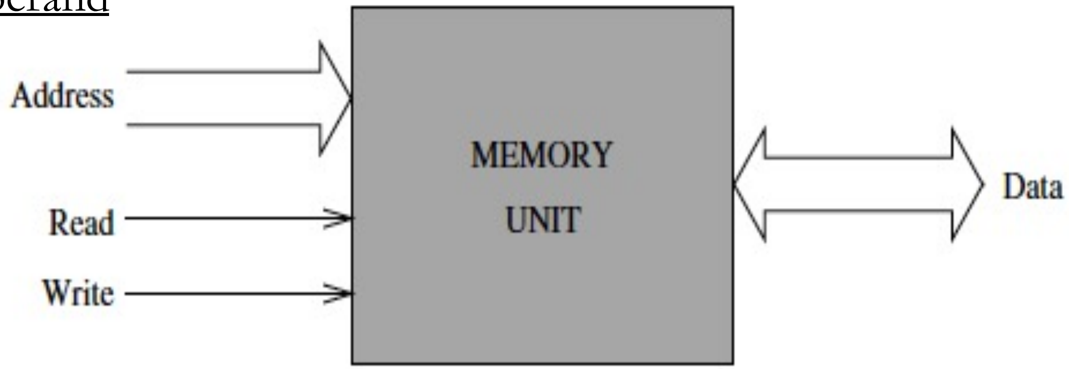
Memory

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



Memory: Reading from Memory

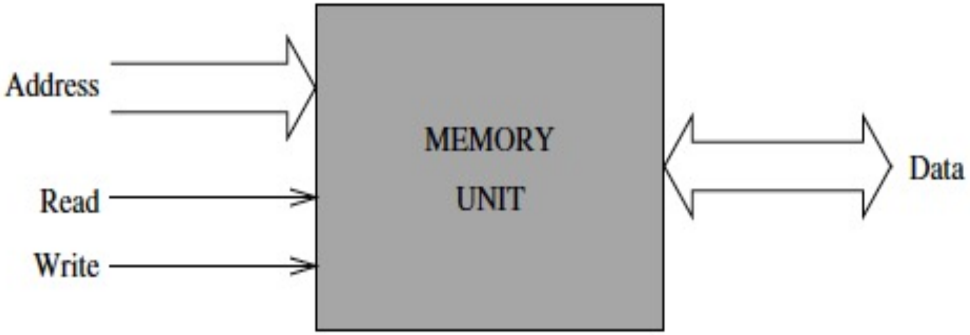
- **Multiple machine cycle** are required when **reading** from memory, Why?
    - Because **it** responds much more slowly than the CPU.
- Steps in a typical **read cycle**
    1. **Place** the address of the value you want to read on the address bus.
    2. **Assert** (changing the value of) the processor's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

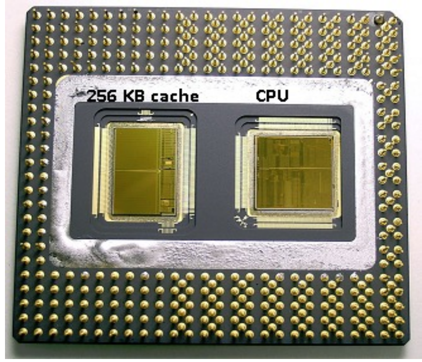
- Steps in a typical **write cycle:**

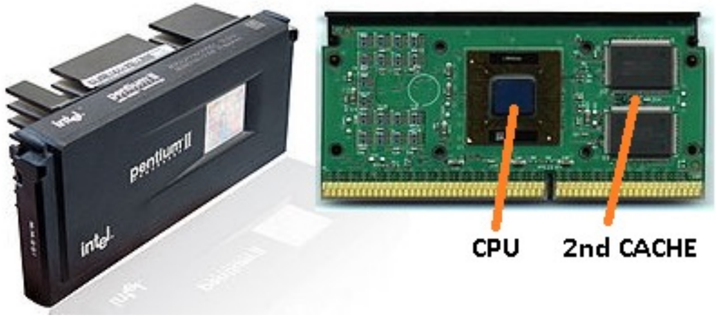
1. **Place** the address of the location to be written on the address bus,
2. **Place** the data to be written on the data bus,
3. **Assert** (changing the value of) the processor's **WR** (write) pin.
4. **Wait** for the memory to store the data at the addressed location

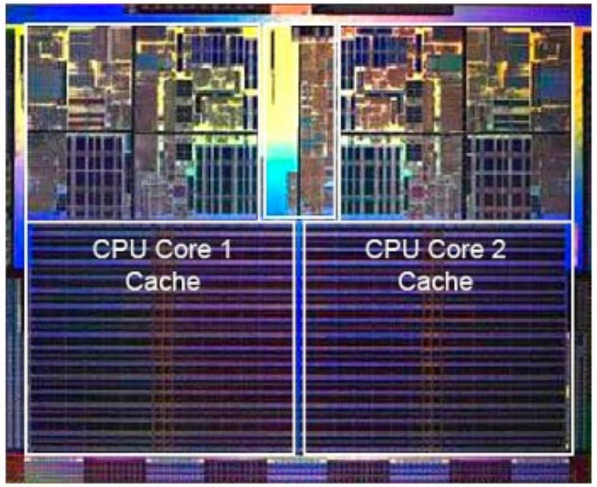


Reading from Memory: Cache Memory

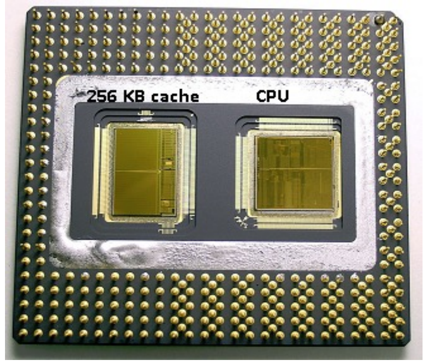
- In practice, **instructions and data** are **not fetched**, most of the time, from the **main memory**.
- There is a **high-speed** **cache memory** that provides
  - faster access to **instructions** and **data** than the main memory.



