



National University  
of Computer & Emerging Sciences

# Computer Organization and Assembly Language

Week: 1

Lecture: 1

Rabia Ahmed  
E: rabia.ansari@nu.edu.pk

# Outline

- Class Policies
- Marks Distribution
- Class Syllabus
- Introduction to the Course
- COAL Essentials

# Class Policies



Allowed	Curiosity
Allowed	Respect
Allowed	Punctuality
Not Allowed	Phones
Not Allowed	Chatting
Not Allowed	Eating
Not Allowed	Talking

# Marks Distribution

- Quizzes 6%
- Class Participation 3%
- Assignments 6%
- Grand Assignment 5%
- Mid 30%
- Final Exam 50%

# Class Syllabus

## ◎ Objective of course:

- Programming Methodology of low-level languages
- How to access computer hardware directly
- Overview of a user-visible architecture (of Intel 80x86 processors)
- Intel 80x86 instruction set, assembler directives, macro.
- How programs interact with the operating system for various services including memory management and input/output services
- How to write programs by understanding the hardware structure

# Book Recommended:



<b>Text Book(s)</b>	<b>Title</b>	Assembly Language for x86 Processors K.Irvine 7 <sup>th</sup> Edition
	<b>Author</b>	Kip R. Irvine
	<b>Publisher</b>	Pearson Education Inc. ( <i>ISBN 978-0-07-338065-0</i> )
<b>Ref. Book(s)</b>	<b>Title</b>	Assembly Language Programming and Org. of the IBM PC
	<b>Author</b>	Ytha Yu, Charles Marut
	<b>Publisher</b>	McGraw Hill
<b>Ref. Book(s)</b>	<b>Title</b>	Computer organization and design: the hardware/software interface
	<b>Author</b>	David A. Patterson and John L. Hennessy
	<b>Publisher</b>	Morgan Kaufmann

# Computer Organization:

## ◎ Definition:

- **Computer organization** is concerned with the way hardware components operate and the way they are connected together to form the computer system.
- The computer organization is concerned with the structure and behavior of digital computers.

# Machine Language

- ◎ A numeric language specifically understood by a computer's processor. It is composed of digital binary numbers and looks like a very long sequence of zeros and ones.
- 10100001 00000000 00000000     Fetch the contents of memory word 0 and put it in register AX
- 00000101 00000100 00000000     Add 4 to AX
- 10100011 00000000 00000000     Store the contents of AX in memory word 0

# Assembly Language

- ◎ An assembly language is a low-level programming language designed for a specific type of processor.
- ◎ Because assembly depends on the machine code instructions, every assembler has its own assembly language which is designed for exactly one specific computer architecture.

# Assembly Language

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

Assembly language has a *one-to-one relationship* with machine language:

- Each assembly language instruction corresponds to a single machine-language instruction.

# Assembly Language

- ◎ Assembly code is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as *assembly*, as in *assembling* the source code

# Example:

- Machine language instruction

- 10100001 00000000 00000000      Fetch the contents of memory word 0 and put it in register AX
- 00000101 00000100 00000000      Add 4 to AX
- 10100011 00000000 00000000      Store the contents of AX in memory word 0

- Assembly language instruction

- MOV AX, A
- ADD AX, 4
- MOV A, AX

# High Language

- ◎ A high-level language (HLL) is a programming language such as C, FORTRAN, or Pascal 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.

# Example:

- Machine language instruction
  - 10100001 00000000 00000000 Fetch the contents of memory word 0 and put it in register AX
  - 00000101 00000100 00000000 Add 4 to AX
  - 10100011 00000000 00000000 Store the contents of AX in memory word 0
- Assembly language instruction
  - MOV AX, A
  - ADD AX, 4
  - MOV A, AX
- High level Language
  - A= A+ 4

## 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

- ◎ 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;
```

# What you ‘ll learn?

- You will learn some basic principles of computer architecture, as applied to the Intel IA-32 processor family.
- You will learn some basic boolean logic and how it applies to programming and computer hardware.
- You will learn about how IA-32 processors manage memory, using real mode, protected mode, and virtual mode.
- You will learn how high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code.
- You will learn how high-level languages implement arithmetic expressions, loops, and logical structures at the machine level.
- You will learn about data representation, including signed and unsigned integers, real numbers, and character data.
- 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.

# Portable Language

- A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable*.

## **Example:**

- C++
  - Compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system.
- Java
  - 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*.

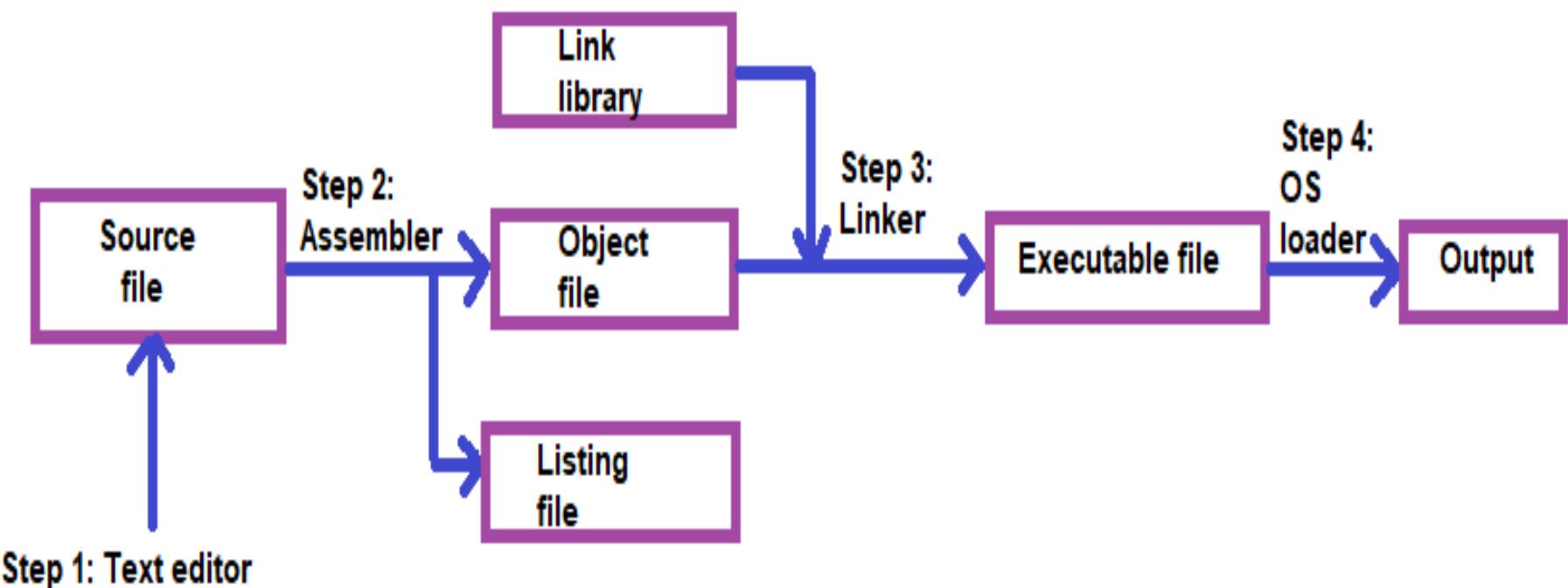
# Applications of Assembly Language

- 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. This permits to take total control over the creation of machine code.
- Assembly language will help you understanding the interaction between the computer hardware, operating system, and application programs. Using assembly language, you can apply and test the theoretical information you are given in computer architecture and operating systems courses.

# *What Are Assemblers and Linkers?*

- ◎ A source program written in assembly language cannot be directly executed on its target computer. It must be translated, or assembled into executable code.

# Assemble-Link-Execute Cycle



# Steps of Assemble-link-Execute Cycle

- ◎ Step1: A programmer uses a text editor to create source file.
- ◎ Step2: The assembler reads source file and produces an object file. Optionally a listing file is also produced.
- ◎ Step3: Linker combines individual files into a single executable program.

- ◎ Step4: The OS loader 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.

# Assembler:

- ◎ **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.

# Linker

- 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

- ◎ **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

- **Debugger** is a utility program, that lets you step through a program while it's running and examine registers and memory

# 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.

# Example of a partial listing file:

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.
- ◎ It provides direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system.

# Translating Languages

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



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



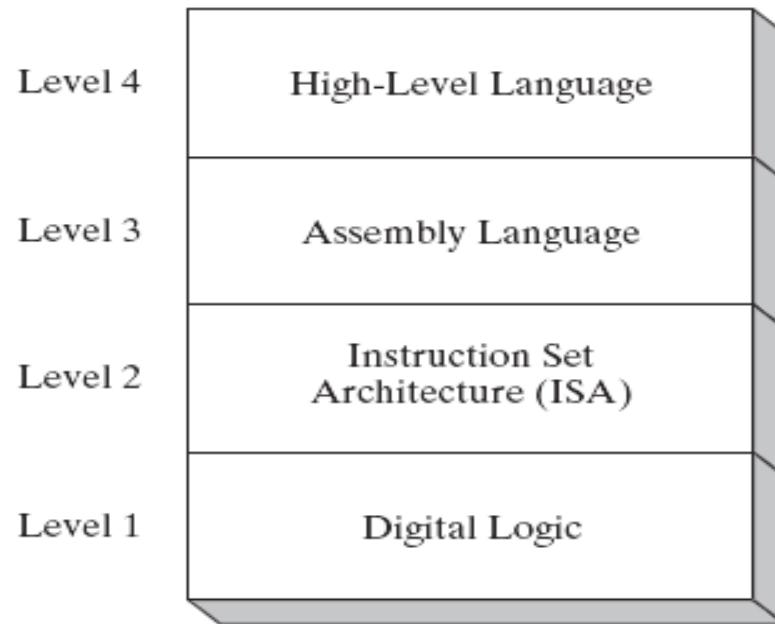
Assembly Language:

```
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

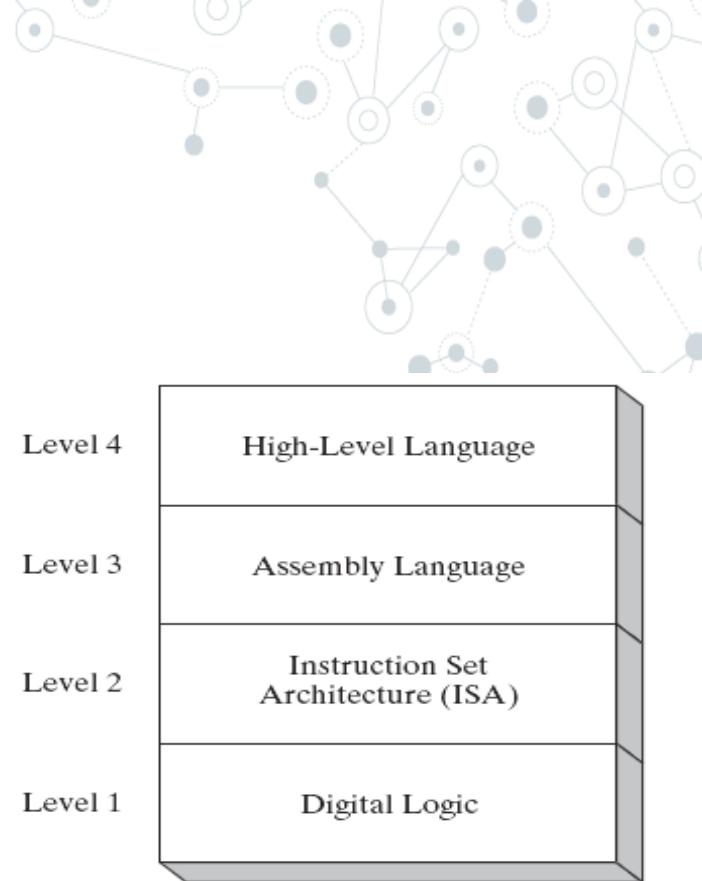


(descriptions of individual levels follow . . . )

# High-Level Language

## Level 4

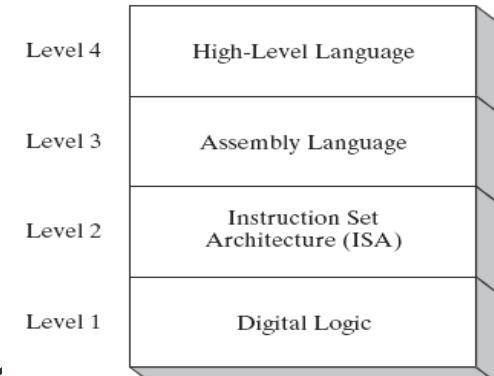
- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- Programs compile into assembly language  
(Level 3)



# Assembly Language

## Level 3

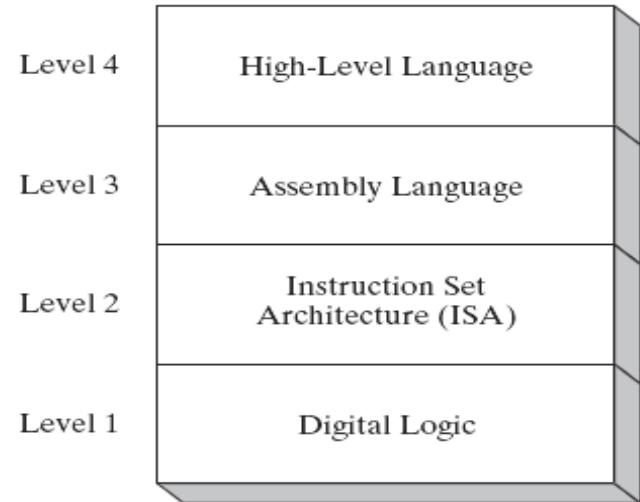
- Instruction mnemonics that have a one-to-one correlation to machine language
- Programs are translated into Instruction Set Architecture Level - machine language (Level 2)
- The instructions in assembly language may directly match the computer's architecture or they may be translated during execution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



# Instruction Set Architecture (ISA)

## Level 2

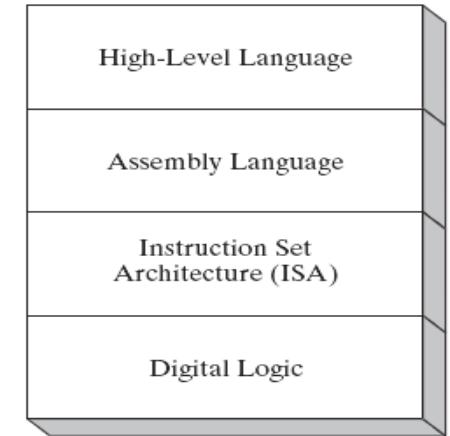
- Also known as **conventional machine language**
- Executed by Level 1 (Digital Logic)



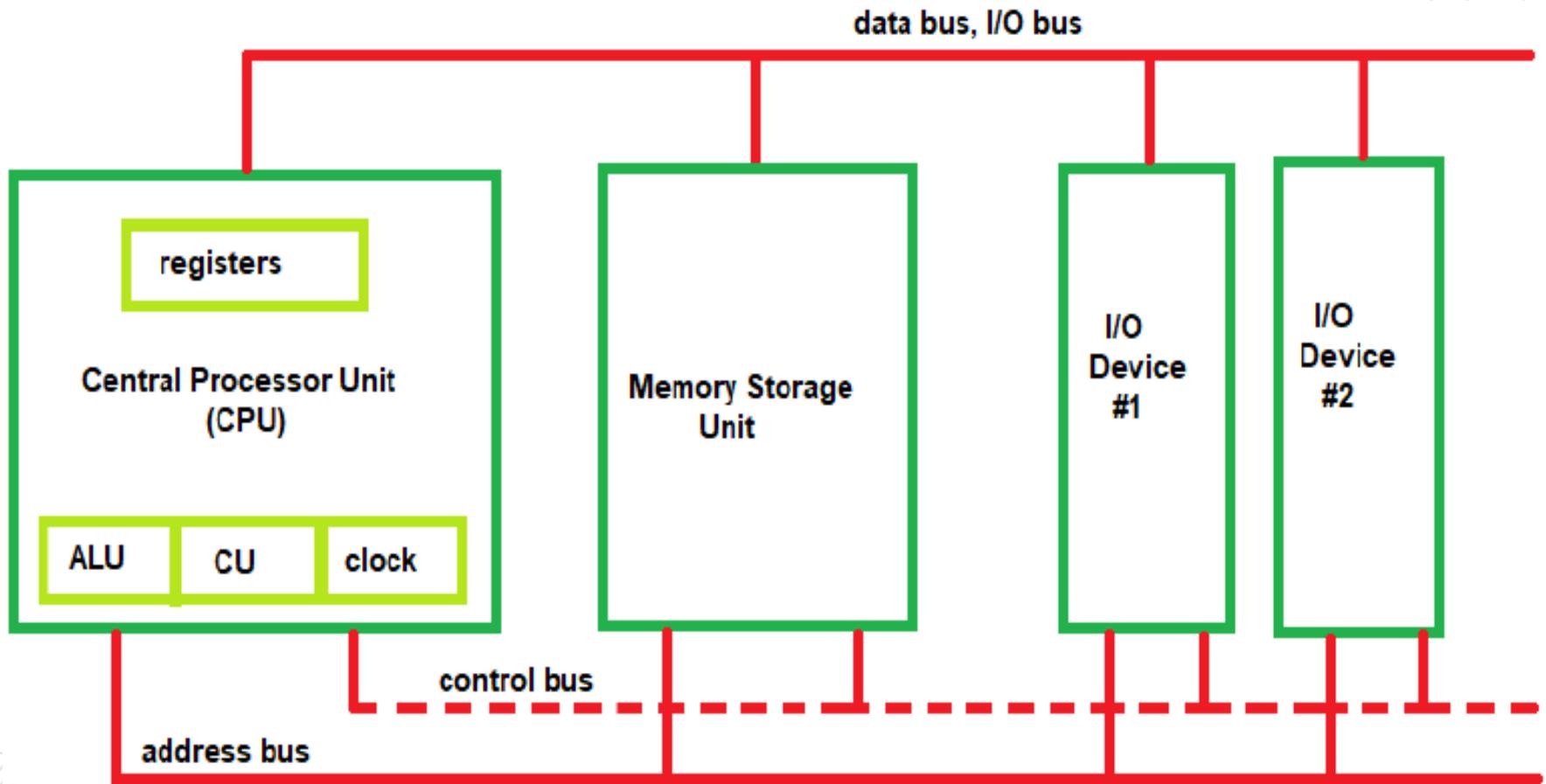
# Digital Logic

## Level 1

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



# Basic Microcomputer Design



# Basic Microcomputer Design

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

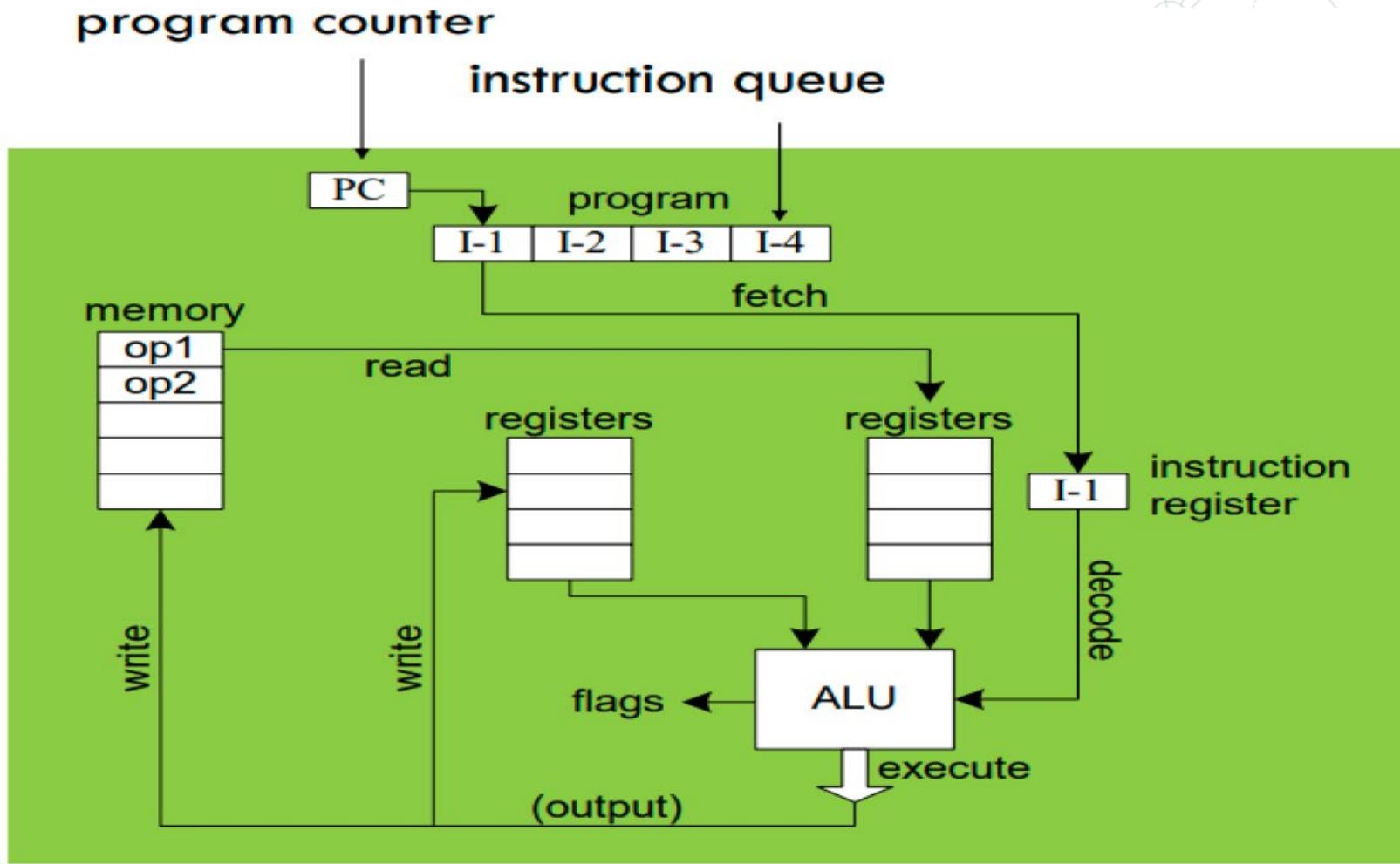
- A *bus* is a group of parallel wires that transfer data from one part of the computer to another.
- A computer system usually contains **four bus types: data, I/O, control, and address.**
- The *data bus* transfers instructions and data between the CPU and memory.
- The *I/O bus* transfers data between 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.

# Clock Cycles

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

# Instruction Execution Cycle

**Fetch**  
**Decode**  
**Fetch operands**  
**Execute**  
**Store output**



# Instruction Execution Cycle

Here are the steps to execute it:

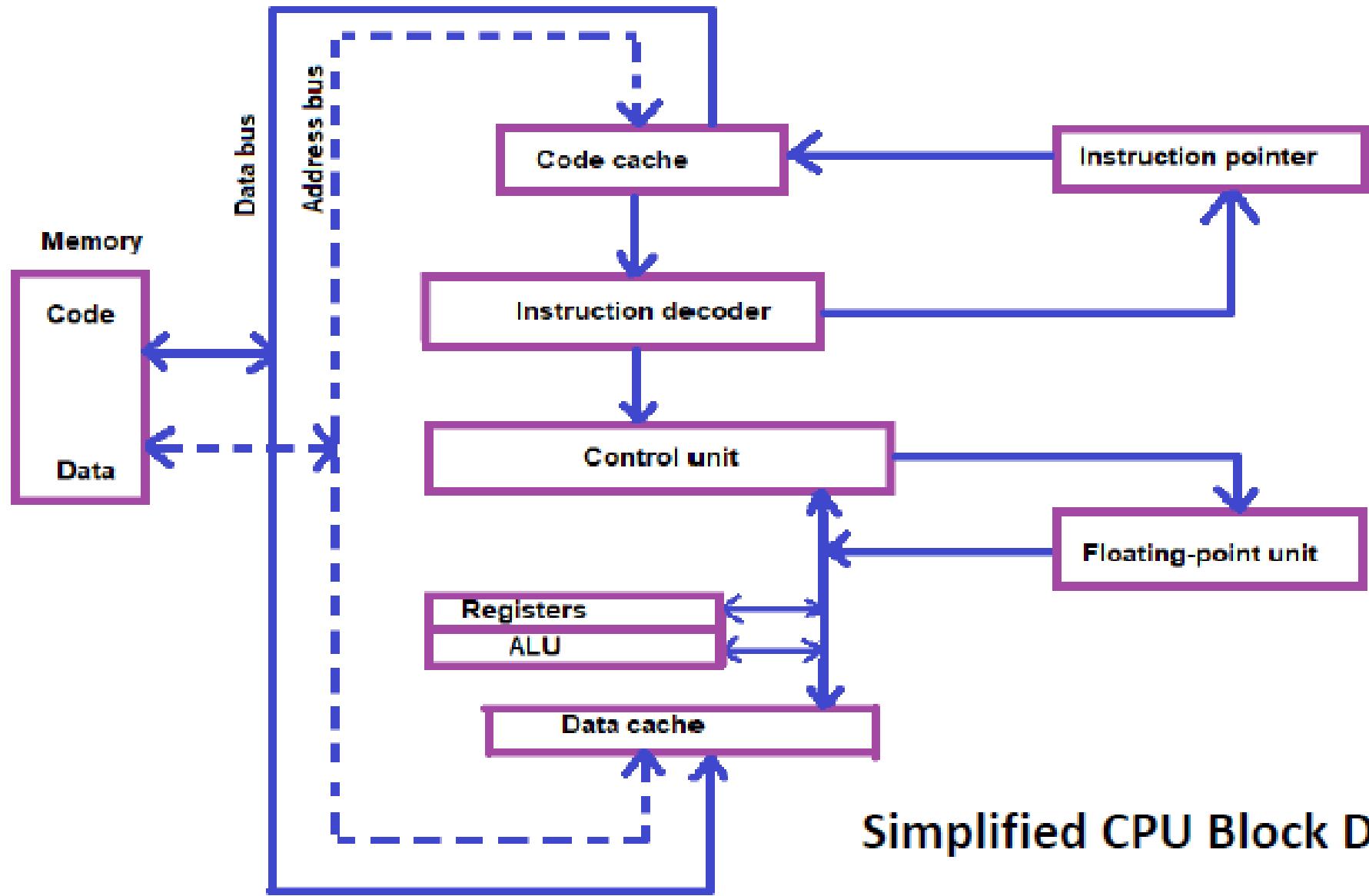
1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction queue*. It then increments the instruction pointer.
2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern.
  - This bit pattern might reveal that the instruction has operands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

# Instruction Execution Cycle

- ◎ An *operand* is a value that is either an input or an output to an operation.
- ◎ For example, the expression  $Z = X + Y$  has **two input operands** ( $X$  and  $Y$ ) and a single **output operand** ( $Z$ ).
- ◎ In order **to read program instructions from memory**, an address is placed on the **address bus**.
- ◎ Next, the **memory controller** places the requested code on the data bus, making the code available inside the code cache.

# Instruction Execution Cycle

- ◎ The **instruction pointer's** value determines which instruction will be executed next.
- ◎ The instruction is analyzed by the **instruction decoder**, causing the appropriate digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit.
- ◎ **Control bus** carries signals that use the system clock to coordinate the transfer of data between different CPU components.



Simplified CPU Block Diagram

# 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

# Computer Organization And Assembly Language

Week 2

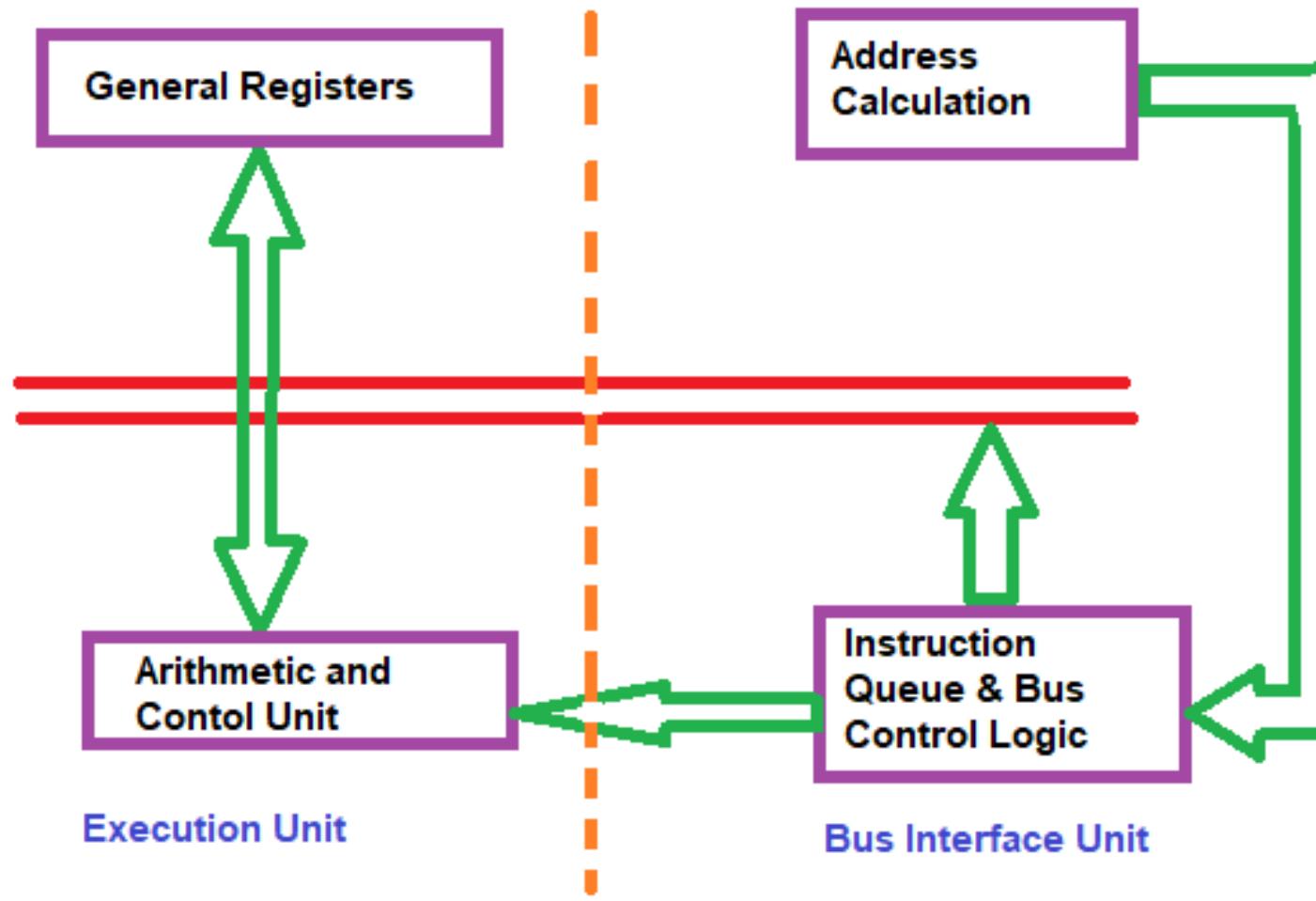
# Execution Unit and Bus Interface Unit

- Instructions executed by a CPU are called its **instruction set**. These instructions are unique for a particular architecture. Machine instructions are simple but their execution is complex.
- The 8086 CPU logic has been divided into 2 functional units, namely Execution Unit (EU) and Bus Interface Unit (BIU).
- **Execution Unit:** The purpose of this unit is to execute instructions. All the actual processing takes place in this unit.
- **Bus Interface Unit:** It facilitates the communication between the EU and the memory or I/O circuits. It is responsible for transmitting addresses, data and control signals on the buses.

# Execution Unit and Bus Interface Unit

- The purpose of this separation is to increase the processing speed of the processor. While the EU is executing an instruction, the BIU fetches up to six byte (for 8086) of the next instruction and places them in the instruction queue. This operation is called **instruction prefetch**.

# Execution Unit and Bus Interface Unit



# 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

- Computers read memory much more slowly than they access internal registers. CPU designers figured out that computer memory creates a speed bottleneck because most programs have to access variables. To reduce the amount of time spent in reading and writing memory the most recently used instructions and data are stored in high-speed memory called *cache*.
- The idea is that a program is more likely to want to access the same memory and instructions repeatedly, so cache keeps these values where they can be accessed quickly.
- Also when the CPU begins to execute a program, it can look ahead and load the next thousand instructions (for example) into cache, on the assumption that these instructions will be needed in the near future.

# Cache

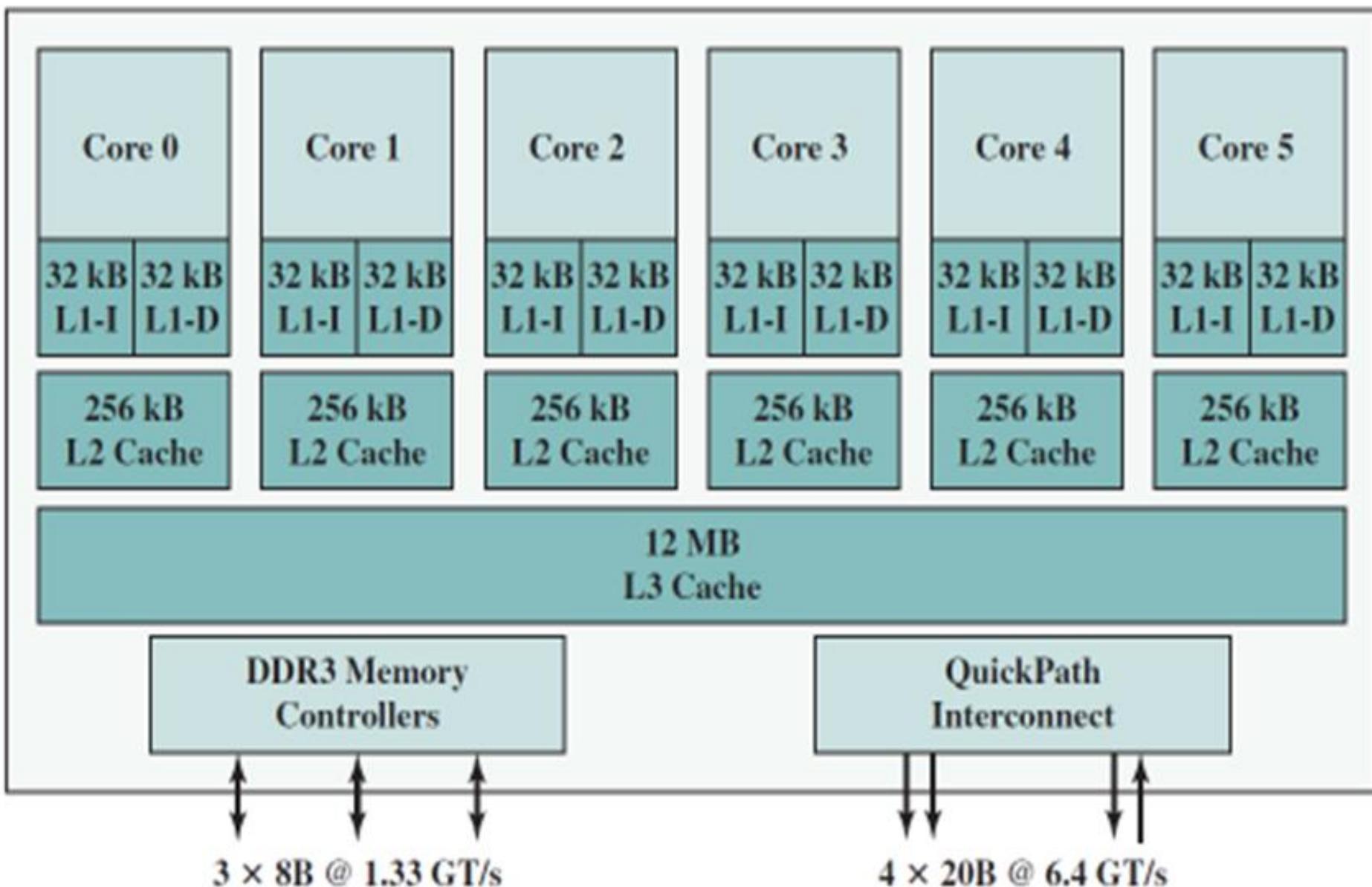
- If there happens to be a **loop** in that block of code, the same instructions will be in cache.
- When the processor is able to find its data in cache memory, we call that a ***cache hit***.
- On the other hand, if the CPU tries to find something in cache and it's not there, we call that a ***cache miss***.

# X86 family cache types

- Cache memory for the x86 family comes in two types.
  - *Level-1 cache* (or *primary cache*) is stored right on the CPU. It is extremely fast but relatively small.
  - *Level-2 cache* (or *secondary cache*) is a little bit slower and attached to the CPU by a high-speed data bus. It is often more capacious than L1 cache.

# 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.
- On the other hand, conventional memory, known as *dynamic RAM*, must be refreshed constantly.
  - It's much slower, but cheaper.



Intel Core i7-990X Block Diagram

# Loading and Executing a Program (1)

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

## Loading and Executing a Program (2)

- 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.

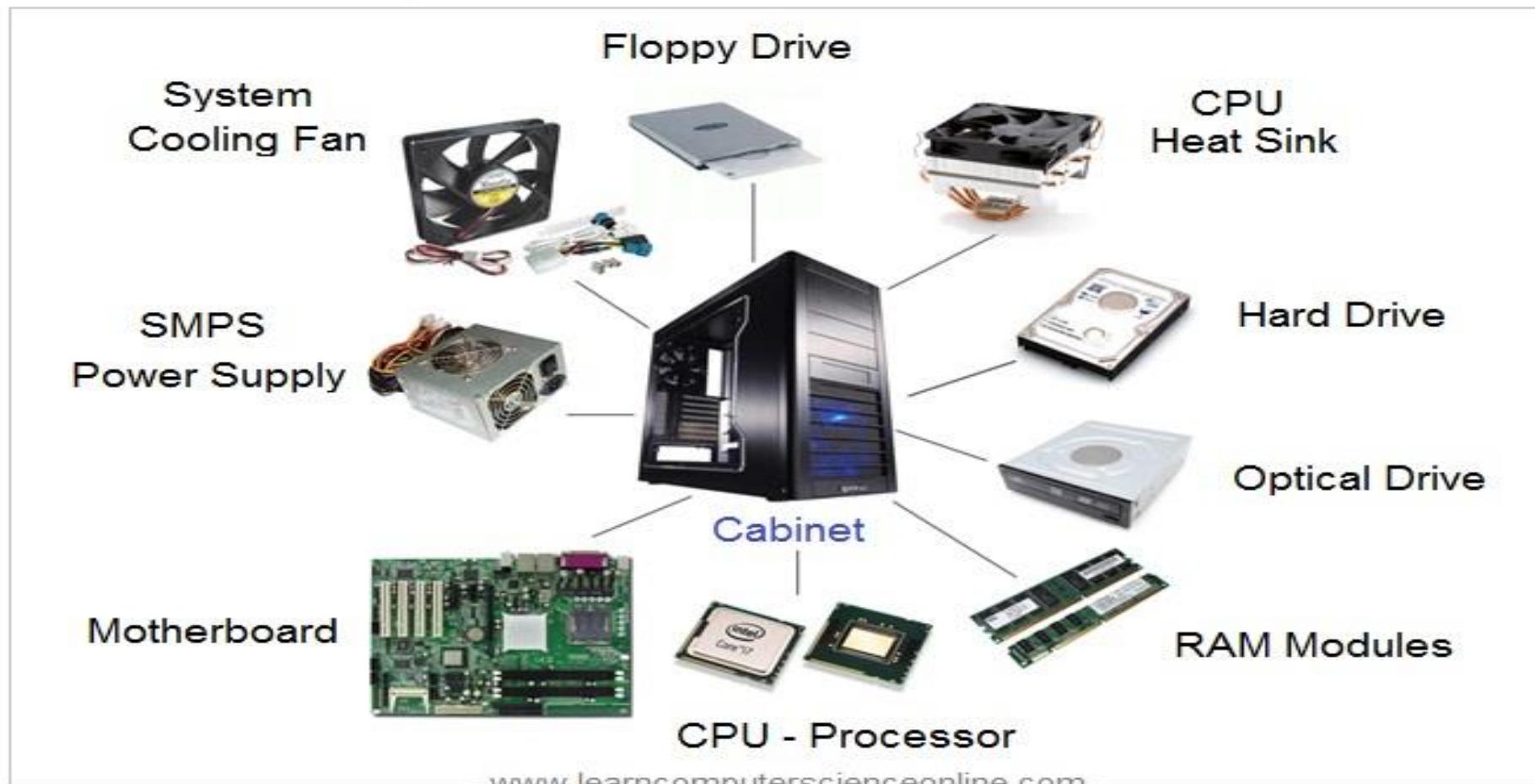
# Virtual Machine

- A virtual machine is a program on a computer that works like it is a separate computer inside the main computer.
- Virtual machines differ and are organized by their function:
- **System virtual machines** (also termed full virtualization VMs) provide a substitute for a real machine. They provide functionality needed to execute entire operating systems. A hypervisor uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine.
- **Process virtual machines** are designed to execute computer programs in a platform-independent environment.

# Virtual Machine Concept

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

**Computer System** - Internal Hardware Components



# Virtual Machine Concept

- SOFTWARES → Makes Computer easy to use.



# Virtual Machine Concept

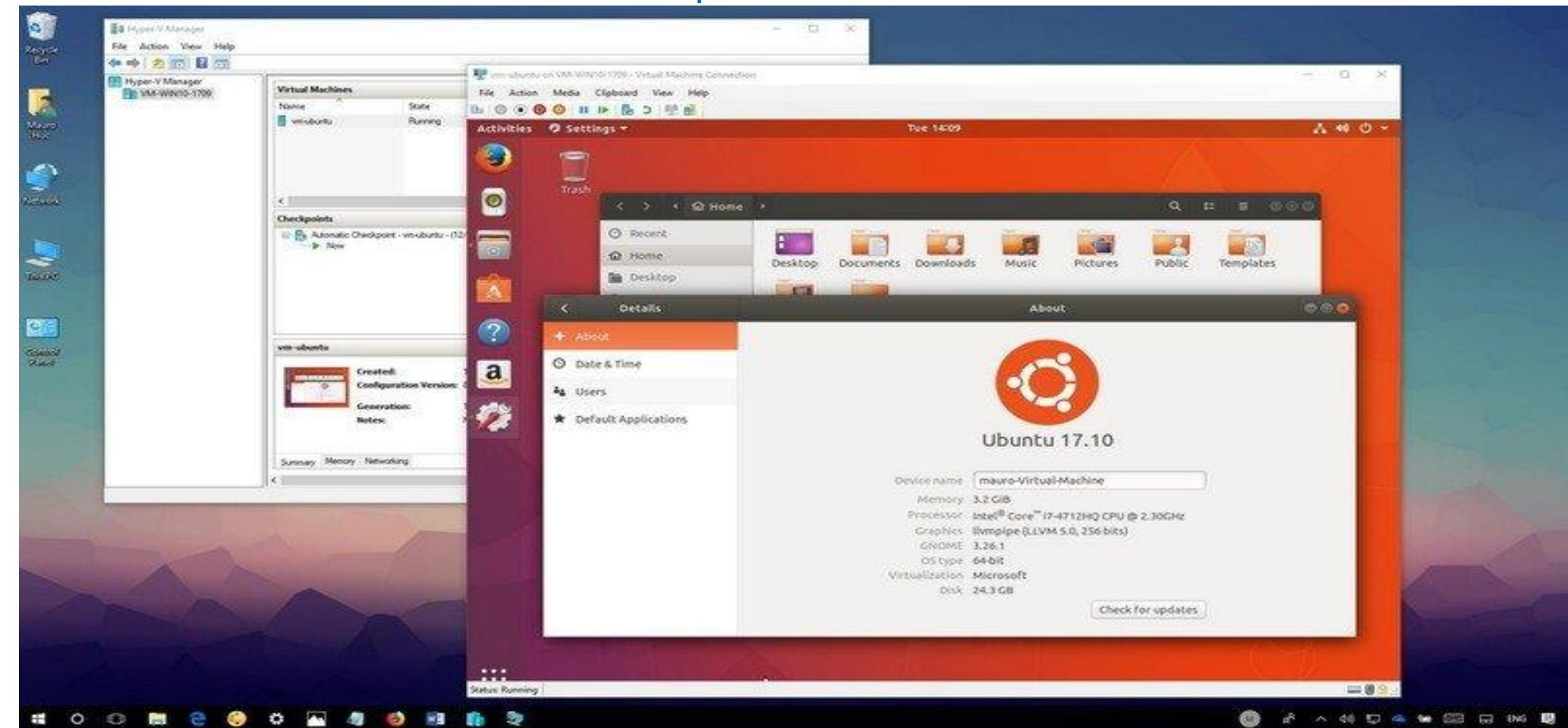
- **OPERATING SYSTEM SOFTWARES:** They are able to control Physical Components of Computer i.e. HARDWARE. They manage computer hardware, **software resources**, and provides common services for computer programs.

# Virtual Machine Concept

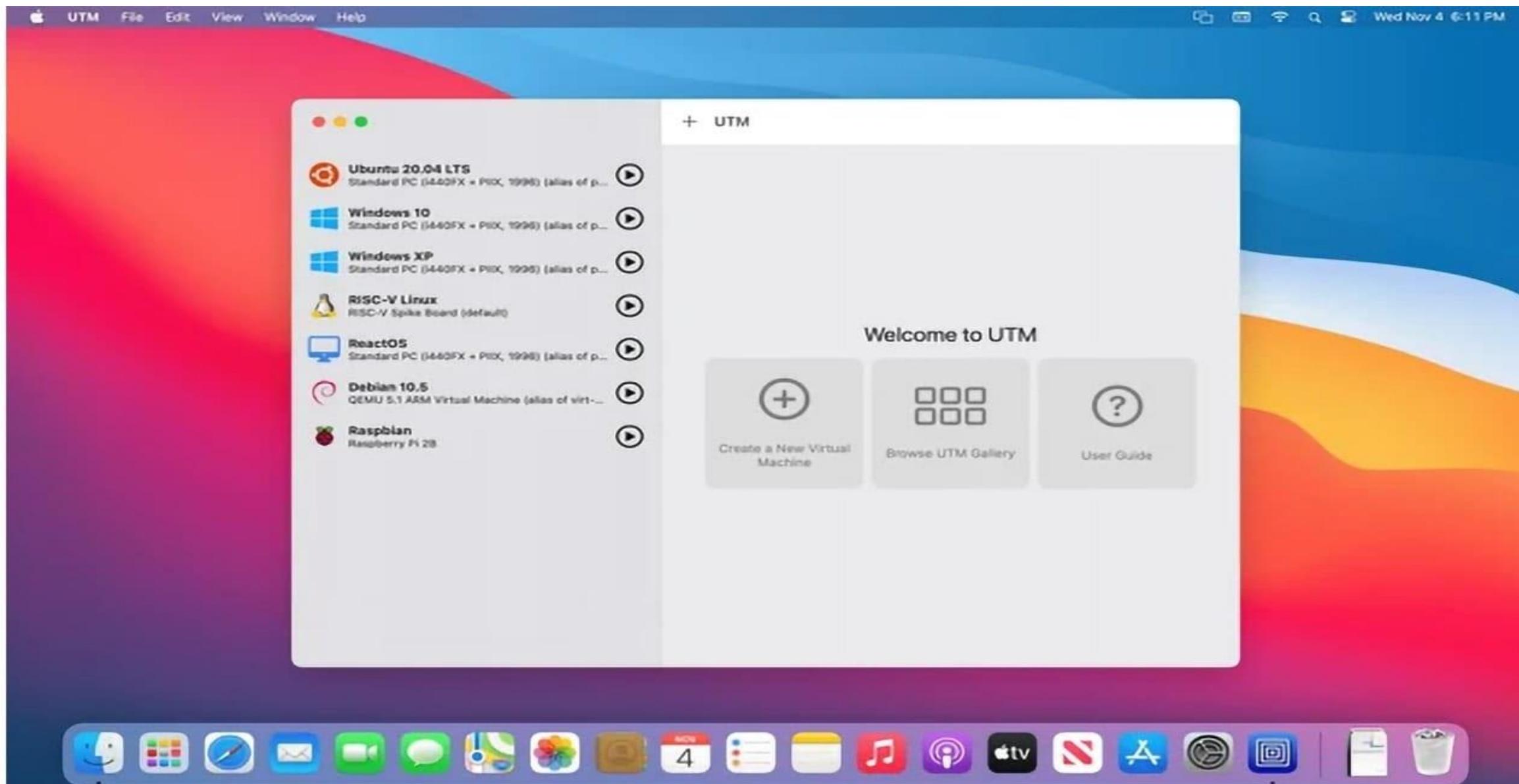
- VIRTUAL MACHINE MONITOR (HYPERVISOR)
- Software → allows to run → an Operating System → Within another Operating System.



# Virtual Machine Concept



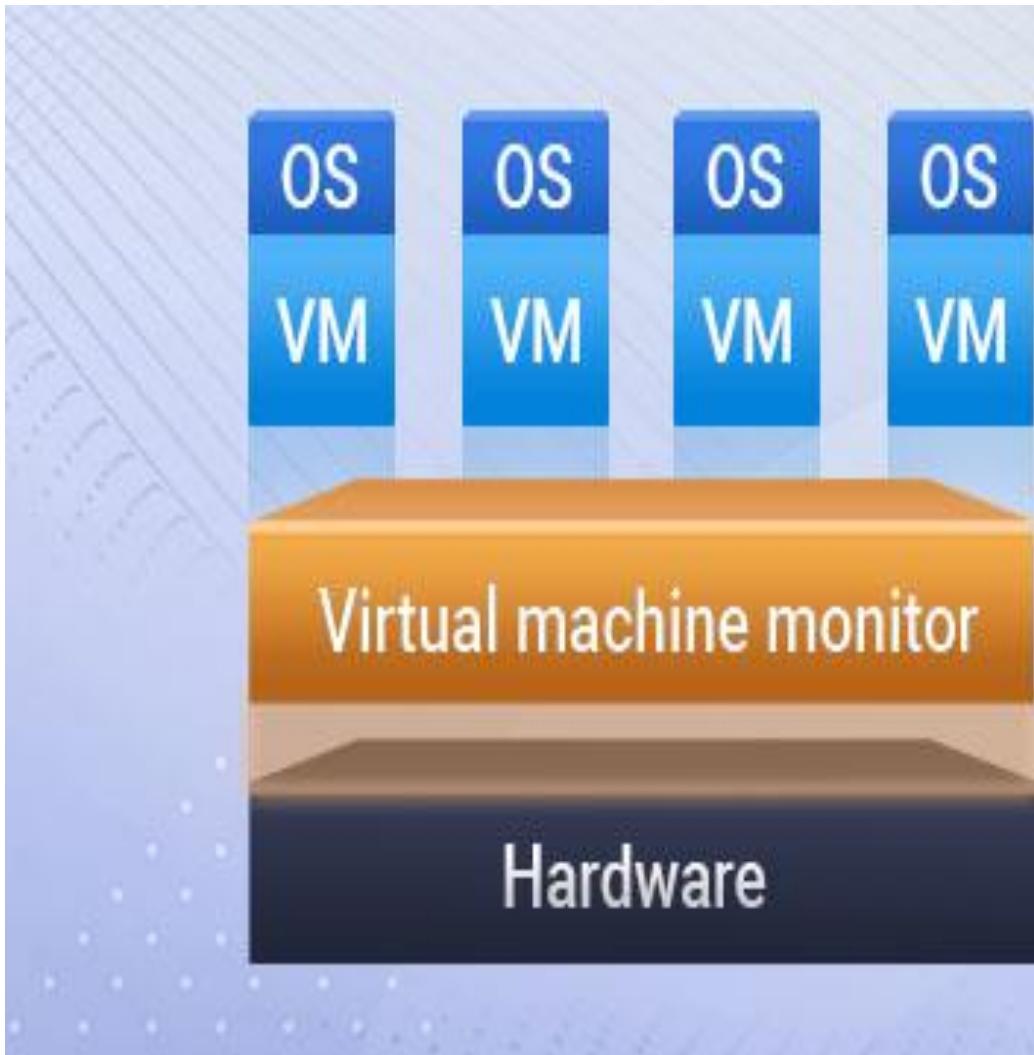
# Virtual Machine Concept



# Virtual Machine Concept



# Virtual Machine Concept

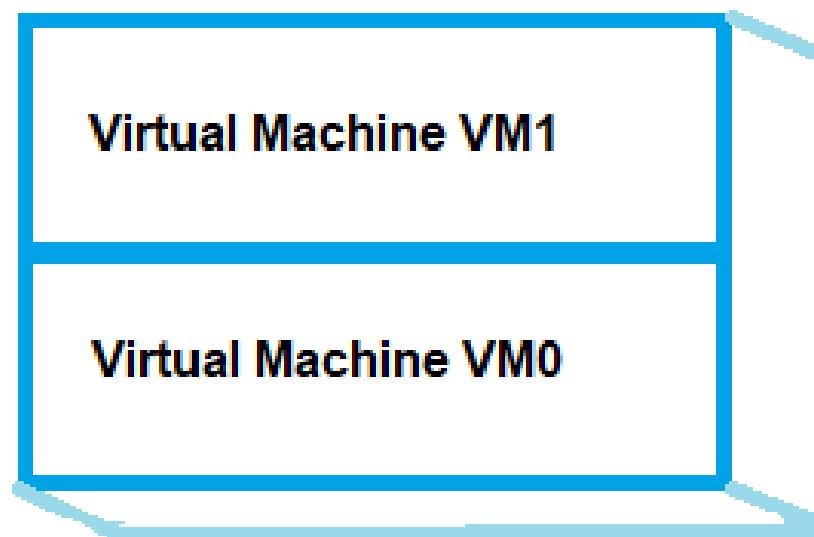


# Virtual Machine Concept (Process Virtual Machine)

- Process virtual machines are designed to execute computer programs in a platform-independent environment.
- 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.

# Virtual Machine Concept

- The virtual machine **VM1**, can execute commands written in language L1.
- The virtual machine **VM0** can execute commands written in language L0.

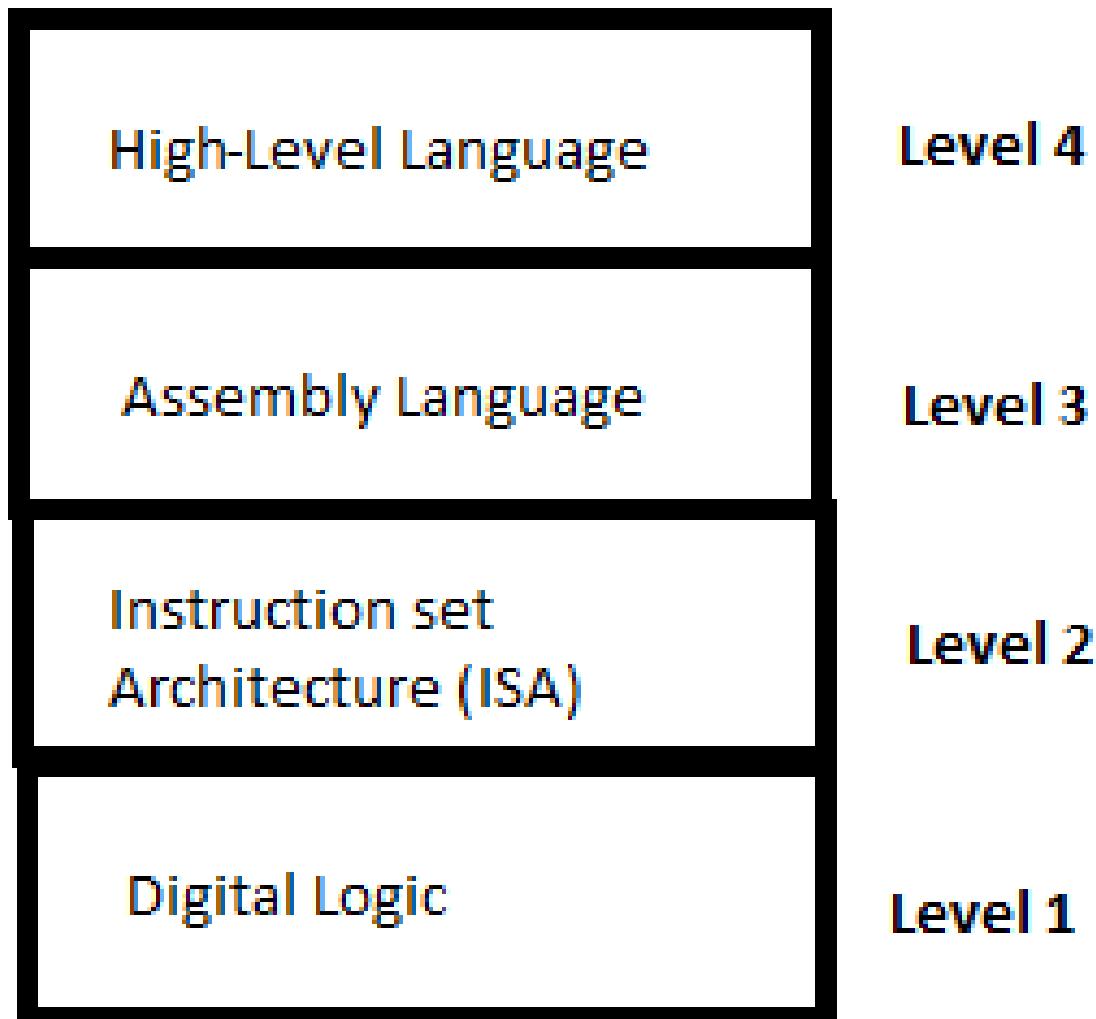


# Virtual Machine Concept

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



# Specific Machine Levels



# High-Level Language

## Level 4

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

# Assembly Language

## Level 3:

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

# 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

# 32-bit x86 processors

- x86 is a family of instruction set architectures based on Intel 8086 microprocessor and its 8088 variant.
- Many additions and extensions have been added over the years.
- Intel introduced its first 32-bit microprocessor, the 80386 in 1985.
- IA-32 (short for "Intel Architecture, 32-bit"), sometimes also called i386 is the 32-bit version of the x86 instruction set architecture, designed by Intel and first implemented in the 80386 microprocessor in 1985.
- In our course we will be discussing basic architectural features of all x86 processors which include Intel IA-32 and 32-bit AMD(Advanced Micro Devices) processors.

# Modes of Operation

- 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)

# Protected Mode (introduced with 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
- ❖ Native mode used by Windows NT, 2000, XP and Linux
- ❖ In x86 protected mode, segment registers hold pointers to segment descriptor tables

- ❖ **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.
  - ❖ 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

# 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

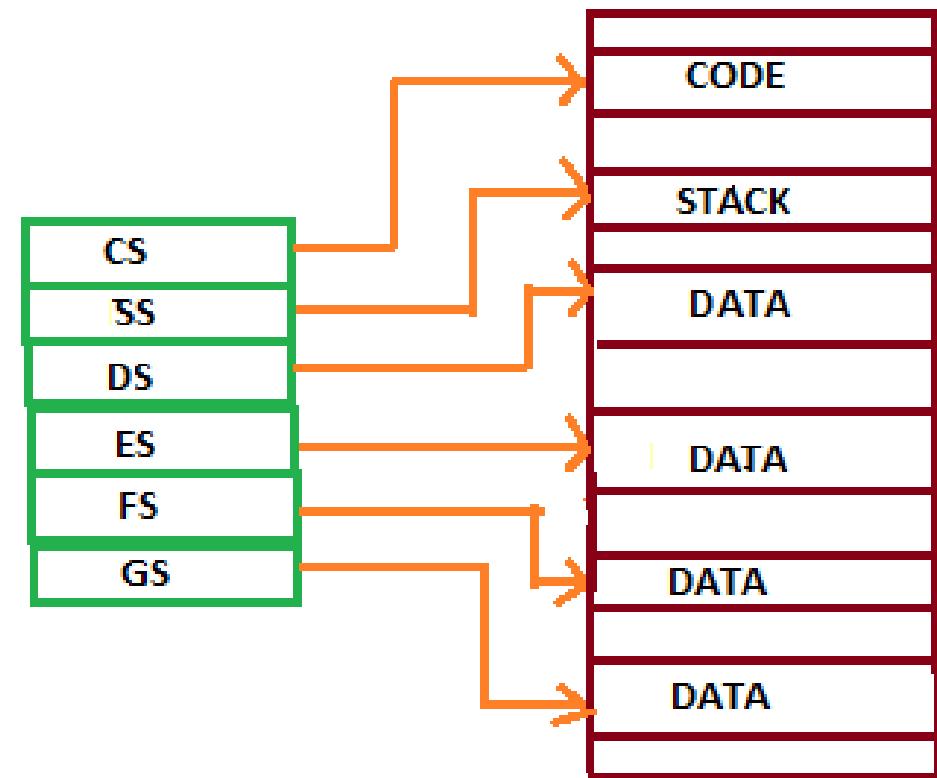
# Basic Execution Environment

# 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

# Real Address Mode(1)

- A program can access up to six segments at any time
  - ❖ Code segment
  - ❖ Stack segment
  - ❖ Data segment
  - ❖ Extra segment ( up to 3)
  - ❖ Logical address
- ❑ Segment = 16 bits
- ❑ Offset = 16 bits
- ❖ Linear (physical) address = 20 bits



# Real Address Mode(2)

- **Segmentation:** Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.
- The number of address lines in 8086 is 20, 8086 BIU will send 20-bit address, so as to access one of the 1MB memory locations. The segment registers actually contain the upper 16 bits of the starting addresses of the memory segments with which the 8086 is working at that instant of time. Each segment is made up of contiguous memory locations.

# Real Address Mode(3)

- **Program Segments:**

A memory segment is a block of consecutive memory bytes. Each segment is identified by a segment number. A segment number is 16 bits. Within a segment, a memory location is specified by giving an offset.

A typical machine language program is composed of instructions(code) and data. Processor also uses stack to implement procedure calls. The program's code, data and stack are loaded into different memory segments(code segment, data segment and memory segment).

The choice of segment is normally defaulted by the processor according to the function being executed. Instructions are always fetched from the code segment. Any stack push or pop or any data reference referring to the stack uses the stack segment. All other references to data use the data segment.

- **Segment Registers:** Segment registers are used to hold base addresses for the program code, data and stack.
- **Code segment register (CS):** is used for addressing memory location in the code segment of the memory, where the executable program is stored.
- **Data segment register (DS):** points to the data segment of the memory where the data is stored.
- **Extra Segment Register (ES):** also refers to a segment in the memory which is another data segment in the memory.
- **Stack Segment Register (SS):** is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data. The segment is also used to store interrupt and subroutine return addresses

## Real Address Mode(4)

- Because of the way the segment address and offset are added, a single linear address can be mapped to more than one segment:offset pairs.

## Real Address Mode(5):

- To obtain a 20-bit physical address, the 8086 microprocessor first shifts the segment address 4 bits to the left(this is equivalent to multiplying by 10 h), and then adds the offset.

Linear Address = Segment x 10(hex) + offset

# Real Address Mode(6):

- Example 1:

Segment = A1F0(hex)

Offset = 04C0(hex)

Logical address = A1F0:04C0 (hex)

What is linear address?

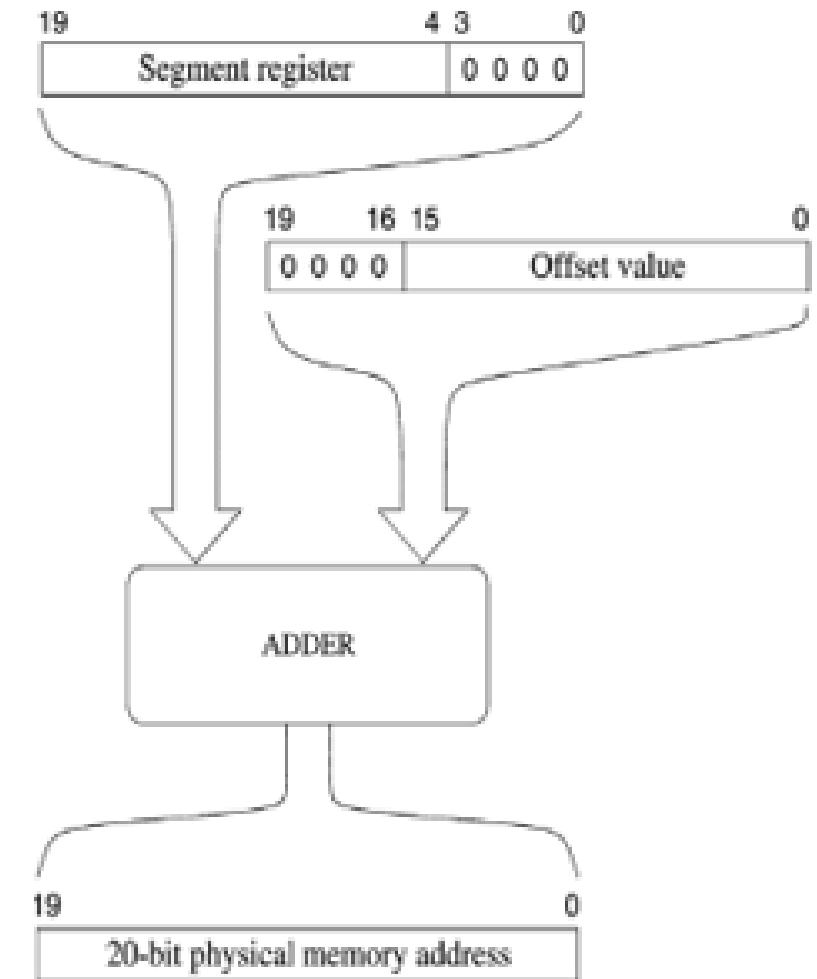
- Solution:

A1F00 ( add 0 to segment in hex)

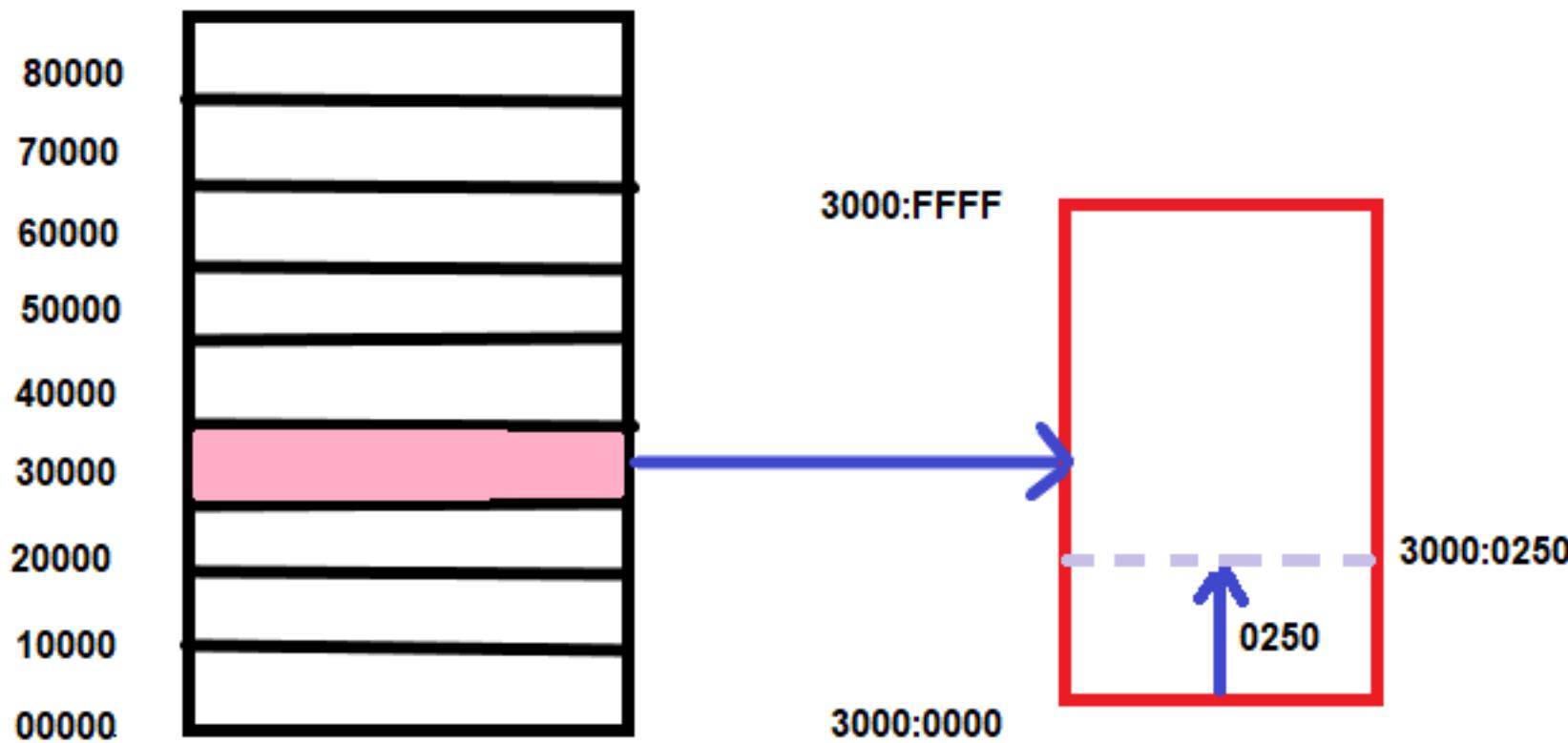
+ 04C0 (offset in hex)

---

A23C0 ( 20-bit linear/physical address in hex)



## Real Address Mode(7):



Segmented Memory map, Real-Address Mode

## Example 2 ( Example 3.1 book2)

For a memory location whose physical address is specified by 1256Ah, give the address in segment:offset form for segments 1256h and 1240h.

**Answers:**

Offset in segment 1256h is Ah

Offset in segment 1240h is 16Ah

1256Ah = 1256:000A = 1240:016A

## References:

- Textbook and reference books mentioned in course outline

# Computer Organization And Assembly Language

Week 3

# Addressable memory (32-bit x86 Processors)

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

# Address Space(32-bit x86 Processors)

- 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

# Protected Mode

In protected mode segment registers hold pointers to segment descriptor tables. Segment descriptor table contains segment descriptors. Segment descriptor contains segment's base address, access rights, size limit, type, and usage.

## Important terminologies:

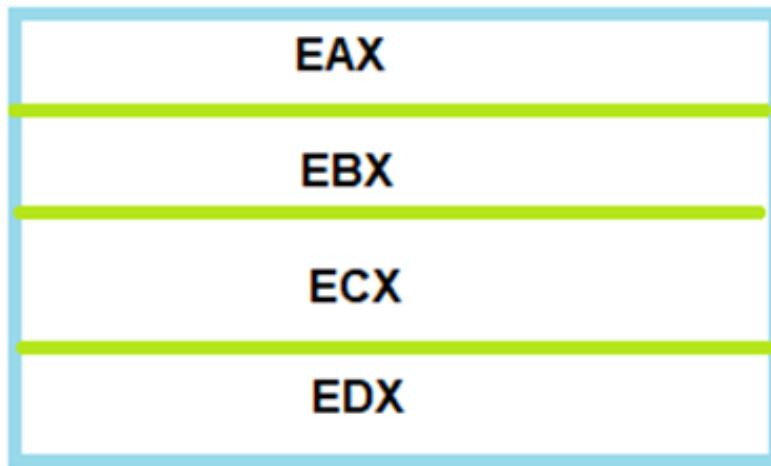
- **Segments**
  - ❑ variable-sized areas of memory for code & data
- **Segment descriptor**
  - ❑ contains segment's base address, access rights, size limit, type, and usage
- **Segment descriptor table**
  - ❑ contains segment descriptors by which OS keep track of locations of individual program segments
- **Segment registers**
  - ❑ points to segment descriptor tables

# Basic Program Execution Registers(32-bit x86 Processors)

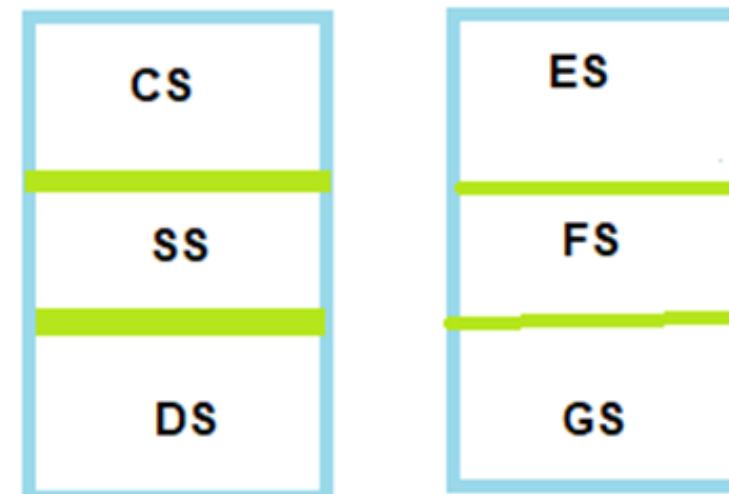
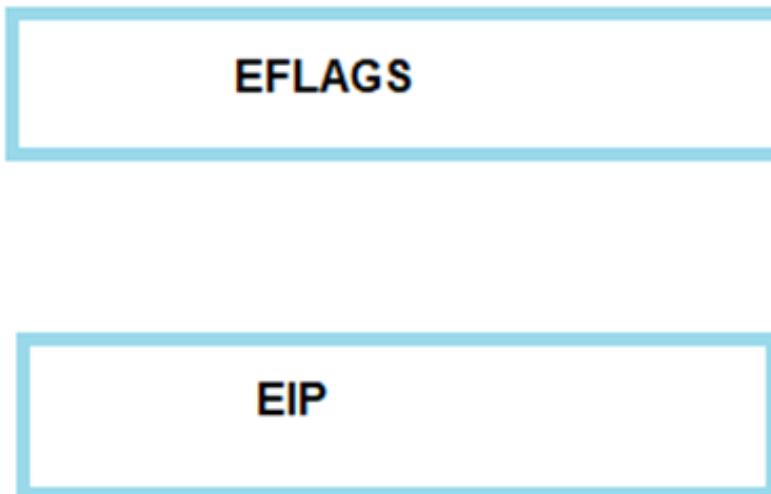
- *Registers* are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory
- There are eight 32-bit general-purpose registers(which can hold either integer data operands or addressing information), six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP). Eight additional registers are available for floating-point instructions. These registers are also used by the multimedia instructions. There is another set of registers used by vector-processing instructions.

# Basic Program Execution Registers

32-bit General-Purpose Registers

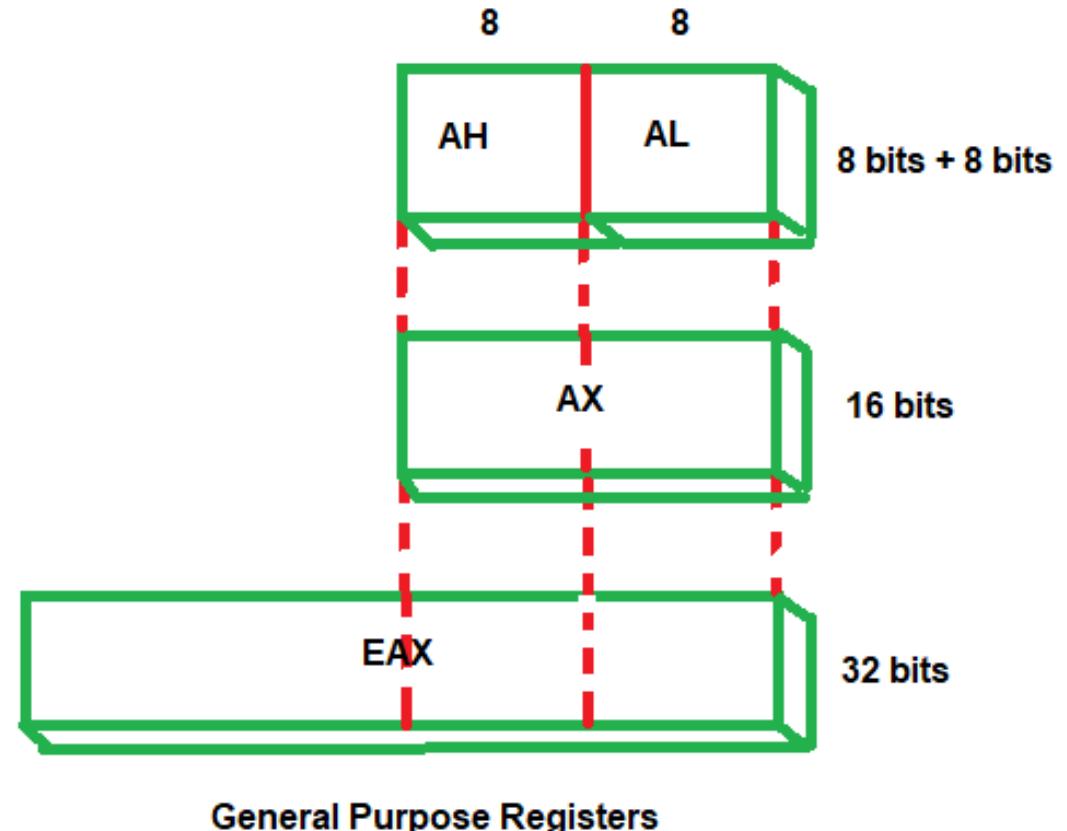


16-bit segment Registers



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

- As shown in Figure below, 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



<b>32-Bit</b>	<b>16-Bit</b>	<b>8-Bit (High)</b>	<b>8-bit (Low)</b>
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:

<b>32-Bit</b>	<b>16-Bit</b>
ESI	SI
EDI	DI
EBP	BP
ESP	SP

# Specialized Uses of general-purpose registers (1)

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(Count)** - The CPU automatically uses **ECX** as a counter for looping operations
- **EDX (Data)** - Used in multiplication and division operations

# Specialized Uses of general-purpose registers(2)

**Pointer and 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 address 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. They are used by high-speed memory transfer instructions.
- **EBP** is used to reference function parameters and local variables on the stack.

- **Segment Registers:**
  - In real-address mode, **16-bit segment registers indicate base addresses of pre-assigned memory areas named segments.**
  - In protected mode, **segment registers hold pointers to segment descriptor tables** (The descriptor describes the location, length and access rights of the memory segment).
  - Some segments hold program instructions (code), others hold variables (data), and another segment named the stack segment holds local function variables and function parameters.
- **Instruction Pointer :**
  - The EIP, or instruction pointer, register contains the address of the next instruction to be executed.
  - Certain machine instructions manipulate EIP, causing the program to branch to a new location.

# EFLAGS Register:

The EFLAGS register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation.

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



<b>O = Overflow</b>	<b>S = Sign</b>
<b>D = Direction</b>	<b>Z = Zero</b>
<b>I = Interrupt</b>	<b>A = Auxiliary Carry</b>
<b>T = Trap</b>	<b>P = Parity</b>
<b>C = Carry</b>	

# Flags:

Two types of flags:

- **Control Flags** determine how instructions are carried out.
- **Status Flags** indicate the results of arithmetic and logical operations performed by CPU.

## Control Flags:

**Direction Flag(DF):** affects the direction of block data transfers( like long character string). It controls the left-to-right or right-to-left direction of string processing.

**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 CPU is halted after every instruction. Used for debugging process.

## Status Flags:

**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 0 result. 1= zero; 0 = not zero.

**Auxiliary carry flag**- set when an operation causes a carry from bit 3 to 4 or borrow (from bit 4 to 3). 1 = carry; 0 = no carry.

**Parity flag**- used to verify memory integrity. Even # of 1s = Even parity; odd # of 1s = odd parity. It is set if the least-significant byte in the result contains an even number of 1 bits.

# Floating point, MMX, XMM registers

## Floating point Registers:

- Eight 80-bit floating-point data registers in the FPU
- ST(0), ST(1), ..., ST(7)
- arranged in a stack
- used for all floating-point arithmetic

## MMX Registers:

- Eight 64-bit MMX registers. The MMX register names are aliases to the same registers used by floating-point unit.

## XMM Registers:

- Eight 128-bit XMM registers used by streaming SIMD (single-instruction multiple-data) extensions to the instruction set.

# X86 Memory Management

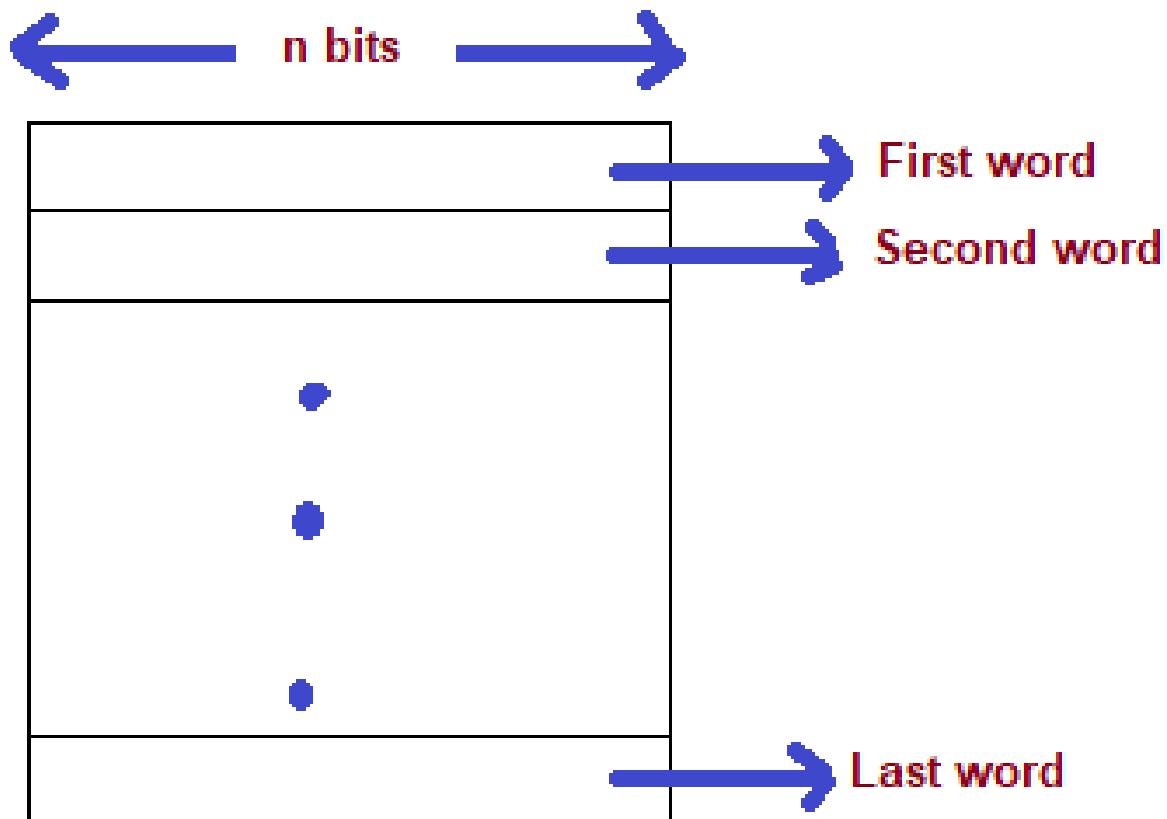
x86 processors manage memory according to the basic mode of operation. Protected mode is the most powerful and robust but it does restrict application programs from directly accessing hardware.

## IA-32 memory addressing

- The IA-32 architecture has different models for accessing the memory.
- The **segmented memory model** associates different areas of the memory, called segments, with different usages. The code segment holds the instructions of a program. The stack segment contains the processor stack, and four data segments are provided for holding data operands.
- The other model is **flat memory model** of the IA-32 architecture, where a 32-bit address can access a memory location anywhere in the code, processor stack, or data areas. Entire address space is described by a 32-bit segment, which provides  $2^{32} = 4$  gigabytes of address space. Program can (in theory) access up to 4 gigabytes of virtual or physical memory.

# Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1.
- The memory is organized so that a group of  $n$  bits can be stored or retrieved in a single, basic operation. Each group of  $n$  bits is referred to as a word of information, and  $n$  is called the word length. The memory of a computer can be schematically represented as a collection of words.



Memory words

# Instructions

- An **instruction** is a statement that becomes executable when a program is assembled.
- Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at run time.
- We use the Intel IA-32 instruction set

An instruction contains:

- Label (optional)
- Mnemonic (required)
- Operand (depends on the instruction)
- Comment (optional)

**Basic syntax:**

- [label:] mnemonic [operands] [ ; comment]

## Labels:

- An identifier that acts as place markers for instructions and data.
- marks the address (offset) of code and data
- Implies the variable's or instruction's address when placed before them
- Follow identifier rules

## Data label:

- Identifies location of a variable
- Used to reference variable in a code
- must be unique within its enclosing procedure

**Example:** (not followed by colon)

```
count  DWORD  100  
Array  DWORD  1024,  2048  
          DWORD  4096,  8192
```

## Code label:

- Code labels are in the code segment, and are offsets for transfer of control instructions.
- target of jump and loop instructions
- Must end with a colon (:) character
- A code label can share the same line with an instruction, or it can be on a line by itself

## Example:

L1:

L2: mov eax, ebx

# Mnemonics and Operands

- **Instruction Mnemonics**

- memory aid
- Short word that identifies an instruction
- They give the hint about type of operation they perform

Examples: MOV, ADD, SUB, MUL, INC, DEC

- **Operands**

- A value used for input or output for an instruction
- Assembly language instruction can have between 0 and 3 operands
- Operands can be a register, memory operand, integer expression, or input-output port

Examples:

— constant	96
— constant expression	2 + 4
— register	eax
— memory (data label)	count

Constants and constant expressions are often called **immediate values**

# Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One operand
  - inc eax ; register
  - inc myByte ; memory
- Two operands
  - add ebx, ecx ; register, register
  - sub myByte, 25 ; memory, constant
  - add eax, 36 \* 25 ; register, constant-expression
- Three operands
  - imul eax,ebx,5 ; register, register, integer constant

# Instruction Format Examples

- There is a natural ordering of operands.
- When instructions have multiple operands, the first one is typically called the destination operand.
- The second operand is usually called the source operand.
- In general, the contents of the destination operand are modified by the instruction.

# Comments

- Ignored by assembler
- Writer provides information about the program through comments.

## Single-line comments:

- Begin with semicolon(;) and all characters occurring after semicolon on the same line are ignored by the assembler.
- Example: MOV AX, 2h ; The value of AX is 2h

## Block Comments:

- Begin with comment directive and user-specified symbol. All subsequent lines of text are considered as comments and ignored by assembler until the same user-specified symbol is written again.
- Example: MOV AX, 2h

```
COMMENT @  
this line is a comment  
this line is comment  
@
```

# NOP ( No Operation) Instruction

- Takes 1 byte of program storage
- Does not do any work
- Used to align code to efficient address boundaries
- **Flags:** Affected none
- **Syntax:** NOP
- **Example:**
  - 00000000 66 8B C3 mov ax,bx
  - 00000003 90 NOP ; align next instruction
  - 00000004 8B D1 mov edx, ecx
- x86 processors load data and code more quickly from even doubleword addresses

# Integer Constants (integer literals)(1)

- Format:

[{+|-}] digits [radix]

- Possible radix values:

h	Hexadecimal
q/o	octal
d	decimal
b	binary
r	encoded real
t	decimal (alternate)
y	binary (alternate)

# Integer Constants (integer literals)(2)

- Examples:

34 ; no radix means it is a decimal value

45d

35q

23o

0Bh

34h

- Practice Problem:

Using the value -35 write it as an integer literal in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax. ( stored in 8 bit format)

## References:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
2. Computer Organization and Embedded Systems by Carl Hamacher, Zvonko Vranesic, Safwat Zaky and Naraig Manjikian ( sixth Edition)
3. Assembly Language Programming and organization of the IBM PC by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 4

# Memory Locations and Addresses

- Accessing the memory to store or retrieve information, requires distinct names or addresses for each location.
- The memory can have up to  $2^k$  addressable locations. The  $2^k$  addresses constitute the address space of the computer. It is customary to use numbers from 0 to  $2^k - 1$ .
- For example, a 32-bit address creates an address space of  $2^{32}$  or 4G (4 giga) locations, where 1G is  $2^{30}$ .

# Byte Addressability

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory. The term **byte-addressable memory** is used for this assignment.
- Byte locations have addresses 0, 1, 2,....

# Big-endian and Little-endian assignment

There are two ways that byte addresses can be assigned across words:

- A **big-endian** stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest.
- A **little-endian** system stores the least-significant byte at the smallest address.

x86 processors store and retrieve data from memory using little-endian order. (low to high). The least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions.

Some other systems use big-endian order.( high to low)

## Example 1:

- Representation of 12345678h in big-endian and little-endian.

<b>0000:</b>	12
<b>0001:</b>	34
<b>0002:</b>	56
<b>0003:</b>	78

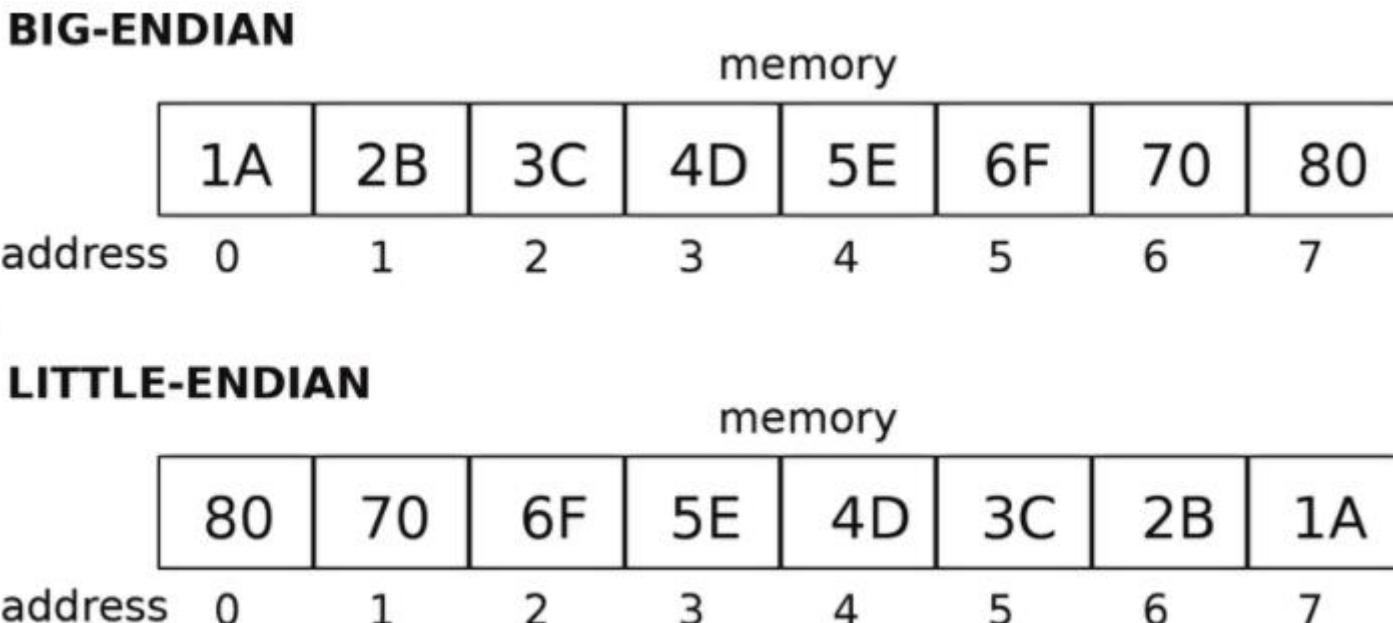
Big-endian

<b>0000:</b>	78
<b>0001:</b>	56
<b>0002:</b>	34
<b>0003:</b>	12

Little-endian

## Example 2:

- Representation of 0x1A2B3C4D5E6F7080 in big-endian and little-endian.