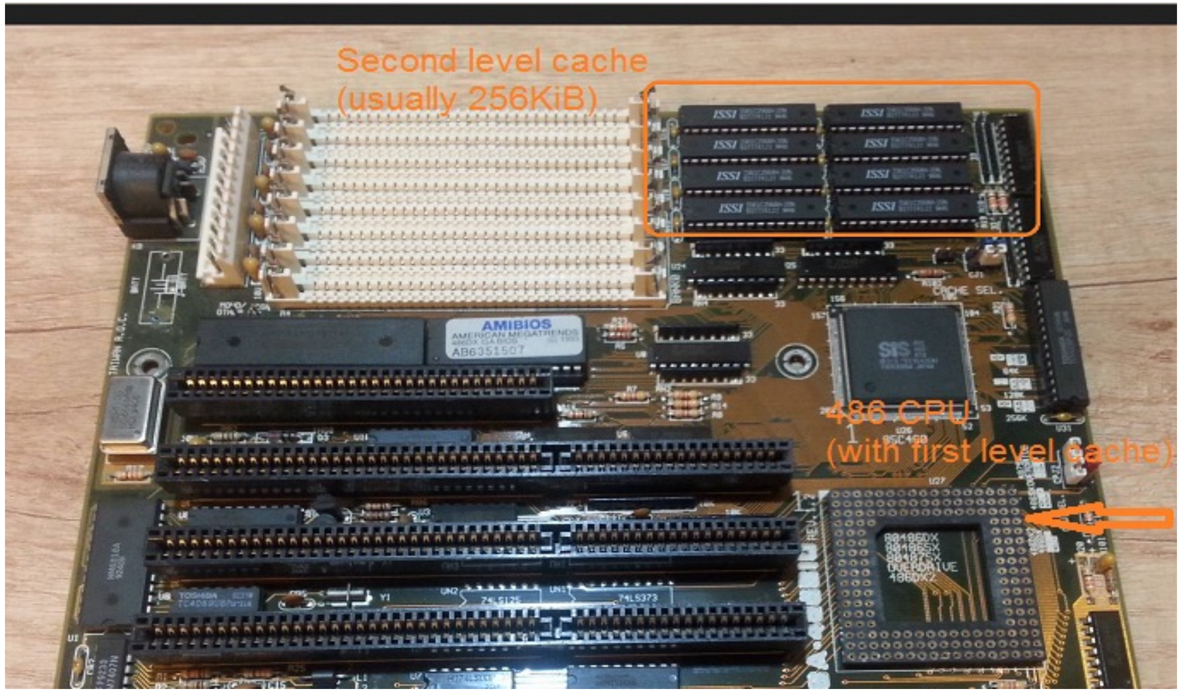




Reading from Memory: Cache Memory



- ○ **Level-1 cache**: inside the CPU



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

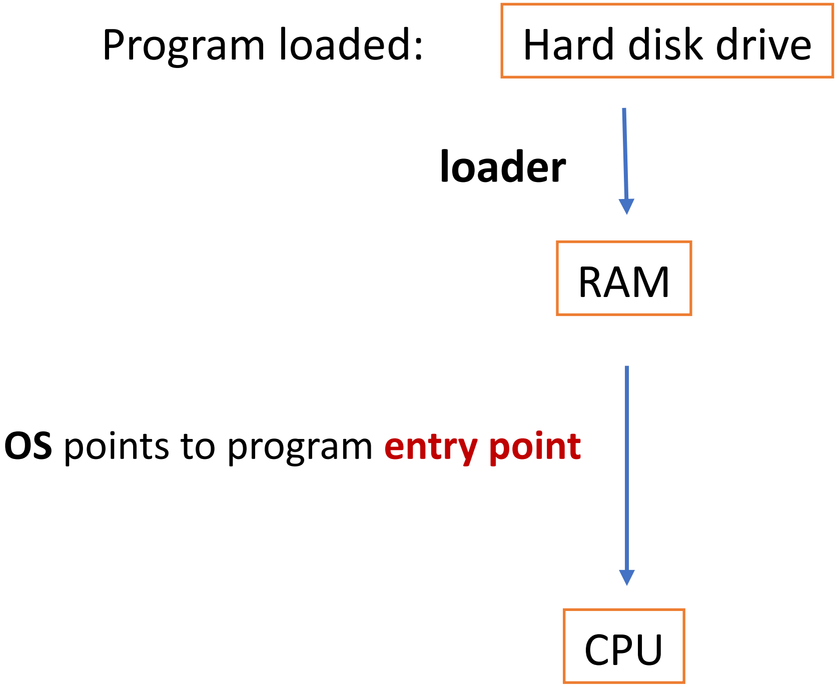
Reading from Memory: Cache Memory

- **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



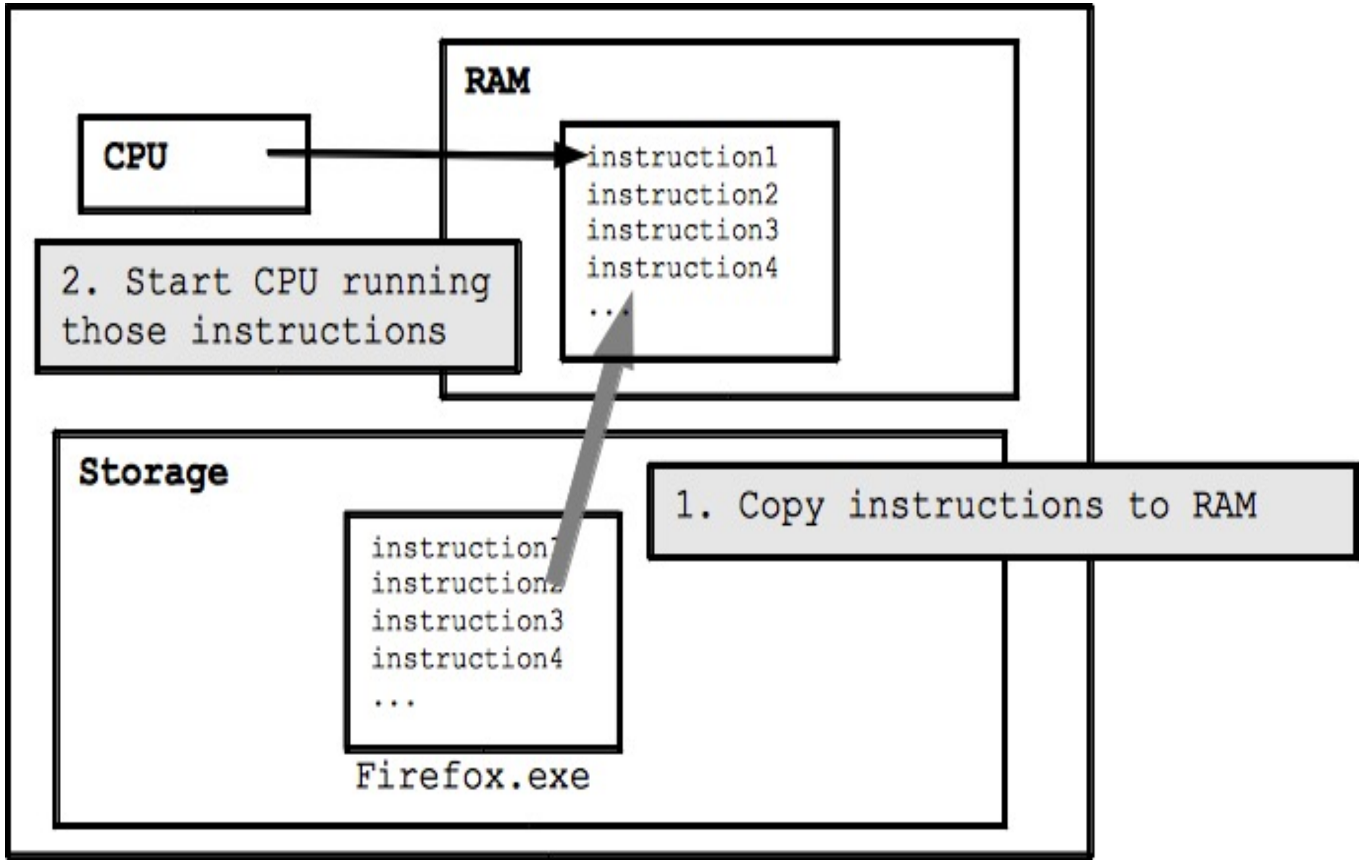
Program loaded: Hard disk drive

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

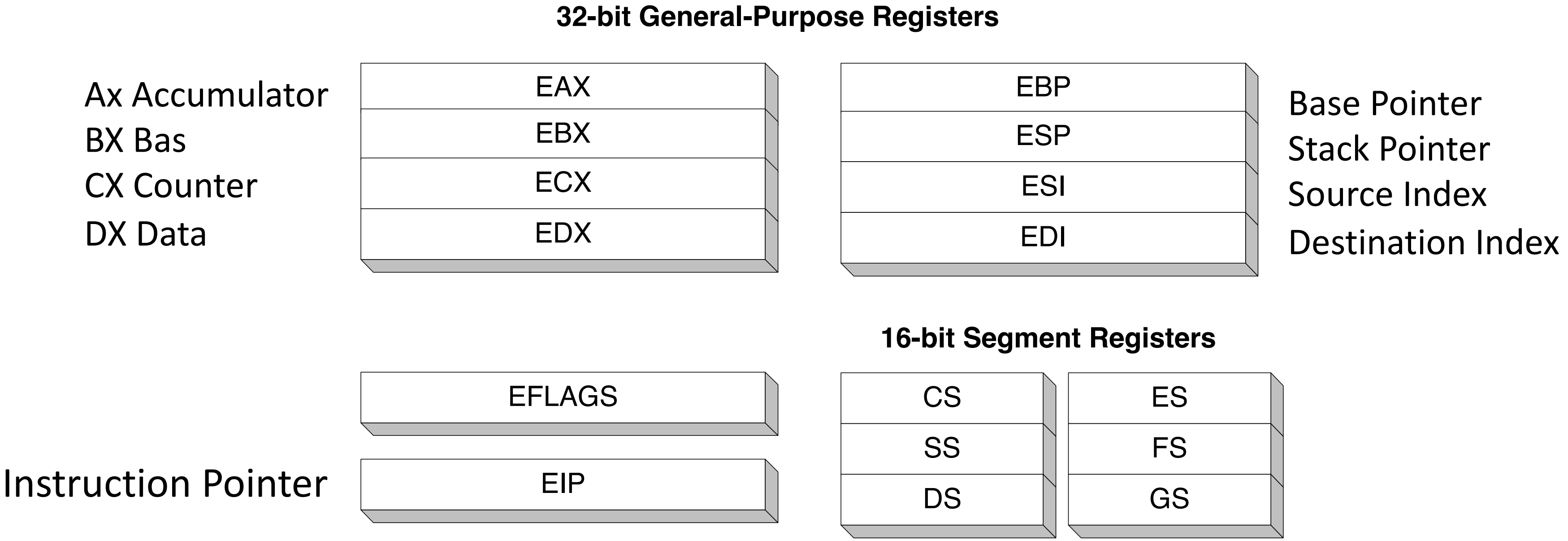
- **Address Space**

- **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?