Representation of 0x1A2B3C4D5E6F7080 in big-endian and little-endian.

# Memory Organization of Intel IA-32 architecture.

In the IA-32 architecture, memory is byte-addressable using 32-bit addresses, and instructions typically operate on data operands of 8 or 32 bits. These operand sizes are called byte and doubleword in Intel terminology. Little-endian addressing is used.

## Practice Problem 1:

What segment addresses correspond to the linear address 28F30h? (in real mode)

Answer:

- Many different segment-offset addresses can produce the linear address 28F30h. Possible answers: 28F0:0030, 28F3:0000, 28B0:0430 ,

... .

## Practice Problem 2:

List the steps needed to execute the machine instruction:

Load R<sub>2</sub>, LOC

in terms of transfers between the components of computer and some simple control commands. Assume that the address of the memory location containing this instruction is initially in register PC. Where R<sub>2</sub> is general purpose register present in the CPU.

# Hexadecimal Addition (1)

- Consider addition of 6A2h and 49Ah. Their sum is B3Ch.

Carry	1		
Number 1	6	A	2
Number 2	4	9	A
Sum	B	3	C

# Hexadecimal Addition (2)

## Method 1:

- In the lowest digit position,  $2 + A =$  decimal 12, so there is no carry and  $12 = C$  in hexadecimal.
- In the next position,  $A+9 =$  decimal 19, so there is a carry because  $19 \geqslant 16$ , the number base. Calculate  $19 \bmod 16$ ,  $19 \bmod 16 = 3$ . So 3 will be placed in sum position and 1 will be carried into the third digit position.
- Finally add 1, 6 and 4.  $1+ 6 + 4 =$  decimal 11, which is denoted by letter B in the third position of the sum and there will be no carry.
- The hexadecimal sum is B3C.

# Hexadecimal Addition (3)

## Method 2:

- Find the decimal equivalent of 6A2h and 49Ah.  $6A2 = 1698$  decimal and  $49A = 1178$  decimal.
- Add the two decimal numbers 1698 and 1178.  $1698 + 1178 = 2876$ .
- Convert 2876 into hexadecimal.  $2876 = B3Ch$

# Hexadecimal Addition (4)

## Method 3:

- Find the binary equivalent of 6A2h and 49Ah. 6A2= 11010100010 binary and 49A= 10010011010 binary.
- Add the two binary numbers

$$\begin{array}{r} 11010100010 \\ +10010011010 \\ \hline 101100111100 \end{array}$$

- Convert 101100111100 into hexadecimal. 101100111100 = B3Ch

# Constant Integer Expressions

- A mathematical expression composed of integer constants and arithmetic operators.
- Expression must evaluate to an integer so it can be stored in 32 bits.
- They can only be evaluated at assembly time

Operator	Name	Precedence Level
()	Parentheses	1
+,-	Unary plus, minus	2
*,/	Multiply, divide	3
MOD	Modulus	3
+,-	Add, subtract	4

Precedence order

# Constant Integer Expressions

- Examples:

Expression	Value
$16/5$	3
$-(3+4)*(6-1)$	-35
$-3+4*6 -1$	20
$25 \bmod 3$	1

# Real Number Literals(floating point literals)(1)

- Represented as either decimal real or encoded (hexadecimal) reals.

Format for decimal real:

[sign] integer.[integer][exponent]

Sign {+,-}

Exponent E[{+,-}]integer

Example:

2.

+3.0

-44.2E+05

26.E5

At least one digit and a decimal point is required.

# Real Number Literals(floating point literals)(2)

- Encoded Real:
- Represents a real number in hexadecimal, using IEEE floating-point format for short reals
- Example:
  - $+1.0 \rightarrow 0011\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000 \rightarrow 3F800000r$

# Character Literals

- Single character enclosed in single or double quotes
- Assembler stores value in memory as the character ‘s binary ASCII code
- Example:
  - ‘A’
  - “d”

# String Literals

- Sequence of characters enclosed in single or double quotes.
- Examples:
  - “ABCD”
  - “ Good morning “
  - ‘ABC’
  - “x”
  - “Goodnight, Gracie”
  - ‘4096’
  - “This isn’t a test”
  - ‘Say “Goodnight, ” Gracie’

# Reserved words

- Special meaning and can be used in their correct context
- Not case-sensitive
- Types of reserved words:
  - Instruction mnemonics (Examples: MOV, ADD)
  - Register names ( Examples: EAX, AX)
  - Directives
  - Attributes ( Examples: BYTE and WORD)
  - Operators
  - Predefined symbols( @data)

# Identifiers

- A programmer-chosen name.
- It may identify variable, constant etc

Rules on how they can be formed:

They may contain characters between 1 and 247 characters

Not case sensitive

First character must be a letter. Underscore, @, ?, or \$

Cannot be same as an assembler reserved word

# Directives(1)

- Command embedded in source code which is recognized and acted upon by the assembler
- Directives have a syntax similar to assembly language but do not correspond to Intel processor instructions.
- Not part of the Intel instruction set
- Directives do not execute at run time, whereas instructions do.
- Used to declare code, data areas, select memory model, define variables, macros and procedures etc.
- not case sensitive: It recognizes .data, .DATA, and .Data as equivalent.
- Example showing difference between directive and instruction

```
myVar DWORD 26      ; here DWORD directive tells assembler to reserve space in program for doubleword  
mov eax, myVar     ; mov instruction executes at runtime, copies the contents of myVar to EAX register
```

# Directives(2)

- Their one important function is to define program segments.
- The .DATA directive identifies the area of a program containing variables  
`.data`
- The .CODE directive identifies the area of a program containing instructions  
`.code`
- The .STACK directive identifies the area of a program holding the runtime stack, setting its size  
`.stack 100h`
- Different assemblers have different directives  
`NASM` not the same as `MASM`

# Intrinsic Data Types

- MASM defines intrinsic data types, each of which describe a set of value that can be assigned to variables and expressions of the given type
- The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80
- Signed, pointer or the floating point are optional and are mainly for the benefit of programmers who want to be reminded about the type of data held in variable
- Example:
  - A variable declared as DWORD; logically holds an unsigned 32-bit integer; It can hold
    - a signed 32-bit integer
    - 32-bit single precision real
    - A 32-bit pointer
- Assembler is not a case sensitive

- Syntax:

[name] directive initializer [,initializer]...

# Intrinsic Data Types

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

# Legacy Data Directives

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer
DQ	64-bit integer or real
DT	Define 80-bit (10 byte) integer

# Intrinsic Data Types

- MASM defines intrinsic data types, each of which describe a set of value that can be assigned to variables and expressions of the given type
- The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80
- Signed, pointer or the floating point are optional and are mainly for the benefit of programmers who want to be reminded about the type of data held in variable
- Example:
  - A variable declared as DWORD; logically holds an unsigned 32-bit integer; It can hold
    - a signed 32-bit integer
    - 32-bit single precision real
    - A 32-bit pointer
- Assembler is not a case sensitive

- Syntax:

[name] directive initializer [,initializer]...

# Intrinsic Data Types

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

# Legacy Data Directives

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer
DQ	64-bit integer or real
DT	Define 80-bit (10 byte) integer

# Defining BYTE and SBYTE (8-bit data)

BYTE and SBYTE are used to allocate storage for an unsigned or signed 8-bit value:

## Examples:

```
value1 BYTE 'A'          ; character constant  
value2 BYTE 0            ; smallest unsigned byte  
value3 BYTE 255          ; largest unsigned byte  
value4 SBYTE -128         ; smallest signed byte  
value5 SBYTE +127         ; largest signed byte  
value6 BYTE ?             ; no initial value
```

## Example to show offset value:

```
.data  
value7 BYTE 10h           ; offset is zero  
value8 BYTE 20h           ; offset is 1
```

# Multiple Initializers(1)

- If a definition has multiple initializers, the label is the offset for the first data item:

.data

list BYTE 10h, 20h, 30h, 40h

Offset	0000	0001	0002	0003
Value:	10	20	30	40

# Multiple Initializers(2)

- Not all definitions need labels:

.data

```
list    BYTE   10h, 20h, 30h, 40h  
        BYTE   50h, 60h, 70h, 80h  
        BYTE   81h, 82h, 83h, 84h
```

Offset	0000	0001	0002	0003	0004	0005
Value:	10	20	30	40	50	60

# Multiple Initializers(3)

- The different initializers can use different radices:

.data

list1 BYTE 10, 32, 41h, 00100010b

list2 BYTE 0aH, 20H, 'A', 22h

- list1 and list2 will have the identical contents, albeit at different offsets.

# Defining Strings

- To create a string data definition, enclose a sequence of characters in single or double quotation marks.
- The most common type of string ends with a null byte (containing 0), called a null-terminated string.
- Each character uses a byte of storage.
- The rule that byte values must be separated by commas does not apply on strings.

greeting1 BYTE “Good afternoon”, 0

is the same as

greeting1 BYTE ‘G’, ‘o’, ‘o’, … 0

# Defining Strings

- Strings can be spread over several lines:

```
greeting2    BYTE    "Welcome to the Encryption"  
              BYTE    " Demo program"
```

# DUP Operator

- Use DUP to allocate (create space for) an array or string.( It allocates space for multiple data items).

- Syntax:

counter DUP ( argument )

- Counter and argument must be constants or constant expressions

- Examples:

var1 BYTE 20 DUP(0) ; 20 bytes, all equal to zero

var2 BYTE 20 DUP(?) ; 20 bytes, uninitialized

var3 BYTE 4 DUP("STACK") ; 20 bytes, "STACKSTACKSTACKSTACK"

## Practice:

- var4 BYTE 10,3 DUP(0), 20

var4	10
	0
	0
	0
	20

# Practice:

list1 BYTE 10,20,30,40

list2 BYTE 10,20,30,40

    BYTE 50,60,70,80

    BYTE 81,82,83,84

list3 BYTE ?,32,41h,00100010b

list4 BYTE 0Ah,20h,'A',22h

	Offset	Value
list1	0000	10
	0001	20
	0002	30
	0003	40
list2	0004	10
	0005	20
	0006	30
	0007	40
	0008	50
	0009	60
	000A	70
	000B	80
	000C	81
	000D	82
list3	000E	83
	000F	84
list3	0010	

## Exercises:

- Define
  - i. Largest unsigned Value (16-bits)
  - ii. Smallest Signed Value (16-bits)
  - iii. Initialized Array of 5 Words.
  - iv. Un-initialized Array of 5 Words.
  
- Define
  - i. Largest unsigned Value (32-bits)
  - ii. Smallest Signed Value (32-bits)
  - iii. Unsigned Array containing 20 variables that are un-initialized. (32- bits)
  - iv. Signed Array containing 5 variables. (32-bits)

# Defining WORD and SWORD (16-bit Data)

- The WORD and SWORD directives allocate storage of one or more 16-bit integers:

```
word1    WORD    65535    ; largest unsigned value  
word2    SWORD   -32768    ; smallest signed value  
word3    WORD    ?         ; uninitialized value
```

- The dw directive can be used to allocated storage for either signed or unsigned integers:

```
val1     DW      65535    ; unsigned  
val2     DW      -32768    ; signed
```

# Arrays of Words

- You can create an array of word values by listing them or using the DUP operator:

myList WORD 1h, 2h, 3h, 4h, 5h

Offset	0000	0002	0004	0006	0008
Value:	1	2	3	4	5

array WORD 5 DUP(?) ; 5 values, uninitialized

# Defining DWORD and SDWORD (32-bit Data)

- The DWORD and SDWORD directives allocate storage of one or more 32-bit integers:

val1 DWORD 12345678h ; unsigned

val2 SDWORD -2147483648 ; signed

val3 DWORD 20 DUP(?) ; unsigned array

- The dd directive can be used to allocated storage for either signed or unsigned integers:

val1 DD 12345678h ; unsigned

val2 DD -2147483648 ; signed

# Arrays of Doublewords

- You can create an array of word values by listing them or using the **DUP** operator:

```
myList  DWORD  1, 2, 3, 4, 5
```

Offset	0000	0004	0008	000C	0010
Value:	1	2	3	4	5

# Defining QWORD (64-bit Data)

- The QWORD directive allocate storage of one or more 64-bit (8-byte) values:

```
quad1 QWORD 1234567812345678h
```

- The dq directive can be used to allocated storage:

```
quad1 DQ 1234567812345678h
```

# Defining TBYTE

- The TBYTE directive allocate storage of one or more 80-bit integers, used mainly for binary-coded decimal numbers:

```
val1    TBYTE    80000000000000001234h
```

- The dq directive can be used to allocated storage:

```
val1    DT      80000000000000001234h
```

# Defining Real number data

- There are three different ways to define real values:
  - REAL4 defines a 4-byte single-precision real value.
  - REAL8 defines a 8-byte double-precision real value.
  - REAL10 defines a 10-byte extended double-precision real value.
- Each requires one or more real constant initializers.

## Examples of Real Data Definitions:

rVal1           REAL4   -2.1

rVal2           REAL8   3.2E-260

rVal3           REAL10  4.6E+4096

ShortArray   REAL4   20 DUP(0.0)

rVal1   DD  -1.2

rVal2   dq  3.2E-260

rVal3   dt  4.6E+4096

# Ranges for Real Numbers

<u>Data Type</u>	<u>Significant Digits</u>	<u>Approximate Range</u>
Short Real	6	$1.18 \times 10^{-38}$ to $3.40 \times 10^{38}$
Long Real	15	$2.23 \times 10^{-308}$ to $1.79 \times 10^{308}$
Extended Real	19	$3.37 \times 10^{-4932}$ to $1.18 \times 10^{4932}$

# Sample Program

```
TITLE Add and Subtract      (AddSubAlt.asm)
; This program adds and subtracts 32-bit integers.
; 32-bit Protected mode version
```

```
.386
.MODEL flat, stdcall
.STACK 4096
```

```
ExitProcess PROTO, dwExitCode : DWORD
DumpRegs PROTO
```

```
.code
main PROC
    mov eax, 10000h ; EAX = 10000h
    add eax, 40000h ; EAX = 50000h
    sub eax, 20000h ; EAX = 30000h
    call DumpRegs
    INVOKE ExitProcess,0
main ENDP
END main
```

Program Output: showing registers and flags

EAX=00030000	EBX=7FFDF000	ECX=00000101	EDX=FFFFFFF
ESI=00000000	EDI=00000000	EBP=0012FFF0	ESP=0012FFC4
EIP=00401024	EFL=00000206	CF=0 SF=0 ZF=0 OF=0	

- **.386 directive** identifies the program as a 32-bit program that can access 32-bit registers and addresses.
- **.model flat, stdcall** selects the program's memory model, and identifies the calling convention. The **stdcall** keyword tells the assembler how to manage the runtime stack when procedures are called. It is a calling convention, that is a scheme for how subroutines receive parameters from their caller and how they return a result.
- **.stack 4096** sets aside **4096 bytes** of storage for the runtime stack. It tells the assembler how many bytes of memory to reserve for the program's runtime stack.
  - Stack are used :
    1. to hold passed parameters
    2. to hold the address of the code that called the function. The CPU uses this address to return when the function call finishes, back to the spot where the function was called.
  - Stack can hold local variables.

- When your program is ready to finish, it calls **ExitProcess** that returns an integer that tells the operating system that your program worked just fine.
- The **ENDP directive** marks the end of a procedure. Our program had a procedure named main, so the endp must use the same name.
- The **END directive** is used to mark the last line to be assembled (end of program), and it identifies the program entry point (main). If you add any more lines to a program after the END directive, they will be ignored by the assembler.

- **ExitProcess** is an MS-Windows function that halts the current program (called a process),  
and
- **DumpRegs** is a procedure from the Irvine32 link library that displays registers.
- **Invoke** is an assembler directive that calls a procedure or function.
  - This program ends by calling the ExitProcess function, passing it a return code of zero.

# Sample Program

```
1 title Add and Subtract          (AddSub.asm)
2 ; This program adds and subtracts 32-bit integers.
3
4 INCLUDE Irvine32.inc
5 .code
6 main PROC
7     mov eax, 10000h             ; EAX = 10000h
8     add eax, 40000h             ; EAX = 50000h
9     sub eax, 20000h             ; EAX = 30000h
10    call DumpRegs              ; Display registers
11    exit
12 main ENDP
13 END main
```

- Program Description
- The `TITLE` directive marks the entire line as a comment
- The `INCLUDE` directive copies necessary definitions and setup information from a text file (`Irvine32.inc`) located in assembler's INCLUDE directory.
- The `.code` directive marks the beginning of the code area of the program
- The `PROC` marks the beginning of a procedure
- The `CALL` statement calls a procedure. `DumpRegs`: Irvine32 procedure
- The `exit` statement calls a predefined MS-Window function that halts the program
- The `ENDP` directive marks the end of the procedure
- The `END` directive marks the last line of the program to be assembled. It identifies the name of the program's startup procedure (the procedure that starts the program execution.)
- Segments – organize the program
  - The code segment (`.code`) contains all of the program's executable instruction
  - The data segment (`.data`) holds variable
  - The stack (`.stack`) holds procedure parameters and local variables

# Virtual Machine Concept ( week 2)

Programs written in L1 can run in two different ways:

- **Interpretation** – L0 program interprets and executes L1 instructions one by one
- **Translation** – L1 program is completely translated into an L0 program, which then runs on the computer hardware

# Difference between compiler and assembler

- Compiler translate programs written in high-level languages into machine code that a computer understands (directly or indirectly). The purpose of an assembler is to translate assembly language into object code.
- Compilers and interpreters generate many machine code instructions for each high-level instruction, assemblers create one machine code instruction for each assembly instruction.

# Important terminologies

Source code is what a programmer writes, but it is not directly executable by the computer. It must be converted into machine language by compilers, assemblers or interpreters.

An object file is a computer file containing object code, that is, machine code output of an assembler or compiler. The object code is not usually directly executable.

An executable file is kind of file that is capable of being executed or run as a program in the computer. In a Disk Operating System or Windows operating system, an executable file usually has a file name extension of . bat, . com, or . exe.

# Advantages of high level languages

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

# Operand Types

x86 instruction format:

[label : ] mnemonic [operands ] [ ; comment ]

Because the number of operands may vary, we can further subdivide the formats to have zero, one, two, or three operands. Label and comment fields are omitted for clarity:

mnemonic

mnemonic [ *destination* ]

mnemonic [ *destination* ],[*source* ]

mnemonic [ *destination* ],[*source-1* ],[*source-2* ]

# Operand Types

There are three basic types of operands:

- Immediate—uses a numeric literal expression
- Register—uses a named register in the CPU
- Memory—references a memory location

# Instruction Operand Notation, 32-bit mode

- Refer to Table 4-1 of your textbook.

# Direct Memory Operands

- Variable names are references to offsets within data segments.
- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler. The brackets imply a deference operation.

```
.data  
var1 BYTE 10h  
.code  
mov al, var1 ; AL = 10h
```

Alternate instruction:

```
mov al, [var1] ; AL = 10h
```

This notation [ ] is used when an arithmetic expression is involved.

Suppose var1 were located at offset 10400h. It would be assembled into following machine instruction:

A0 00010400

# MOV instruction

- The MOV instruction copies data from a source operand to a destination operand.
- Syntax:

*MOV destination, source*

- destination operand's contents change
- source operand's contents do not change
- Both operands must be the same size
- Both operands cannot be memory operands
- CS, EIP, and IP cannot be the destination operand
- Flags Affected:
- None

- Example:

```
.data
count BYTE 100
wVal WORD 2
.code
mov bl, count
mov ax, wVal
mov count, al
mov al, wVal      ; error, AL is 8 bits
mov ax, count     ; error, AX is 16 bits
mov eax, count    ; error, EAX is 32 bits
.data
bVal BYTE 100
bVal2 BYTE ?
.code
mov bVal2, bVal   ; error, memory-to-memory move not permitted
```

- Here is a list of the general variants of MOV, excluding segment registers:

MOV reg, reg

MOV mem, reg

MOV reg, mem

MOV mem, imm

MOV reg, imm

Segment registers should not be directly modified

- Memory to Memory

A single MOV instruction does not permit memory to memory move. A temporary register must be used for data transfer.

```
.data  
var1 WORD ?  
var2 WORD ?  
.code  
mov ax, var1  
mov var2, ax
```

- Overlapping Values:

```
.data
```

```
oneByte BYTE 78h
```

```
oneWord WORD 1234h
```

```
oneDword DWORD 12345678h
```

```
.code
```

```
mov eax, 0          ; EAX = 00000000h
```

```
mov al, oneByte    ; EAX = 00000078h
```

```
mov ax, oneWord    ; EAX = 00001234h
```

```
mov eax,oneDword  ; EAX = 12345678h
```

```
mov ax,0           ; EAX = 12340000h
```

# Zero/Sign Extension of Integers

- Copying Smaller values to larger ones:

Example 1: (Unsigned)

```
.data  
count WORD 1  
.code  
mov ecx,0  
mov cx, count
```

## Example 2: (For signed integer)

### Incorrect Approach:

```
.data  
signedVal SWORD -16    ; FFF0h (-16)  
.code  
mov ecx,0  
mov cx,  signedVal      ; ECX = 0000FFF0h (+65,520)
```

### Correct Approach:

```
mov ecx, OFFFFFFFFh  
mov cx,  signedVal      ; ECX = FFFFFFF0h (-16)
```

### Sign Extension Technique:

# MOVZX Instruction

- Zero Extension: MOVZX instruction
- When copying a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.
- Only used with unsigned integers.
- The destination must be a register
- Source operand cannot be a constant

- Syntax:

MOVZX reg32, reg/mem8

MOVZX reg32, reg/mem16

MOVZX reg16, reg/mem8

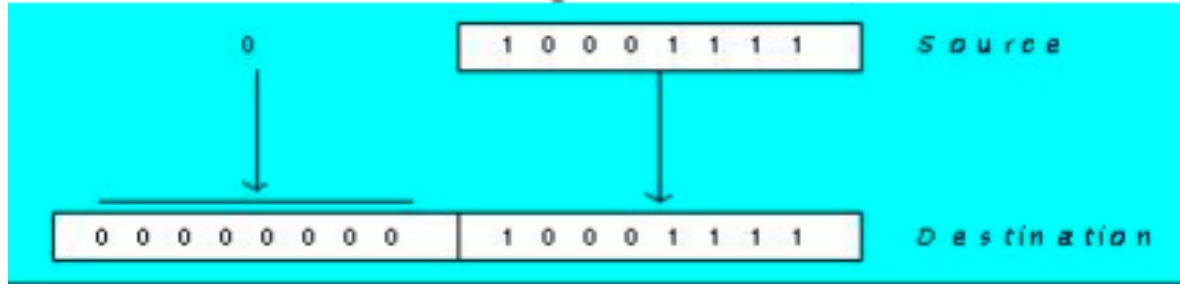


Diagram using MOVZX to copy a byte into a 16-bit destination

Example:

```
.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh

.code
movzx eax, word1    ; EAX = 0000A69Bh
movzx edx, byte1    ; EDX = 0000009Bh
movzx cx, byte1     ; CX = 009Bh
```

# MOVsx Instruction

- Sign Extension: MOVsx instruction
- The MOVsx instruction copies the contents of a source operand into a destination operand and sign-extend the value to 16-32 bits.
- The MOVsx instruction fills the upper half of the destination with a copy of the source operand's sign bit(highest bit).
- Only used with signed integers

Syntax:

MOVsx reg32, reg/mem8

MOVsx reg32, reg/mem16

MOVsx reg16, reg/mem8

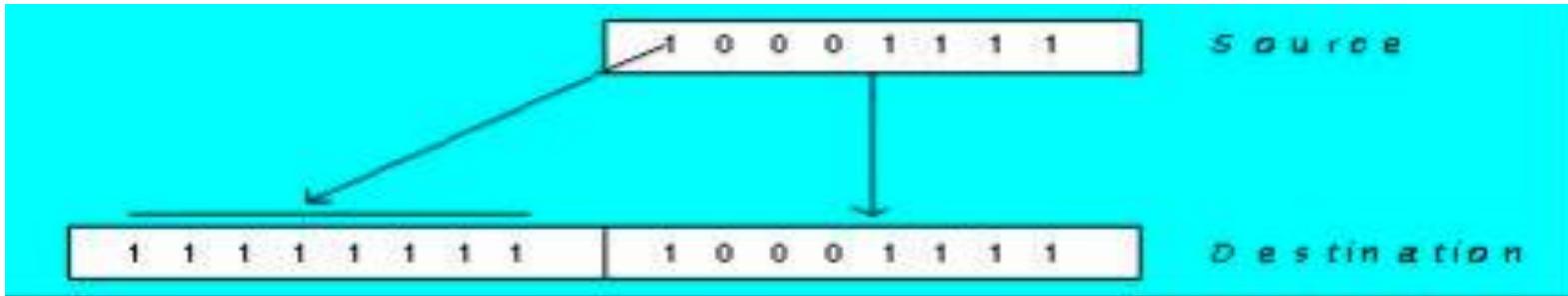


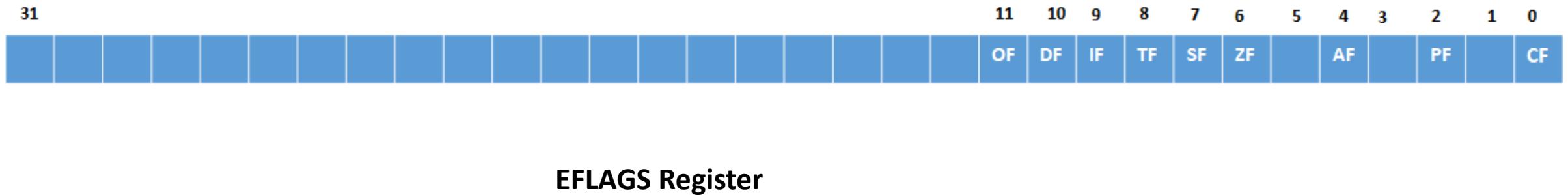
Diagram using MOVSX to copy a byte into a 16-bit destination

Example:

```
.data  
byteVal BYTE 10001111b  
.code  
movsx ax, byteVal ; AX = 111111110001111b
```

# LAHF and SAHF Instructions

- LAHF (load status flags into AH) copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry.
- SAHF (store AH into status flags) instruction copies AH into the low byte of the EFLAGS register. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry.



# Example code for LAHF and SAHF

```
.data  
saveflags BYTE ?  
.code  
lahf          ; load flags into AH  
mov saveflags, ah ; save them in a variable  
mov ah, saveflags ; load saved flags into AH  
sahf          ; copy into Flags register
```

# XCHG Instruction

- XCHG (exchange data) instruction exchanges the values of two operands.
- At least one operand must be a register
- No immediate operands are permitted.

## Syntax:

XCHG reg, reg

XCHG reg, mem

XCHG mem, reg

## Example:

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax, bx      ; exchange 16-bit regs
xchg ah, al      ; exchange 8-bit regs
xchg var1, bx    ; exchange mem, reg
xchg eax, ebx    ; exchange 32-bit regs
xchg var1,var2   ; error: two memory operands
```

- To exchange two memory operands, a register will be used as a temporary container:

```
mov ax, val1  
xchg ax, val2  
mov val1, ax
```

# Direct-Offset Operands

- A constant offset is added to a data label to produce an effective address (EA).
- Adding a displacement to the name of a variable, creates a direct-offset operand.
- Memory locations that may not have explicit labels can be accessed using this method.
- The address is dereferenced to get the value inside its memory location

```
.data  
arrayB BYTE 10h, 20h, 30h, 40h, 50h  
.code  
mov al, arrayB      ; AL = 10h
```

To access the second byte in the array add 1 to the offset of arrayB:

```
mov al, [arrayB+1]    ; AL = 20h . Highly recommended  
mov al, arrayB+1      ; AL = 20h alternate notation but not recommended
```

To access third byte add 2:

```
mov al, [arrayB+2]    ; AL = 30h
```

- MASM has no built-in range checking for effective addresses. If we type the following instruction in the above example then the instruction retrieves a byte of memory outside the array.

```
mov al, [arrayB+20] ; AL = ??
```

# Word Arrays

```
.data  
arrayW WORD 100h, 200h, 300h  
.code  
mov ax, arrayW          ; AX = 100h  
mov ax, [arrayW+2]       ; AX = 200h.  
; Here offset of array' each element is 2 bytes beyond the previous one
```

# Doubleword Arrays

```
.data  
arrayD DWORD 10000h, 20000h  
.code  
mov eax , arrayD          ; EAX = 10000h  
mov eax, [arrayD+4]        ; EAX = 20000h
```

# INC and DEC Instructions

- Add 1, subtract 1 from destination operand, operand may be register or memory
- INC Syntax: **INC reg**  
**INC mem**
- Logic:  $\text{destination} \leftarrow \text{destination} + 1$
- DEC Syntax: **DEC reg**  
**DEC mem**
- Logic:  $\text{destination} \leftarrow \text{destination} - 1$
- Flags Affected:  
The Overflow, Sign, Zero, Auxiliary Carry and Parity flags are changed according to the value of the destination operand. The INC and DEC instructions do not affect the Carry flag.

# INC and DEC Instructions

- INC and DEC Examples

```
.data  
myWord WORD 1000h  
.code  
inc myWord      ; myWord =1001h  
mov bx, myWord  
dec bx          ; bx = 1000h
```

# ADD Instruction:

- ADD destination, source
- Logic: destination  $\leftarrow$  destination + source
- Examples:

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov eax, var1          ; EAX = 00010000h
add eax, var2          ; EAX = 00030000h
add var2, 3h            ; var2 = 00020003h
```

## Flags Affected:

The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

# SUB Instruction

- SUB destination, source

Logic: destination  $\leftarrow$  destination – source

## Examples:

```
.data  
var1 DWORD 30000h  
var2 DWORD 10000h  
.code  
mov eax, var1      ; EAX = 00030000h  
sub eax, var2      ; EAX = 00020000h
```

## Flags Affected:

The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

# NEG Instruction

- NEG (negate) Instruction
- Reverses the sign of an operand
- It reverses the sign of number by converting the number to its 2's complement.
- Operand can be a register or memory operand
- Syntax: NEG reg  
          NEG mem

```
.data
valB BYTE -1
valW WORD +32767
.code
mov al, valB      ; AL = -1
neg al            ; AL = +1
neg valW          ; valW = -32767
```

## Flags Affected:

The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand. Applying the NEG instruction to a nonzero operand always sets the carry flag.

The processor implements NEG using the following internal operation:

SUB 0,operand

## Examples

```
.data  
valB BYTE 1,0  
valC SBYTE -128  
.code  
neg valB          ; CF = 1, OF = 0  
neg [valB + 1]    ; CF = 0, OF = 0  
neg valC          ; CF = 1, OF = 1
```

# Implementing Arithmetic Expressions

- Translate mathematical expressions into assembly language
- Example:  $Rval = -Xval + (Yval - Zval)$

```
; Addition and Subtraction (AddSubTest.asm)
```

```
.386
```

```
.model flat,stdcall
```

```
.stack 4096
```

```
ExitProcess proto,dwExitCode:dword
```

```
.data
```

```
Rval SDWORD ?
```

```
Xval SDWORD 26
```

```
Yval SDWORD 30
```

```
Zval SDWORD 40
```

```
.code
```

```
main PROC
```

```
; INC and DEC
```

```
mov ax,1000h
```

```
inc ax      ; 1001h
```

```
dec ax      ; 1000h
```

```
; Expression: Rval = -Xval + (Yval - Zval)
```

```
mov eax, Xval
```

```
neg eax          ; -26
mov ebx,Yval
sub ebx,Zval    ; -10
add eax,ebx
mov Rval,eax    ; -36
Invoke ExitProcess,0
main ENDP
END main
```

# Flags Affected by Arithmetic

- The CPU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations based on the contents of the destination operand
- The MOV instruction never affects the flags.
- **Zero flag:** Set when destination operand equals zero

```
mov cx,1  
sub cx,1          ; CX = 0, ZF = 1  
mov ax,0FFFFh  
inc ax           ; AX = 0, ZF = 1  
inc ax           ; AX = 1, ZF = 0
```

Note: A flag is set when it equals 1

A flag is clear when it equals 0

- **Sign flag**: Set when the destination operand is **negative**. Clear when the destination is positive.

```
mov cx,0  
sub cx,1      ; CX = -1, SF = 1  
add cx,2      ; CX = 1, SF = 0  
mov ax, 7FFFh  
add ax, 2      ; AX =8001h , SF = 1
```

Note: The sign flag is a copy of the destination's highest bit

- **Carry flag**: Set when **unsigned destination** operand value is out of range. Indicates unsigned integer overflow

```
mov al, 7Fh  
add al, 1      ; AL = 80, CF = 0  
mov al, OFFh  
add al, 1      ; AL = 00, CF = 1, Too big  
mov al, 1  
sub al, 2      ; AL = FF, CF = 1, larger unsigned integer is subtracted from a smaller one
```

- **Auxiliary Carry:** Set when carry out of bit 3 in the destination operand

```
mov al,0Fh  
add al,1           ; AL = 10h, AC = 1
```

- **Parity flag:** Set when the least significant byte of the destination has even number of 1 bits.

```
mov al, 10001100b  
add al, 00000010b      ; AL = 10001110, PF = 1  
sub al, 10000000b      ; AL = 00001110, PF = 0
```

- **Overflow flag:** Set when signed destination operand value is out of range

```
mov al,+ 127  
add al, 1       ; OF = 1  
mov al, -128  
sub al, 1       ; OF = 1
```

- Note: When adding two integers, the Overflow flag is only set
  - When two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

## A hardware viewpoint of unsigned and signed operations:

- All CPU instructions operate exactly the same on signed and unsigned integers
  - The programmers are solely responsible for using the correct data type with each instruction
  - A hardware viewpoint of Overflow and Carry flags
- 
- How the ADD instruction modifies CF:
    - CF = (carry out of the MSB of the destination operand)
  - How the SUB instruction modifies CF:
    - NEG the source and ADD it to the destination
    - CF = 1 when larger unsigned integer is subtracted from a smaller one

- How the ADD instruction modifies OF :
  - $OF = (\text{carry out of the MSB}) \oplus (\text{carry into the MSB})$
- How the SUB instruction modifies OF :
  - NEG the source and ADD it to the destination
  - $OF = (\text{carry out of the MSB}) \oplus (\text{carry into the MSB})$

- Examples:

mov al,-128	; AL = 10000000b
neg al	; AL = 10000000b, CF = 1, OF = 1
mov al,80h	; AL = 10000000b
add al,2	; AL = 10000010b, CF = 0, OF = 0
mov al,1	; AL = 00000001b
sub al,2	; AL = 11111111b, CF = 1, OF = 0
mov al,7Fh	
add al,2	; AL = 10000001b, CF = 0, OF = 1

NEG instruction produces an invalid result if the destination operand cannot be stored correctly

## References:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
2. Computer Organization and Embedded Systems by Carl Hamacher, Zvonko Vranesic, Safwat Zaky and Naraig Manjikian ( sixth Edition)
3. Assembly Language Programming and organization of the IBM PC by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 5

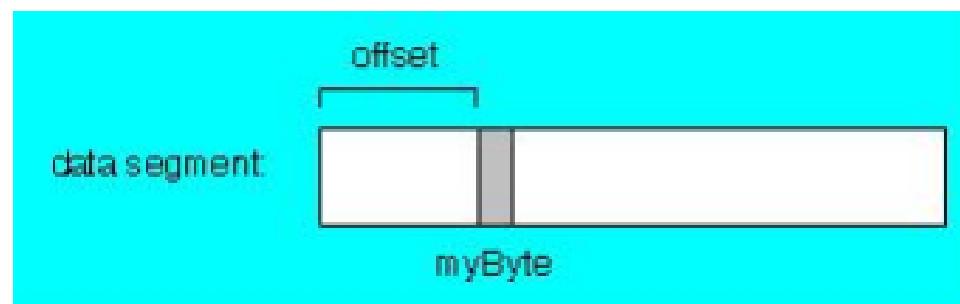
Data Transfer, Addressing and Arithmetic

# Data-Related Operators and Directives

- Operators and directives are not executable instructions; instead, they are interpreted by the assembler.
- They are used to get information about the addresses and size characteristics of data.

# OFFSET Operator

- It returns the offset of a data label.
- OFFSET represents the distance in bytes, of a label from the beginning of its enclosing segment.
- **Protected mode:** Offset are 32 bits
- Real mode: Offset are 16 bits



- **OFFSET Example**

Assume that the data segment begins at 00404000h

```
.data  
bVal BYTE ?  
wVal WORD ?  
dVal DWORD ?  
dVal2 DWORD ?  
.code  
mov esi, OFFSET bVal ; ESI = 00404000  
mov esi, OFFSET wVal ; ESI = 00404001  
mov esi, OFFSET dVal ; ESI = 00404003  
mov esi, OFFSET dVal2 ; ESI = 00404007
```

## Relating to C/C++

The value returned by OFFSET is a pointer

Compare the following code written for both C++ and assembly language

```
// C++ version:  
char array[1000];  
char * p = array;  
;  
; Assembly version  
.data  
array BYTE 1000 DUP(?)  
.code  
mov esi, OFFSET array
```

OFFSET can also be applied to a direct-offset operand. Suppose myArray contains five 16-bit words. The following MOV instruction obtains the offset of myArray, adds 4, and moves the resulting address to ESI. We can say that ESI points to the third integer in the array:

```
.data  
myArray WORD 1,2,3,4,5  
.code  
mov esi, OFFSET myArray + 4
```

# ALIGN DIRECTIVE

- The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary.
- **Syntax:**  
    ALIGN bound
  - Bound can be 1, 2, 4, 8, or 16.
  - A value of 1 aligns the next variable on a 1- byte boundary.

# ALIGN(cont.)

## Example

Assume that the data segment begins at 00404000h

```
.data  
    bVal BYTE ? ; 00404000  
    ALIGN 2  
    wVal WORD ? ; 00404002  
    bVal2 BYTE ? ; 00404004  
    ALIGN 4  
    dVal DWORD ? ; 00404008  
    dVal2 DWORD ? ; 0040400C
```

# Reason for aligning data

- Bound can be 1, 2, 4, 8, or 16.
- A value of 1 aligns the next variable on a 1- byte boundary.

Why bother aligning data? Because the CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses.

# PTR Operator

- It is used to override the declared size of an operand
- Provides the flexibility to access part of a variable
- Must be used in combination with one of the standard assembly data type: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TWORD

In little-endian, multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address .

For example, the doubleword 12345678h would be stored as:

Byte	offset
78	0000
56	0001
34	0002
12	0003

# Moving the part (bits) of a larger variable into smaller destination

Example:

```
.data  
myDouble DWORD 12345678h  
.code  
mov ax, myDouble          ; error  
mov ax, WORD PTR myDouble ; AX = 5678h  
mov ax, WORD PTR [myDouble+2] ; AX = 1234h  
mov al, BYTE PTR myDouble ; AL = 78h  
mov al, BYTE PTR [myDouble+1] ; AL = 56h  
mov al, BYTE PTR [myDouble+2] ; AL = 34h
```

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
	56		0001	myDouble + 1
1234	34		0002	myDouble + 2
	12		0003	myDouble + 3

## Moving smaller values into larger destinations

- PTR operator can combine elements of a smaller data type and move them into a larger destination operand.

### Example 1:

```
.data  
myBytes BYTE 12h,34h,56h,78h  
.code  
mov ax, WORD PTR [myBytes]      ; AX = 3412h  
mov ax, WORD PTR [myBytes+2]    ; AX = 7856h  
mov eax, DWORD PTR myBytes     ; EAX = 78563412h
```

## Example 2:

```
.data  
wordList WORD 5678h,1234h  
.code  
mov eax,DWORD PTR wordList ; EAX = 12345678h
```

# TYPE Operator

- TYPE operator returns the size, in bytes, of a single element of a data declaration/ variable
- Example:

```
.data  
var1 BYTE ?  
var2 WORD 3245h, 098Fh  
var3 DWORD ?  
var4 QWORD ?  
.code  
mov eax, TYPE var1    ; 1  
mov eax, TYPE var2    ; 2  
mov eax, TYPE var3    ; 4  
mov eax, TYPE var4    ; 8
```

# LENGTHOF Operator

- LENGTHOF operator counts the number of elements in an array.
- Example:

```
.data ;LENGTHOF
byte1 BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 32
array2 WORD 5 DUP(3 DUP(?)) ; 15
array3 DWORD 1,2,3,4 ; 4
digitStr BYTE "1234567 8",0 ; 10
.code
    mov ecx, LENGTHOF array1 ; 32
    Mov edx, LENGTHOF digitStr ; 10
```

- If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array. Given the following data, LENGTHOF myArray would return the value 5:

myArray BYTE 10,20,30,40,50

BYTE 60,70,80,90,100 ; LENGTHOF myArray will return 5

- Alternatively, you can end the first line with a comma and continue the list of initializers onto the next line. Given the following data, LENGTHOF myArray would return the value 10:

myArray BYTE 10,20,30,40,50,

60,70,80,90,100 ; LENGTHOF myArray will return 10

# SIZEOF Operator

- SIZEOF Operator returns a value that is equivalent to multiplying LENGTHOF by TYPE
- Example

```
.data ; SIZEOF
byte1 BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 64
array2 WORD 5 DUP(3 DUP(?)) ; 30
array3 DWORD 1,2,3,4 ; 16
digitStr BYTE "12345678",0 ; 9
.code
mov ecx, SIZEOF array1 ; 64
```

## Example:

```
.data  
array WORD 10,20,  
      30,40,  
      50,60  
.code  
mov eax,LENGTHOF array      ; 6  
mov ebx,SIZEOF array        ; 12
```

# LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- LABEL does not allocate any storage of its own.
- Example 1:

```
.data  
val16 LABEL WORD  
val32 DWORD 12345678h  
.code  
mov ax, val16          ; AX = 5678h  
mov dx, [val16+2]      ; DX = 1234h
```

- Example 2:

```
.data  
LongValue LABEL DWORD  
val1 WORD 5678h  
val2 WORD 1234h  
.code  
mov eax, LongValue    ; EAX = 12345678h
```

# Addressing modes

- The way an operand is specified is known as its addressing mode. The addressing modes we have used so far are
  - register mode, which means that an operand is a register
  - immediate mode, when an operand is a constant
  - direct mode, when an operand is variable

```
MOV AX,0          ;destination AX is a register mode and source 0 is an immediate mode  
ADD ALPHA,AX     ;destination Alpha is direct mode, source AX is register mode
```

- There are other modes used to address memory operands indirectly.

## Operands Types:

- Operands can be of different types that includes immediate operand and register.
- Other types include direct operand that is the name of a variable, and represents the variable's address.
- A direct-offset operand adds a displacement to the name of a variable, generating a new offset. This new offset can be used to access data in memory.
- An indirect operand is a register containing the address of data. By surrounding the register with brackets (as in [esi]), a program dereferences the address and retrieves the memory data.
- An indexed operand combines a constant with an indirect operand. The constant and register value are added, and the resulting offset is dereferenced. For example, [array +esi] and array[esi] are indexed operands.

# Indirect Addressing

- A register is used as pointer and the register's value is manipulated to address array elements etc.

## Indirect Operands:

- When an operand uses indirect addressing, it is called indirect operand.
- An indirect operand holds the address of a variable, usually an array or string.
- It can be dereferenced (just like a pointer).

## Protected Mode:

In this mode an indirect operand can be any 32-bit general-purpose register (EAX, EBX, ESI, EDI, EBP and ESP) surrounded by brackets.

## Example:

```
.data  
val1 BYTE 10h,20h,30h  
.code  
mov esi, OFFSET val1      ; ESI = the address of Val1  
mov al, [esi]              ; AL = 10h, dereference ESI  
inc esi  
mov al, [esi]              ; AL = 20h  
inc esi  
mov al, [esi]              ; AL = 30h
```

- Using PTR with Indirect Operands:
- Use PTR to clarify the size attribute of a memory operand

```
.data  
myCount WORD 0  
.code  
mov esi, OFFSET myCount  
inc [esi]           ; error: ambiguous  
inc WORD PTR [esi] ; ok
```

- Arrays:
- Indirect operands are ideal for traversing an array
- The register in brackets must be incremented by a value that matches the array type
- Example:

The figure shows initial value of esi

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,OFFSET arrayW           ; ESI = the address of Val1
mov ax,[esi]                     ; AX = 1000h
add esi,2                         ; or: add esi,TYPE arrayW
add ax,[esi]                      ; AX = 3000h
add esi,2
add ax,[esi]                      ; AX = 6000h
```

Offset	Value
10200	1000h
10202	2000h
10204	3000h

# Indexed Operands

- An indexed operand adds a constant to a register to generate an effective address.
- Two basic formats are permitted by MASM
  - Constant[reg]
  - [constant + reg]
- Indexed operands can appear in one of two different formats, as either a variable name combined with a register, or as a constant integer combined with a register.

- In the first format, the variable name is translated by the assembler into a constant that represents the variable's offset.
- The index register should be initialized to zero before accessing the first array element
- Example

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
mov esi,0  
mov ax, arrayW[esi]           ; AX =1000h  
add esi,2  
mov ax, arrayW[esi]           ; AX =2000h  
mov ax,[arrayW + esi]         ; AX = 2000h
```

- Adding Displacements:
- The second type of indexed addressing combines a register with a constant offset, in either order. The index register holds the base address of an array or structure, and the constant identifies offsets of various array elements.
- Example:

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
mov esi, OFFSET arrayW  
mov ax, [esi]           ; AX = 1000h  
mov ax, [esi+2]         ; AX = 2000h  
mov ax, [esi+4]         ; AX = 3000h  
mov ax, [4+esi]         ; AX = 3000h
```

- Scale Factors in Indexed Operands
- You can scale indexed operand to the offset of an array element
- This is done by multiplying the index by the array's TYPE
- Example:

```
.data
arrayB BYTE    0,1,2,3,4,5
arrayW WORD    0,1,2,3,4,5
arrayD DWORD   0,1,2,3,4,5
.code
mov esi,4          ; ESI = index of array
mov al,  arrayB[esi*TYPE arrayB] ; AL = 04h
mov bx,  arrayW[esi*TYPE arrayW] ; BX = 0004h
mov edx, arrayD[esi*TYPE arrayD] ; EDX = 00000004h
```

# Pointers

Declare a pointer variable that contains the offset of another variable.

Example:

```
.data  
arrayW WORD 1000h,2000h,3000h  
ptrW DWORD arrayW ; ptrW (pointer variable)
```

```
.code
```

```
mov esi,ptrW  
mov ax,[esi] ; AX = 1000h
```

Alternate format for declaring ptrW:

```
ptrW DWORD OFFSET arrayW ; ptrW = Offset (address) of arrayW
```

# Transfer of Control or branch instructions

- Transfer of control is a way of altering the order in which statements are executed.
- By default CPU loads and executes programs sequentially. There are certain instructions that transfer the control to a new location in the program.
- There are two basic types of transfers:

## Unconditional Transfer:

Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction does this.

## Conditional Transfer:

The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

# JMP Instruction

- JMP is an unconditional jump to a destination, identified by a code label, translated by the assembler into an offset that is usually within the same procedure.
- Syntax: JMP destination
- Logic: EIP  $\leftarrow$  destination

When CPU executes an unconditional transfer, the offset of destination is moved into the instruction pointer, causing execution to continue at the new location.

- Example

## Creating a loop

top:

.

.

jmp top ; repeat the endless loop

# LOOP Instruction

- The LOOP instruction, formally known as Loop According to ECX Counter, repeats a block of statements a specific number of times.
- ECX is automatically used as a counter and is decremented each time the loop repeats.
- **Syntax:**  
    LOOP destination
- **Logic:**
  - $ECX \leftarrow ECX - 1$
  - if  $ECX \neq 0$ , jump to destination
- **Implementation:**
  - The assembler calculates the distance, in bytes, between the current location and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.
  - The loop destination must be within -128 to +127 bytes of the current location counter.

## Loop Example 1:

- The following loop calculates the sum of the integers 5+ 4+3+2+1 :

offset	machine code	source code
00000000	66 B8 0000	mov ax,0
00000004	B9 00000005	mov ecx,5
00000009	66 03 C1	L1:add ax,cx
0000000C	E2 FB	loop L1
0000000E		

When `LOOP` is assembled, the current location = 0000000E. Looking at the `LOOP` machine code, we see that -5 (FBh) is added to the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

## Example 2:

```
mov ax,0  
mov ecx,5  
L1:  
inc ax  
loop L1
```

In the above example, we add 1 to AX each time the loop repeats.  
When the loop ends, AX =5 and ECX =0

# Programming Errors with loop

- A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.

# Programming Errors with loop

- Occasionally, you might create a loop that is large enough to exceed the allowed relative jump range of the LOOP instruction. Following is an example of an error message generated by MASM because the target label of a LOOP instruction was too far away:

```
error A2075: jump destination too far : by 14 byte(s)
```

# Programming Errors with loop

- Rarely should you explicitly modify ECX inside a loop. If you do, the LOOP instruction may not work as expected. In the following example, ECX is incremented within the loop. It never reaches zero, so the loop never stops:

```
top:  
.  
. .  
inc ecx  
loop top
```

## Exercise 1:

If the relative offset is encoded in a single byte,

- (a) what is the largest possible backward jump?
- (b) what is the largest possible forward jump?

(a) -128

(b) +127

Average sizes of machine instructions are about 3 bytes, so a loop might contain, on average, a maximum of 42 instructions!

## Exercise 2:

What will be the final value of AX?

```
mov ax, 6  
mov ecx, 4  
L1:  
    inc ax  
    loop L1
```

How many times will the loop execute?

```
mov ecx, 0  
X2:  
    inc ax  
    loop X2
```

What will be the final value of AX?

10

How many times will the loop execute?

4,294,967,296

```
mov ax, 6  
mov ecx, 4  
L1:  
    inc ax  
    loop L1
```

```
mov ecx, 0  
X2:  
    inc ax  
    loop X2
```

## Example 3:

```
.data  
count DWORD ?  
.code  
mov ecx,100          ; set loop count  
top:  
    mov count, ecx    ; save the count  
    .  
    mov ecx,20          ; modify ECX  
    .  
    mov ecx, count      ; restore loop count  
loop top
```

## Exercise 3:

- Find the value of AX and ESI at end?
- Times loop is executed?

---

```
.data
intArray WORD 100, 200, 300, 400, 500
.code
main PROC
    mov esi, 0
    mov ax, 0

    mov ecx, LENGTHOF intArray
L1:

    mov ax, intArray [esi]
    add esi, TYPE intArray
    loop L1
```

## Exercise 4:

- Find the value of AX and ESI at end?
- Times loop is executed?

---

```
INCLUDE Irvine32.inc

.DATA
number    DWORD    20
.CODE
main      PROC
          mov     EAX, 0
          mov     ECX, number
forCount: add     EAX, ECX
          loop   forCount

          exit
main      ENDP
END main
```

## Exercise 5:

- Write a program in assembly that prints numbers from 1 till N. You can select any value of N. Move the value in eax instead of using printing statement. Following is the helping program that print numbers from 1 to 10.

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
    int i;

    for (i = 1; i < 11; ++i)
    {
        printf("%d ", i);
    }
    return 0;
```

## Exercise 6: (Summing an Integer Array)

- Write a code to add 16-bit integers stored in an pre-defined array.
- LOOP instructions must be used
- Also use OFFSET, LENGTHOF and TYPE Operators.
- Array at least contain 4-unsigned integers.

## Exercise 6: (Solution)

```
.data  
intarray WORD 100h,200h,300h,400h  
.code  
    mov edi,OFFSET intarray      ; address  
    mov ecx,LENGTHOF intarray   ; loop counter  
    mov ax,0                     ; zero the sum  
L1:  
    add ax,[edi]                ; add an integer  
    add edi,TYPE intarray       ; point to next  
    loop L1                     ; repeat until ECX = 0
```

## Exercise 7: (Copying a string)

- Write a code to copy elements of an array of characters(string) to other array of characters.

# Exercise 7: (solution)

```
TITLE My Program
INCLUDE Irvine32.inc

.data
source BYTE "This is the source string",0
target BYTE SIZEOF source DUP(0)

.code
main PROC
    mov esi,0                      ; index register
    mov ecx,SIZEOF source          ; loop counter
L1:
    mov al,source[esi]              ; get a character from source
    mov target[esi],al              ; store it in the target
    inc esi                        ; move to next character
    loop L1                         ; repeat for entire string
    call DumpRegs
    exit
main ENDP
END main
```

## Exercise 8:

- Write a loop that iterates through a doubleword array and calculates the sum of its elements using a scale factor with indexed addressing.

# Example(Copying a word array to doubleword array

```
.data
```

```
array1 word 100h, 200h, 300h, 400h
```

```
array2 dword 4 dup(?)
```

```
.code
```

```
main PROC
```

```
    mov esi, offset array1
```

```
    mov edi, offset array2
```

```
    mov ecx, lengthof array1
```

```
L1:
```

```
    mov eax, 0
```

```
    mov ax, [esi]
```

```
    mov [edi], eax
```

```
    add esi, type array1
```

```
    add edi, type array2
```

```
loop L1
```

# Nested Loop

- A nested loop is a **loop within a loop**, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes.
- When creating **a loop inside another loop, special consideration must be given to the outer loop counter in ECX**. You should save it in a variable because CPU uses ECX counter for all loops.

# Nested Loop: (Example 1)

```
.data  
count DWORD ?  
.code  
mov ecx, 100      ; set outer loop count  
L1:  
    mov count, ecx   ; save outer loop count  
    mov ecx, 20      ; set inner loop count  
L2:  
    .  
    .  
    loop L2         ; repeat the inner loop  
    mov ecx, count   ; restore outer loop count  
    loop L1         ; repeat the outer loop
```

# Nested Loop: (Example 2)

```
INCLUDE Irvine32.inc
.code
main PROC
    mov ecx, 100
    L1:
        push ecx
        mov ecx, 20
        L2:
            mov eax, ecx
            Call WriteInt
            loop L2
        pop ecx
        loop L1
    exit
main ENDP
END main
```





# Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data  
intarray WORD 100h,200h,300h,400h  
.code  
    mov edi,OFFSET intarray      ; address  
    mov ecx,LENGTHOF intarray   ; loop counter  
    mov ax,0                     ; zero the sum  
L1:  
    add ax,[edi]                ; add an integer  
    add edi,TYPE intarray       ; point to next  
    loop L1                     ; repeat until ECX = 0
```



# Copying a String

The following code copies a string from source to target.

```
.data
source    BYTE    "This is the source string",0
target    BYTE    SIZEOF source DUP(0),0

.code
    mov    esi,0           ; index register
    mov    ecx,SIZEOF source ; loop counter
L1:
    mov    al,source[esi]   ; get char from source
    mov    target[esi],al    ; store in the target
    inc    esi               ; move to next char
    loop   L1                ; repeat for entire string
```

Adapted from:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)

# Computer Organization and Assembly Language

Week 7

Chapter 5 “Procedures”

# Outline

- Symbolic Constants
- Stack Operations
- Defining and using Procedures
- Program Design using Procedures
- Defining and using Procedures
- Program Design using Procedures

# Symbolic Constants

## Equal Sign Directive:

- The equal-sign directive associates a symbol name with an integer expression.

## Syntax:

`name = expression`

Here expression is a 32-bit integer value. When a program is assembled, all occurrences of name are replaced by expression during the assembler's preprocessor step.

## Example:

`count = 40`

## Keyboard Definitions:

Symbols can be defined for numeric keyboard codes as well.

### Example:

TAB = 9 ; ASCII code for Tab

Later in program symbol TAB can be used in place of its value.

mov al, TAB

Current location counter (\$):

The symbol \$ is called the **current location counter**.

Example:

```
selfPtr DWORD $ ; selfPtr is initialized with it's offset value
```

# Calculating sizes of Arrays and strings using \$ operator

- The \$ operator (current location counter) returns the offset associated with the current program statement. It can be used to calculate the size of arrays.

- Example 1:

```
list BYTE 10, 20, 30, 40
```

```
ListSize = ($ - list) ; listSize will give value = 4
```

ListSize must follow immediately after list.

The following, for example, produces too large a value for ListSize because the storage used by var2 affects the distance between the current location counter and the offset of list:

```
list BYTE 10h, 20h, 30h, 40h
```

```
var2 BYTE 20 DUP(?)
```

```
ListSize = ($ - list)
```

- Example 2: Calculating the length of a string

```
myString BYTE "This is a long string, containing"
```

```
        BYTE "any number of characters"
```

```
myString_len = ($ - myString)
```

- Example 3: Calculating the length of array of words

```
list WORD 1000h, 2000h, 3000h, 4000h
```

```
ListSize = ($ - list) / 2
```

- Example 4: Calculating the length of array of doublewords

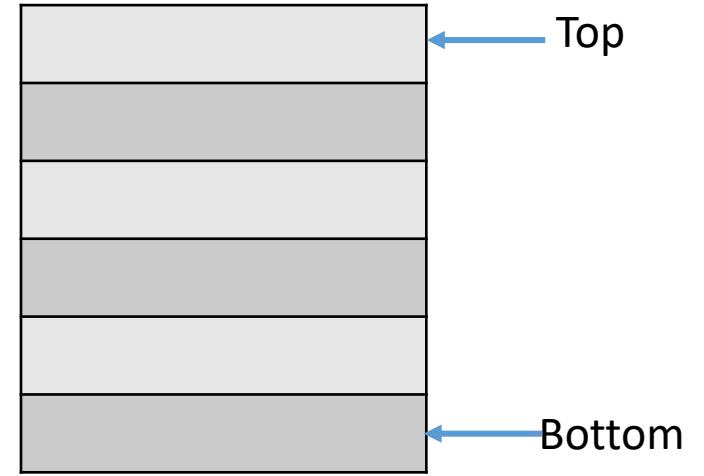
```
list DWORD 10000000h, 20000000h, 30000000h, 40000000h
```

```
ListSize = ($ - list) / 4
```

# Stack Operations

# Stack

- Consider a stack of plates...
  - Plates are only added to the top
  - Plates are only removed from the top
- LIFO Structure (Last-In, First-Out)
- Stack Abstract data type
  - studied in data structure course
  - useful for a variety of programming applications
  - students must have an experience of implementing it in high-level languages.



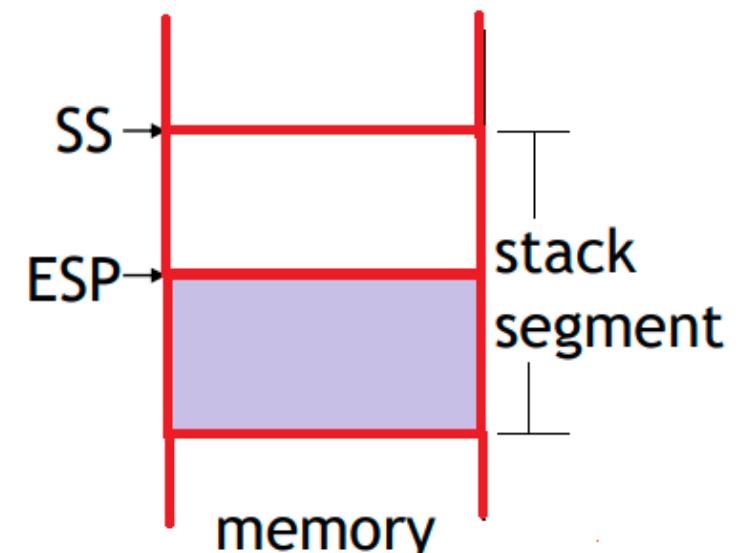
# Runtime Stack

- The runtime stack is a memory array managed directly by the CPU, using the ESP (extended stack pointer) register, known as stack pointer register. It is essential for calling and returning from procedures.
- Runtime stack stores information about the active subroutines of a computer program.

# Runtime Stack

- Managed by the CPU, using two registers
  - SS (stack segment) : not modified by user program
  - ESP (stack pointer) \* : points to the top of the stack(last data added/pushed on the stack) usually modified by CALL, RET, PUSH and POP, in 32-bit mode ESP register holds 32-bit offset into some memory location on the stack
- An Intel stack grows from high memory to low memory

\*SP in Real-address mode



## Difference between runtime stack and stack abstract data type

- The **runtime stack** works at the **system level** to handle subroutine calls.
- The **stack abstract data type (ADT)** is a programming construct typically written in a high-level programming language such as C++ or Java. It is used when implementing algorithms that depend on last-in, first-out operations.

# Push and Pop Instructions

- **PUSH syntax:**

- PUSH r/m16
- PUSH r/m32
- PUSH imm32

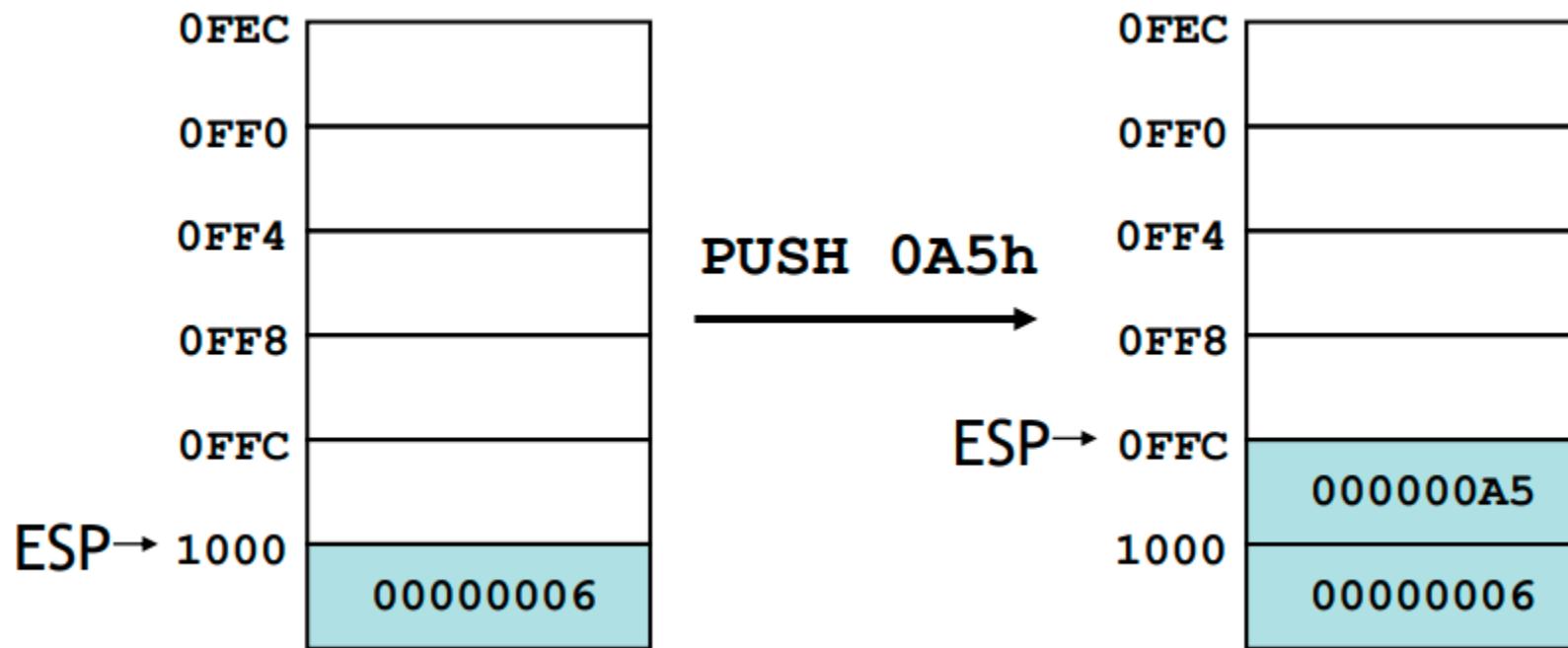
- **POP syntax:**

- POP r/m16
- POP r/m32

# Push Operation

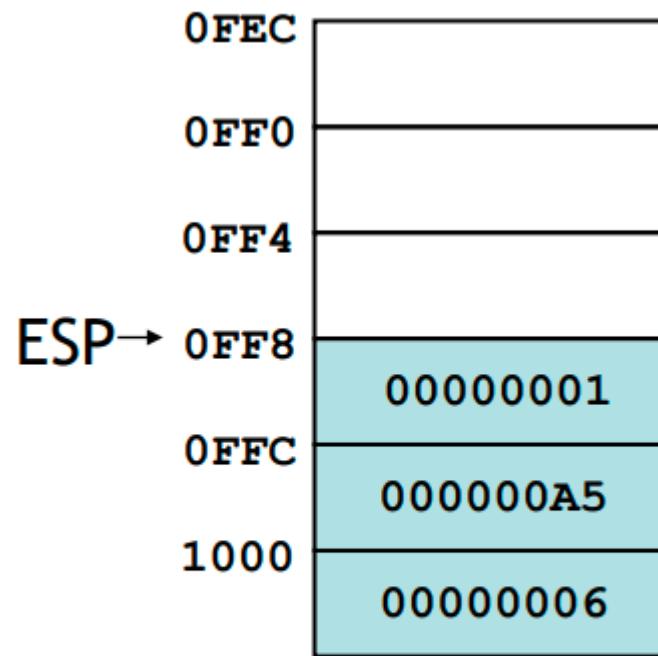
- A push operation first decrements the stack pointer by 2 or 4 (depending on operands) and then copies source operand into the location in the stack pointed to by the stack pointer.
- A 16-bit/32-bit operand causes ESP to be decremented by 2 and 4, respectively

# Pushing an integer on the stack(1 of 2)

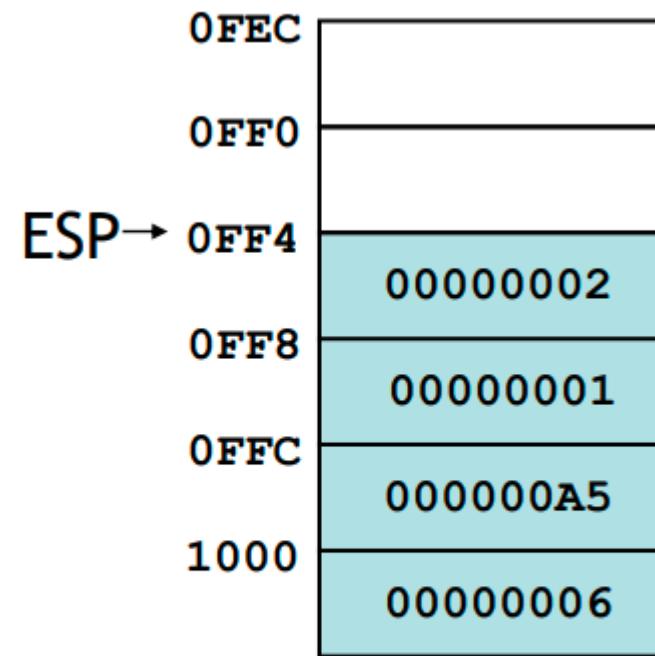


## Pushing an integer on the stack(2 of 2)

- The same stack after pushing two more integers:

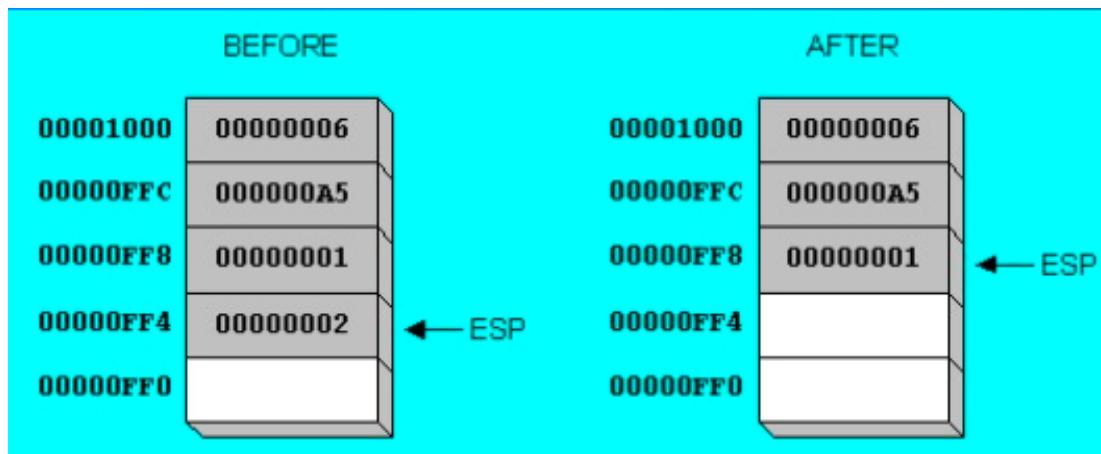
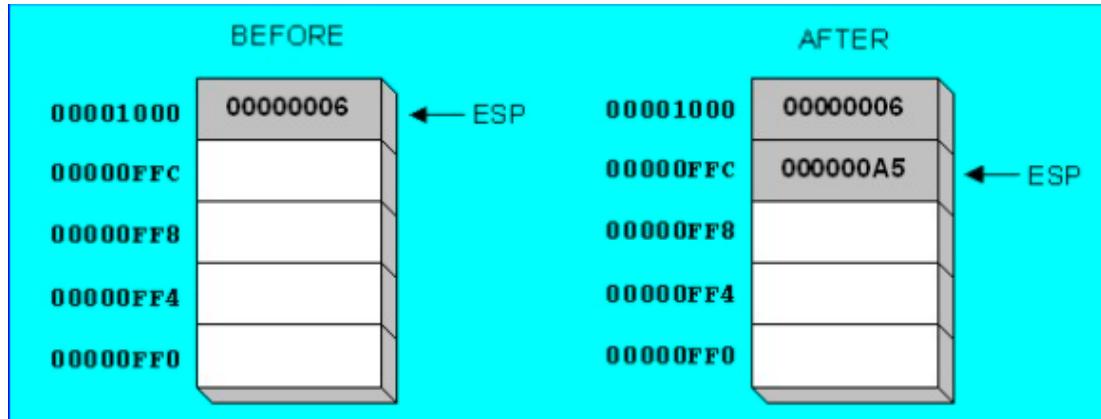


PUSH 01h



PUSH 02h

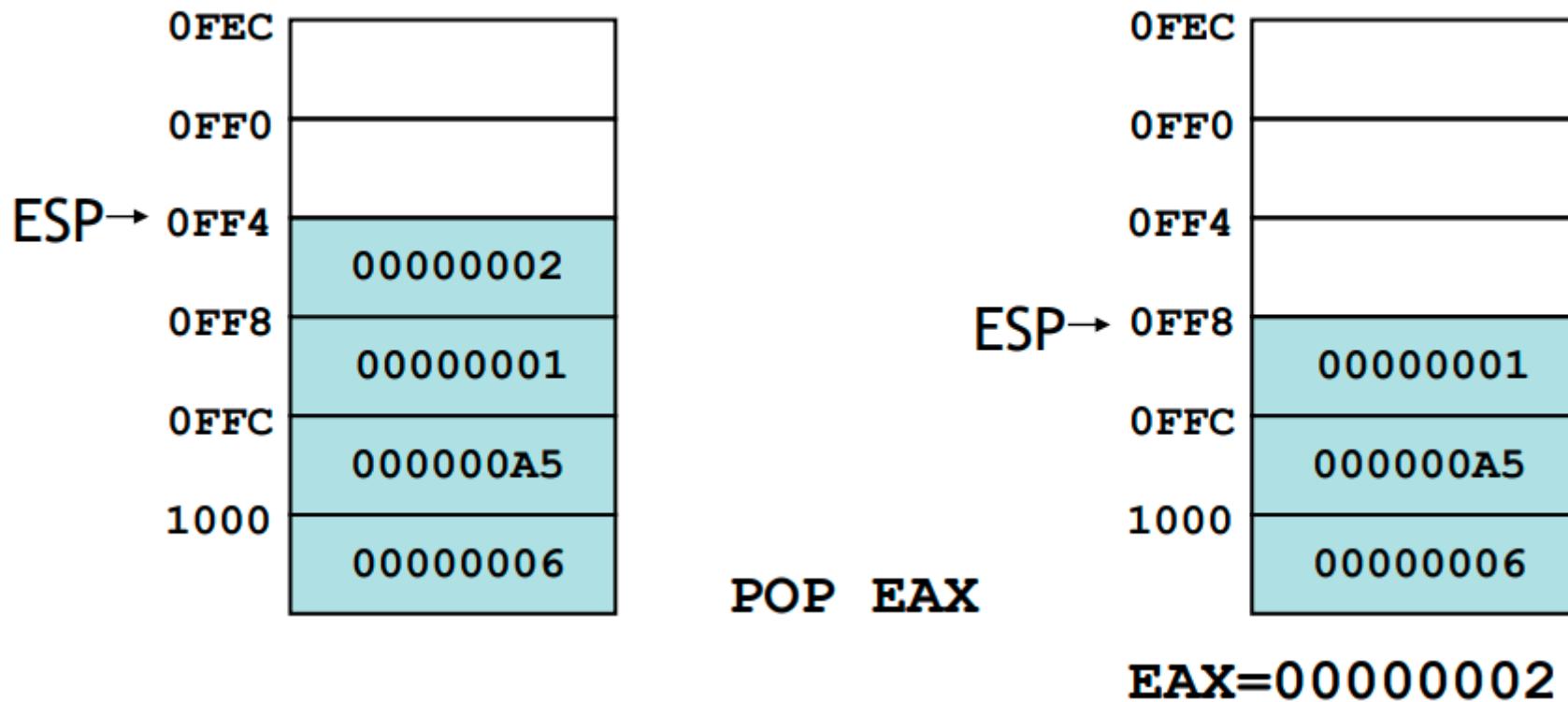
Pushing an integer on the stack(In this diagram top of stack moves downward when stack pointer decreases in value because of arrangement of offset addresses).( It is an alternate version of previous diagrams)



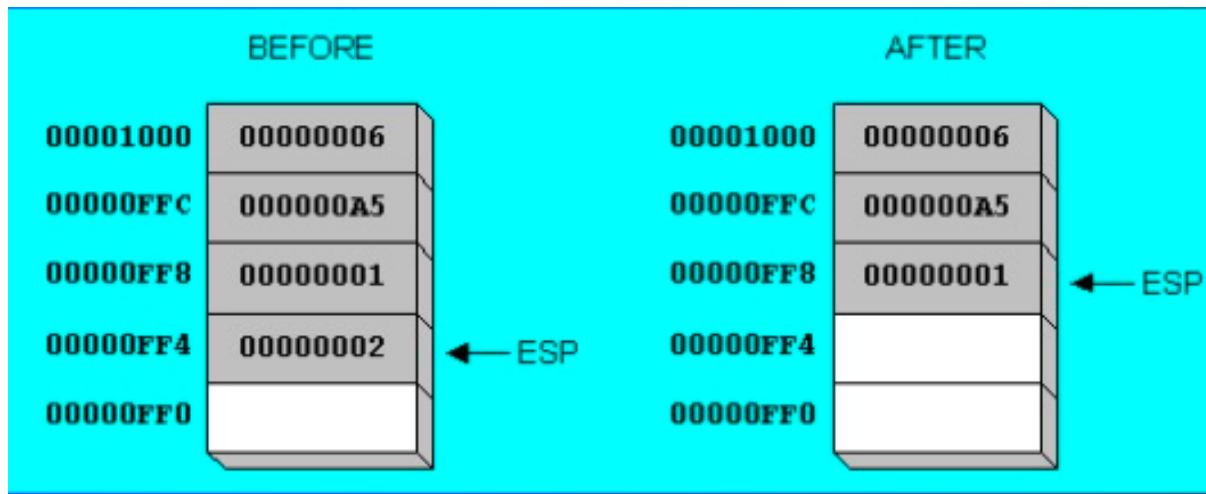
# Pop Operation

- First Copies value at stack[ESP] into a register or variable then adds  $n$  to ESP, where  $n$  is either 2 or 4 depending on the attribute of the operand receiving the data

# Popping a value from the runtime stack



Popping a value from the runtime stack(this is the alternate version of previous diagram)



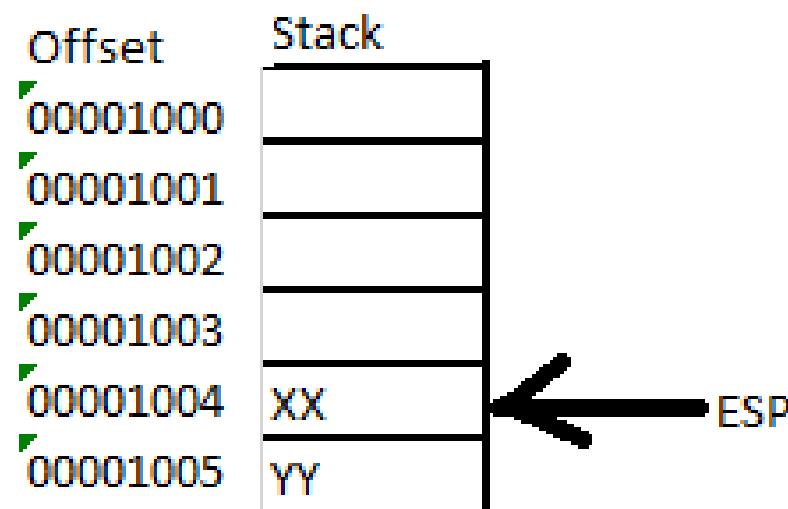
# Exercise 1:

After the execution of the given instructions where will the ESP point? Also show runtime stack

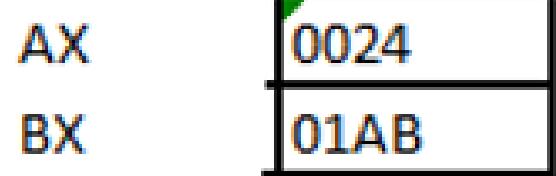
AX  
BX

0024
01AB

PUSH AX  
PUSH BX



# Solution



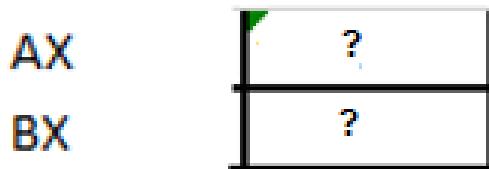
PUSH AX  
PUSH BX

Offset	Stack
00001000	AB
00001001	01
00001002	24
00001003	00
00001004	XX
00001005	YY

A black arrow points from the label 'ESP' to the top of the stack table, indicating the current position of the stack pointer.

## Exercise 2:

After the execution of the given instructions where will the ESP point? Also show runtime stack



POP      BX  
POP      AX

Offset	Stack
00001000	AB
00001001	01
00001002	24
00001003	00
00001004	XX
00001005	YY

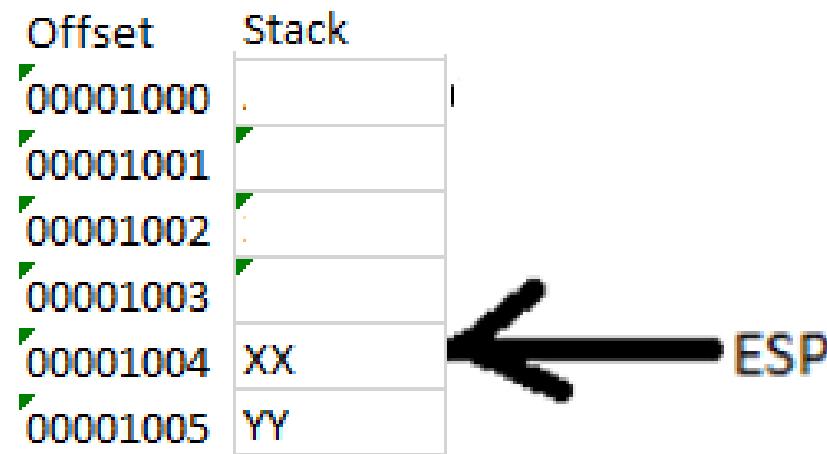


ESP

# Solution



POP BX  
POP AX



**Note:** If you observe the actual memory while running your program, you will find that after pop instruction is executed the values are copied into the destination, but they still are present in the same location of memory. However we consider this portion that is lower addresses as logically empty, these values will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

# When to use Stacks( Applications)

- Temporary save area for registers
- To save return address for CALL
- To pass arguments
- Local variables
- Applications which have LIFO nature, such as reversing a string

## Example(1) of using stacks:

- Save and restore registers when they contain important values. Note that the PUSH and POP instructions are in the opposite order:

```
push esi          ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal ; starting OFFSET
mov ecx,LENGTHOF dwordVal; number of units
mov ebx,TYPE dwordVal ;size of a doubleword
call DumpMem        ; display memory

pop ebx           ; opposite order
pop ecx
pop esi
```

## Example(2): Nested Loop

- When creating a nested loop, push the outer loop counter before entering the inner loop:

```
    mov ecx,100      ; set outer loop count
L1:
    push ecx        ; save outer loop count

    mov ecx,20       ; set inner loop count
L2:
    ;
    ;
    loop L2         ; repeat the inner loop

    pop ecx         ; restore outer loop count
    loop L1         ; repeat the outer loop
```

## Example(3a): reversing a string (1 of 2)

```
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC
; Push the name on the stack.
    mov ecx, nameSize
    mov esi, 0
L1:
    movzx eax, aName[esi]      ; get character
    push eax                  ; push on stack
    inc esi
Loop L1
```

## Example(3a): reversing a string (2 of 2)

```
; Pop the name from the stack, in reverse,  
; and store in the aName array.  
mov ecx, nameSize  
mov esi, 0  
L2:  
    pop eax          ; get character  
    mov aName[esi], al ; store in string  
    inc esi  
    Loop L2  
  
exit  
main ENDP  
END main
```

## Example(3b): reversing a string (alternate code)(1/3)

```
INCLUDE Irvine32.inc
.data
s BYTE "Good Morning",0
.code
main PROC
mov esi, 0
mov ecx, LENGTHOF s
dec ecx
L1:
movzx eax, s[esi]
push eax
inc esi
loop L1
```

## Example(3b): reversing a string (alternate code)(2/3)

```
mov esi, 0
mov ecx, LENGTHOF s
dec ecx
L2:
pop eax
mov s[esi], al
inc esi
loop L2
mov edx, OFFSET s
call WriteString
; The writestring procedure writes a null-terminated string
; to the console window, pass the offset of string in EDX
```

## Example(3b): reversing a string (alternate code)(3/3)

```
call Crlf ; advances the cursor to the beginning of the next line in console window  
call DumpRegs  
exit  
main ENDP  
END main
```

## Exercise 3:

- a. Prompt the user for five 32-bit signed integer values, then output the values in reverse order.
- b. If value of ESP is initially 0041FAD0h. Draw the runtime stack after the complete execution of loop L1. Also state the value of ESP each time the L1 executes.
- c. State the value of final value of ESP at the end of the program. Also state the value of ESP each time the loop L2 executes.

## Solution( part a)

```
INCLUDE Irvine32.inc
.data
a BYTE "Enter a value:",0
.code
main PROC
mov ecx, 5
L1:
mov edx, OFFSET a
call WriteString
call ReadInt      ; reads 32-bit signed integer from keyboard and returns the value in EAX
push eax
Loop L1
mov ecx, 5
L2:
pop eax
call WriteInt    ; writes 32-bit signed integer to console window, pass the integer to EAX
Loop L2
main ENDP
END main
```

# Related Instructions

- **PUSHFD** and **POPFD**
  - push and pop the (32-bit) EFLAGS register on the stack
  - LAHF, SAHF are other ways to save flags
  - The MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags.
- **PUSHAD** pushes the 32-bit general-purpose registers on the stack in the following order
  - EAX, ECX, EDX, EBX, ESP(value before executing PUSHAD), EBP, ESI and EDI
- **POPAD** pops the same registers off the stack in reverse order
- **PUSHA** and **POPA** do the same for 16-bit registers

# Example:(PUSHAD and POPAD)

```
MySub PROC
pushad    ; save general-purpose registers
...
; modify some registers
...
popad    ; restore general-purpose registers
ret
MySub ENDP
```

Note: Do not use this if your procedure uses registers for return values

# When not to use PUSHAD

- Example:

```
ReadValue PROC
    pushad          ; save general-purpose registers
    .
    .
    mov eax, return_value
    .
    .
    popad          ; error: overwrites EAX!
    ret
ReadValue ENDP
```

# Example(PUSHFD and POPFD)

```
.data  
saveFlags DWORD ?  
  
.code  
pushfd          ; push flags on stack  
pop saveFlags   ; copy into a variable  
....  
push saveFlags  ; push saved flag values  
popfd           ; copy into the flags
```

# Test run: Run this code and observe the values at each step

```
INCLUDE Irvine32.inc
.data
saveflags DWORD ?
.code
main PROC
mov eax, 90
neg eax
inc eax
call DumpRegs
pushfd
pop saveflags
mov eax, 5
sub eax,6
call DumpRegs
push saveflags
popfd
call DumpRegs
exit
main ENDP
End main
```

# Defining and using Procedures

- Concepts:

Blocks of code that are called and must be returned from

1. A procedure begins with **itsname PROC** and terminate with **itsname ENDP**
2. To end a procedure other than the program startup procedure (main), use **ret** instruction
3. Use call **itsname** to call the procedure

It is highly desirable to preserve the registers when writing a procedure.

# Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function
- Following is an assembly language procedure named sample:

```
sample PROC
.
.
ret
sample ENDP
```

- A named block of statements that ends with a **return**

# Documenting Procedures:

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- **Receives:** A list of input parameters; state their usage and requirements.
- **Returns:** A description of values returned by the procedure.
- **Requires:** Optional list of requirements called preconditions that must be satisfied before the procedure is called.
  - For example, a procedure of drawing lines could assume that video display adapter is already in graphics mode.

# Example: Sum of three integers