32-bit General-Purpose Registers

Ax Accumulator
BX Bas
CX Counter
DX Data

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

Base Pointer
Stack Pointer
Source Index
Destination Index

16-bit Segment Registers

EFLAGS

Instruction Pointer

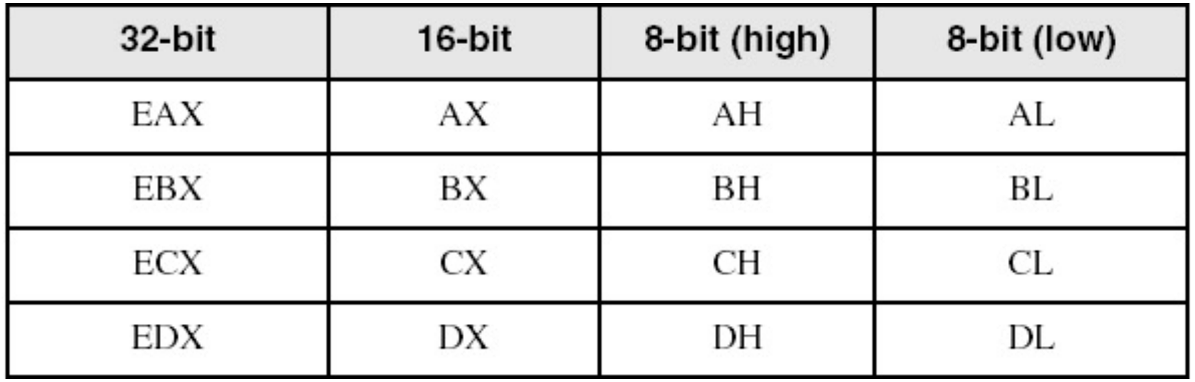
EIP

CS
SS
DS

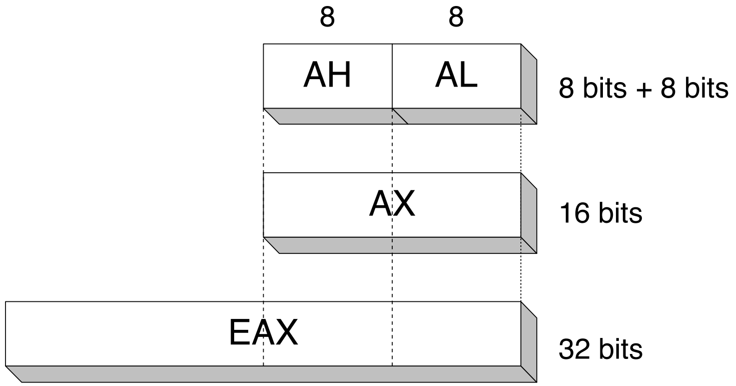
ES
FS
GS

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

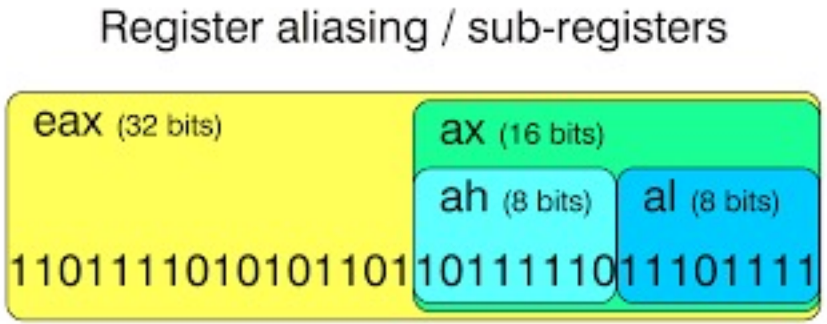
- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to **EAX**, **EBX**, **ECX**, and **EDX**

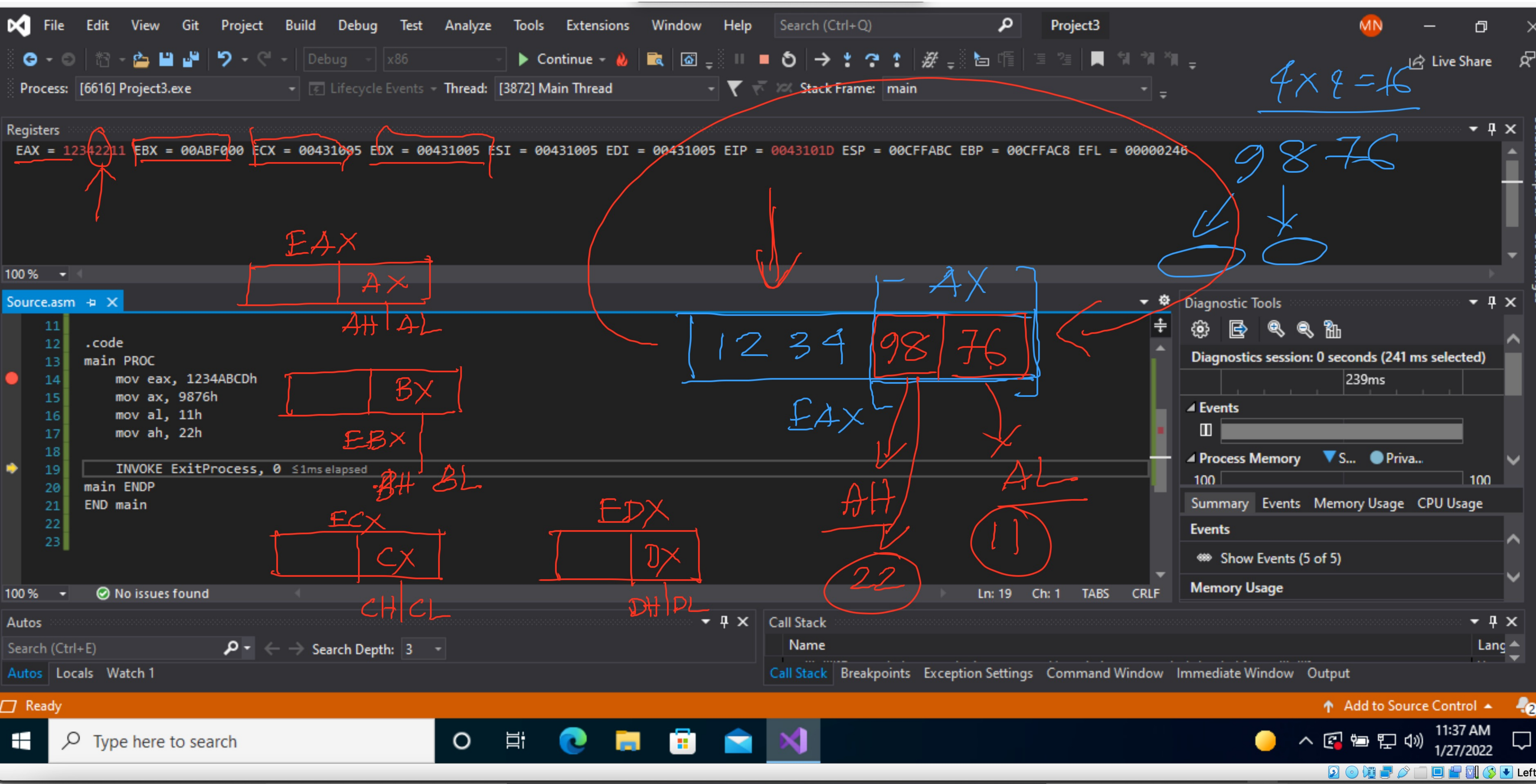


32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL



Ex:





```
EAX = 12342211 EBX = 00ABF000 ECX = 00431005 EDX = 00431005 ESI = 00431005 EDI = 00431005 EIP = 0043101D ESP = 00CFFABC EBP = 00CFFAC8 EFL = 00000246
```