```
;-----  
SumOf PROC  
;  
; Calculates and returns the sum of three 32-bit integers  
;Receives: EAX, EBX, ECX, the three integers.  
; May be signed or unsigned.  
;Returns: EAX = sum, and the status flags ;  
(Carry, Overflow, etc.) are changed.  
; Requires: nothing  
;  
add eax, ebx  
add eax, ecx  
ret  
SumOf ENDP
```

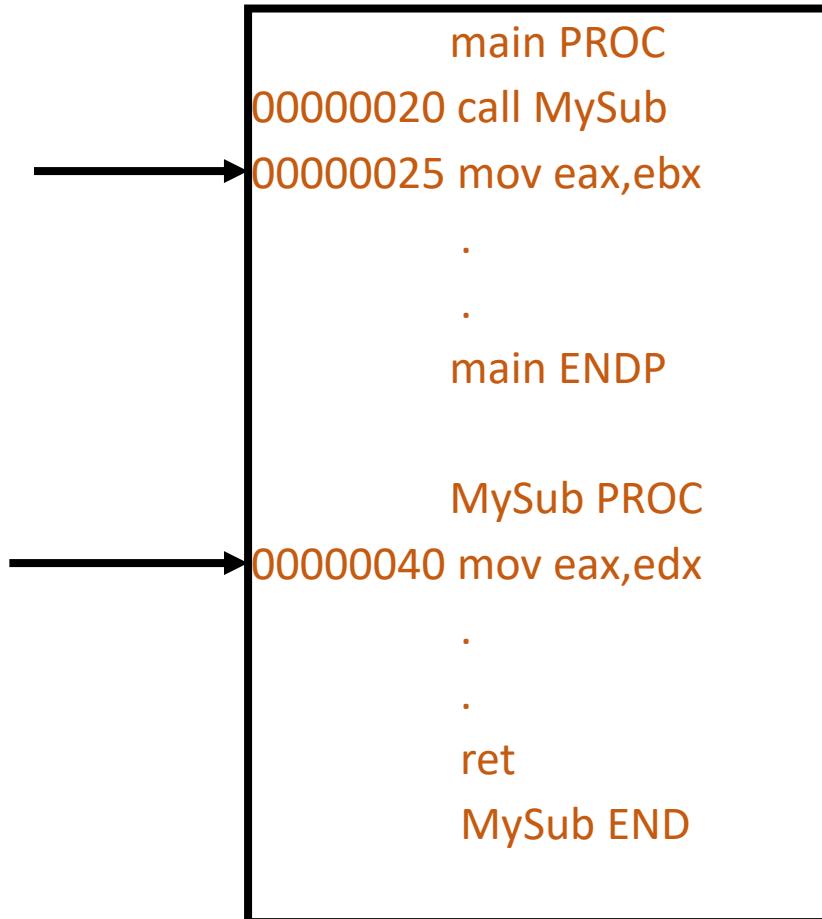
# CALL and RET instructions

- The **CALL** instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into EIP
- The **RET** instruction returns from a procedure
  - pops top of stack into EIP

# CALL-RET example (1 of 2)

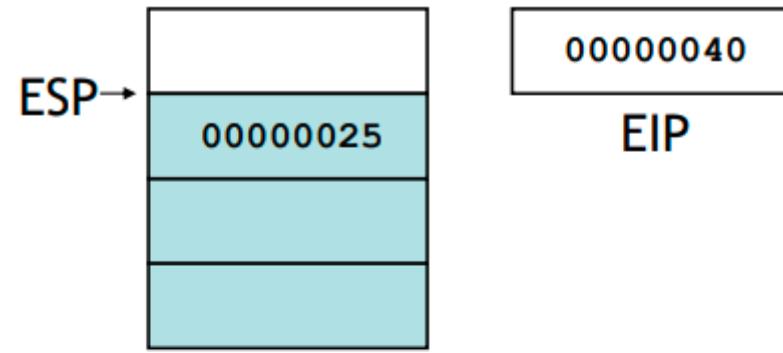
00000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub .

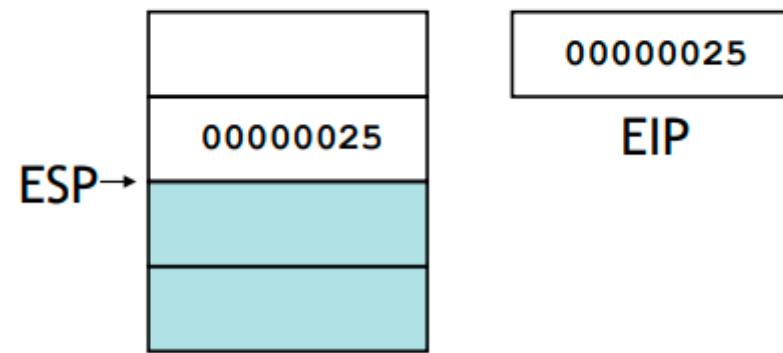


## CALL-RET example (2 of 2)

The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP



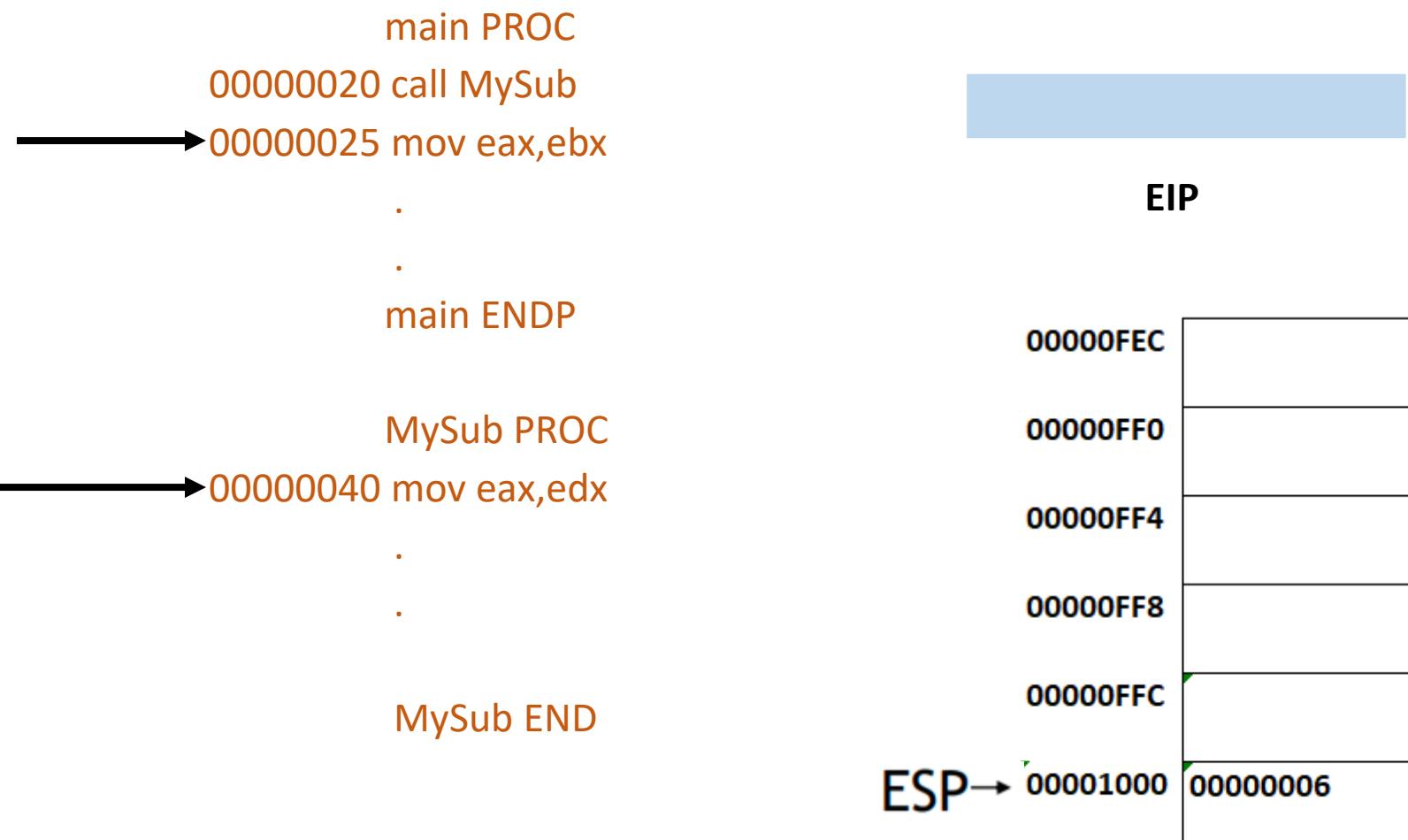
The RET instruction pops 00000025 from the stack into EIP



# CALL Instruction Detail example (before CALL)

00000025 is the offset of the instruction immediately following the CALL instruction.(In this case it is the return address of the calling program)

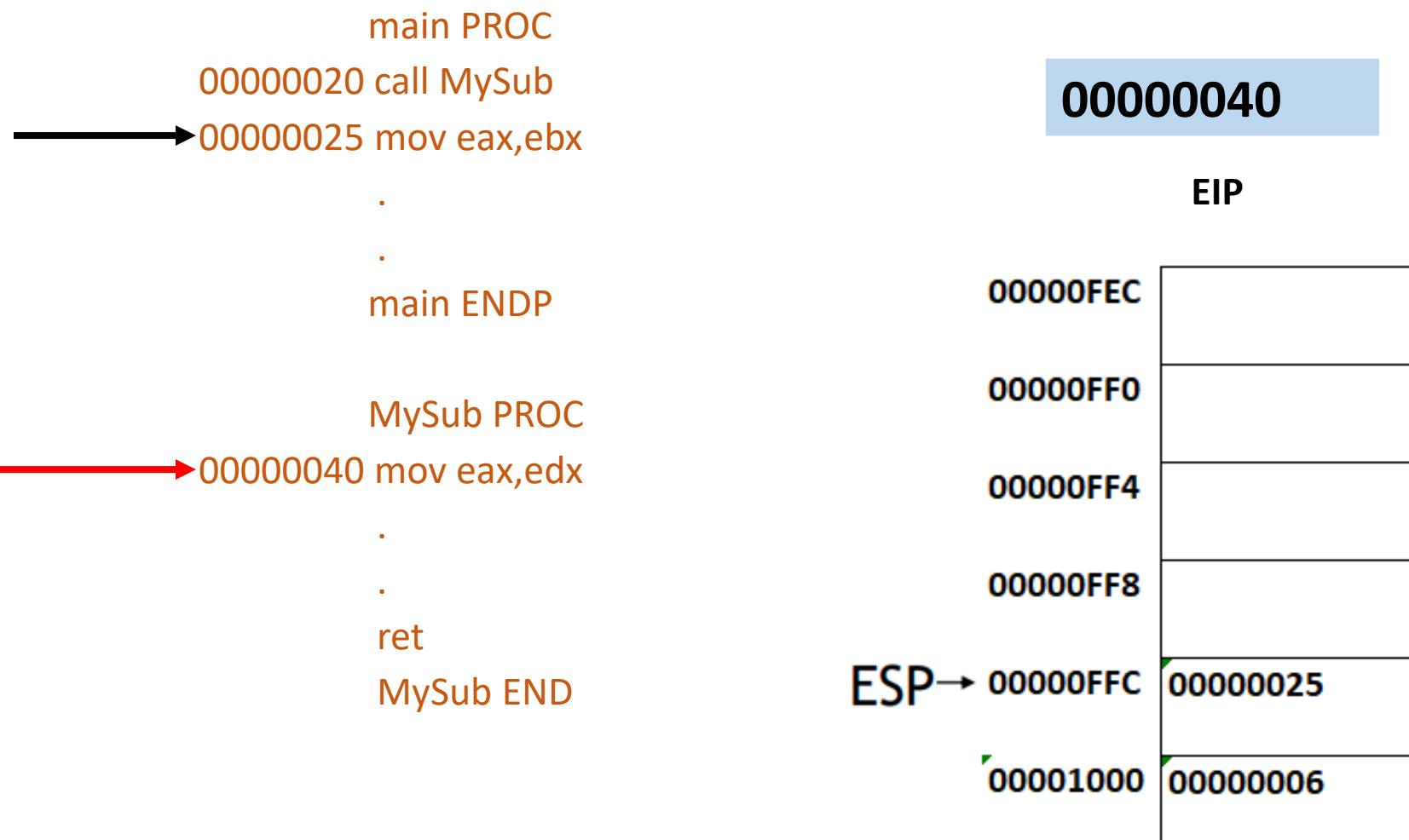
00000040 is the offset of the first instruction inside MySub .



# CALL Instruction Detail example (after CALL)

00000025 is the offset of the instruction immediately following the CALL instruction.(In this case it is the return address of the calling program)

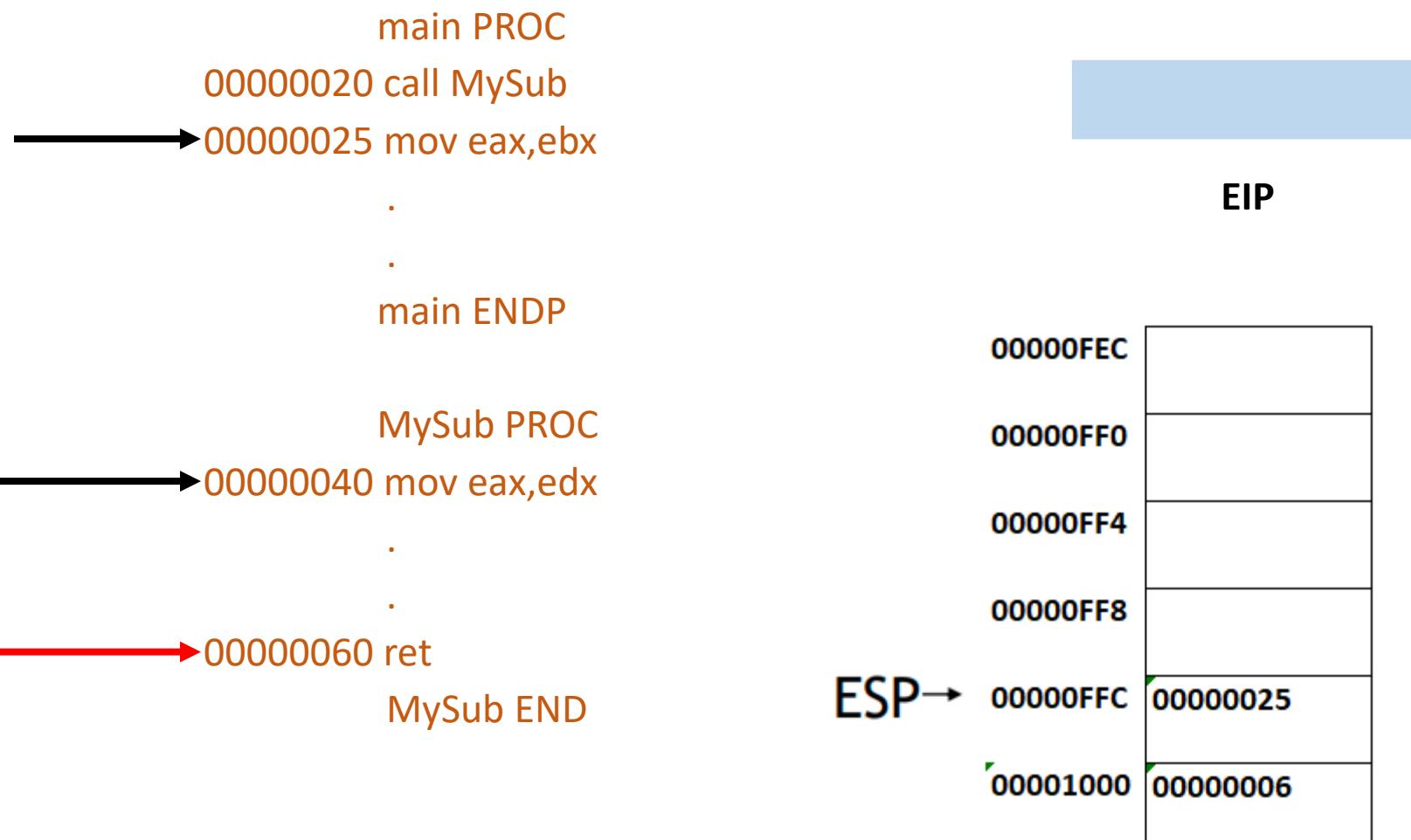
00000040 is the offset of the first instruction inside MySub .



# RET Instruction Detail example (before RET)

00000025 is the offset of the instruction immediately following the CALL instruction.(In this case it is the return address of the calling program)

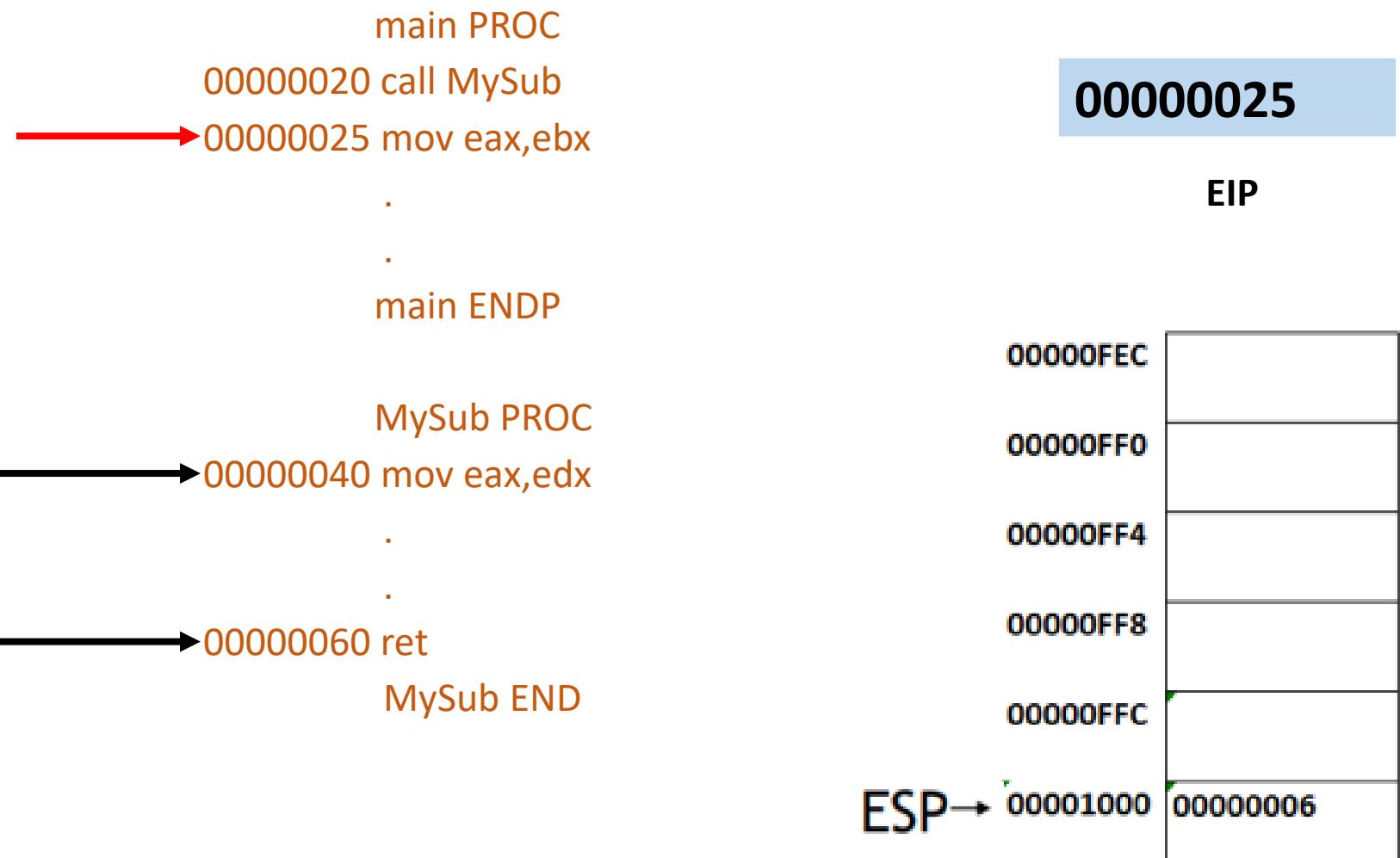
00000040 is the offset of the first instruction inside MySub .



# RET Instruction Detail example (after RET)

00000025 is the offset of the instruction immediately following the CALL instruction.(In this case it is the return address of the calling program)

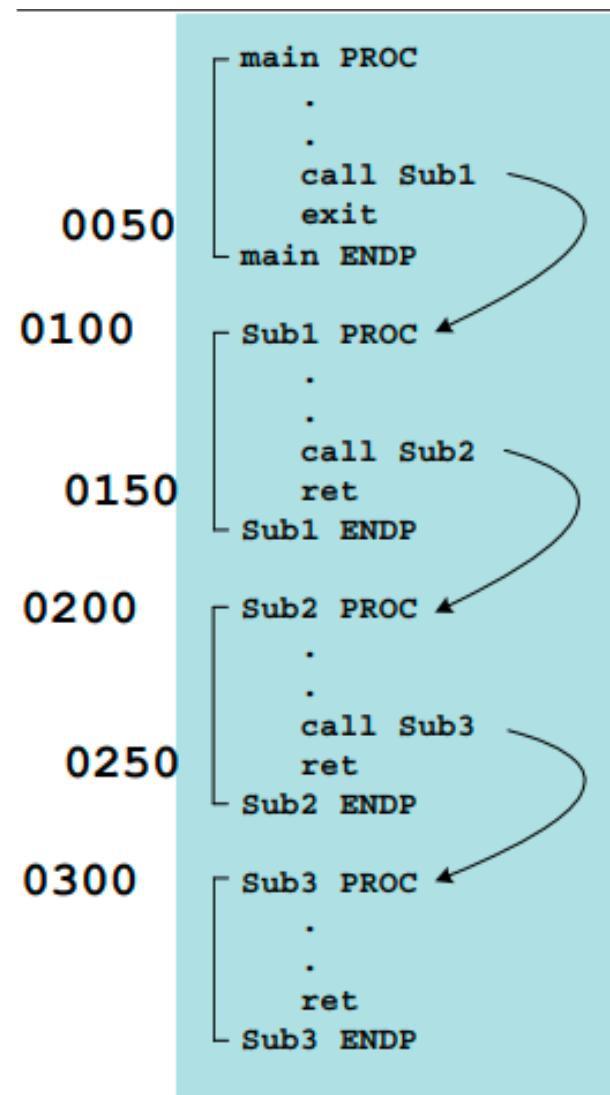
00000040 is the offset of the first instruction inside MySub .



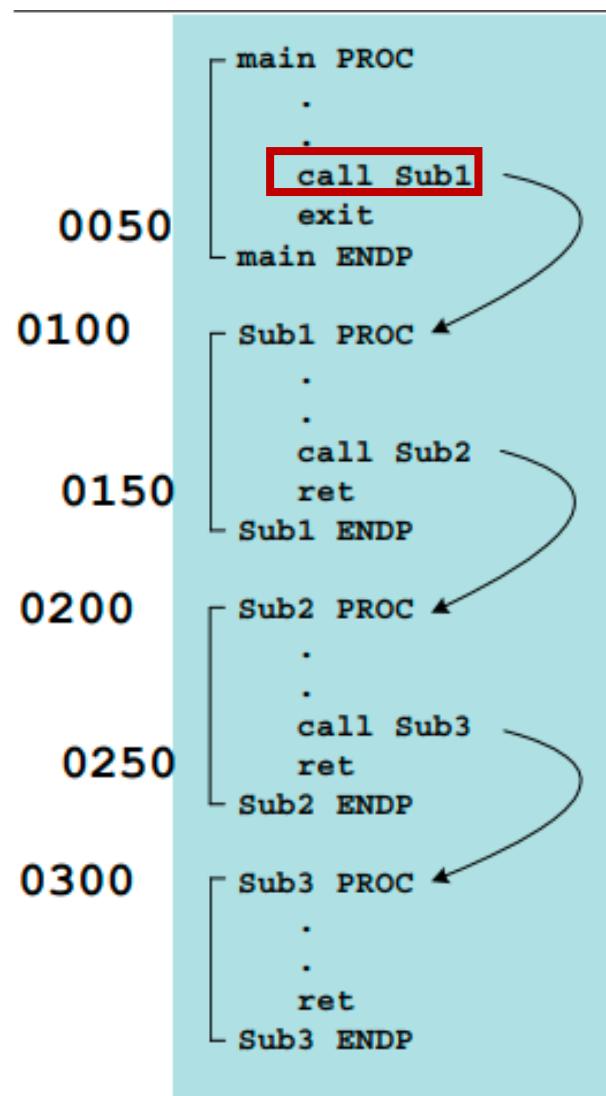
# Nested Procedure Call

- Occurs when a called procedure calls another procedure before the first procedure returns

# Nested Procedure calls



# Nested Procedure calls



0050  
0100  
0150  
0200  
0250  
0300

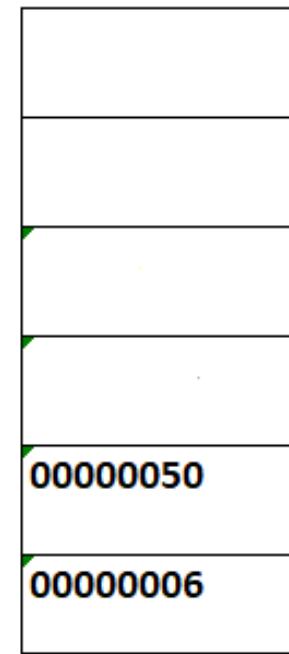
main PROC  
.  
. .  
call Sub1  
exit  
main ENDP

Sub1 PROC  
.  
. .  
call Sub2  
ret  
Sub1 ENDP

Sub2 PROC  
.  
. .  
call Sub3  
ret  
Sub2 ENDP

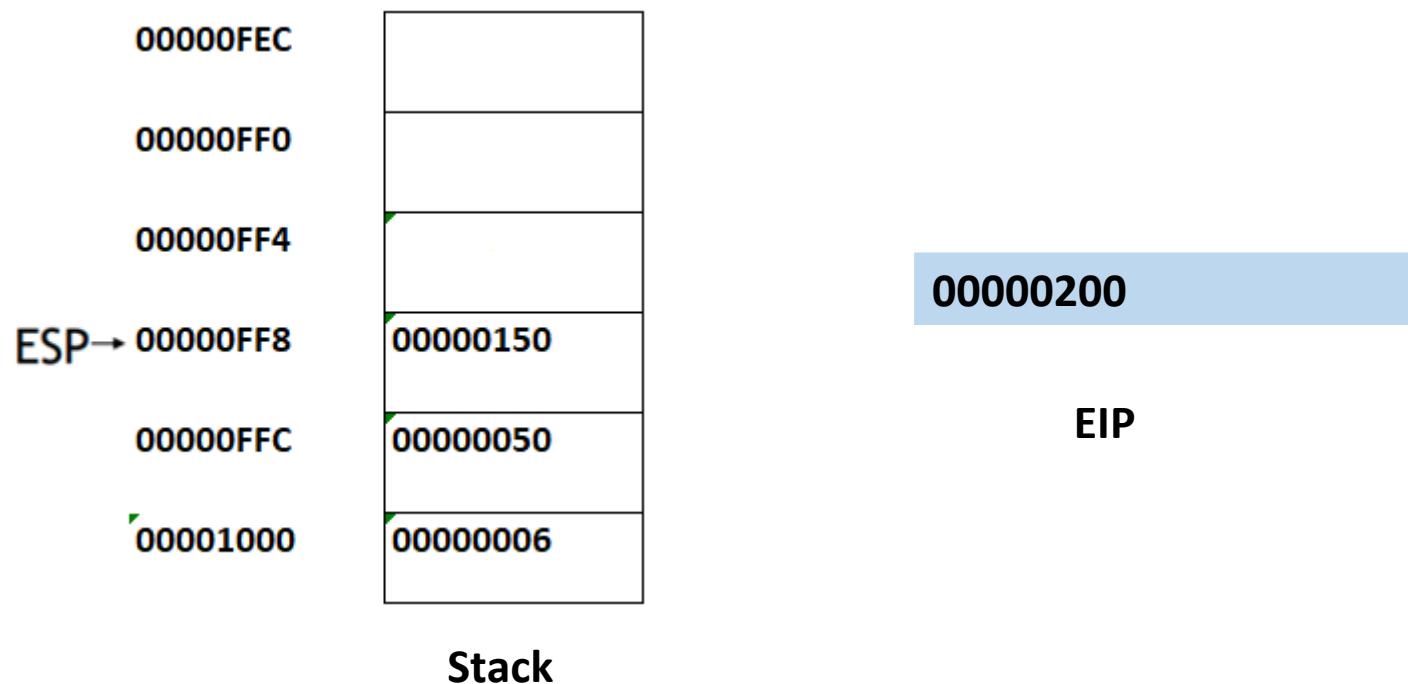
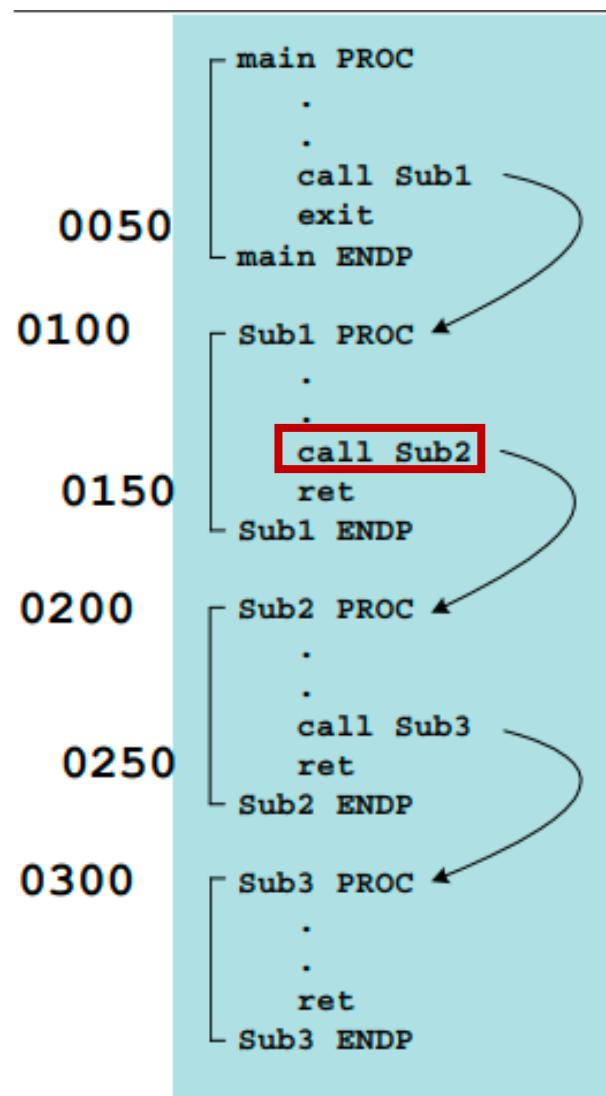
Sub3 PROC  
.  
. .  
ret  
Sub3 ENDP

ESP → 00000FFC  
00000FEC  
00000FF0  
00000FF4  
00000FF8  
00000050  
00000006  
00001000

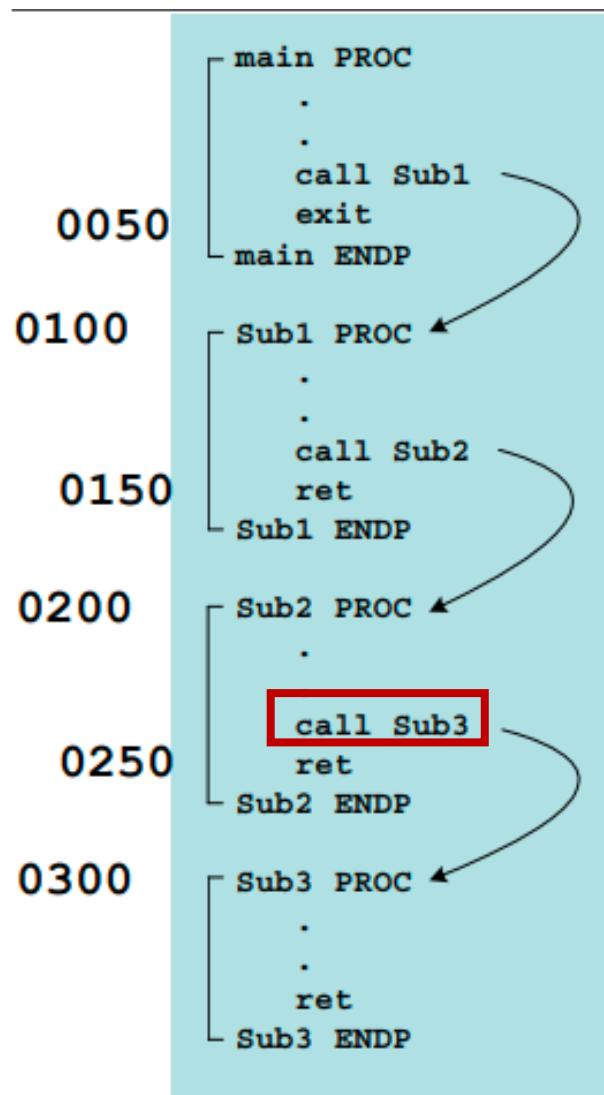


Stack

# Nested Procedure calls



# Nested Procedure calls



00000FEC  
00000FF0  
ESP → 00000FF4  
00000FF8  
00000FFC  
00001000

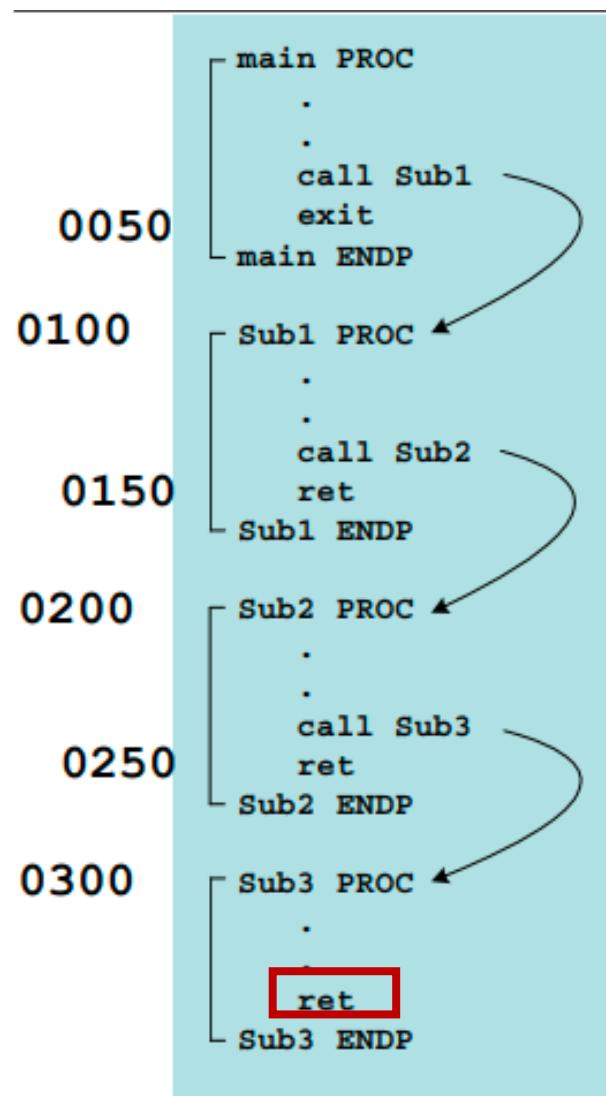
00000250
00000150
00000050
00000006

Stack

00000300

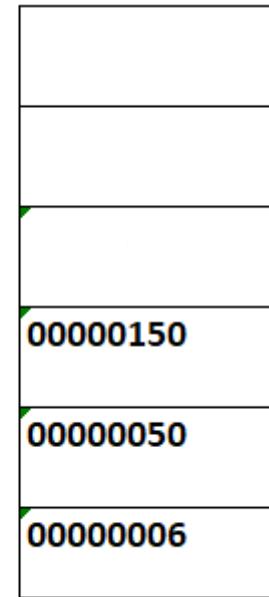
EIP

# Nested Procedure calls



0050  
0100  
0150  
0200  
0250  
0300

00000FEC  
00000FF0  
00000FF4  
ESP → 00000FF8  
00000FFC  
00001000

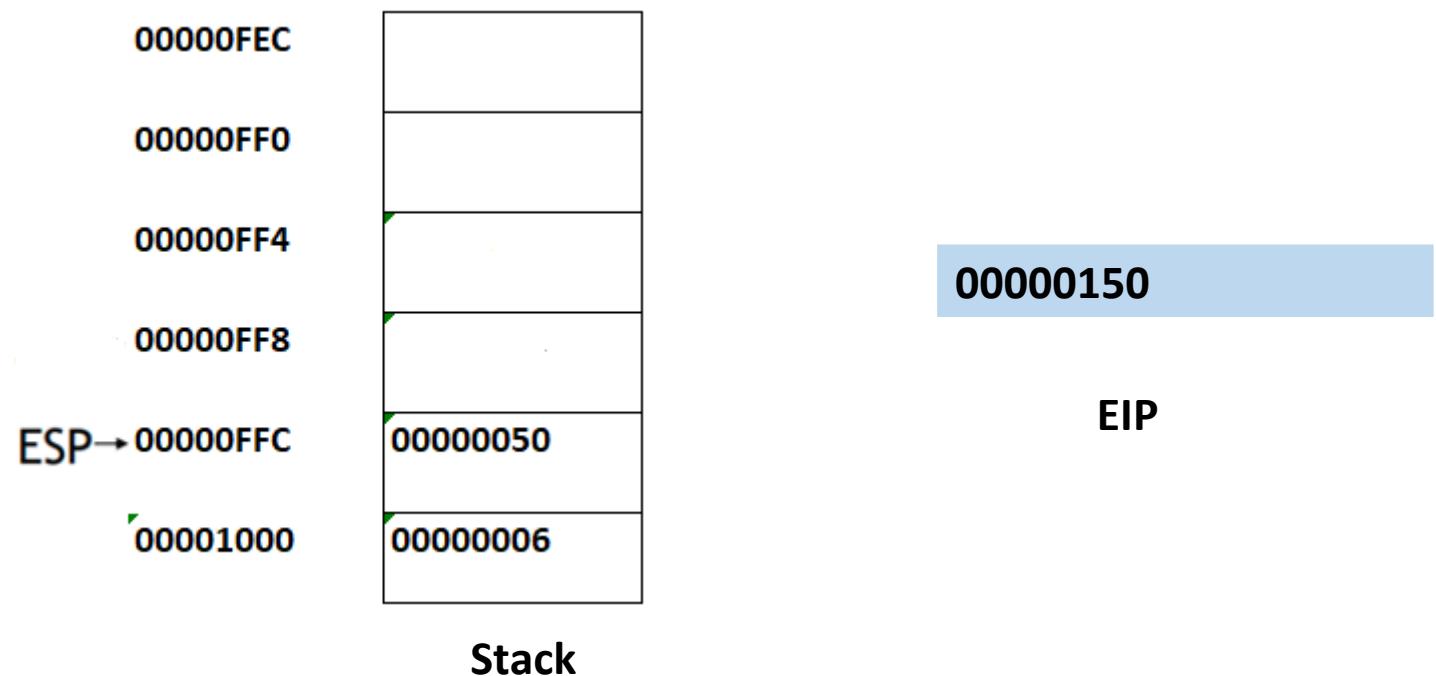
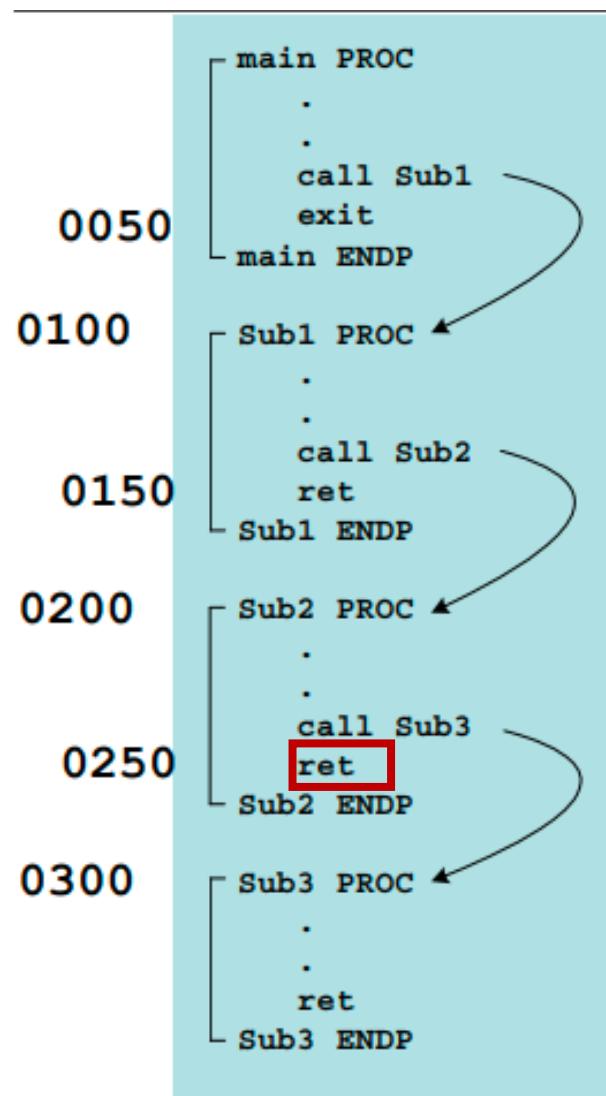


Stack

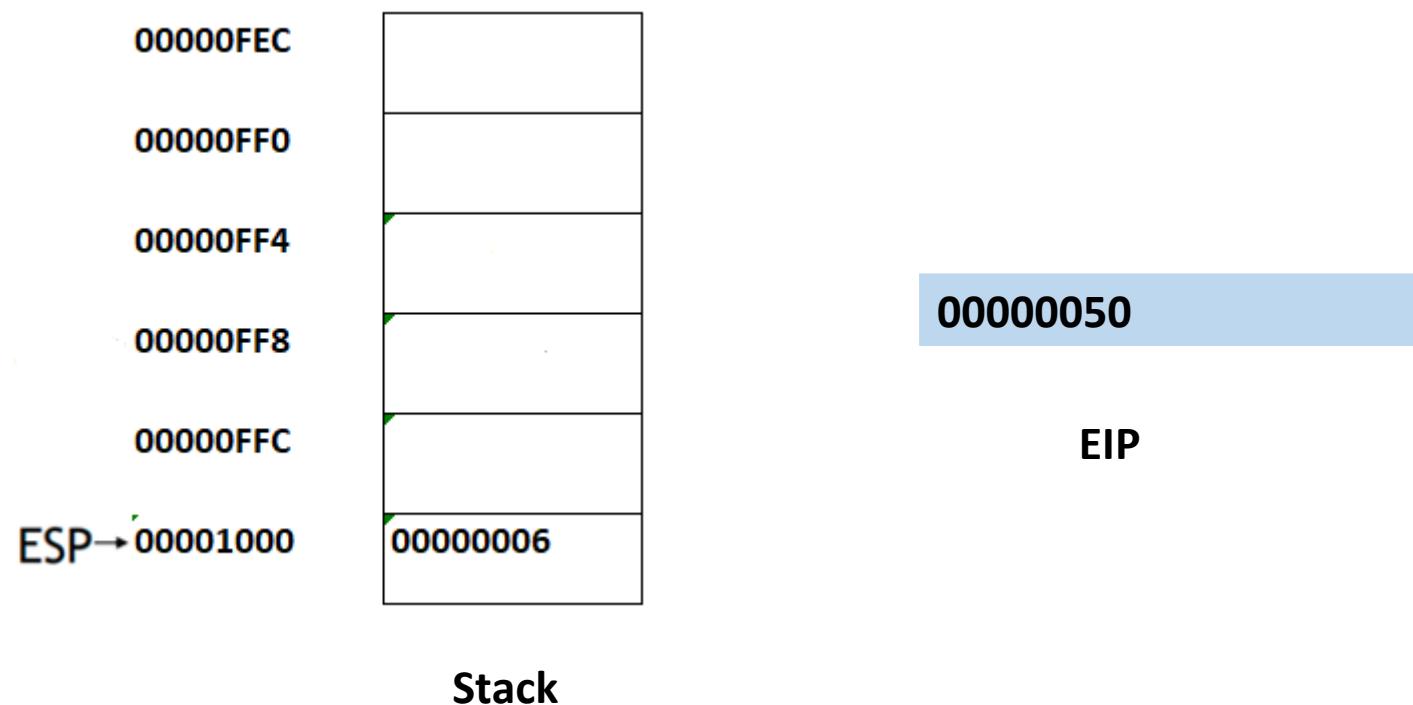
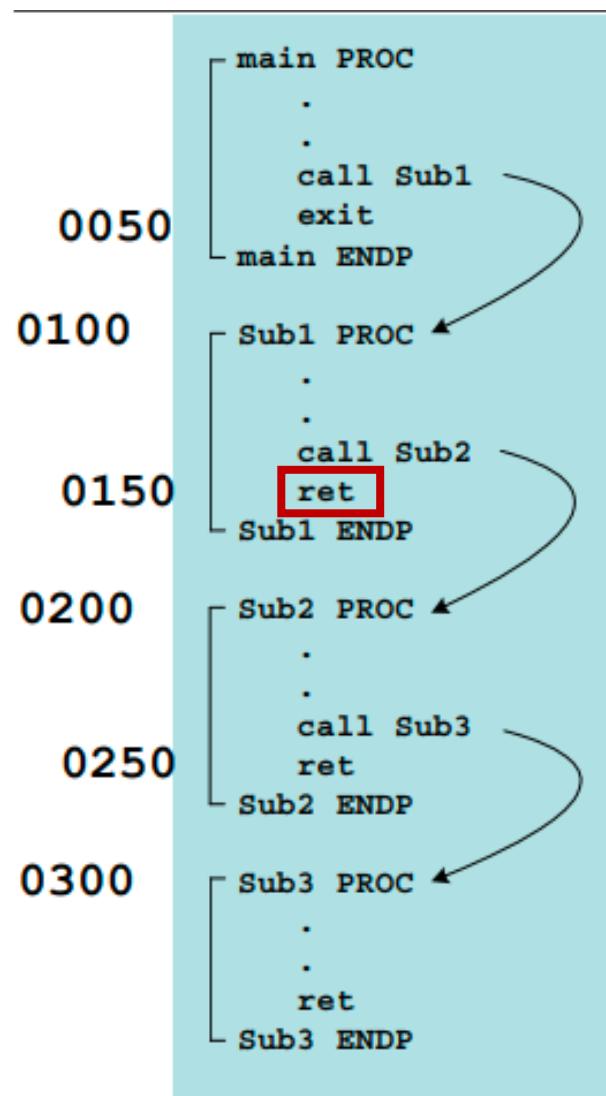
00000250

EIP

# Nested Procedure calls



# Nested Procedure calls



# Local and Global Labels

- A local label is visible only to statements inside the same procedure. A global label is visible everywhere.
- By default, a code label followed by a single colon(:) has local scope, making it visible only to statements inside its enclosing procedure
- Global labels are followed by two colons(::), making them visible to the whole program

# Example:

```
main PROC
    jmp L2          ; error!
    L1: ::          ; global label
    exit
main ENDP

sub2 PROC
    L2:             ; local label
    jmp L1          ; ok
    ret
sub2 ENDP
```

# Procedure Parameters(1 of 3)

- A good procedure might be usable in many different programs
  - But not if it refers to specific variable names
- **Parameters** help to make procedures flexible because parameter values can change at runtime
- General registers can be used to pass parameters
- **Passing Parameters:**
- In registers – Fastest
- In global variables- Hard to reuse, poor programming practice
- On the stack – Used by high level languages

# Procedure Parameters(2 of 3)

- The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names.(not a good practice)

```
ArraySum PROC
```

```
    mov esi, 0           ; array index  
    mov eax, 0           ; set the sum to zero  
    mov ecx, LengthOf myArray
```

```
L1:
```

```
    add eax, myArray[esi]      ; add each integer to sum  
    add esi, 4                 ; point to next integer  
    loop L1                   ; repeat for array size
```

```
    mov theSum, eax           ; store the sum  
    ret
```

```
ArraySum ENDP
```

# Procedure Parameters(3 of 3)

This version returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Calculates the sum of an array of 32-bit integers.
; Recevies: ESI points to an array of doublewords,
; ECX = number of array elements.
; Returns: EAX = sum
;-----
        push esi
        push ecx
        mov eax, 0          ; set the sum to zero
L1:   add eax, [esi]      ; add each integer to sum
        add esi, TYPE DWORD ; point to next integer
        loop L1             ; repeat for array size
        pop ecx
        pop esi
        ret
ArraySum ENDP
```

# Calling ArraySum

It is better to pass the offset of an array to the procedure than to include references to specific variable names inside the procedure

```
.data  
array DWORD 10000h, 20000h, 30000h, 40000h  
theSum DWORD ?  
.code  
main PROC  
    mov esi, OFFSET array  
    mov ecx, LENGTHOF array  
    call ArraySum  
    mov theSum, eax
```

# Testing ArraySum (complete program)

```
; Testing the ArraySum procedure (TestArraySum.asm)

.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?

.code
main PROC
    mov    esi,OFFSET array          ; ESI points to array
    mov    ecx,LENGTHOF array        ; ECX = array count
    call   ArraySum                 ; calculate the sum
    mov    theSum,eax               ; returned in EAX

    INVOKE ExitProcess,0
main ENDP
;-----;
; ArraySum
```

```
; Calculates the sum of an array of 32-bit integers.  
; Receives: ESI = the array offset  
; ECX = number of elements in the array  
; Returns: EAX = sum of the array elements  
;-----
```

```
ArraySum PROC
```

```
    push    esi          ; save ESI, ECX  
    push    ecx  
    mov     eax, 0       ; set the sum to zero
```

```
L1:
```

```
    add     eax, [esi]   ; add each integer to  
sum  
    add     esi, TYPE DWORD ; point to next integer  
    loop   L1           ; repeat for array size  
    pop     ecx          ; restore ECX, ESI  
    pop     esi          ;  
    ret                ; sum is in EAX
```

```
ArraySum ENDP
```

```
END main
```

# USES Operator

- lets you list the names of all registers you want to save and restore within a procedure.
- USES tells the assembler to do two things:
  - First, generate PUSH instructions that save the registers on the stack at the beginning of the procedure.
  - Second, generate POP instructions that restore the register values at the end of the procedure.
- The USES operator immediately follows PROC, and is itself followed by a list of registers on the same line separated by spaces or tabs (not commas).
- To avoid side effects return register shouldn't be saved

## Example:

```
ArraySum PROC USES esi ecx
```

```
    mov eax,0           ; set the sum to zero
```

```
L1:
```

```
    add eax,[esi]      ; add each integer to sum
```

```
    add esi,TYPE DWORD ; point to next integer
```

```
    loop L1            ; repeat for array size
```

```
    ret                ; sum is in EAX
```

```
ArraySum ENDP
```

MASM generates the following code:

```
ArraySum PROC
```

```
    push esi
```

```
    push ecx
```

```
    mov eax,0           ; set the sum to zero
```

```
L1:
```

```
    add eax,[esi]      ; add each integer to sum
```

```
    add esi,TYPE DWORD ; point to next integer
```

```
    loop L1            ; repeat for array size
```

```
    pop ecx
```

```
    pop esi
```

```
    ret
```

```
ArraySum ENDP
```

# When not to push a register

```
SumOf PROC      ; sum of three integers  
    push eax      ;line 1  
    add eax, ebx   ;line 2  
    add eax, ecx   ;line 3  
    pop eax       ;line 4  
    ret
```

SumOf ENDP

The sum of three integers is stored in EAX on line 3 but pop instruction replaces it with the starting value of EAX on line 4

Adapted from:

Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)

# Computer Organization and Assembly Language

Week 8

Chapter 5 “Procedures”

# Outline

- Link Library Overview
- Calling a Library Procedure
- Linking to a Library
- Library Procedures – Overview
- Six Examples
- Conditional Branching
- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures

# Link Library Overview

- A file containing procedures that have been compiled into machine code
  - constructed from one or more OBJ files
- To build a library, . . .
  - start with one or more ASM source files
  - assemble each into an OBJ file
  - create an empty library file (extension .LIB)
  - add the OBJ file(s) to the library file, using the Microsoft LIB utility

# Linking to External Library

- **Link Library**

A file containing procedures that have been assembled into machine code

- In your program, these procedures could be included and called
- The assembler would leave the target address of the Call instruction blank, which will be filled by the linker

- **Linker Command**

- The linker utility combines a program's object file with one or more object files and link libraries. The following command, for example, links hello.obj to the irvine32.lib and kernel32.lib library files:

- `link hello.obj irvine32.lib kernel32.lib`

- **Link Libraries:**

- Irvine32.lib
- Irvine64.obj

# Calling a Library Procedure

- Call a library procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.code
    mov eax,1234h          ; input argument
    call WriteHex          ; show hex number
    call Crlf              ; end of line
```

# Library Procedures - Overview

**CloseFile** – Closes an open disk file

**Clrscr** - Clears console, locates cursor at upper left corner

**CreateOutputFile** - Creates new disk file for writing in output mode

**Crlf** - Writes end of line sequence to standard output

**Delay** - Pauses program execution for  $n$  millisecond interval

**DumpMem** - Writes block of memory to standard output in hex

**DumpRegs** – Displays general-purpose registers and flags (hex)

**GetCommandtail** - Copies command-line args into array of bytes

**GetDateTime** – Gets the current date and time from the system

**GetMaxXY** - Gets number of cols, rows in console window buffer

**GetMseconds** - Returns milliseconds elapsed since midnight

# Library Procedures - Overview

**GetTextColor** - Returns active foreground and background text colors in the console window

**Gotoxy** - Locates cursor at row and column on the console

**IsDigit** - Sets Zero flag if AL contains ASCII code for decimal digit (0–9)

**MsgBox**, **MsgBoxAsk** – Display popup message boxes

**OpenInputFile** – Opens existing file for input

**ParseDecimal32** – Converts unsigned integer string to binary

**ParseInteger32** - Converts signed integer string to binary

**Random32** - Generates 32-bit pseudorandom integer in the range 0 to FFFFFFFFh

**Randomize** - Seeds the random number generator

**RandomRange** - Generates a pseudorandom integer within a specified range

**ReadChar** - Reads a single character from standard input

# Library Procedures - Overview

**ReadDec** - Reads 32-bit unsigned decimal integer from keyboard

**ReadFromFile** – Reads input disk file into buffer

**ReadHex** - Reads 32-bit hexadecimal integer from keyboard

**ReadInt** - Reads 32-bit signed decimal integer from keyboard

**ReadKey** – Reads character from keyboard input buffer

**ReadString** - Reads string from standard input, terminated by [Enter]

**SetTextColor** - Sets foreground and background colors of all subsequent console text output

**Str\_compare** – Compares two strings

**Str\_copy** – Copies a source string to a destination string

**StrLength** – Returns length of a string

**Str\_trim** - Removes unwanted characters from a string.

# Library Procedures - Overview

**Str\_Ucase** - Converts a string to uppercase letters.

**WaitMsg** - Displays message, waits for Enter key to be pressed

**WriteBin** - Writes unsigned 32-bit integer in ASCII binary format.

**WriteBinB** – Writes binary integer in byte, word, or doubleword format

**WriteChar** - Writes a single character to standard output

**WriteDec** - Writes unsigned 32-bit integer in decimal format

**WriteHex** - Writes an unsigned 32-bit integer in hexadecimal format

**WriteHexB** – Writes byte, word, or doubleword in hexadecimal format

**WriteInt** - Writes signed 32-bit integer in decimal format

# Library Procedures - Overview

**WriteStackFrame** - Writes the current procedure's stack frame to the console.

**WriteStackFrameName** - Writes the current procedure's name and stack frame to the console.

**WriteString** - Writes null-terminated string to console window

**WriteToFile** - Writes buffer to output file

**WriteWindowsMsg** - Displays most recent error message generated by MS-Windows

**Note:** Do remember the name of registers in which input parameters for these procedures are passed and output is returned

# Irvine Library Help

- A Windows help file showing:
- Irvine Library Procedures
  - Procedure Purpose
  - Calling & Return Arguments
  - Example of usage
- Some other information

[IrvineLibHelp.chm](#)

# Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code  
    call Clrscr  
    mov  eax,500  
    call Delay  
    call DumpRegs
```

Sample output:

```
EAX=00000613  EBX=00000000  ECX=000000FF  EDX=00000000  
ESI=00000000  EDI=00000100  EBP=0000091E  ESP=000000F6  
EIP=00401026  EFL=00000286  CF=0  SF=1  ZF=0  OF=0
```

## Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data
    str1 BYTE "Assembly language is easy!",0
.code
    mov edx,OFFSET str1
    call WriteString
    call Crlf
```

## Example 2a

Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)

```
.data  
    str1 BYTE "Assembly language is easy!",0Dh,0Ah,0  
.code  
    mov edx,OFFSET str1  
    call WriteString
```

# Example 3

Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line.

```
IntVal = 35
.code
    mov  eax,IntVal
    call WriteBin           ; display binary
    call Crlf
    call WriteDec           ; display decimal
    call Crlf
    call WriteHex           ; display hexadecimal
    call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

## Example 4

Input a string from the user. EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter.

```
.data  
    fileName BYTE 80 DUP(0)  
.code  
    mov edx, OFFSET fileName  
    mov ecx, SIZEOF fileName - 1  
    call ReadString
```

A null byte is automatically appended to the string.

## Example 5

Generate and display ten pseudorandom signed integers in the range 0 – 99. Pass each integer to WriteInt in EAX and display it on a separate line.

```
.code
    mov ecx,10          ; loop counter
L1: mov eax,100        ; ceiling value
    call RandomRange   ; generate random int
    call WriteInt      ; display signed int
    call CrLf          ; goto next display line
    loop L1            ; repeat loop
```

# Example 6

Display a null-terminated string with yellow characters on a blue background.

```
.data  
    str1 BYTE "Color output is easy!",0  
.code  
    mov eax,yellow + (blue * 16)  
    call SetTextColor  
    mov edx,OFFSET str1  
    call WriteString  
    call Crlf
```

The background color is multiplied by 16 before being added to the foreground color.

# Activity:(To be done by students)

## Integer Summation Program

**Description:** Write a program that prompts the user for multiple 32-bit integers, stores them in an array, calculates the sum of the array, and displays the sum on the screen

## Integer Summation Program

Main steps:

Prompt user for multiple integers

Calculate the sum of the array

Display the sum

# Procedure Design

Main

  Clrscr ; clear screen

  PromptForIntegers

    WriteString ; display string

    ReadInt ; input integer

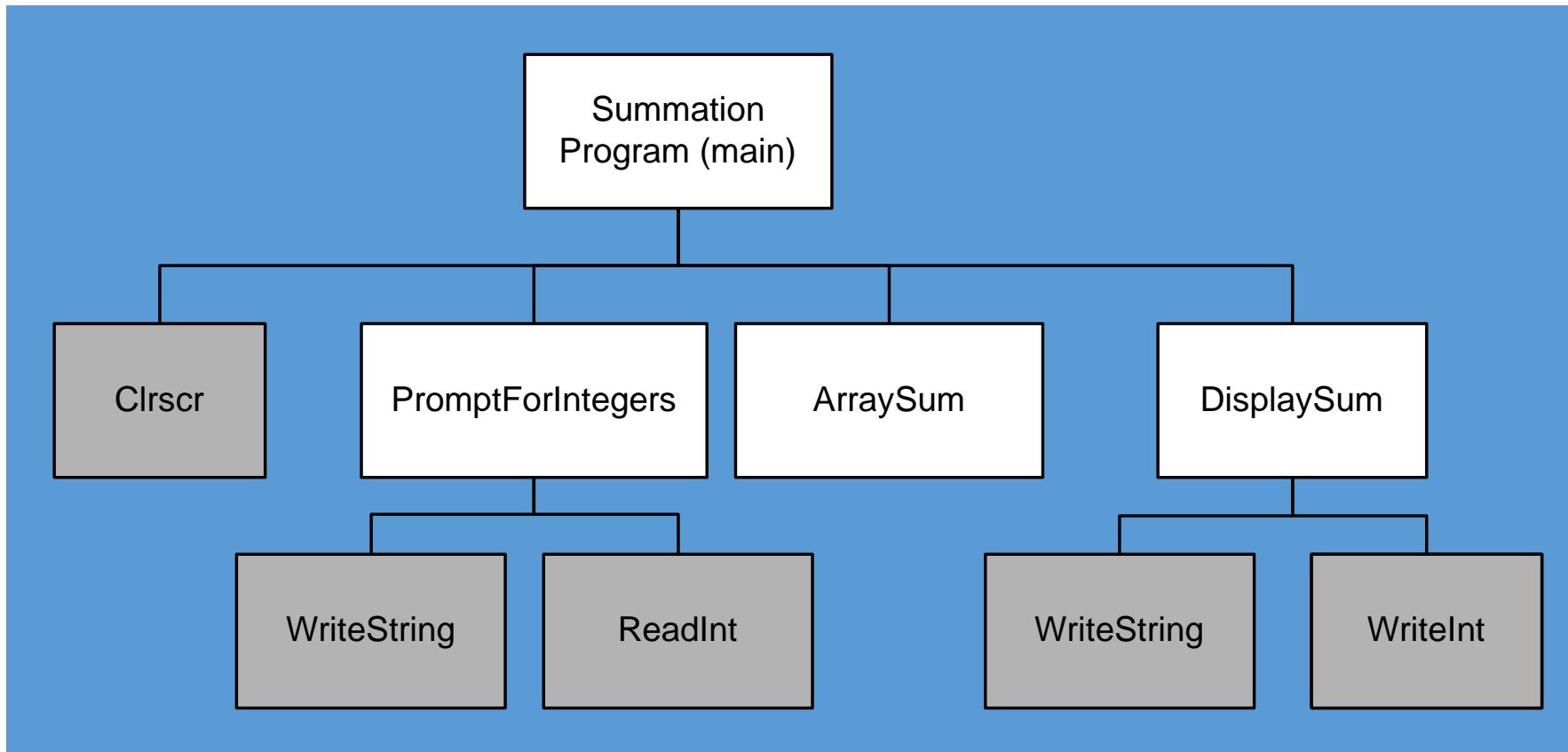
  ArraySum ; sum the integers

  DisplaySum

    WriteString ; display string

    Writeln ; display integer

# Structure Chart



gray indicates library procedure

# Sample Output

```
Enter a signed integer: 550
Enter a signed integer: -23
Enter a signed integer: -96
The sum of the integers is: +431
```

# Chapter 6 “Conditional Processing”

# Boolean and Comparison Instructions

# NOT Instruction

- Performs a bitwise Boolean NOT operation(inverts all bits) on a single destination operand.
- Syntax:
  - NOT reg
  - NOT mem
- Flags
  - No flags affected
- Example:

```
mov al,00111011b  
not al           ; AL = 11000100b
```

X	$\neg X$
F	T
T	F

$$\begin{array}{r} \text{NOT} \quad \underline{0\ 0\ 1\ 1\ 1\ 0\ 1\ 1} \\ \quad \quad \quad 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0 \end{array}$$

# AND Instruction

- Performs a bitwise Boolean AND operation between each pair of matching bits in two operands
- Syntax:

*AND destination, source*

AND reg,reg

AND reg,mem

AND reg,imm

AND mem,reg

AND mem,imm

- Flags:

CF=0, OF =0, sign, parity and zero are modified according to the value assigned to destination operand

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

- Example:

```
mov al,10101110b  
and al,11110110b      ; result in AL = 10100110b
```

- Application:

Clear 1 or more bits in an operand without affecting other bits. The technique is called bit masking.

### Example:

```
mov al, 00111011b  
and al, 00001111b
```

00111011	
AND 00001111	
<hr/>	
cleared	0 0 0 0   1 0 1 1
bit extraction	
unchanged	

# OR Instruction

- Performs a bitwise Boolean OR operation between each pair of matching bits in two operands
- Syntax :
  - *OR destination, source*

OR reg,reg

OR reg,mem

OR reg,imm

OR mem,reg

OR mem,imm

- Flags:

CF=0, OF =0, sign, parity and zero are modified according to the value assigned to destination operand

- Note:

ORing any number with itself does not change its value.

X	Y	X V Y
0	0	0
0	1	1
1	0	1
1	1	1

- Example:

```
mov al, 11100011b  
or al, 00000100b ; result in AL = 11100111b
```

- Application:

set 1 or more bits in an operand without affecting any other bits.

Example:

```
mov dl, 00111011b  
or dl, 00001111b
```

0 0 1 1 1 0 1 1		
OR 0 0 0 0 1 1 1 1		
unchanged	0 0 1 1   1 1 1 1	set

# XOR Instruction

- Performs a bitwise Boolean exclusive-OR operation between each pair of matching bits in two operands
- **Syntax:**
  - *XOR destination, source*
- **Flags:**

CF=0, OF =0, sign, parity and zero are modified according to the value assigned to destination operand

**Note:**

XORing any number with itself clears the value.

```
xor al,al ; al = 00000000b
```

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

- Application:

XOR is a useful way to invert the bits in an operand and data encryption. XOR reverses itself when applied twice to the same operand.

### Example:

```
mov dl, 00111011b  
xor dl, 00001111b
```

$$\begin{array}{r} 00111011 \\ \text{XOR } 00001111 \\ \hline \end{array}$$

unchanged

0	0	1	1	0	1	0
---	---	---	---	---	---	---

inverted

# Parity checking:

- Check the parity of 8-bit operand

```
mov al, 10110101b      ; 5 bits = odd parity  
xor al, 0                ; Parity flag clear (odd)  
mov al, 11001100b      ; 4 bits = even parity  
xor al, 0                ; Parity flag set (even)
```

- 16-bit parity

```
mov ax, 64C1h          ; ax = 0110 0100 1100 0001b  
xor ah, al              ; Parity flag set (even)
```

## Applications(1/4)

Task: Convert the character in AL to upper case.

Solution:

Use the AND instruction to clear bit 5.

```
mov al,'a'          ; AL = 01100001b  
and al,11011111b  ; AL = 01000001b
```

- In this example all characters in an array are converted to uppercase:

```
.data  
array BYTE 50 DUP(?)  
.code  
mov ecx, LENGTHOF array  
mov esi, OFFSET array  
L1:  
and BYTE PTR [esi],11011111b      ; clear bit 5  
inc esi  
loop L1
```

- In this example we have used indirect operands for accessing elements of array. Note the use of PTR operator

- In this example all characters in an array are converted to lowercase:

```
.data  
array BYTE 50 DUP(?)  
.code  
mov ecx, LENGTHOF array  
mov esi, OFFSET array  
L1:  
or BYTE PTR [esi], 00100000b      ; set bit 5  
inc esi  
loop L1
```

- In this example we have used indirect operands for accessing elements of array. Note the use of PTR operator

## Applications(2/4)

Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.

Solution:

Use the OR instruction to set bits 4 and 5.

```
mov al,6           ; AL = 00000110b  
or  al, 00110000b ; AL = 00110110b
```

The ASCII digit '6' = 00110110b

## Applications(3/4)

Task: Jump to a label if an integer is even.

Solution:

AND the lowest bit with a 1. If the result is Zero, the number was even.

```
mov ax, wordVal  
and ax, 1           ; low bit set? (here 1 means 0000000000000001b)  
jz EvenValue       ; jump if Zero flag set
```

## Applications(4/4)

Task: Jump to a label if the value in AL is not zero.

Solution:

OR the byte with itself, then use the JNZ (jump if not zero) instruction.

```
or al, al  
jnz IsNotZero ; jump if not zero
```

# Test Instruction

- Performs a nondestructive **AND** operation between each pair of matching bits in two operands.
- No operands are modified, but the flags are affected
- Example: jump to a label if either bit 0 or bit 1 in AL is set
  - `test al, 00000011b`
  - `jnz ValueFound`
- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.
  - `test al, 00000011b`
  - `jz ValueNotFound`

# Test Instruction

- **Flags:**

CF=0, OF =0, sign, parity and zero are modified according to the value assigned to destination operand

# CMP Instruction

- The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand.
- Neither operand is modified.

- **Syntax:**

CMP destination, source

- **Flags:**

Overflow, carry, sign, parity and zero flags are modified according to the value the destination operand would have had if actual subtraction had taken place

# Example:(unsigned)

- Example: destination == source

```
mov al,5  
cmp al,5          ; Zero flag set
```

- Example: destination < source

```
mov al,4  
cmp al,5          ; Carry flag set
```

- Example: destination > source

```
mov al,6  
cmp al,5          ; ZF = 0, CF = 0
```

(both the Zero and Carry flags are clear)

The comparisons shown so far were unsigned.

## Example:(signed)

The comparisons shown here are performed with signed integers.

- Example: destination > source

```
mov al,5  
cmp al,-2      ; Sign flag == Overflow flag
```

- Example: destination < source

```
mov al,-1  
cmp al,5      ; Sign flag != Overflow flag
```

# Conditions:

unsigned	ZF	CF
destination < source	0	1
destination > source	0	0
destination = source	1	0

signed	flags
destination < source	SF != OF
destination > source	SF == OF
destination = source	ZF=1

# Setting and Clearing individual flag:

```
and al, 0          ; set Zero
or  al, 1          ; clear Zero
or  al, 80h        ; set Sign
and al, 7Fh        ; clear Sign
stc               ; set Carry
clc               ; clear Carry

mov al, 7Fh
inc al            ; set Overflow

or eax, 0         ; clear Overflow
```

# Important points to remember

- The result of the operation of AND, OR, NOT and XOR is stored in the destination, which must be a register or memory location.
- The source may be a constant, register, or memory location. However, memory-to-memory operations are not allowed.

# Masking

- Selectively modify the bits in the destination.
- For the mask bits, consider following properties:
- if  $b$  represents a bit (0 or 1)

$$b \text{ AND } 1 = b \quad b \text{ OR } 0 = b \quad b \text{ XOR } 0 = b$$

$$b \text{ AND } 0 = 0 \quad b \text{ OR } 1 = 1 \quad b \text{ XOR } 1 = \sim b \text{ (complement of } b\text{)}$$

- The AND instruction can be used to **clear** specific destination bits while preserving the others. A 0 mask bit clears the corresponding destination bit; a 1 mask bit preserves the corresponding destination bit.
- The OR instruction can be used to **set** specific destination bits while preserving the others. A 1 mask bit sets the corresponding destination bit; a 0 mask bit preserves the corresponding destination bit.
- The XOR instruction can be used to **complement** specific destination bits while preserving the others. A 1 mask bit complements the corresponding destination bit; a 0 mask bit preserves the corresponding destination bit.

# Conditional Structures

- There are no high-level logic structures such as if-then-else, in the IA-32 instruction set. But, you can use combinations of comparisons and jumps to implement any logic structure.
- First, an operation such as **CMP**, **AND** or **SUB** is executed to modified the CPU flags. Second, a conditional jump instruction tests the flags and changes the execution flow accordingly.

**CMP AL, 0**

**JZ L1 :**

# Jcond Instruction

- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Syntax:  
**Jcond *destination***
- Flags:  
**None is affected.**

Four groups: (some are the same)

- based on specific flag values
- based on equality between operands or value of (E)CX
- based on comparisons of unsigned operands
- based on comparisons of signed operands

# Jump Based on Specific Flags

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

# Jump Based on Equality

Mnemonic	Description
JE	Jump if equal ( $leftOp = rightOp$ )
JNE	Jump if not equal ( $leftOp \neq rightOp$ )
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

# Jumps Based on unsigned comparisons

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$ )
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if $leftOp \geq rightOp$ )
JNB	Jump if not below (same as JAE)
JB	Jump if below (if $leftOp < rightOp$ )
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if $leftOp \leq rightOp$ )
JNA	Jump if not above (same as JBE)

# Jumps Based on signed comparisons

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$ )
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightOp$ )
JNL	Jump if not less (same as JGE)
JL	Jump if less (if $leftOp < rightOp$ )
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if $leftOp \leq rightOp$ )
JNG	Jump if not greater (same as JLE)

# Tables

Signed Jumps		
Symbol	Description	Condition for Jumps
JG/JNLE	jump if greater than jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	jump if greater than or equal to jump if not less than or equal to	SF = OF
JL/JNGE	jump if less than jump if not greater than or equal	SF <> OF
JLE/JNG	jump if less than or equal jump if not greater than	ZF = 1 or SF <> OF

JA/JNBE	jump if above jump if not below or equal	CF = 0 and ZF = 0
JAE/JNB	jump if above or equal jump if not below	CF = 0
JB/JNAE	jump if below jump if not above or equal	CF = 1
JBE/JNA	jump if equal jump if not above	CF = 1 or ZF = 1

Symbol	Description	Condition for Jumps
JE/JZ	jump if equal	ZF = 1
JNE/JNZ	jump if equal to zero jump if not equal	ZF = 0
JC	jump if carry	CF = 1
JNC	jump if no carry	CF = 0
JO	jump if overflow	OF = 1
JNO	jump if no overflow	OF = 0
JS	jump if sign negative	SF = 1
JNS	jump if nonnegative sign	SF = 0
JP/JPE	jump if parity even	PF = 1
JNP/JPO	jump if parity odd	PF = 0

## Examples:

- Compare unsigned AX to BX, and copy the larger of the two into a variable named large

```
mov Large,bx  
cmp ax,bx  
jna Next  
mov Large,ax  
Next:
```

- Compare signed AX to BX, and copy the smaller of the two into a variable named small

```
mov Small,ax  
cmp bx,ax  
jnl Next  
mov Small,bx  
Next:
```

## Exercise:

- Find the first even number in an array of unsigned integers

```
.data  
intArray DWORD 7,9,3,4,6,1  
.code  
...  
        mov    ebx, OFFSET intArray  
        mov    ecx, LENGTHOF intArray  
L1:     test   DWORD PTR [ebx], 1  
        jz    found  
        add    ebx, 4  
        loop  L1  
...
```

# BT (Bit Test) Instruction

- Copies bit  $n$  from an operand into the Carry flag
- Syntax:

**BT *bitBase*, *n***

- *bitBase* may be *r/m16* or *r/m32*
- *n* may be *r16*, *r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

```
bt AX, 9           ; CF = bit 9
jc L1             ; jump if Carry
```

- **BTC** *bitBase, n*: bit test and complement
- **BTR** *bitBase, n*: bit test and reset (clear)
- **BTS** *bitBase, n*: bit test and set

**BTC** copies bit n to the Carry flag and complements bit n in the destination operand. **BTR** copies bit n to the Carry flag and clears bit n in the destination. **BTS** copies bit n to the Carry flag and sets bit n in the destination.

# Conditional Loops

# LOOPZ and LOOPE

- Syntax:
  - **LOOPE destination**
  - **LOOPZ destination**
- Logic:
  - $ECX \leftarrow ECX - 1$
  - if  $ECX \neq 0$  and  $ZF=1$ , jump to *destination*
- The destination label must be between -128 and +127 bytes from the location of the following instruction
- Useful when scanning an array for the first element that meets some condition.

# LOOPNZ and LOOPNE

- Syntax:

**LOOPNZ *destination***

**LOOPNE *destination***

- Logic:

- $\text{ECX} \leftarrow \text{ECX} - 1;$
- if  $\text{ECX} \neq 0$  and  $\text{ZF}=0$ , jump to *destination*

## Example 1:

- Write assembly code that finds the first positive value in an array.

# Example 1:

- Write assembly code that finds the first positive value in an array:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h      ; test sign bit
    pushfd                         ; push flags on stack
    add esi,TYPE array
    popfd                           ; pop flags from stack
    loopnz next                     ; continue loop
    jnz quit                         ; none found
    sub esi,TYPE array              ; ESI points to value
quit:
```

If no value is found then after adding 2 to the ESI it points to sentinel, so we need to skip the next line

ESI was incremented due to which it points to the next value rather than the desired value after the loop terminates. In this step we are subtracting its value by 2 so that it points to the first non negative value in array.

In this code the ESI points to the desired value, There are 2 situations:

1. If positive value is found it points to the desired value.
2. If no value is found it points to the sentinel

## Example 2:

- Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0      ; check for zero
quit:
```

# Solution:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh

.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1:cmp WORD PTR [esi],0 ; check for zero
    pushfd                ; push flags on stack
    add esi,TYPE array
    popfd                 ; pop flags from stack
    loope L1               ; continue loop
    jz quit                ; none found
    sub esi,TYPE array     ; ESI points to value
quit:
```

# Conditional Structures

# IF-THEN

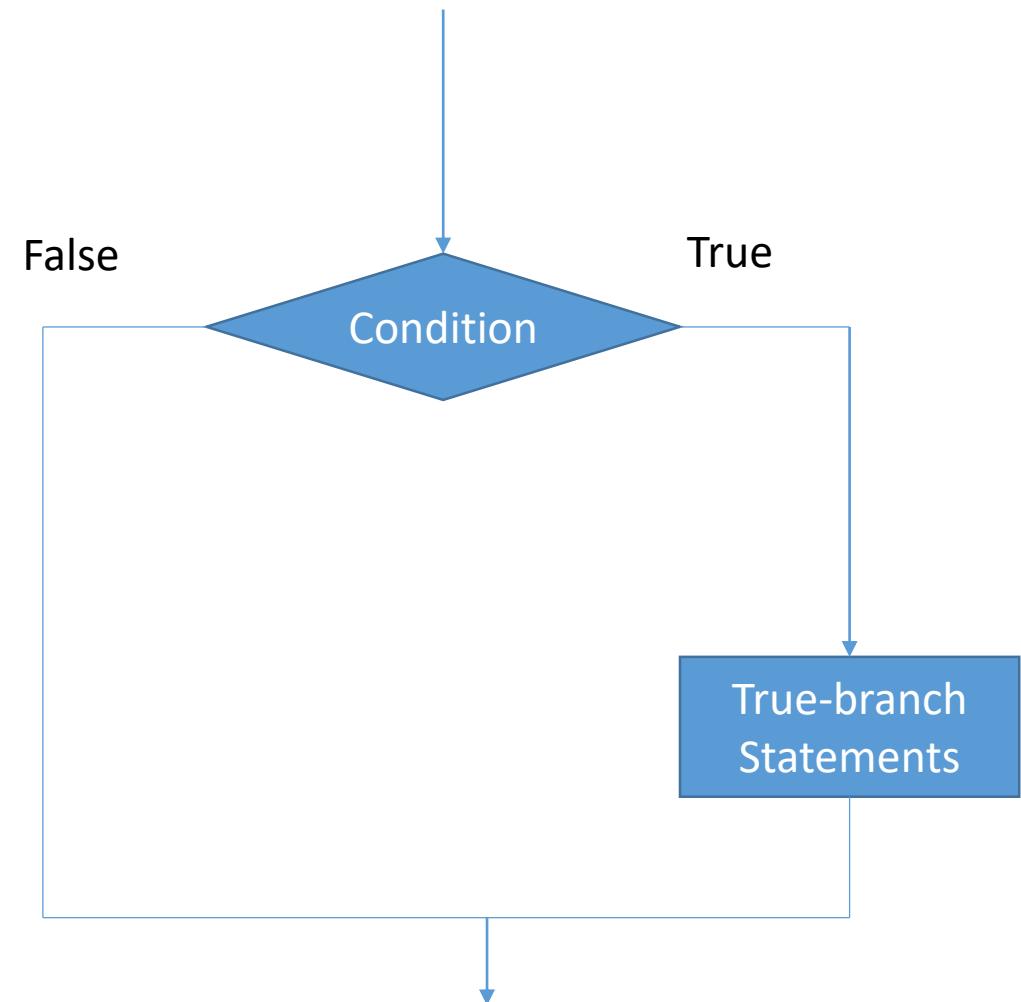
Pseudocode:

IF condition is true

THEN

execute true-branch statements

END\_IF



## Example:

Replace the number in AX by its absolute value.

```
; if AX < 0  
CMP AX, 0  
JNL END_IF  
;then  
NEG AX  
END_IF:
```

# IF-THEN-ELSE

Pseudocode:

IF condition is true

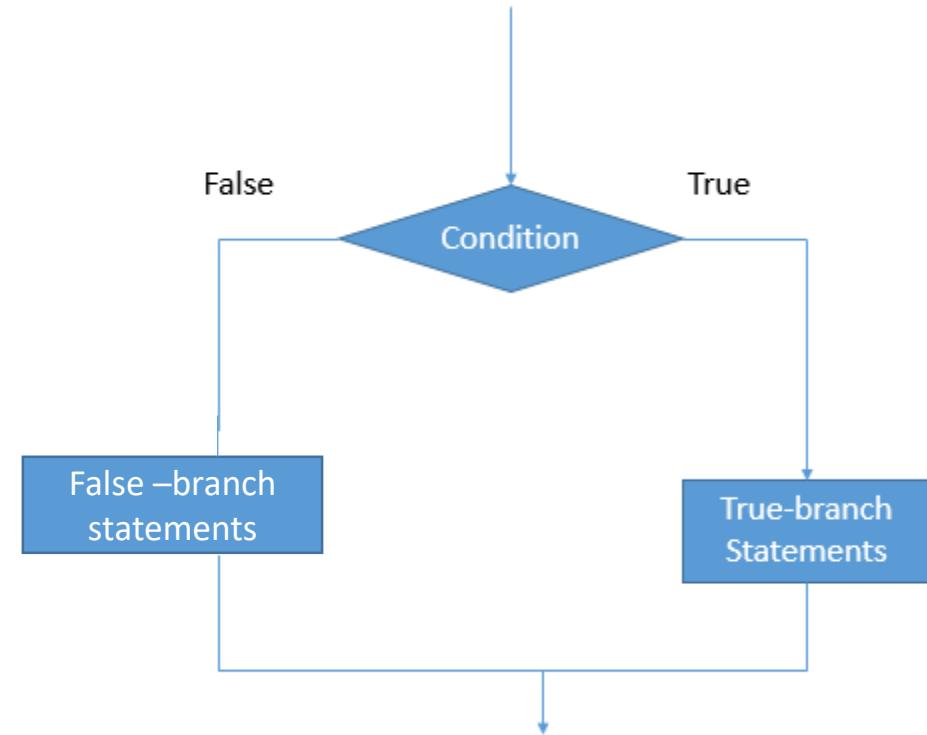
THEN

execute true-branch statements

ELSE

execute false-branch statements

END\_IF



# IF-THEN-ELSE

if **C** then **T** else **E**

**C**

**JNE else**

**T**

**JMP endif**

**else:**

**E**

**endif:**

# Block-Structured IF Statements

- Assembly language programmers can easily translate logical statements written in C++/Java into assembly language.

## Example 1:

```
if( op1 == op2 )  
    x = 1;  
else  
    x = 2;
```

```
mov eax,op1  
cmp eax,op2  
jne L1  
mov x,1  
jmp L2  
L1: mov x,2  
L2:
```

## Example 2:

- Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
cmp ebx,ecx  
ja next  
mov eax,5  
mov edx,6  
next:
```

## Example 3:

- Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )  
    var3 = 10;  
else  
{  
    var3 = 6;  
    var4 = 7;  
}
```

```
mov eax,var1  
cmp eax,var2  
jle L1  
mov var3,6  
mov var4,7  
jmp L2  
L1: mov var3,10  
L2:
```

# Compound Expression with AND

When implementing the logical AND operator, consider that HLLs use short-circuit evaluation

In the following example, if the first expression is false, the second expression is skipped:

```
if (a1 > b1) AND (b1 > c1)  
    x = 1;
```

## One possible solution:

```
if (al > bl) AND (bl > cl)
    x = 1;
```

```

        .
        .
        .

    cmp al,bl           ; first expression...
    ja L1
    jmp next

L1:
    cmp bl,cl           ; second expression...
    ja L2
    jmp next

L2:                      ; both are true
    mov x,1             ; set X to 1

next:
```

## Efficient Solution:

```
if (al > bl) AND (bl > cl)
    x = 1;
```

- But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
cmp al,bl           ; first expression...
jbe next            ; quit if false
cmp bl,cl           ; second expression...
jbe next            ; quit if false
mov x,1              ; both are true
next:
```

## Example:

- Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx  
  && ecx > edx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
cmp ebx,ecx  
ja next  
cmp ecx,edx  
jbe next  
mov eax,5  
mov edx,6  
next:
```

# Compound Expression with OR

- In the following example, if the first expression is true, the second expression is skipped:

```
if (al > bl) || (bl > cl)
    x = 1;
```

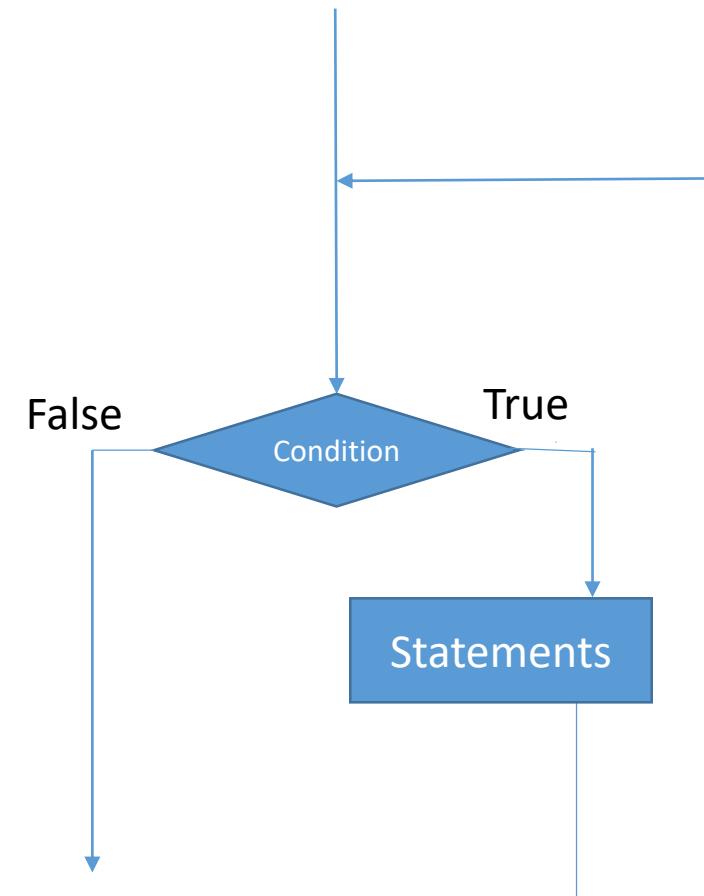
- We can use "fall-through" logic to keep the code as short as possible:

```
cmp al,bl      ; is AL > BL?
ja L1          ; yes
cmp bl,cl      ; no: is BL > CL?
jbe next       ; no: skip next statement
L1:mov X,1      ; set X to 1
next:
```

# While Loop

- Pseudocode

```
WHILE condition DO  
    statements  
END WHILE
```



# While Loops

- A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx)
    eax = eax + 1;
```

```
_while:
    cmp eax,ebx      ; check loop condition
    jae _endwhile   ; false? exit loop
    inc eax         ; body of loop
    jmp _while      ; repeat the loop
_endwhile:
```

## Example :

- Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
_while:
    cmp ebx,val1      ; check loop condition
    ja _endwhile      ; false? exit loop
    add ebx,5         ; body of loop
    dec val1
    jmp while         ; repeat the loop
_endwhile:
```

## Exercise:

Write the assembly code for given pseudocode

```
while(eax < ebx)
{
    eax++;
    if (ebx==ecx)
        x=2;
    else
        x=3;
}
```

```
_while:  cmp    eax,  ebx
          jae   _endwhile
          inc    eax
          cmp    ebx,  ecx
          jne   _else
          mov    x,  2
          jmp   _while
_else:   mov    x,  3
          jmp   _while
_endwhile:
```

# REPEAT UNTIL

- Pseudocode:

```
REPEAT  
statements  
UNTIL condition
```

# CASE

Pseuocode:

CASE expression

values\_1: statements\_1

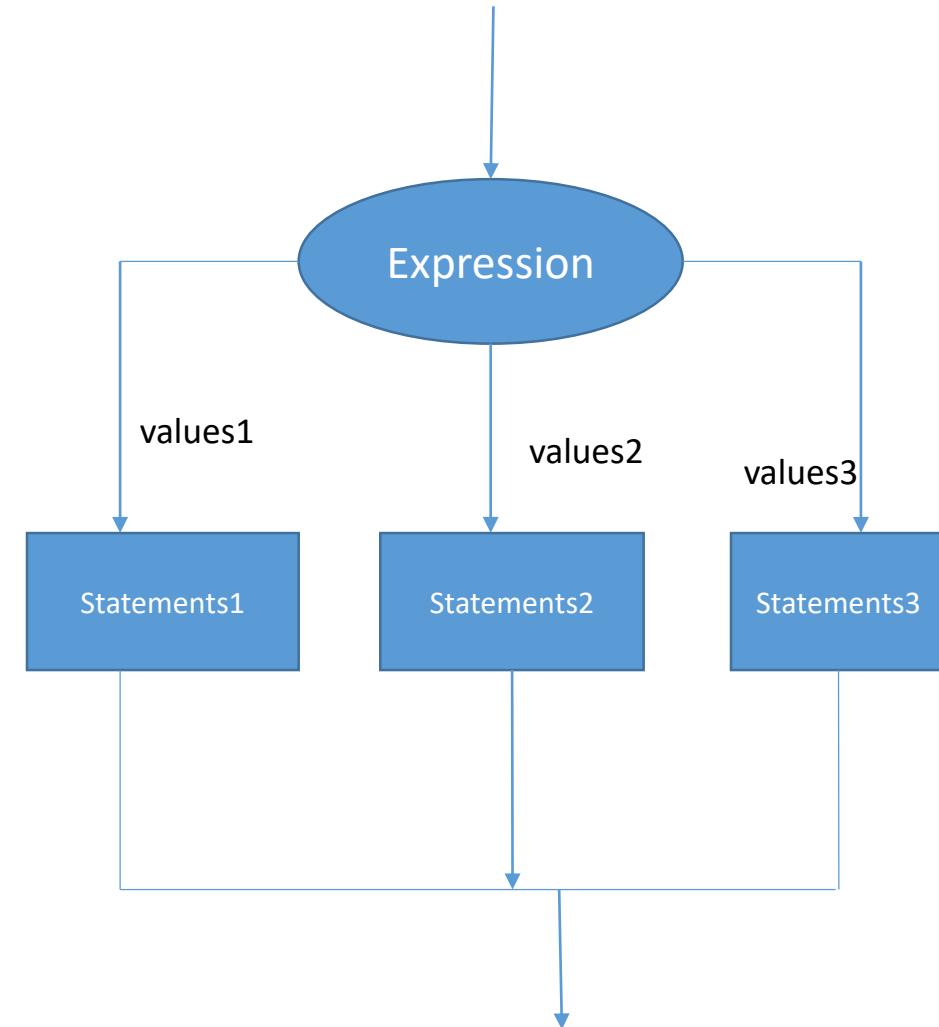
Values\_2: statements\_2

.

.

values\_n: statements\_n

END\_CASE



# Example

- If AX contains a negative number, put -1 in BX; if AX contains 0, put 0 in BX; if AX contains a positive number, put 1 in BX.

```
CMP AX,0
JL NEGATIVE
JE ZERO
JG POSITIVE
NEGATIVE:
MOV BX,-1
JMP END_CASE
ZERO:
MOV BX,0
JMP END_CASE
POSITIVE:
MOV BX, 1
END_CASE:
```

# Do While

- A DO loop is really an IF statement, here the body of the loop followed by an IF statement to unconditional jump

# Example:

```
Do  
{  
eax = eax + 1;  
}while( eax < ebx)
```

```
Do:  
Inc eax  
CMP eax, ebx  
JAE end_do  
Jmp Do  
End_do:
```

## Exercise 1:

- Write some code for given nested if statements:

```
if op1 == op2  
if X > Y  
call Routine1  
else  
call Routine2  
end if  
else  
call Routine3  
end if
```

# One possible solution:

```
mov eax,op1  
cmp eax,op2          ; op1 == op2?  
jne L2              ; no: call Routine3  
; process the inner IF-ELSE statement  
mov ebx,X  
cmp ebx,Y          ; X > Y?  
jg L1              ; yes: call Routine1  
call Routine2        ; no: call Routine2  
jmp L3              ; and exit  
L1: call Routine1    ; call Routine1  
jmp L3              ; and exit  
L2: call Routine3  
L3:
```

## Adapted from:

1. Intel x86 Instruction Set Architecture, Computer Organization and Assembly Languages Yung-Yu Chuang
2. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
3. Assembly Language Programming and organization of the IBM PC by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 10 part a

Chapter 7 “Integer Arithmetic”

# Outline

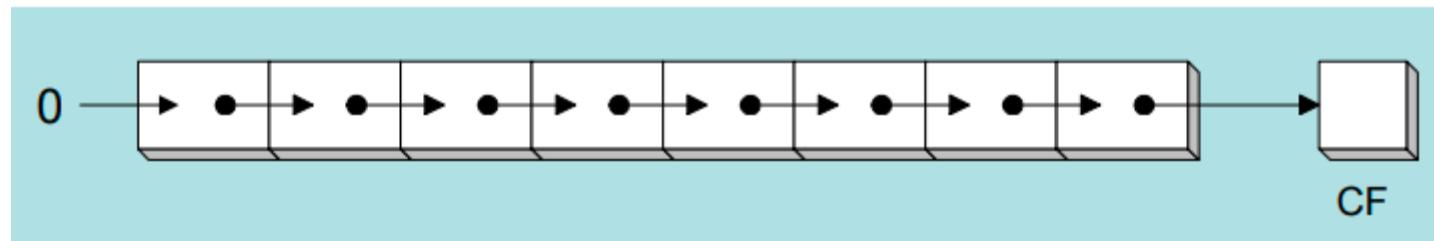
- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction

# Shift and Rotate Instructions

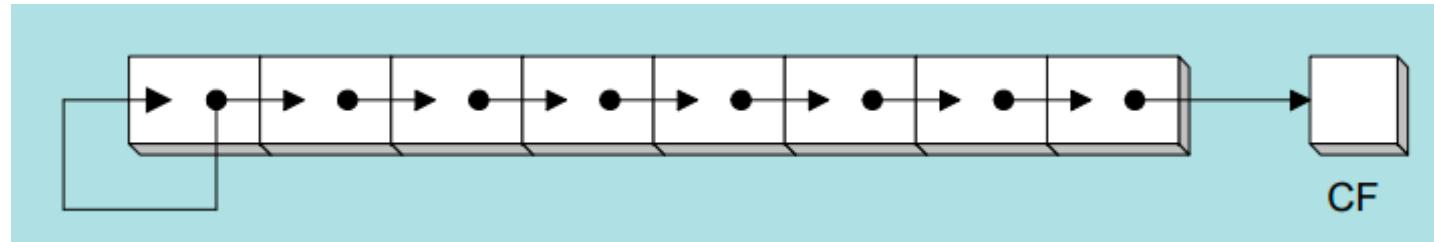
- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

# Logical vs Arithmetic Shifts

- A logical shift fills the newly created bit position with zero:

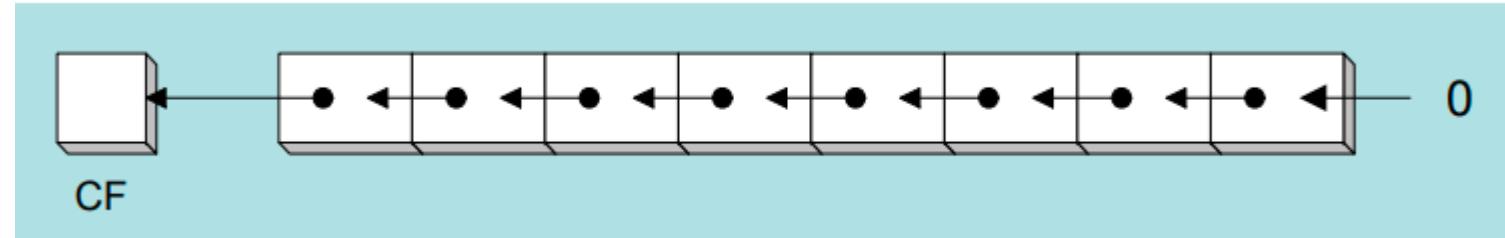


- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



# SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.
- The MSB is moved to CF.



- Operand Types:  
**SHL destination, count**

- *SHL reg, imm8*
- *SHL mem, imm8*
- *SHL reg, CL*
- *SHL mem, CL*

Imm8 must be between 0-255

## Effect on flags

- SF, PF, ZF reflect the result
- AF is undefined
- CF= last bit shifted out
- OF= 1 if result changes sign on last shift

# Fast Multiplication/ Bitwise Multiplication

- Shifting left 1 bit multiplies a number by 2

```
mov dl,5  
shl dl,1
```

Before: 

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 5

After: 

0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---

 = 10

- Shifting left  $n$  bits multiplies the operand by  $2^n$ .

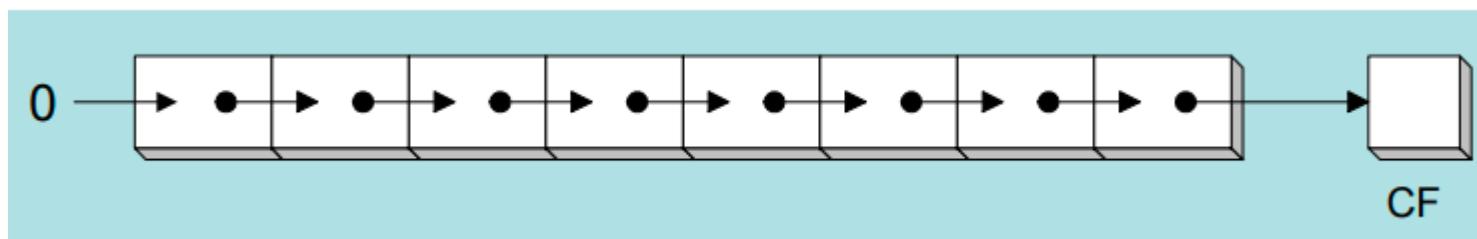
For example,  $5 * 2^2 = 20$

```
mov dl,5  
shl dl,2
```

; DL = 00000101b  
; DL = 20 or DL = 00010100b

# SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand.
- The highest bit position is filled with a zero and LSB is copied into CF.



# Bitwise Division

- Shifting right n bits divides the operand by  $2^n$ .

Example:

```
mov dl, 80
```

```
shr dl, 1      ; DL = 40
```

```
shr dl, 2      ; divide by 4, DL = 10
```

## Exercise:

Suppose DH contains 8Ah and CL contains 2 . What are the values of DH and CF after the Instruction SHR DH,CL is executed?

## Solution:

The value of DH in binary is 10001010.

After two right shifts, CF = 1.

The new value of DL is obtained by erasing the rightmost two bits and adding two 0 bits to the left end, thus DH =00100010b = 22h.

## SAL instructions

- SAL (shift arithmetic left) is identical to SHL.

## Exercise:

Write some code to multiply the value of AX by 8. Assume that overflow will not occur.

## Solution:

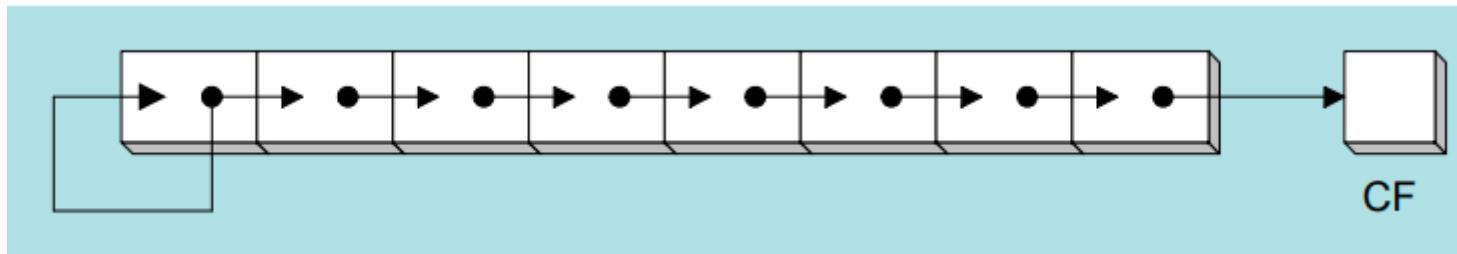
To multiply by 8, we need to do three left shifts.

MOV CL, 3 ;number of shifts to do

SAL AX,CL ;multiply by 8

# SAR instructions

- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.
- An arithmetic shift preserves the numbers sign



Example:

mov dl, -80

sar dl, 1 ; DL = -40

sar dl, 2 ; DL = -10

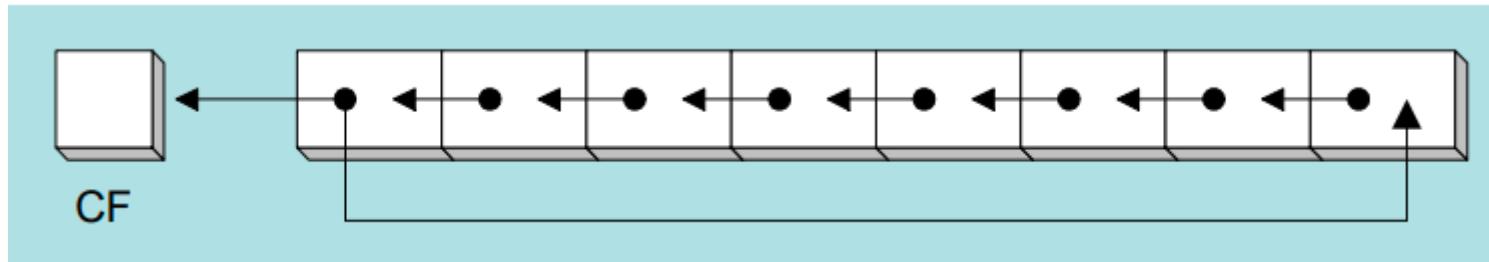
# Signed Division

You can divide a signed operand by a power of 2, using the SAR instruction. Consider the following example:

```
mov dl, -128 ; DL = 10000000b  
sar dl, 3      ; DL = 11110000b
```

# ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost



- Example:

```
    mov al,11110000b  
    rol al,1           ; AL = 11100001b
```

## Multiple Rotations:

- When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the MSB position.