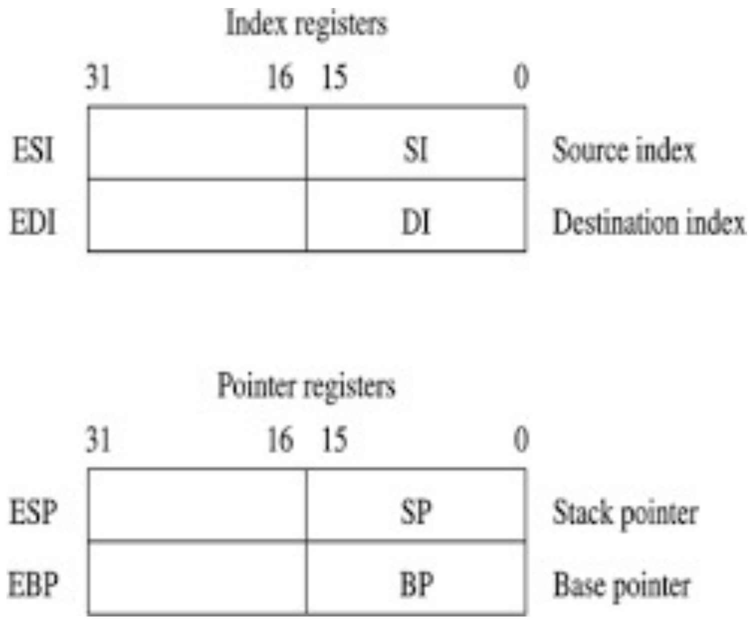
```
.code
main PROC
    mov eax, 1234ABCDh
    mov ax, 9876h
    mov al, 11h
    mov ah, 22h

    INVOKE ExitProcess, 0  ≤1ms elapsed
main ENDP
END main
```

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

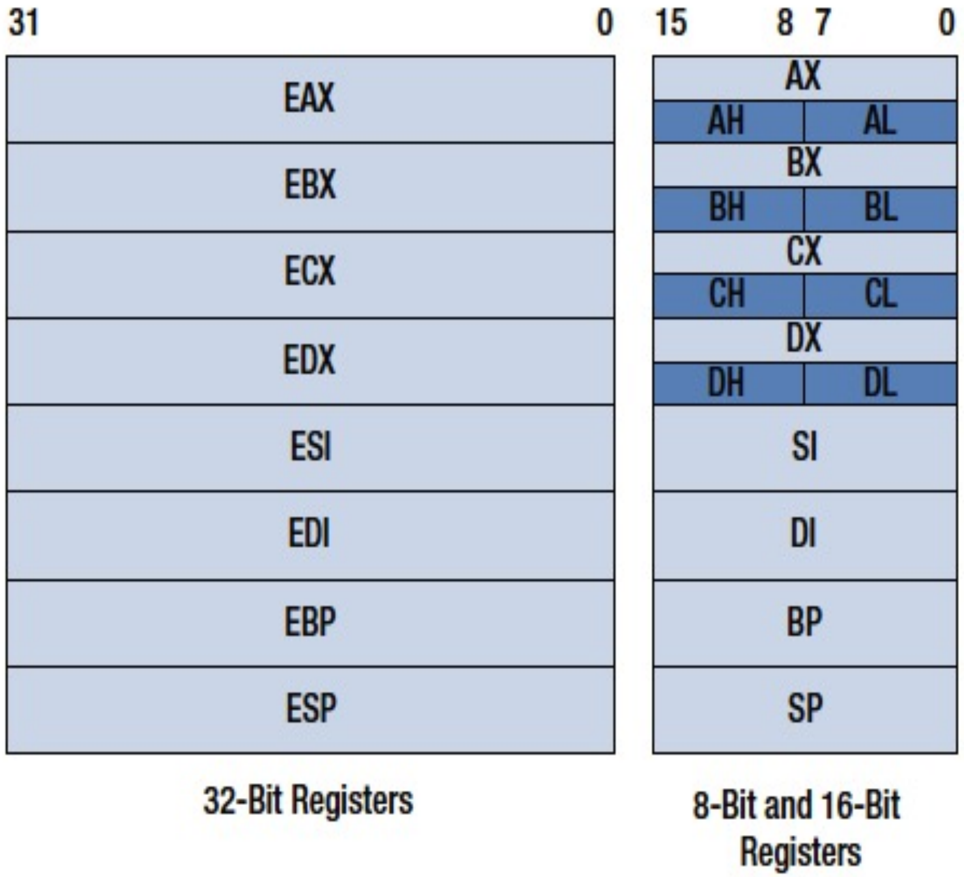
- Some registers have only a 16-bit name for their lower half:

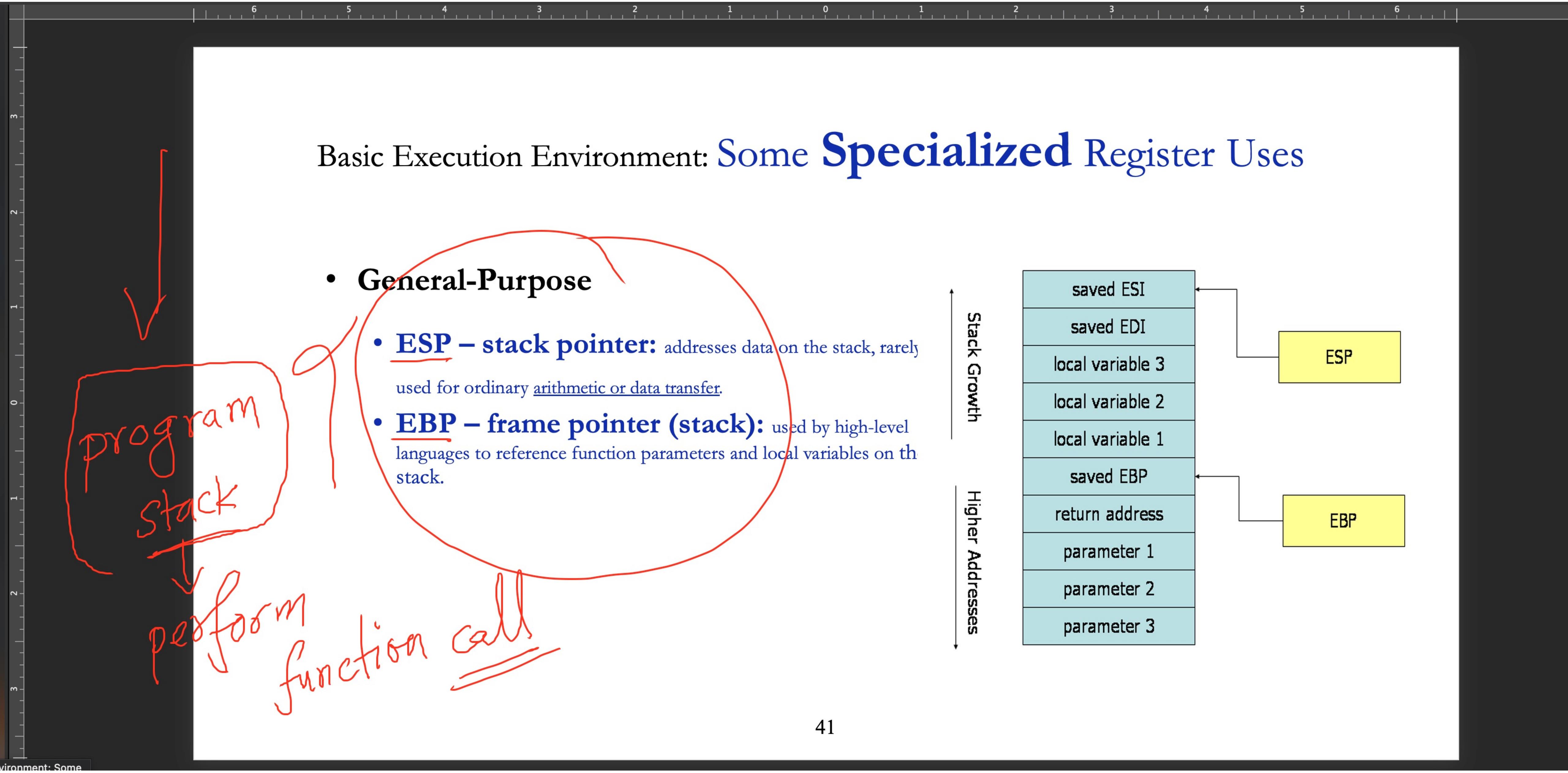
32-bit	**16-bit**
ESI	SI
EDI	DI
EBP	BP
ESP	SP



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, rarel used for ordinary arithmetic or data transfer.
  - **EBP – frame pointer (stack):** used by high-level languages to reference function parameters and local variables on th stack.

Stack Growth

Higher Addresses

saved ESI
saved EDI
local variable 3
local variable 2
local variable 1
saved EBP
return address
parameter 1
parameter 2
parameter 3

ESP

EBP

program stack

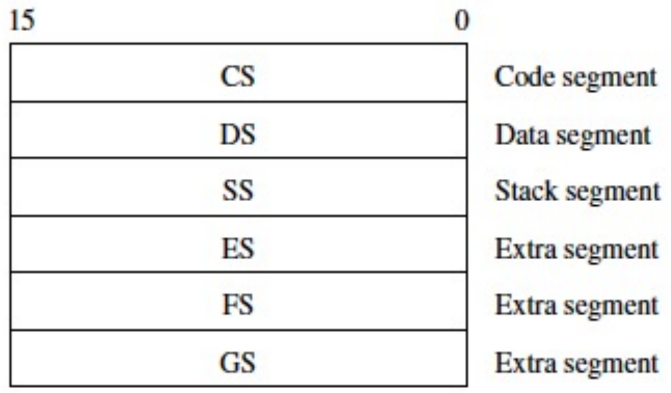
perform function call

41

Attendance!

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.



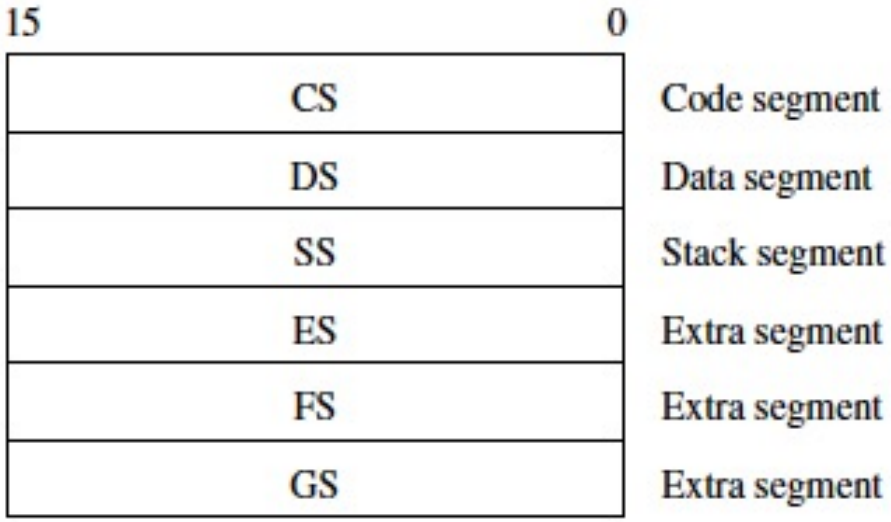
15 … 0	
CS	Code segment
DS	Data segment
SS	Stack segment
ES	Extra segment
FS	Extra segment
GS	Extra segment

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 **<u>data segment</u>**, we could use two segment registers to point to the two data segments.