```
mov al, 00100000b  
rol  al, 3           ; CF = 1, AL = 00000001b
```

# Exchanging Groups of Bits:

- You can use ROL to exchange the upper (bits 4–7) and lower (bits 0–3) halves of a byte.

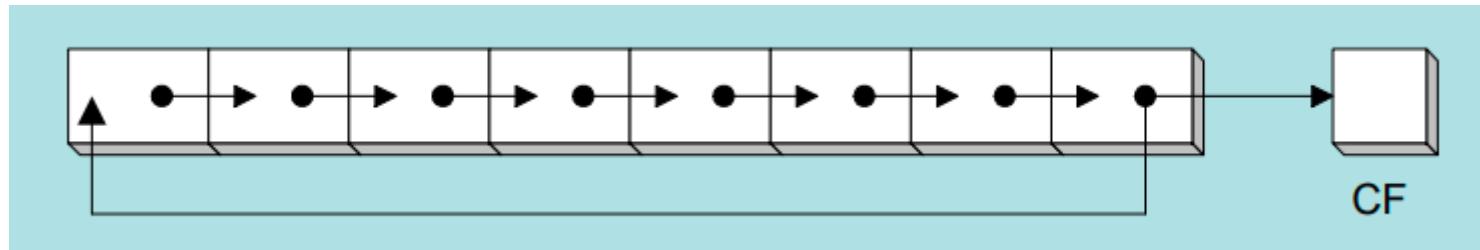
```
mov dl, 3Fh  
rol dl, 4          ; DL = F3h
```

- When rotating a multibyte integer by four bits, the effect is to rotate each hexadecimal digit one position to the right or left. Here, for example, we repeatedly rotate 6A4Bh left four bits, eventually ending up with the original value:

```
mov ax, 6A4Bh  
rol ax, 4          ; AX = A4B6h  
rol ax, 4          ; AX = 4B6Ah  
rol ax, 4          ; AX = B6A4h  
rol ax, 4          ; AX = 6A4Bh
```

# ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit position
- No bits are lost



## Example:

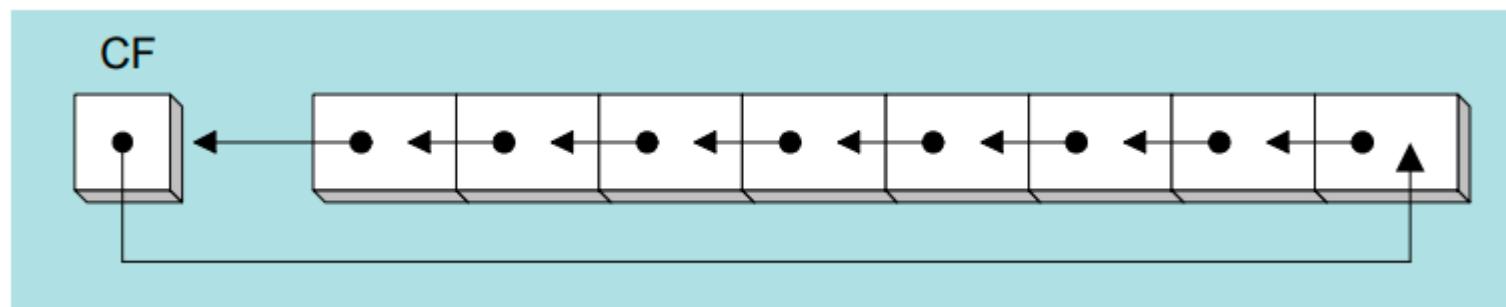
---

```
mov al,11110000b
ror al,1           ; AL = 01111000b

mov dl,3Fh
ror dl,4          ; DL = F3h
```

# RCL Instruction

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit (LSB)
- Copies the most significant bit (MSB) to the Carry flag

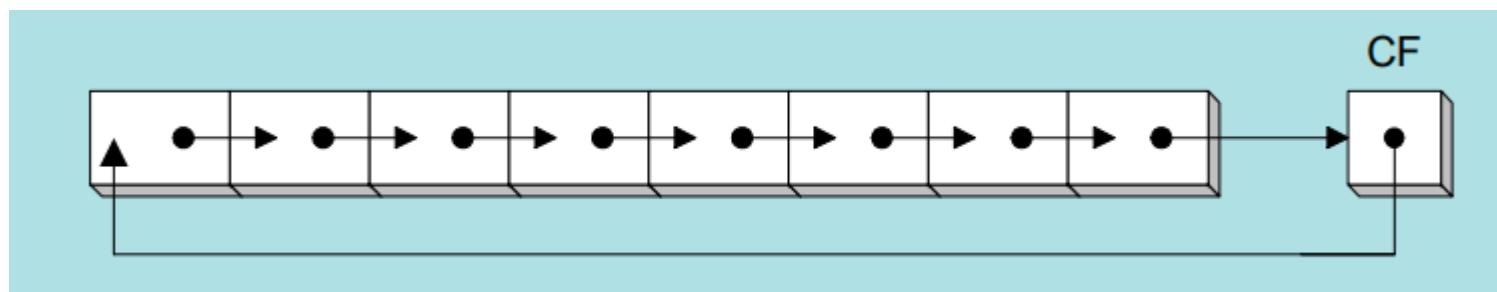


## Example:

```
clc          ; CF = 0
mov bl,88h    ; CF,BL = 0 10001000b
rcl bl,1      ; CF,BL = 1 00010000b
rcl bl,1      ; CF,BL = 0 00100001b
```

# RCR Instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit (MSB)
- Copies the least significant bit (LSB) to the Carry flag



## Example:

**stc**

**mov ah,10h**

**rcr ah,1**

; CF = 1

; ;

= 00010000 1  
= 10001000 0

AH

CF

# SHLD Instruction

- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected

# SHLD Instruction

- Syntax: (shift left double)  
**SHLD *destination, source, count***

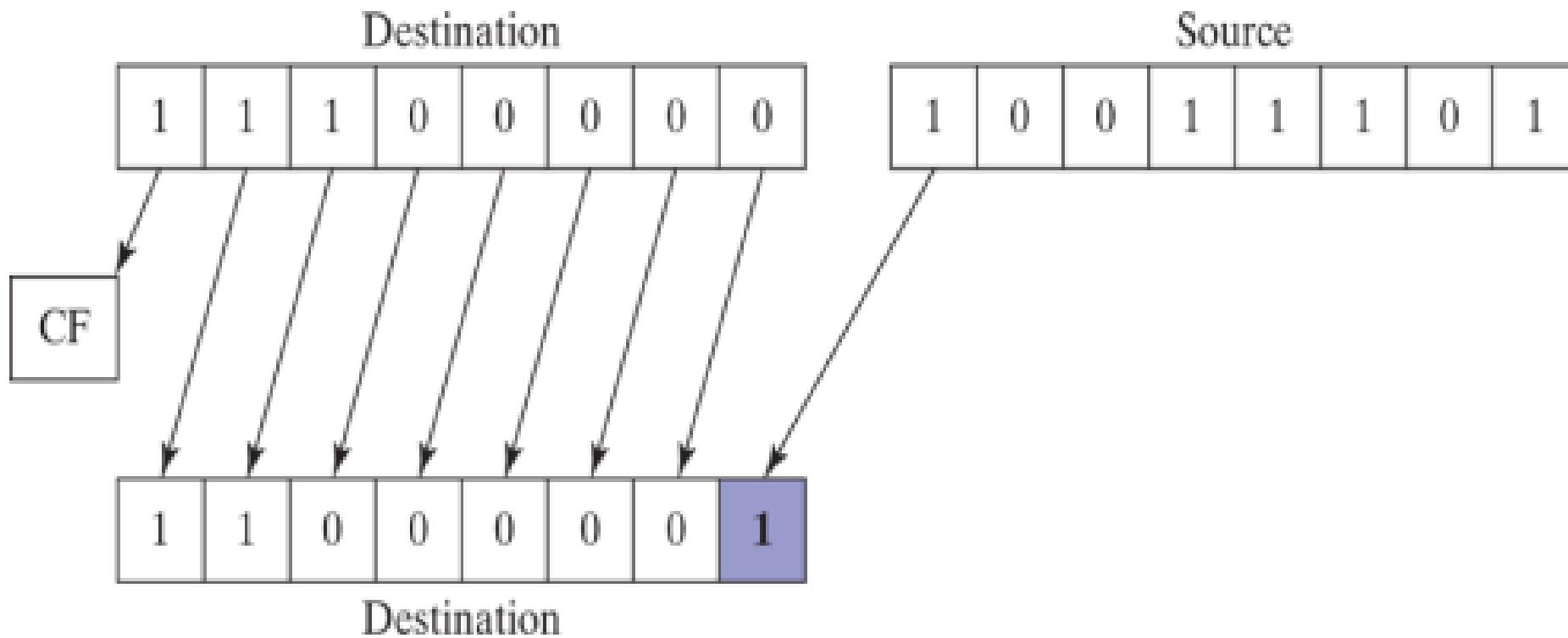
SHLD reg16, reg16, CL/imm8

SHLD mem16, reg16, CL/imm8

SHLD reg32, reg32, CL/imm8

SHLD mem32, reg32, CL/imm8

# SHLD Instruction

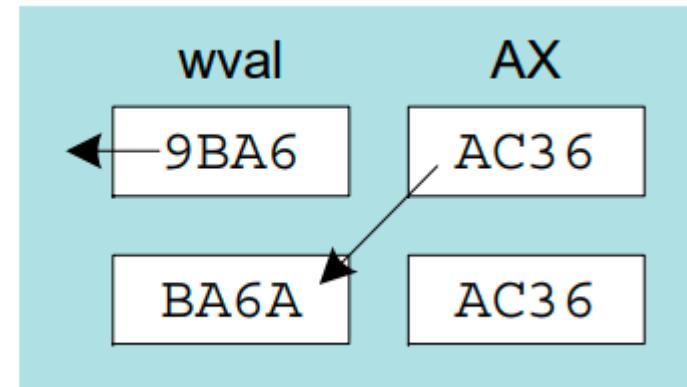


## Example:

- Shift **wval** 4 bits to the left and replace its lowest 4 bits with the high 4 bits of **AX**:

```
.data  
wval WORD 9BA6h  
.code  
mov ax,0AC36h  
shld wval,ax,4
```

Before:

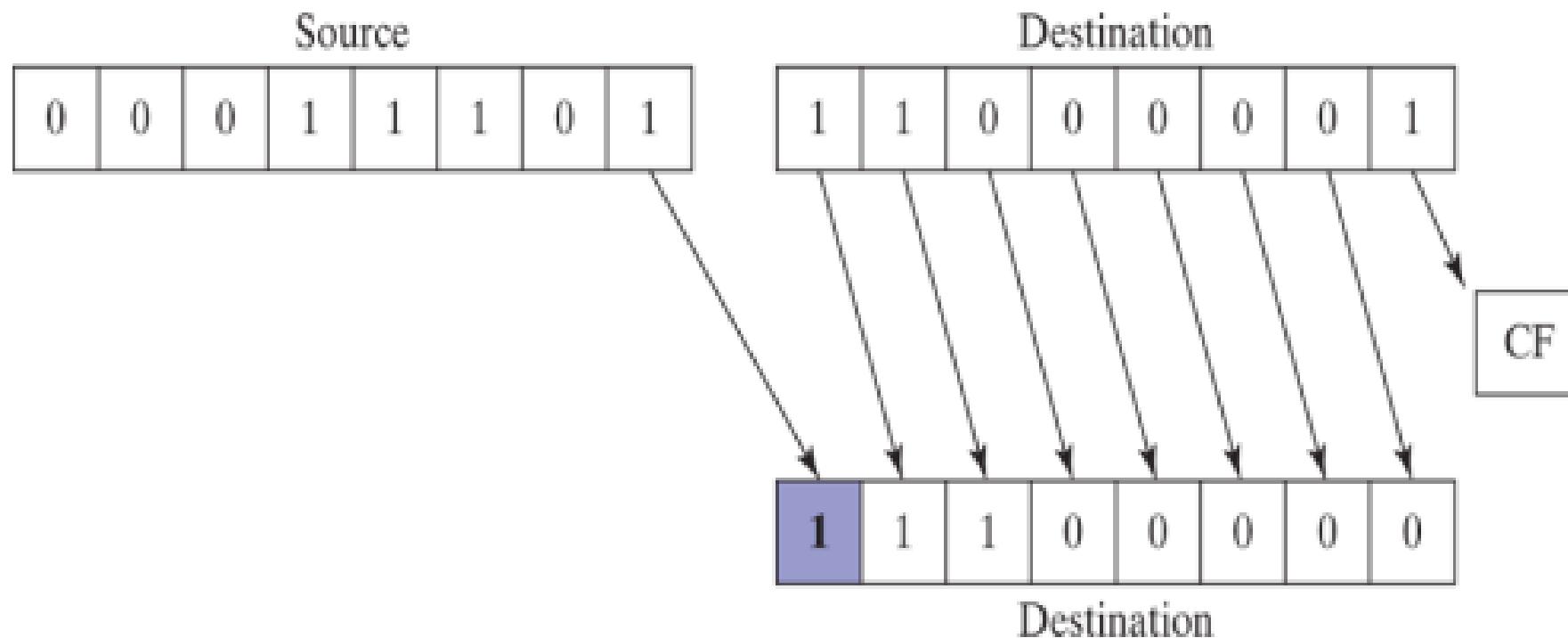


After:

# SHRD Instruction

- Shifts a destination operand a given number of bits to the right
- The bit positions opened up by the shift are filled by the least significant bits of the source operand
- The source operand is not affected
- Syntax:  
**SHRD *destination, source, count***

# SHRD Instruction

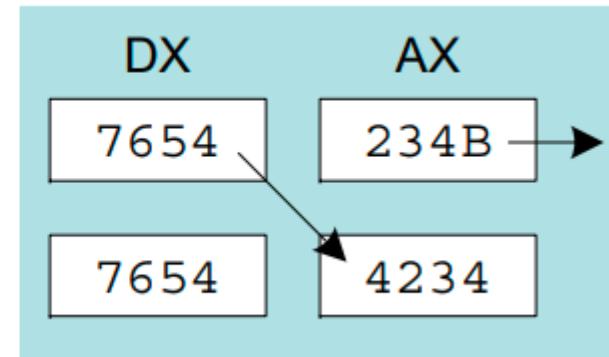


# SHRD Example:

- Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov ax, 234Bh  
mov dx, 7654h  
shrd ax, dx, 4
```

Before:  
After:



## Example:

Demonstrates SHRD by shifting an array of doublewords to the right by 4 bits:

# Applications:

- SHLD and SHRD can be used to manipulate bit-mapped images, when groups of bits must be shifted left and right to reposition images on the screen.
- Another potential application is data encryption, in which the encryption algorithm involves the shifting of bits.
- Finally, the two instructions can be used when performing fast multiplication and division with very long integers.

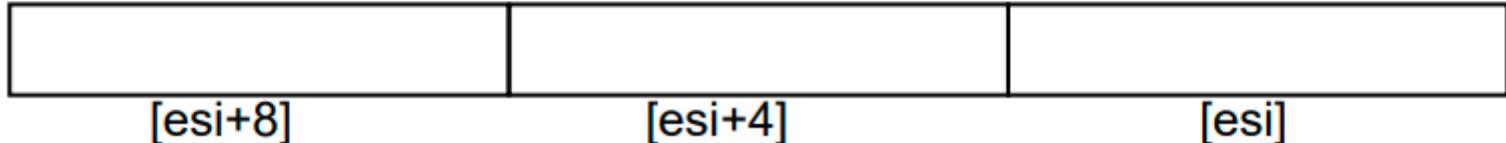
# Shift and Rotate Applications

- Shifting Multiple Doublewords
- Binary Multiplication
- Displaying in Binary Bits
- Isolating a Bit String

# Shifting multiple doublewords

- Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.
- The following shifts an array of 3 doublewords 1 bit to the right:

```
shr array[esi + 8],1 ; high dword  
rcr array[esi + 4],1 ; middle dword,  
rcr array[esi],1      ; low dword,
```



# Binary Multiplication

- We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
- Factor any binary number into powers of 2.
  - For example, to multiply EAX \* 36, factor 36 into  $2^5 + 2^2$  and use the distributive property of multiplication to carry out the operation:

```
EAX * 36  
= EAX * (32 + 4)  
= (EAX * 32) + (EAX * 4)
```

```
mov eax,123  
mov ebx,eax  
shl eax,5  
shl ebx,2  
add eax,ebx
```

$$\begin{array}{r} 01111011 & 123 \\ \times & 00100100 & 36 \\ \hline 01111011 & 123 \text{ SHL } 2 \\ + & 01111011 & 123 \text{ SHL } 5 \\ \hline 0001000101001100 & 4428 \end{array}$$

# Displaying Binary Bits

- *Algorithm:* Shift MSB into the Carry flag; If CF = 1, append a "1" character to a string; otherwise, append a "0" character. Repeat in a loop, 32 times.

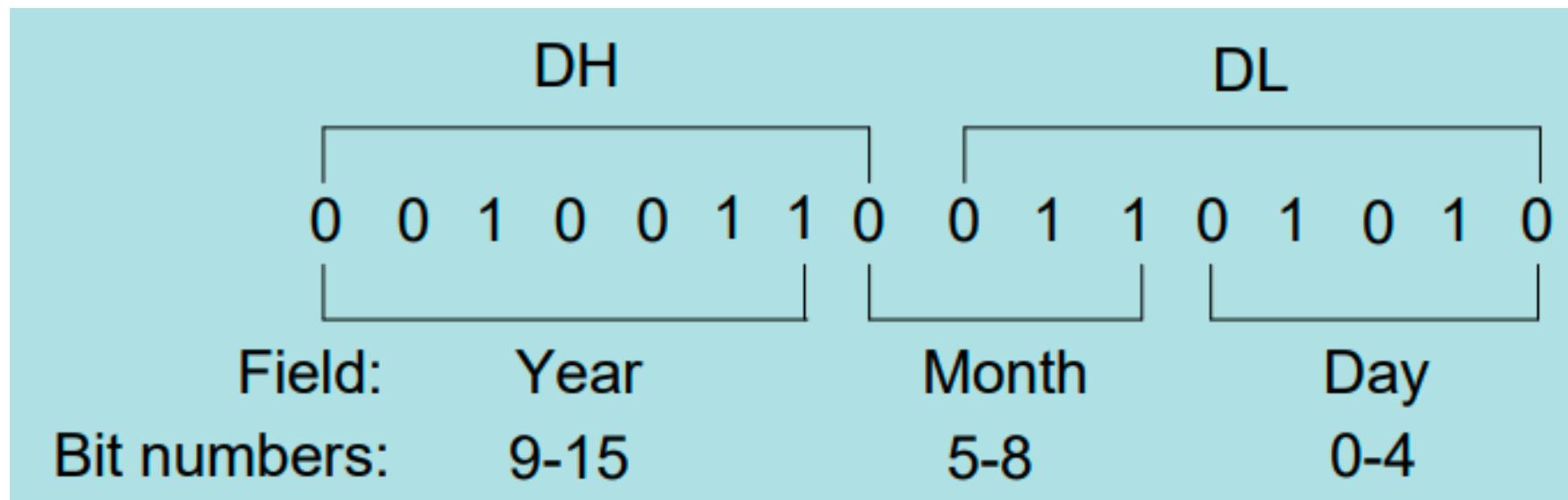
```
        mov ecx,32
        mov esi,offset buffer
L1:  shl eax,1
        mov BYTE PTR [esi], '0'
        jnc L2
        mov BYTE PTR [esi], '1'
L2:  inc esi
        loop L1
```

# Isolating bit string

- To extract a single bit string, shift its bits into the lowest part of a register and clear the irrelevant bit positions.

# Isolating a bit string

- The MS-DOS file date field packs the year (relative to 1980), month, and day into 16 bits:



# Isolating a bit string

The following code example extracts the day number field of a date stamp integer by making a copy of DL and masking off bits not belonging to the field:

```
mov al,d1          ; make a copy of DL
and al,00011111b ; clear bits 5-7
mov day,al        ; save in day variable
```

To extract the month number field, we shift bits 5 through 8 into the low part of AL before masking off all other bits. AL is then copied into a variable:

```
mov ax,dx          ; make a copy of DX
shr ax,5           ; shift right 5 bits
and al,00001111b ; clear bits 4-7
mov month,al       ; save in month variable
```

# Isolating a bit string

The year number (bits 9 through 15) field is completely within the DH register. We copy it to AL and shift right by 1 bit:

```
mov al,dh          ; make a copy of DX
shr al,1          ; shift right 1 bit
mov ah,0          ; clear AH to 0
add ax,1980        ; year is relative to 1980
mov year,ax        ; save in year
```

# Multiplication and Division

# MUL Instruction

- The MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.
- The multiplier and multiplicand must always be the same size, and the product is twice their size.
- The instruction formats are:
  - MUL r/m8**
  - MUL r/m16**
  - MUL r/m32**

Implied Operands

Multiplicand	Multiplier	Product
AL	r/m8	AX
AX	r/m16	DX:AX
EAX	r/m32	EDX:EAX

# MUL Instruction

- Effect on Flags:
- MUL sets the Carry and Overflow flags if the upper half of the product is not equal to zero.
- SF, ZF, AF, PF:                          undefined
- CF/OF:
  - = 0 if the upper half of the result is zero.
  - = 1 otherwise.

## Example 1:

100h \* 2000h, using 16-bit operands:

```
.data  
val1 WORD 2000h  
val2 WORD 100h  
.code  
mov ax, val1  
mul val2 ; DX:AX=00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

## Example 2:

12345h \* 1000h, using 32-bit operands:

```
mov eax,12345h  
mov ebx,1000h  
mul ebx ; EDX:EAX=0000000012345000h, CF=0
```

# IMUL Instruction

- IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX (there are one/two/three operand formats)
- Preserves the sign of the product by sign-extending it into the upper half of the destination register

# Instruction Format

## Single operand:

IMUL r/m8

IMUL r/m16

IMUL r/m32

- The one-operand formats store the product in AX, DX:AX, or EDX:EAX.
- Also, the Carry and Overflow flags are set if the upper half of the product is not a sign extension of the lower half.

## Two operands:

IMUL r16, r/m16

IMUL r16, imm8

IMUL r32, r/m32

IMUL r32, imm8

IMUL r16, imm16

IMUL r32, imm32

- The two-operand version stores the product in the first operand, which must be a register. The second operand (the multiplier) can be a register, a memory operand, or an immediate value.

## Three operands:

IMUL r16, r/m16, imm8

IMUL r16, r/m16, imm16

IMUL r32, r/m32, imm8

IMUL r32, r/m32, imm32

- The three-operand formats store the product in the first operand. The second operand can be a register or memory operand, which is multiplied by the third operand, an 8- or 16-bit or 32-bit immediate value.

## Example:

- multiply  $48 * 4$ , using 8-bit operands:

```
mov al, 48  
mov bl, 4  
imul bl ; AX = 00C0h, OF=1
```

OF=1 because AH is not a sign extension of AL.

- multiply  $48 * 4$ , using 16-bit operands:

```
mov ax, 48  
mov bx, 4  
imul bx ; DX:AX = 000000C0h, OF = 0
```

- In the above instructions DX is a sign extension of AX, so the Overflow flag is clear.

# IMUL Instruction

- Effect on Flags:
- SF, ZF, AF, PF: undefined
- CF/OF:
  - = 0 if the upper half of the result is the sign extension of the lower half (this means that the bits of the upper half are the same as the sign bit of the lower half).
  - = 1 otherwise.

## Example: ( IMUL with two operands)

```
.data  
  
word1 SWORD 4  
  
dword1 SDWORD 4  
  
.code  
  
mov ax ,-16          ; AX = -16  
  
mov bx ,2            ; BX = 2  
  
imul bx ,ax          ; BX = -32  
  
imul bx ,2            ; BX = -64  
  
imul bx, word1        ; BX = -256  
  
mov eax, -16          ; EAX = -16  
  
mov ebx, 2            ; EBX = 2  
  
imul ebx, eax          ; EBX = -32  
  
imul ebx, 2            ; EBX = -64  
  
imul ebx, dword1        ; EBX = -256
```

## Example: (IMUL with three operands)

```
.data  
word1  SWORD  4  
dword1 SDWORD 4  
.code  
imul bx, word1, -16          ; BX = word1 * -16  
imul ebx ,dword1, -16        ; EBX = dword1 * -16  
imul ebx, dword1, -2000000000 ; signed overflow!
```

# Important points to remember

- The two-operand and three-operand formats truncate the product to the length of the destination. If significant digits are lost, the Overflow and Carry flags are set. Be sure to check one of these flags after performing an IMUL operation with two and three operands.
- The two-operand and three-operand IMUL formats may also be used for unsigned multiplication because the lower half of the product is the same for signed and unsigned numbers. There is a small disadvantage to doing so: The Carry and Overflow flags will not indicate whether the upper half of the product equals zero.

# DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
- A single operand is supplied (register or memory operand), which is assumed to be the divisor
- Instruction formats:

**DIV r/m16**

**DIV r/m32**

**DIV r/m8**

Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	<i>r/m8</i>	AL	AH
DX:AX	<i>r/m16</i>	AX	DX
EDX:EAX	<i>r/m32</i>	EAX	EDX

# DIV Instruction

- Effect on Flags:
- All arithmetic status flag values are undefined after executing DIV and IDIV.

# Examples:

- Make sure you clear the upper half of your dividend, if your dividend completely fits in lower half.
- If dividend is big adjust it properly in lower and upper halves

Divide 8003h by 100h, using 16-bit operands:

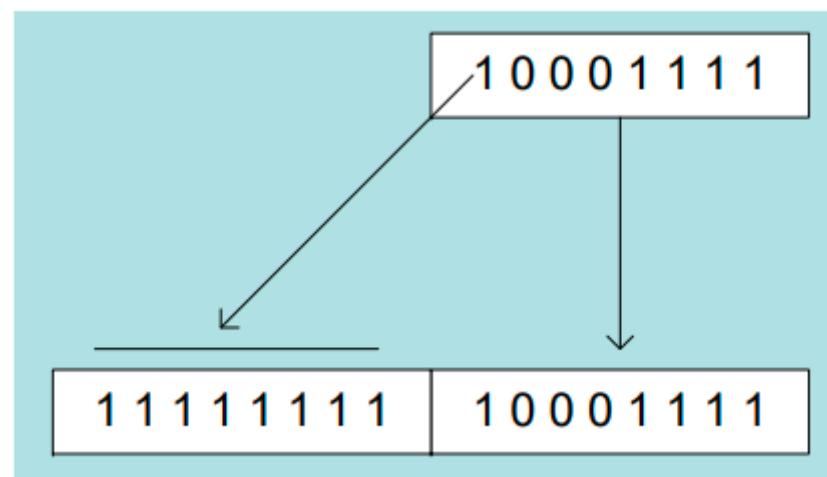
```
mov dx,0          ; clear dividend, high  
mov ax,8003h     ; dividend, low  
mov cx,100h      ; divisor  
div cx           ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

```
mov edx,0         ; clear dividend, high  
mov eax,8003h     ; dividend, low  
mov ecx,100h      ; divisor  
div ecx          ; EAX=00000080h , EDX=3
```

# Signed Integer Division

- Signed integers must be sign-extended before division takes place  
fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit
- For example, the high byte contains a copy of the sign bit from the low byte:



# CBW, CWD, CDQ instructions

- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
  - CBW (convert byte to word) extends AL into AH
  - CWD (convert word to doubleword) extends AX into DX
  - CDQ (convert doubleword to quadword) extends EAX into EDX
- Effect on Flags:

No flag is effected

- Example:

```
mov eax,0FFFFFF9Bh      ; -101 (32 bits)
cdq          ; EDX:EAX = FFFFFFFFFFFFFF9Bh
              ; -101 (64 bits)
```

# IDIV Instruction

- IDIV (signed divide) performs signed integer division
- Uses same operands as DIV
- Before executing division, the dividend must be completely sign-extended using CBW, CWD, CDQ
- The remainder always has the same sign as the dividend.

# IDIV Instruction

- Effect on Flags:
- All arithmetic status flag values are undefined after executing DIV and IDIV.

# Examples:

## Example 1:

- 8-bit division of -48 by 5

```
mov al,-48  
cbw          ; extend AL into AH  
mov bl,5  
idiv bl      ; AL = -9, AH = -3
```

## Example 2:

- 32-bit division of -48 by 5

```
mov eax,-48  
cdq          ; extend EAX into EDX  
mov ebx,5  
idiv ebx     ; EAX = -9, EDX = -3
```

# Divide Overflow

Divide overflow happens when the quotient is too large to fit into the destination.

```
mov ax, 100h  
mov bl, 10h  
div bl           ; AL cannot hold 100h
```

- It causes a CPU interrupt and halts the program. (divided by zero cause similar results)

# ADC Instruction

- ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand.
- The instruction format is same as for ADD instruction
- Example:
- Add two 32-bit integers (FFFFFFFh + FFFFFFFh), producing a 64-bit sum:

```
    mov edx,0  
    mov eax,0FFFFFFFh  
    add eax,0FFFFFFFh  
    adc edx,0 ;EDX:EAX = 00000001FFFFFFEh
```

## Example(1/3):

- Task:
- Add two integers of any size
- Pass pointers to the addends (ESI, EDI) and sum
- (EBX), ECX indicates the number of doublewords

## Example:(2/3)

```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
sum DWORD 3 dup(?)  
    ; = 0000000122C32B0674BB5736
.code
...
mov  esi,OFFSET op1 ; first operand
mov  edi,OFFSET op2 ; second operand
mov  ebx,OFFSET sum ; sum operand
mov  ecx,2           ; number of doublewords
call Extended_Add
...
```

## Example (3/3)

```
Extended_Add PROC
    Pushad
    clc

    L1:
        mov eax,[esi] ; get the first integer
        adc eax,[edi] ; add the second integer
        pushfd          ; save the Carry flag
        mov [ebx],eax ; store partial sum
        add esi,4      ; advance all 3 pointers
        add edi,4
        add ebx,4
        popfd          ; restore the Carry flag
        loop L1          ; repeat the loop
        adc word ptr [ebx],0 ; add leftover carry

    popad
    ret
Extended_Add ENDP
```

# SBB Instruction

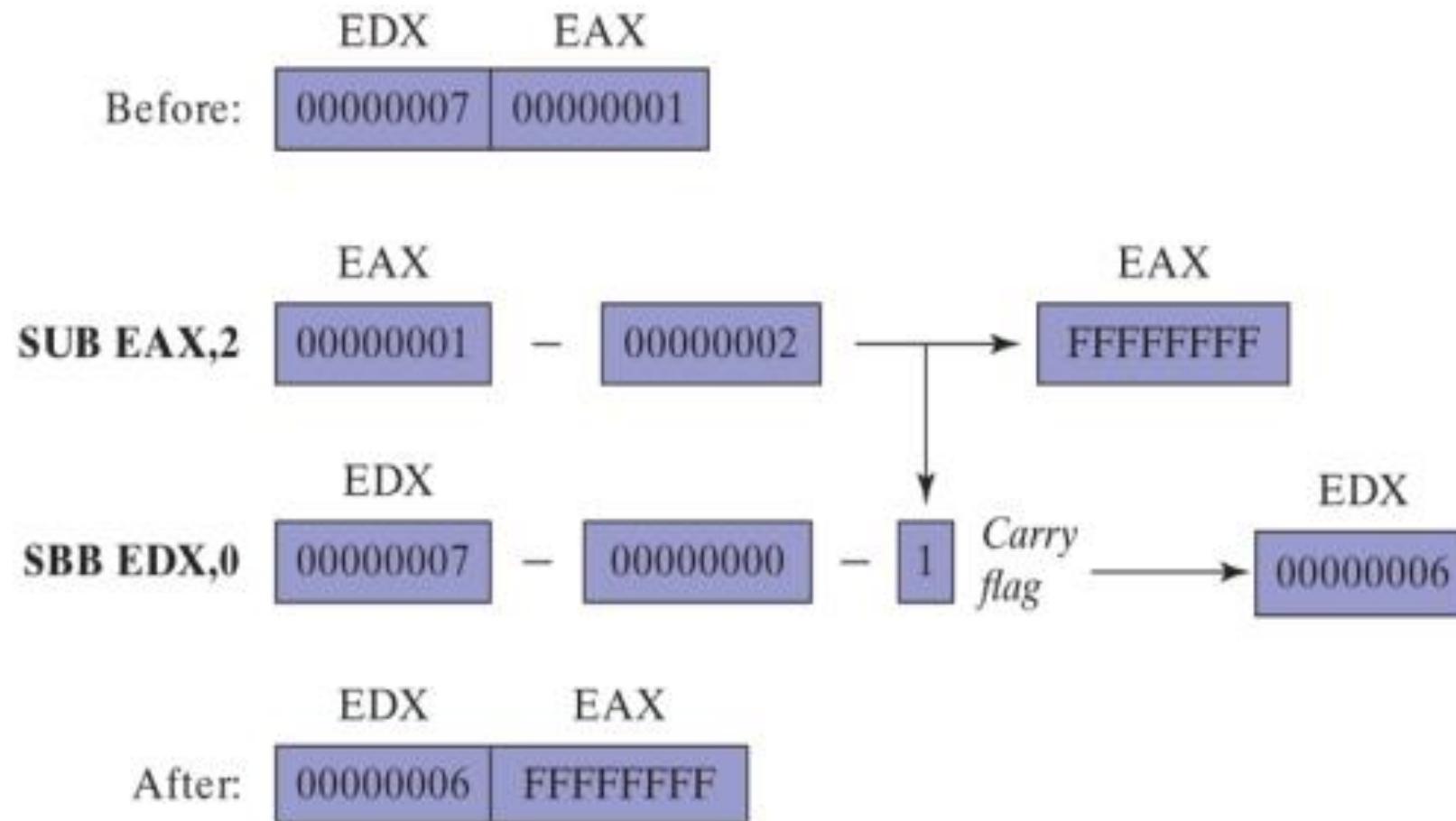
- The SBB (subtract with borrow) instruction subtracts source operand from a destination operand and then subtracts the carry flag from the destination.
- The possible operands are same as for ADC instruction

## Example:

- The following example code performs 64-bit subtraction. It sets EDX:EAX to **0000000700000001h** and subtracts 1 from this value. The lower 32 bits are subtracted first, setting the Carry flag. Then the upper 32 bits are subtracted, including the Carry flag:

```
mov edx, 7      ; upper half  
mov eax, 1      ; lower half  
sub eax, 2      ; subtract 2  
sbb edx, 0      ; subtract upper half
```

# Steps:



# Implementing Arithmetic Expressions:

## Exercise:

```
var4 = (var1 * -5) / (-var2 % var3);
```

```
mov  eax,var2      ; begin right side
neg  eax
cdq
idiv var3          ; EDX = remainder
mov  ebx,edx        ; EBX = right side
mov  eax,-5         ; begin left side
imul var1           ; EDX:EAX = left side
idiv ebx            ; final division
mov  var4,eax        ; quotient
```

Adapted from:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
2. Assembly Language Programming and organization of the IBM PC  
by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 11

Chapter 8 “Advanced Procedures”

# Important Terminologies

- Functions/ Methods/ Procedures/ Subroutines

- Arguments:

Values passed to a subroutine by a calling program

- Parameters

Values received by the called subroutine.

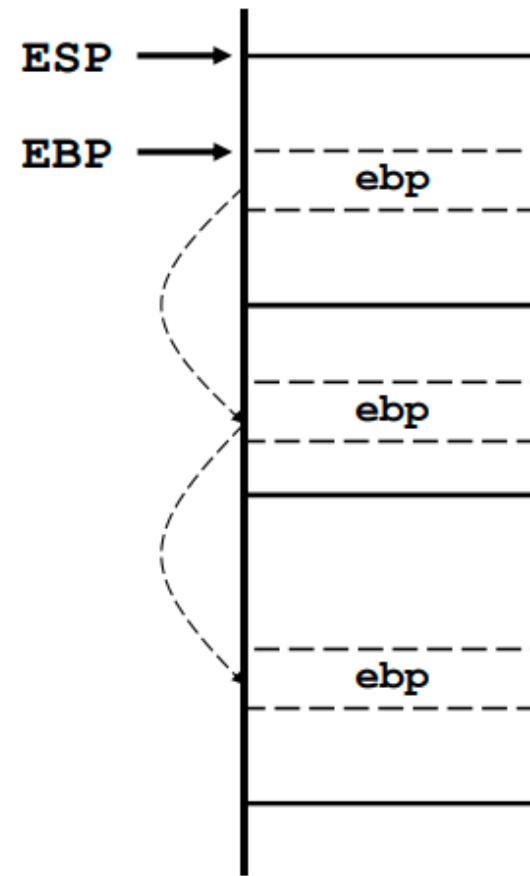
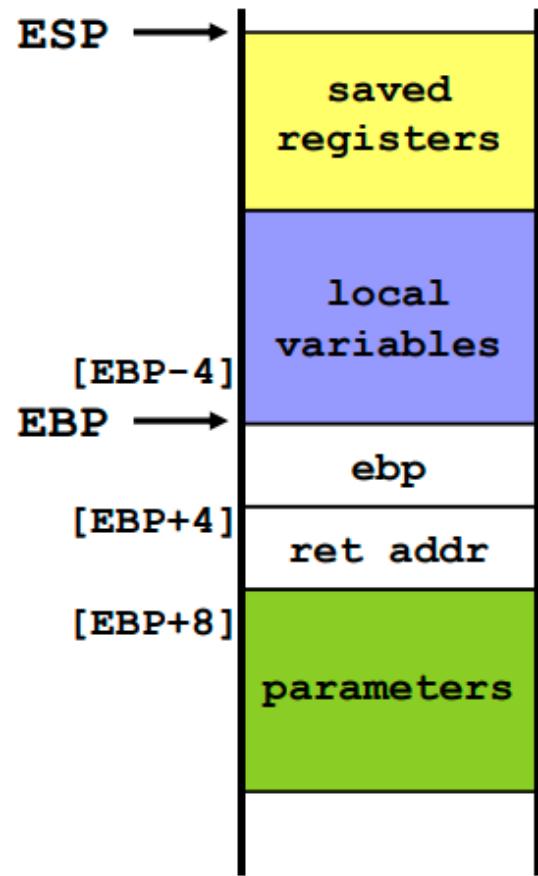
# Stack Frame( 1/3)

- Also known as an **activation record**
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables

## Stack Frame (2/3)

- Created by following sequential steps:
  - Calling procedure pushes **arguments** on the stack and calls the procedure.
  - The subroutine is called, causing the **return address** to be pushed on the stack.
  - The called procedure pushes **EBP** on the stack, and **sets EBP to ESP**. (from this point EBP acts as a base reference for all subroutine parameters)
  - If **local variables** are needed, a constant is subtracted from ESP to make room on the stack.
  - The **registers needed to be saved** are pushed on the stack.

# Stack Frame (3/3)



# Explicit access to stack parameters

- A procedure can explicitly access stack parameters using constant offsets from EBP.  
Example: [ebp + 8]
- EBP is often called the **base pointer** or **frame pointer** because it holds the **base address of the stack frame**.
- EBP does not change value during the procedure.
- EBP must be restored to its original value when a procedure returns

# Parameters

- Two types:
  - register parameters
  - stack parameters.
- 
- Stack parameters are more convenient than register parameters.

# Example 1 demonstrates using different parameters

- Example demonstrates calling DumpMem using register parameters and stack parameters

```
; Register Parameters  
pushad  
mov esi, OFFSET array  
mov ecx, LENGTHOF array  
mov ebx, TYPE array  
call DumpMem  
popad
```

```
;Stack Parameters  
push TYPE array  
push LENGTHOF array  
push OFFSET array  
call DumpMem
```

## Example 2 demonstrates using different parameters

- Consider the following max procedure

```
int max ( int x, int y, int z ) {  
    int temp = x;  
    if (y > temp) temp = y;  
    if (z > temp) temp = z;  
    return temp;  
}
```

Calling procedure: `mx = max(num1, num2, num3)`

### Register Parameters

```
mov eax, num1  
mov ebx, num2  
mov ecx, num3  
call max  
mov mx, eax
```

### Stack Parameters

```
push num3  
push num2  
push num1  
call max  
mov mx, eax
```

} Reverse  
Order

# Register versus Stack Parameters

- **Passing Parameters in Registers**

- **Pros:** Convenient, easier to use, and faster to access
- **Cons:** Only few parameters can be passed
  - A small number of registers are available
  - Often these registers are used and need to be saved on the stack
  - Pushing register values on stack negates their advantage

- **Passing Parameters on the Stack**

- **Pros:** Many parameters can be passed
  - Large data structures and arrays can be passed
- **Cons:** Accessing parameters is not simple
  - More overhead and slower access to parameters

# Arguments pushed on the stack

- Two general types of arguments are pushed on the stack during subroutine calls:
- **Value arguments** (values of variables and constants)
- **Reference arguments** (addresses of variables)

- **Passing by value:**

When an argument is passed by value, a copy of the value is pushed on the stack.

- **Passing by Reference**

An argument passed by reference consists of the address (offset) of an object.

- **Passing Arrays**

High-level languages always pass arrays to subroutines by reference. That is, they push the address of an array on the stack. One would not want to pass an array by value, because doing so would require each array element to be pushed on the stack separately. Such an operation would be very slow, and it would use up precious stack space.