15 0

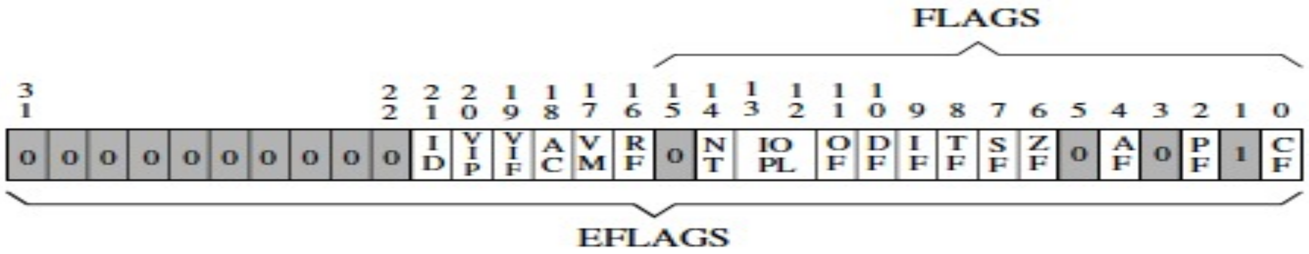
CS	Code segment
DS	Data segment
SS	Stack segment
ES	Extra segment
FS	Extra segment
GS	Extra segment

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

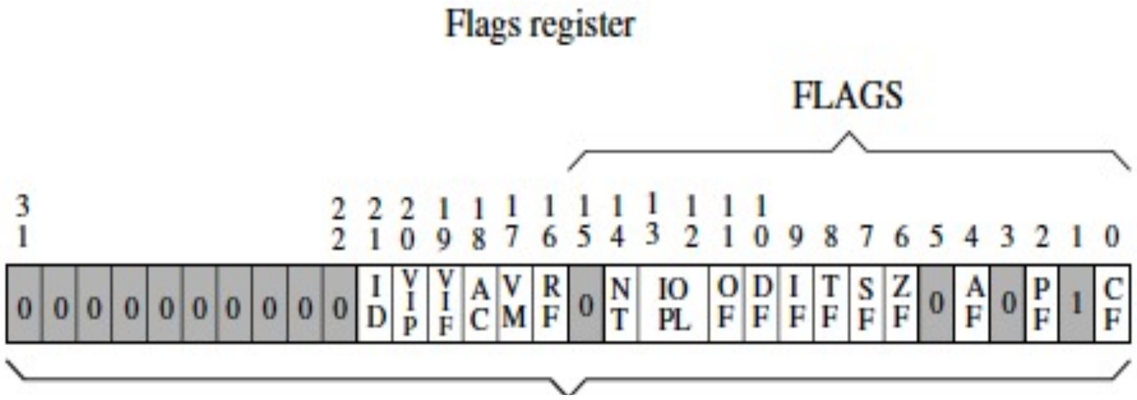
- **EIP** – instruction pointer (also called **program counter**): contains the address of the **next instruction to be executed** .
- **EFLAGS-** a register consists of **individual binary bits** that control the **operation** of the CPU or reflect **the outcome** of **some CPU operation.**
  - **status**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

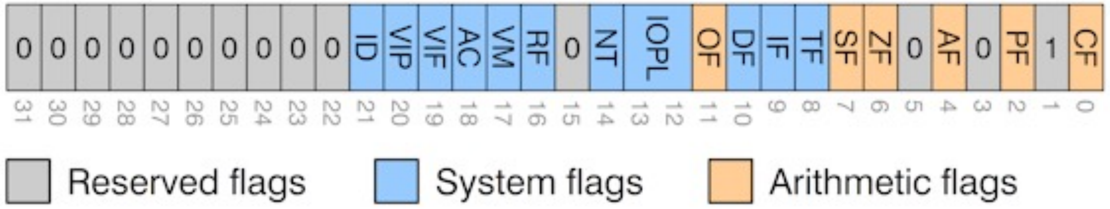
A flag is **set** when it equals 1; it is **clear** (or reset) when it equals 0.

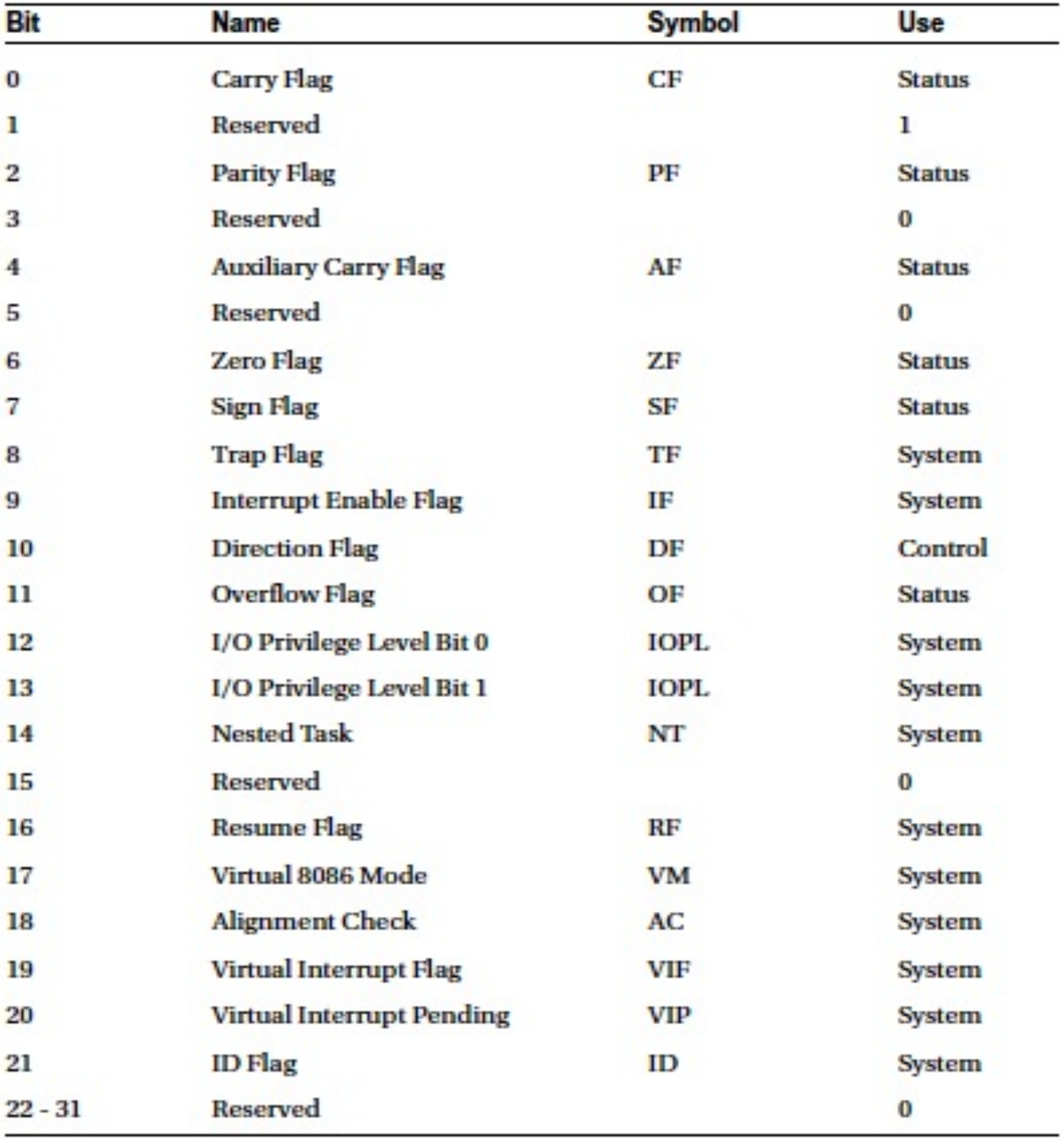
Flags register



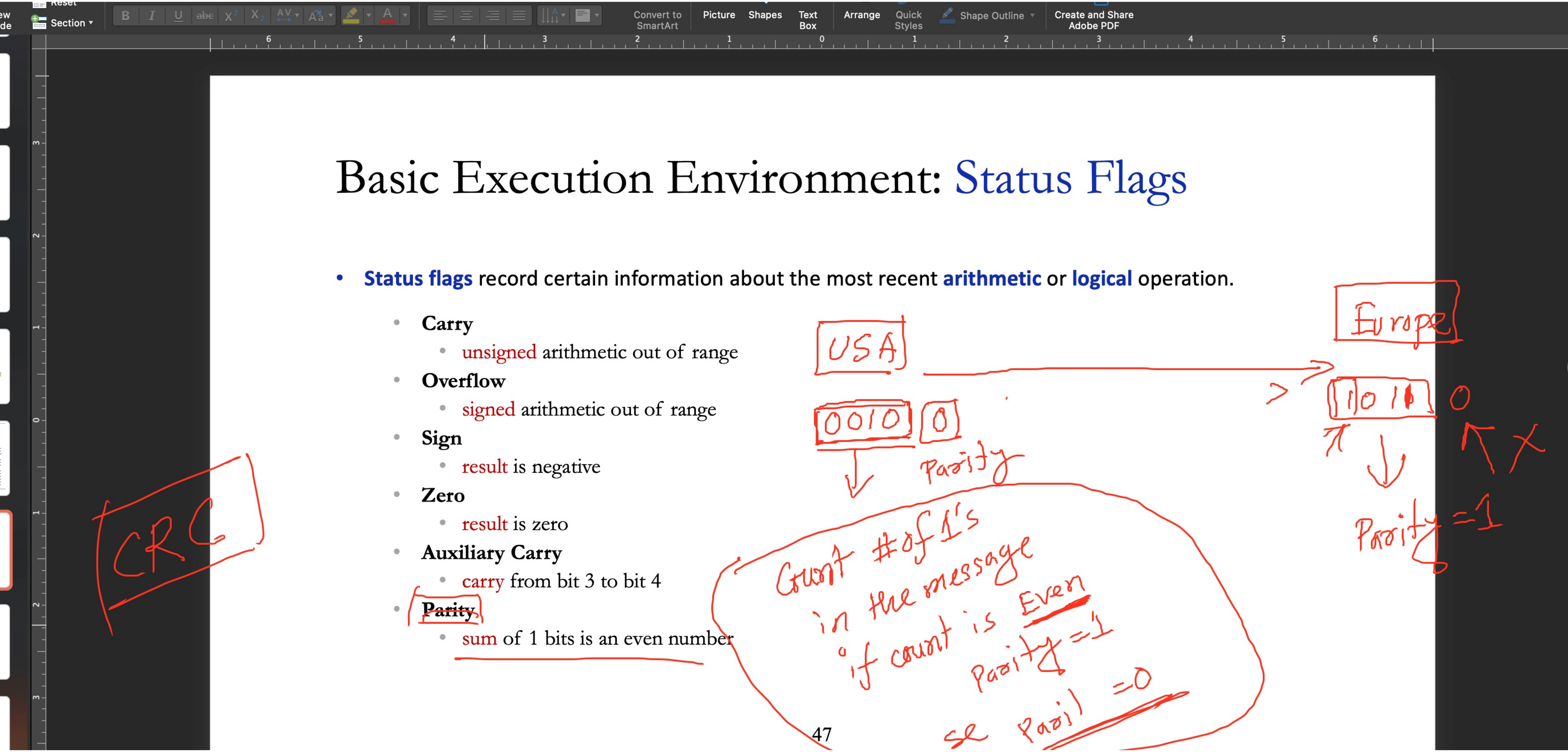
Flags Table







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



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

Count #of 1's in the message

if count is Even
Parity =1
se Parail =0

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.