# Passing by value and passing by reference

call by value

```
int sum=AddTwo(a, b);
```

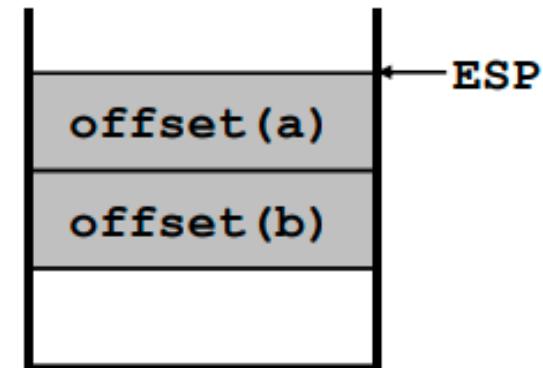
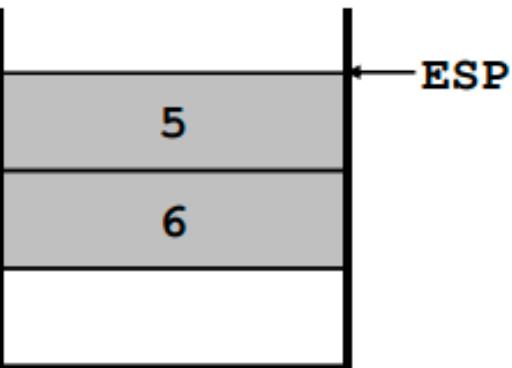
call by reference

```
int sum=AddTwo(&a, &b);
```

```
.data  
a    DWORD   5  
b    DWORD   6
```

```
push b  
push a  
call AddTwo
```

```
push OFFSET b  
push OFFSET a  
call AddTwo
```

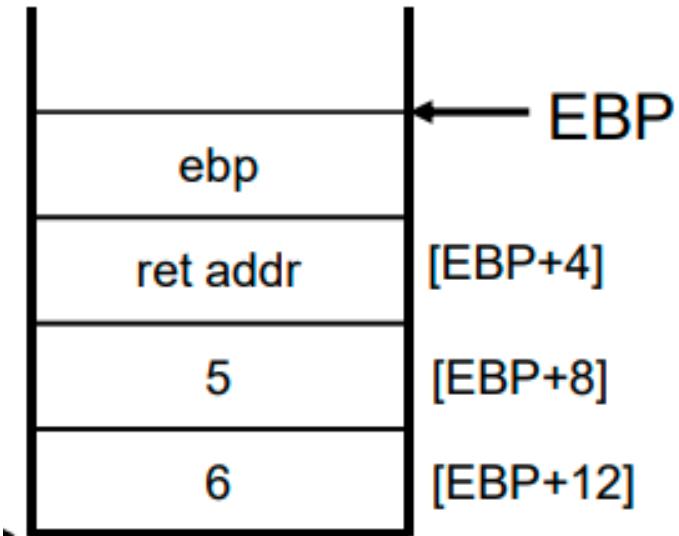


# Stack Frame Example:

```
.data  
sum DWORD ?  
.code  
    push 6          ; second argument  
    push 5          ; first argument  
    call AddTwo    ; EAX = sum  
    mov  sum,eax   ; save the sum
```

```
int AddTwo( int x, int y )  
{  
    return x + y;  
}
```

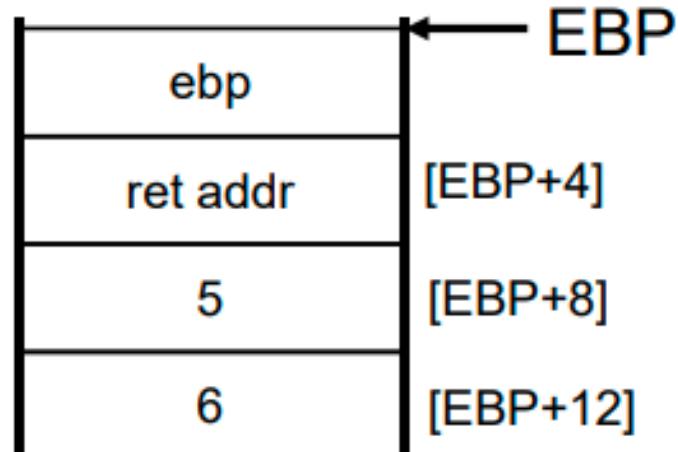
```
AddTwo PROC  
    push ebp  
    mov  ebp,esp  
    .  
    .
```



# Stack Frame Example:

```
AddTwo PROC  
    push ebp  
    mov ebp,esp          ; base of stack frame  
    mov eax,[ebp + 12]   ; second argument (6)  
    add eax,[ebp + 8]    ; first argument (5)  
    pop ebp  
    ret 8                ; clean up the stack  
AddTwo ENDP            ; EAX contains the sum
```

Who should be responsible to remove arguments? It depends on the language model.



# Accessing Parameters on the Stack

- We could have used ESP to access the parameter values
  - [ESP+8] for 5, [ESP+12] for 6
  - However, ESP might change inside procedure
  - Therefore it not used
- The accepted version is to use the EBP register
  - EBP is called the **base pointer**
  - EBP does not change during procedure
  - Start by copying ESP into EBP
  - Use EBP to locate parameters
  - EBP must be restored when a procedure returns

# Base-Offset Addressing

- We will use base-offset addressing to access stack parameters. EBP is the base register and the offset is a constant. 32-bit values are usually returned in EAX. The following implementation of AddTwo adds the parameters and returns their sum in EAX:

```
AddTwo PROC  
push ebp  
mov ebp, esp      ; base of stack frame  
mov eax, [ebp + 12] ; second parameter  
add eax, [ebp + 8]  ; first parameter  
pop ebp  
ret  
AddTwo ENDP
```

# Explicit Stack Parameters

- When stack parameters are referenced with expressions such as [ebp+8] we call them explicit stack parameters . The reason for this term is that the assembly code explicitly states the offset of the parameter as a constant value.

# Cleaning up the stack ( Passed Parameters)

- There must be a way for parameters to be removed from the stack when a subroutine returns. Otherwise, a memory leak would result, and the stack would become corrupted.
- Example:

```
main PROC
    call Example1
    exit
main ENDP
Example1 PROC
    push 6
    push 5
    call AddTwo
    ret          ; stack is corrupted!
Example1 ENDP
```

# Who Should Clean up the Stack?

- When returning for a procedure call ...
  - Who should remove parameters and clean up the stack?
- Clean-up can be done by the calling procedure (**C calling convention**)
  - `add ESP,12 ; will clean up stack`
- Clean-up can be done also by the called procedure (**STDCALL calling convention**)
  - We can specify an **optional integer** in the **ret** instruction
  - `ret 12 ; will return and clean up stack`

We will be using this convention

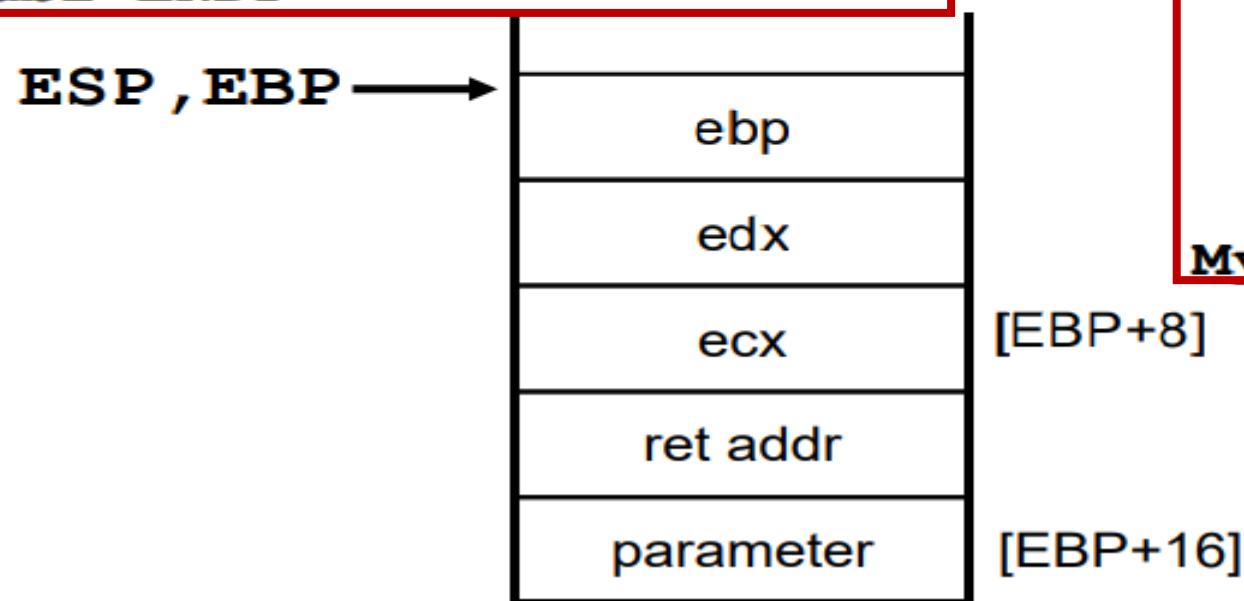
# Ret Instruction

- Return from subroutine
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - RET
  - RET n
- Optional operand n causes n bytes to be added to the stack pointer after EIP (or IP) is assigned a value.
- Return instruction is used to clean up stack
  - `ret n ; n is an integer constant`
  - Actions taken
    - EIP = [ESP]
    - ESP = ESP + 4 + n

# Saving and restoring Registers

- When using stack parameters, avoid USES

```
MySub2 PROC USES ecx, edx
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    pop ebp
    ret 4
MySub2 ENDP
```



```
MySub2 PROC
    push ecx
    push edx
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    pop ebp
    pop edx
    pop ecx
    ret 4
MySub2 ENDP
```

Note:  
The lines in blue are generated by MASM when you use USES

# Reference Parameters

- Usually accessed by procedures using base-offset addressing (from EBP).
- Because each reference parameter is a pointer, it is usually loaded into a register for use as an indirect operand.
- Suppose, for example, that a pointer to an array is located at stack address . The following statement copies the pointer into ESI:

`mov esi, [ebp+12] ; points to the array`

# Passing Arguments by reference

## ArrayFill Example:

- The ArrayFill procedure fills an array with 16-bit random integers

## Calling Program:

- The calling program passes the address of the array, along with a count of the number of array elements:

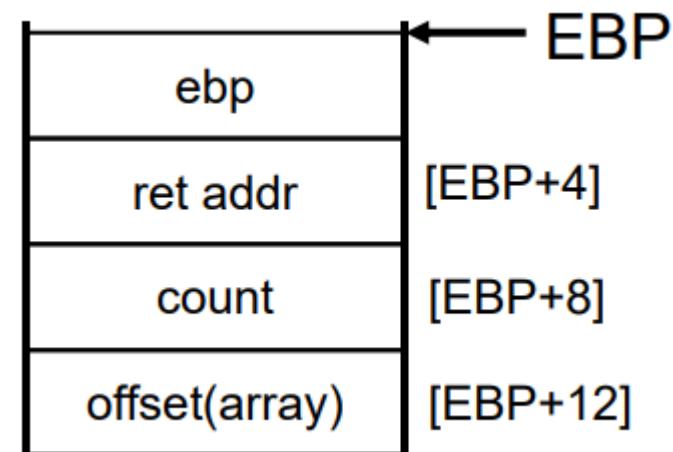
```
.data  
count = 100  
array WORD count DUP(?)  
.code  
    push OFFSET array  
    push COUNT  
    call ArrayFill
```

# Passing arguments by reference

Called Procedure (ArrayFill):

- ArrayFill can reference an array without knowing the array's name:

```
ArrayFill PROC  
    push ebp  
    mov  ebp,esp  
    pushad  
    mov  esi,[ebp+12]  
    mov  ecx,[ebp+8]
```



# Passing 8-bit and 16-bit arguments

- When passing stack arguments, it is best to push 32-bit operands to keep ESP aligned on a doubleword boundary.

```
Uppercase PROC  
    push ebp  
    mov  ebp, esp  
    mov  al, [ebp+8]  
    cmp  al, 'a'  
    jb   L1  
    cmp  al, 'z'  
    ja   L1  
    sub  al, 32  
L1:  pop  ebp  
    ret  4  
  
Uppercase ENDP
```

```
.data  
charVal BYTE 'x'  
.code  
push charVal           ; syntax error!  
call Uppercase
```

```
.data  
charVal  BYTE  'x'  
.code  
movzx eax, charVal  
push  eax  
Call   Uppercase
```

# Local variables

- The variables defined in the data segment can be taken as *static global variables*

Lifetime = program duration

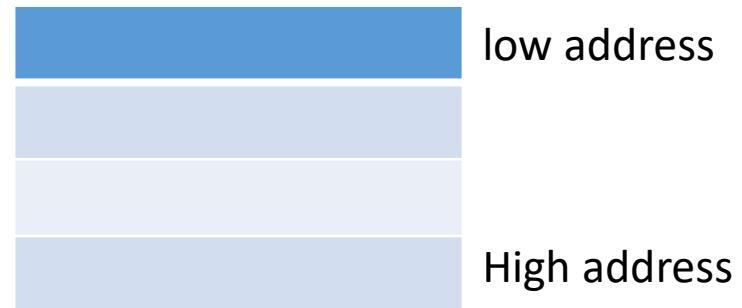
- A local variable is created, used, and destroyed within a single procedure (block)

- Advantages of local variables:
  - Restricted access: easy to debug less error prone
  - Efficient memory usage
  - Same names can be used in two different procedures
  - Essential for recursion

visibility = the whole program

# Creating Local variables (1/2)

- Local variables are created on the runtime stack usually above EBP.  
(remember this is the case when you have drawn your stack frame like this)

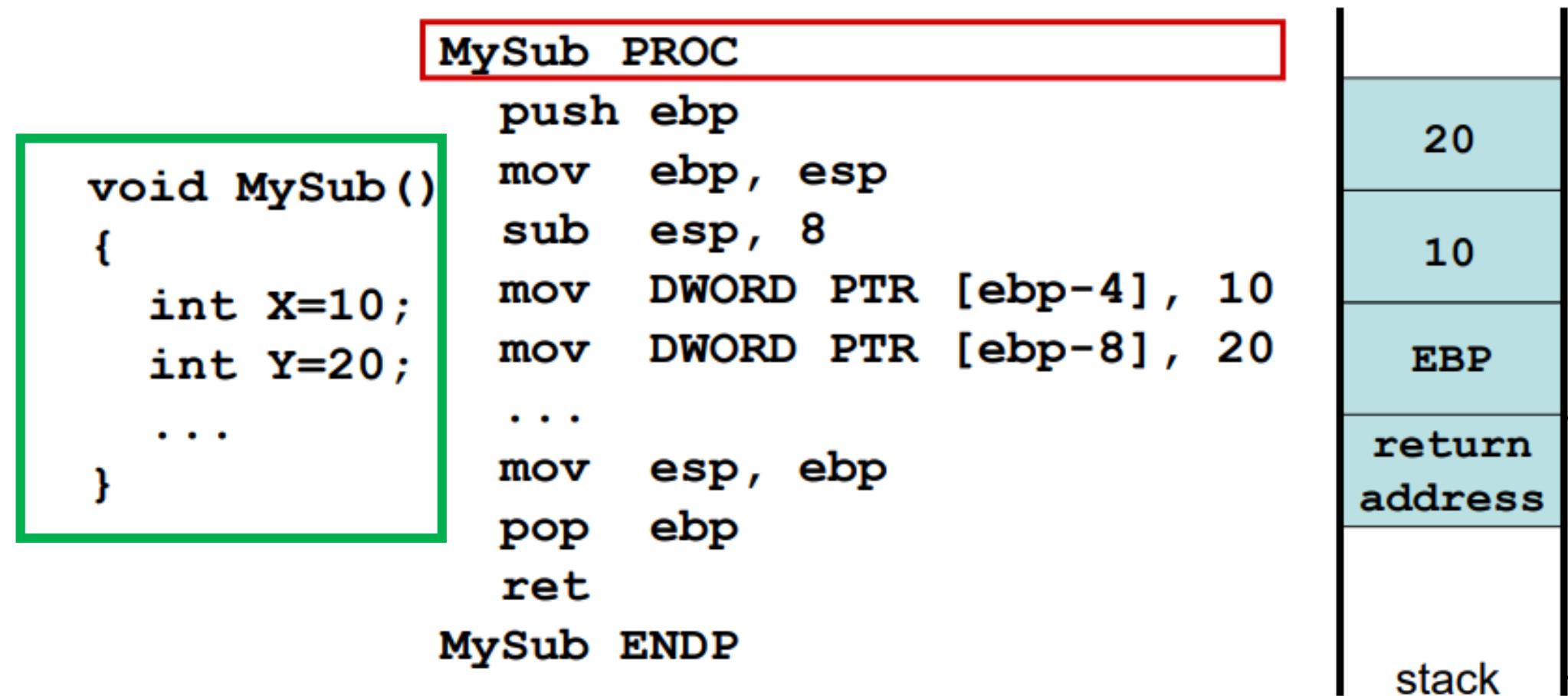


Remember stack always grows from high memory to low memory.

## Creating Local variables (2/2)

- To explicitly create local variables, subtract their total size from ESP.
- They can't be initialized at assembly time but can be assigned to default values at runtime.

# Example 1 of Local Variables



# Removing local variables from stack:

- Before finishing, the function resets the stack pointer ESP by assigning it the value of EBP. The effect is to release the local variables from the stack:

mov esp, ebp ; remove locals from stack

- If this step is omitted, the POP EBP instruction would set EBP to 20 and the RET instruction would branch to memory location 10, causing the program to halt with a processor exception

# LEA Instruction

- The LEA instruction returns offsets of both direct and indirect operands at run time.
- OFFSET only returns constant offsets (assemble time). It is not possible to use OFFSET to get the address of a stack parameter because OFFSET only works with addresses known at compile time. The following statement would not assemble:  
**mov esi, OFFSET [ebp-30] ; error**
- LEA is required when obtaining the offset of a stack parameter or local variable.

# LEA Example

```
void makeArray()
{
    char myString[30];
    for (int i=0; i<30; i++)
        myString[i]='*';
}
```

- Use LEA to move the address of [ebp-30] in esi.
- Remember the address of this local variable will be moved to esi.

```
makeArray PROC
    push ebp
    mov ebp, esp
    sub esp, 32
    lea esi, [ebp-30]
    mov ecx, 30
L1: mov BYTE PTR [esi], '*'
    inc esi
    loop L1
    add esp 32; mov esp, ebp
    pop ebp
    ret
makeArray ENDP
```

Better option for removing locals

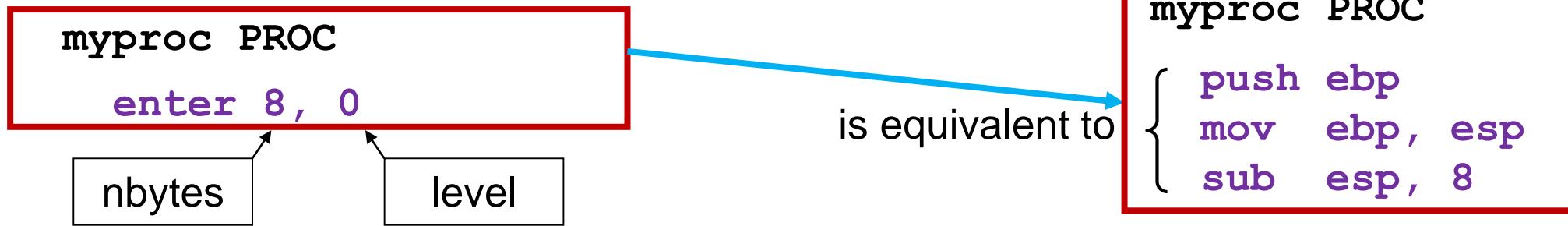
# ENTER and LEAVE Instructions

- **ENTER** instruction creates stack frame for a procedure

- Pushes EBP on the stack
- Sets EBP to the base of the stack frame
- Reserves space for local variables

**push ebp**  
**mov ebp, esp**  
**sub esp, nbytes**

Example:



- **LEAVE** instruction is equivalent to

**mov esp, ebp**  
**pop ebp**

# ENTER instruction Example:

MASM generated code

```
MySub PROC  
    enter 8,0
```

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack.

```
MySub PROC  
    push ebp  
    mov ebp,esp  
    sub esp,8
```

# LEAVE Instruction

**LEAVE** reverses the action of a previous **ENTER** instruction.

## MASM Generated Code

```
MySub PROC  
    enter 8, 0  
    .  
    .  
    .  
    .  
    leave  
    ret  
MySub ENDP
```

```
MySub PROC  
    push ebp  
    mov ebp, esp  
    sub esp, 8  
    .  
    .  
    mov esp, ebp  
    pop ebp  
    ret  
MySub ENDP
```

leave

mov esp, ebp  
pop ebp

ret

ret

# LOCAL Directive

- The LOCAL directive declares a list of local variables
  - Immediately follows the PROC directive
  - Each variable is assigned a type
  - Syntax: LOCAL varlist
- Syntax:  
`LOCAL var1:type1, var2:type2, . . .`
- Example:  
`myproc PROC  
 LOCAL var1:DWORD, ; var1 is a DWORD  
 var2:WORD, ; var2 is a WORD  
 var3[20]:BYTE, ; array of 20 bytes  
 parray:PTR WORD ; pointer to 16-bit word`

# LOCAL Directive Example

- Given **myproc** procedure

```
myproc PROC  
    LOCAL var1:DWORD,  
          var2:WORD,  
          var3[20]:BYTE  
  
    mov eax, var1  
    mov bx, var2  
    mov dl, var3  
  
    . . .  
  
    ret  
  
myproc ENDP
```

- MASM generates:

```
myproc PROC  
    push ebp  
    mov ebp, esp  
    add esp, -28 ;equal to sub esp,28  
    mov eax, [EBP-4]  
    mov bx, [EBP-6]  
    mov dl, [EBP-26]  
    . . .  
    leave  
    ret  
  
myproc ENDP
```

28 bytes reserved (multiple of 4, doubleword boundary).

# Example on Local Variables

- Consider the following procedure: **median**
- To compute the median of an array of integers
  - First, copy the array (to avoid modifying it) into a local array
  - Second, sort the local array
  - Third, find the integer value at the middle of the sorted array

```
int median (int array[], int len) {  
    int local[100];           // local array (100 int)  
    for (i=0; i<len; i++)  
        local[i] = array[i];  // Copy the array  
    bubbleSort(local,len);   // Sort the local array  
    return local[len/2];     // Return middle element  
}
```

# Reserving Stack Space

- `.STACK 4096`
- `Sub1` calls `Sub2`, `Sub2` calls `Sub3`, how many bytes will you need in the stack?

```
Sub1 PROC  
    LOCAL array1[50]:DWORD ; 200 bytes
```

```
Sub2 PROC  
    LOCAL array2[80]:WORD   ; 160 bytes
```

```
Sub3 PROC  
    LOCAL array3[300]:WORD ; 300 bytes
```

660+8(ret addr)+saved registers...

## INVOKE Directive

- The **INVOKE** directive is a powerful replacement for Intel's **CALL** instruction that lets you pass multiple arguments

- Syntax:

```
INVOKE procedureName [, argumentList]
```

- **ArgumentList** is an optional comma-delimited list of procedure arguments
- Arguments can be:

- immediate values and integer expressions
- variable names
- address and ADDR expressions
- register names

## Example:

- Using the **CALL** instruction, for example, we could call a procedure named DumpArray after executing several PUSH instructions:

```
push TYPE array  
push LENGTHOF array  
push OFFSET array  
call DumpArray
```

- The equivalent statement using **INVOKE** is reduced to a single line in which the arguments are listed in reverse order (assuming STDCALL is in effect):

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

## Example(cont)

- **Invoke** permits almost any number of arguments, and individual arguments can appear on separate source code lines. The following **Invoke** statement includes helpful comments:

```
Invoke DumpArray,      ; displays an array  
Offset array,          ; points to the array  
LengthOf array,        ; the array length  
Type array             ; array component size
```

# Homework Questions

- There are 5 homework questions
- Draw stack frame for all.

# Practice Question 1:

Create a procedure named **Swap** that exchanges the contents of two 32-bit integers. Swap receives two input–output parameters named **pValX** and **pValY**, which contain the addresses of data to be exchanged. The two parameters in the Swap procedure, **pValX** and **pValY**, are input–output parameters. Their existing values are input to the procedure, and their new values are also output from the procedure. Write a test program that displays the array before the exchange then calls the procedure Swap and finally displays the array after exchange.

Data to be exchanged: **Array DWORD 10000h, 20000h**

# One possible Solution:

```
INCLUDE Irvine32.inc
.data
Array DWORD 10000h,20000h
.code
main PROC
; Display the array before the exchange:
    mov esi, OFFSET Array
    mov ecx, 2          ; count = 2
    mov ebx, TYPE Array
    call DumpMem        ; dump the array values
    PUSH OFFSET [Array +4] ;pValY
    PUSH OFFSET Array   ;pValX
    CALL Swap
; Display the array after the exchange:
    mov esi, OFFSET Array
    mov ecx, 2          ; count = 2
    mov ebx, TYPE Array
    call DumpMem
    exit
main ENDP
```

```
;-----  
Swap PROC  
;-----  
    push ebp  
    mov ebp, esp  
    mov esi, [ebp+8]    ; get pointer pValX  
    mov edi, [ebp +12]  ; get pointer pValY  
    mov eax, [esi]      ; get first integer  
    xchg eax, [edi]    ; exchange with second  
    mov [esi], eax     ; replace first integer  
    mov esp, ebp  
    pop ebp  
    ret 8              ; Clean the stack  
Swap ENDP  
END main
```

## Practice Question 2:

Create a procedure named **ArraySum** that sums the element of a doubleword array. ArraySum receives two parameters: a pointer to a unsigned doubleword array, and a count of the array's length. ArraySum returns the result in EAX. Write a test program that calls the procedure ArraySum and then stores the result in a doubleword variable named theSum.

Sample array : Array DWORD 10, 20, 30, 40, 50  
theSum DWORD ?

# One possible Solution

```
INCLUDE Irvine32.inc
.data
array DWORD 10,20,30,40,50
theSum DWORD ?
.code
main PROC
    mov ebx, LENGTHOF array
    PUSH ebx
    PUSH OFFSET array
    CALL ArraySum
    mov theSum, eax      ; store the sum
    call DumpRegs
    call WriteDec
    exit
main ENDP
ArraySum PROC
    push ebp
    mov ebp,esp
    push ecx
    push esi
```

```
mov esi, [ebp + 8]          ; address of the array
mov ecx, [ebp +12]          ; size of the array
mov eax,0                   ; set the sum to zero
cmp ecx,0                  ; length = zero?
je L2                      ; yes: quit
L1: add eax, [esi]          ; add each integer to sum
    add esi, 4              ; point to next integer
    loop L1                 ; repeat for array size

L2:
pop esi
pop ecx
mov esp, ebp
pop ebp
ret 8

ArraySum ENDP
END main
```

## Practice Question 3:

Create the **ArrayFill** procedure, which fills an array with a pseudorandom sequence of numbers. It receives four arguments: a pointer to the array, the array length, the max possible random value (assuming 0 is the minimum) and the array type. The first is passed by reference and the others are passed by value. Write a test program that calls ArrayFill once and fills an array.

# One possible Solution:

```
INCLUDE Irvine32.inc
.data
count = 10
array WORD count DUP(0)
.code
main PROC
call Randomize
push type array
push 100
push count
push OFFSET array
call ArrayFill
mov esi, OFFSET array
mov ecx, LENGTHOF array
mov ebx, TYPE array
call DumpMem
exit
main ENDP
```

ArrayFill PROC

push ebp

mov ebp, esp

pushad

; save registers

mov esi, [ebp+8]

; offset of array

mov ecx, [ebp+12]

; array length

L1:

mov eax, [ebp+16]

; get random number from 0-99(n-1)

call RandomRange

; from the link library

mov [esi], ax

; insert value in array

add esi, TYPE word

; move to next element

loop L1

L2: popad

; restore registers

mov esp,ebp

pop ebp

ret 16

; clean up the stack

ArrayFill ENDP

END main

## Practice Question 4:

Create the **ArrayFillandSum** procedure, which fills a local variable that is an array of 5 words(16-bit integer) with a pseudorandom sequence of integers, displays them and then adds those numbers and returns the result in EAX. It receives one argument: the max possible random value (assuming 0 is the minimum). Write a test program that calls ArrayFillandSum once and saves the sum in a Dword variable Sum.

# One possible Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
Maxrange = 100
```

```
Sum Dword ?
```

```
.code
```

```
main PROC
```

```
call Randomize
```

```
push Maxrange
```

```
call ArrayFillAndSum
```

```
mov Sum, eax
```

```
call DumpRegs
```

```
exit
```

```
main ENDP
```

## ArrayFillandSum PROC

```
push ebp
mov ebp,esp
sub esp, 12          ; ESP decremented by 12 to align with doubleword boundary
lea esi, [ebp-10]    ; load address of array
mov ecx, 5           ; array length

L1:
mov eax,[ebp+8]      ; get random number from 0 to (n-1)
call RandomRange     ; from the link library
mov [esi],ax          ; insert value in array
```

```
add esi,TYPE word      ; move to next element
loop L1
    lea esi, [ebp-10]
    mov edx,0
    mov ecx, 5
    mov eax, 0
L2:
    mov ax, [esi]
    add edx, eax
    call WriteDec
    call crlf
    add esi, TYPE word      ; move to next element
loop L2
    mov eax, edx
    mov esp, ebp            ; clean the locals
    pop ebp
    ret 4                  ; clean up the stack
ArrayFillandSum ENDP
END main
```

## Practice Question 5:

Write a procedure named **Sumarrayelements** that receives pointers to three arrays of unsigned byte, word and doubleword respectively, and a fourth parameter that indicates the length of the three arrays. The procedure adds each element  $x_i$  in the first array(byte type) to the corresponding  $y_i$  in the second array(word type) and store the result in  $z_i$  which is the  $i$ th element of third array(dword). Write a test program that calls your procedure and passes the pointers to three different arrays and length of those arrays.

### Sample Variables:

- **Arr1** byte 2, 23, 45, 75, 23
- **Arr2** word 3, 100, 720, 350, 6
- **Arr3 Dword LENGTHOF Arr1 Dup(?)**

# One possible solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
Arr1 byte 2, 23, 45, 75, 23
```

```
Arr2 word 3, 100, 720, 350, 6
```

```
Arr3 Dword LENGTHOF Arr1 Dup(?)
```

```
.code
```

```
main PROC
```

```
push LENGTHOF Arr1
```

```
push OFFSET Arr3
```

```
push OFFSET Arr2
```

```
push OFFSET Arr1
```

```
call SumArrayElements
```

```
mov esi, OFFSET Arr3
```

```
mov ecx, LENGTHOF Arr3
```

```
mov ebx, TYPE Arr3
```

```
call DumpMem
```

```
exit
main ENDP
SumArrayElements PROC
    push ebp
    mov ebp,esp
    pushad
    mov esi, [ebp +8]
    mov edi, [ebp +12]
    mov ebx, [ebp+ 16]
    mov ecx, [ebp+20]
    mov eax, 0
    L1:
    movzx eax, byte ptr [esi]
    add ax, [edi]
    mov [ebx], eax
    add esi, type byte
    add edi, type word
    add ebx, type dword
    Loop L1
    popad
    mov esp, ebp          ; clean the locals
    pop ebp
    ret 16                ; clean up the stack
SumArrayElements ENDP
END main
```

## Adapted from:

1. Intel x86 Instruction Set Architecture, Computer Organization and Assembly Languages Yung-Yu Chuang
2. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
3. Assembly Language Programming and organization of the IBM PC by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 13

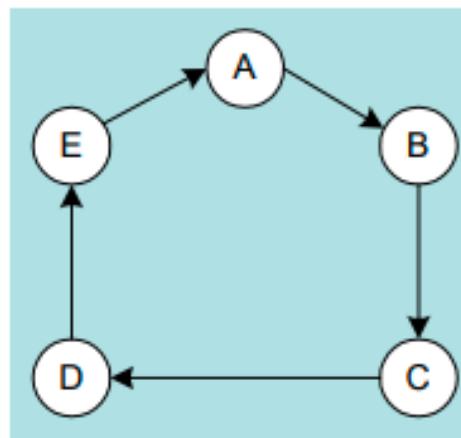
Chapter 8 “Advanced Procedures”

# Recursion

- A recursive subroutine is one that calls itself, either directly or indirectly.
- Recursion , the practice of calling recursive subroutines, can be a powerful tool when working with data structures that have repeating patterns. Examples are linked lists and various types of connected graphs where a program must retrace its path.

# Recursion

- The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:



# Endless Recursion:(Example)

```
; Endless Recursion (Endless.asm)
INCLUDE Irvine32.inc
.data
endlessStr BYTE "This recursion never stops",0
.code
main PROC
    call Endless
    exit
main ENDP
Endless PROC
    mov edx, OFFSET endlessStr
    call WriteString
    call Endless
    ret           ; never executes
Endless ENDP
END main
```

In this program each time the procedure calls itself, it uses up 4 bytes of stack space when the CALL instruction pushes the return address. The RET instruction is never executed, and the program halts when the stack overflows.

# Useful recursive subroutines

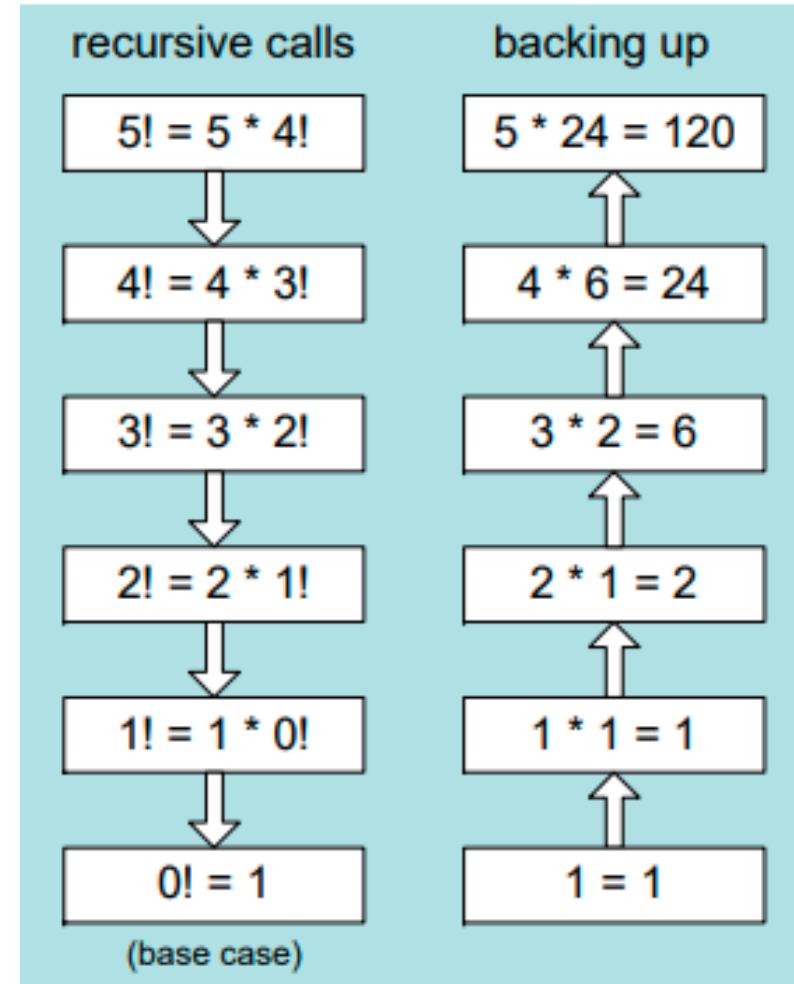
- Useful recursive subroutines always contain a terminating condition (base case). When the terminating condition becomes true, the stack unwinds when the program executes all pending RET instructions.

# Example: Calculating Factorial

- This function calculates the factorial of integer n. A new value of n is saved in each stack frame:

```
int function factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(5);
```



# Calculating Factorial (Assembly code)

```
; Calculating a Factorial (Fact.asm)
INCLUDE Irvine32.inc
.code
main PROC
    push 5          ; calc 5!
    call Factorial ; calculate factorial (EAX)
    call WriteDec ; display it
    call Crlf
    exit
main ENDP
;-----
```

# Calculating Factorial(Assembly code continued)

```
Factorial PROC
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]           ; get n
    cmp eax,0                 ; n > 0?
    ja L1                     ; yes: continue
    mov eax,1                 ; no: return 1
    jmp L2
L1:dec eax
    push eax                  ; Factorial(n-1)
    call Factorial

ReturnFact:
    mov ebx,[ebp+8]           ; get n
    mul ebx                   ; edx:eax=eax*ebx

L2:pop ebp                  ; return EAX
    ret 4                     ; clean up stack
Factorial ENDP
```

# Calculating a factorial

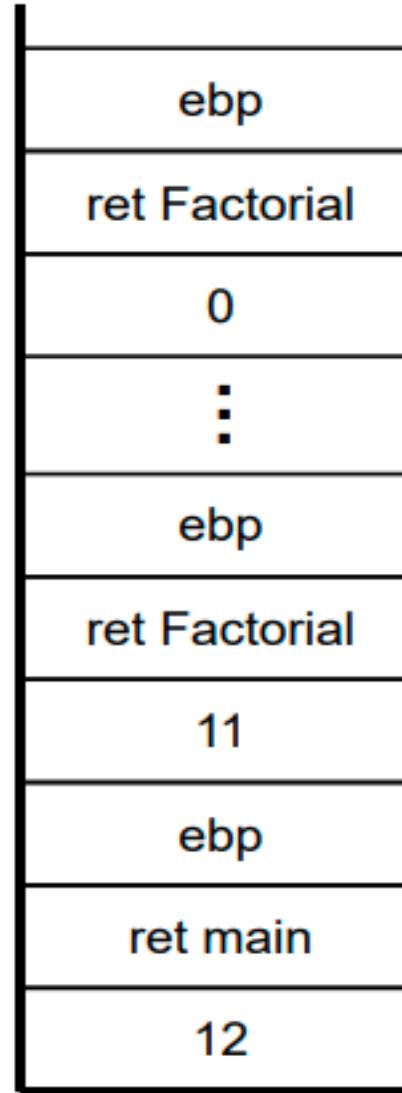
```
push 12  
call Factorial
```

```
Factorial PROC  
    push ebp  
    mov  ebp,esp  
    mov  eax,[ebp+8]  
    cmp  eax,0  
    ja   L1  
    mov  eax,1  
    jmp  L2  
L1: dec  eax  
    push eax  
    call Factorial
```

```
ReturnFact:  
    mov  ebx,[ebp+8]  
    mul  ebx
```

```
L2: pop  ebp  
    ret  4
```

```
Factorial ENDP
```



## .MODEL Directive

- **.MODEL** directive specifies a program's memory model and model options (language-specifier).
- Syntax:

```
.MODEL memorymodel [,modeloptions]
```

- **memorymodel** can be one of the following:
  - tiny, small, medium, compact, large, huge, or flat
- **modeloptions** includes the language specifier:
  - procedure naming scheme
  - parameter passing conventions
- **.MODEL flat, STDCALL**

```
[[, langtype]] [[, stackoption]]
```

# .MODEL Directive

**.MODEL memorymodel [[, langtype]] [[, stackoption]]**

Initializes the program memory model.

- The memorymodel can be TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, or FLAT.
- The langtype can be C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL.
- The stackoption can be NEARSTACK or FARSTACK.

# Memory Models

- A program's memory model determines the number and sizes of code and data segments.
- Real-address mode supports tiny, small, medium, compact, large, and huge models.
- Protected mode supports only the flat model.

Small model: code < 64 KB, data (including stack) < 64 KB.  
All offsets are 16 bits.

Flat model: single segment for code and data, up to 4 GB.  
All offsets are 32 bits.

# Language Specifier

- STDCALL (used when calling Windows functions)
  - procedure arguments pushed on stack in reverse order (right to left)
  - called procedure cleans up the stack
    - `_name@nn` (for example, `_AddTwo@8`)
- C
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack (variable number of parameters such as `printf`)
    - `_name` (for example, `_AddTwo`)
- PASCAL
  - arguments pushed in forward order (left to right)
  - called procedure cleans up the stack
- BASIC, FORTRAN, SYSCALL

## INVOKE Directive

- The **INVOKE** directive is a powerful replacement for Intel's **CALL** instruction that lets you pass multiple arguments
- Syntax:  

**INVOKE procedureName [, argumentList]**
- **ArgumentList** is an optional comma-delimited list of procedure arguments
- Arguments can be:
  - immediate values and integer expressions
  - variable names
  - address and ADDR expressions
  - register names

## Example:

- Using the **CALL** instruction, for example, we could call a procedure named DumpArray after executing several PUSH instructions:

```
push TYPE array  
push LENGTHOF array  
push OFFSET array  
call DumpArray
```

- The equivalent statement using **INVOKE** is reduced to a single line in which the arguments are listed in reverse order (assuming STDCALL is in effect):

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

## Example(cont)

- **INVOKE** permits almost any number of arguments, and individual arguments can appear on separate source code lines. The following **INVOKE** statement includes helpful comments:

INVOKE DumpArray,	; displays an array
OFFSET array,	; points to the array
LENGTHOF array,	; the array length
TYPE array	; array component size

## Important points to remember

If you pass arguments smaller than 32 bits to a procedure, **INVOKE** frequently causes the assembler to overwrite **EAX** and **EDX** when it widens the arguments before pushing them on the stack.

You can avoid this behavior by always passing 32-bit arguments to **INVOKE**, or you can save and restore **EAX** and **EDX** before and after the procedure call.

## ADDR Operator (Used with INVOKE to pass a pointer argument)

- Returns a near or far pointer to a variable, depending on which memory model your program uses:
  - Small model: returns 16-bit offset
  - Large model: returns 32-bit segment/offset
  - Flat model: returns 32-bit offset
- Simple example:

```
.data  
myWord WORD ?  
.code  
INVOKE mySub, ADDR myWord
```

# ADDR Example

```
.data  
Array DWORD 20 DUP (?)  
.code  
...  
INVOKE Swap, ADDR Array, ADDR [Array+4]
```

Code generated by Assembler assuming STDCALL is in effect

```
push OFFSET Array+4  
push OFFSET Array  
Call Swap
```

# PROC Directive

- The **PROC** directive declares a procedure with an optional list of named parameters.
- Syntax:

```
label PROC [attributes] [USES] paramList
```

- **paramList** is a list of parameters separated by commas. Each parameter has the following syntax:

*paramName:type*

*type* must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

- Example: `foo PROC C USES eax, param1:DWORD`

## PROC Example (1)

- The AddTwo procedure receives two integers and returns their sum in EAX.
- C++ programs typically return 32-bit integers from functions in EAX.

```
AddTwo PROC,  
    val1:DWORD,  
    val2:DWORD  
  
    mov eax, val1  
    add eax, val2  
    ret  
  
AddTwo ENDP
```



```
AddTwo PROC,  
    push ebp  
    mov ebp, esp  
    mov eax, dword ptr [ebp+8]  
    add eax, dword ptr [ebp+0Ch]  
    leave  
    ret 8  
AddTwo ENDP
```

## PROC Example(2)

```
Read_File PROC USES eax ebx,  
pBuffer:PTR BYTE  
LOCAL fileHandle:DWORD  
mov esi, pBuffer  
mov fileHandle, eax  
.  
.  
ret  
Read_File ENDP
```

### MASM-generated code

```
Read_File PROC  
push ebp  
mov ebp, esp  
add esp, 0FFFFFFFCh      ; equal to sub esp, 4  
push eax                ; save EAX  
push ebx                ; save EBX  
mov esi, dword ptr [ebp+8] ; pBuffer  
mov dword ptr [ebp-4], eax ; fileHandle  
. .  
pop ebx  
pop eax  
leave  
ret 4  
Read_File ENDP
```

# PROTO Directive

- Creates a procedure prototype
- Syntax:
  - *label* **PROTO** *paramList*
- Every procedure called by the **INVOKE** directive must have a prototype
- A complete procedure definition can also serve as its own prototype

# PROTO Directive

- Standard configuration: **PROTO** appears at top of the program listing, **INVOKE** appears in the code segment, and the procedure implementation occurs later in the program:

```
MySub PROTO      ; procedure prototype  
  
.code  
INVOKE MySub     ; procedure call  
  
MySub PROC      ; procedure implementation  
.  
.MySub ENDP
```

# Steps for creating Prototype

- Prototype for a procedure can be created by copying the PROC statement and making the following changes:
  - Change the word PROC to PROTO.
  - Remove the USES operator if any, along with its register list.

# PROTO Example 1:

- Prototype for the ArraySum procedure, showing its parameter list:

```
ArraySum PROTO,  
    ptrArray:PTR DWORD, ; points to the array  
    szArray:DWORD         ; array size
```

```
ArraySum PROC USES esi  ecx,  
    ptrArray:PTR DWORD, ; points to the array  
    szArray:DWORD         ; array size
```

## PROTO and INVOKE Example 2:

- The following example is used to explain valid calls and errors detected by MASM and not detected by MASM when using INVOKE and PROTO.

- Suppose prototype for Sub1 is declared as:  
Sub1 PROTO, p1:BYTE, p2:WORD, p3:PTR BYTE
- Also assume following variables are defined in data segment:

```
.data  
byte_1 BYTE 10h  
word_1 WORD 2000h  
word_2 WORD 3000h  
dword_1 DWORD 12345678h
```

- An example of valid call to Sub1:
  - **INVOKE Sub1, byte\_1, word\_1, ADDR byte\_1**
- Equivalent code generated by MASM for this INVOKE:

```
push 404000h          ; ptr to byte_1  
sub esp,2             ; pad stack with 2 bytes  
push word ptr ds:[00404001h] ; value of word_1  
mov al,byte ptr ds:[00404000h] ; value of byte_1  
push eax  
call 00401071
```

**EAX** is overwritten and the **sub esp, 2** instruction pads the subsequent stack entry to 32 bits

## Errors detected by MASM:

- If an argument exceeds the size of a declared parameter, MASM generates an error:

```
    INVOKE Sub1, word_1, word_2, ADDR byte_1      ; arg 1 error
```

- MASM generates errors if Sub1 is invoked using too few or too many arguments:

```
    INVOKE Sub1, byte_1, word_2                  ; error: too few arguments
```

```
    INVOKE Sub1, byte_1,  
    word_2, ADDR byte_1, word_2                ; error: too many arguments
```

## Errors not detected by MASM:

- If an argument's type is smaller than a declared parameter, MASM does not detect an error:

```
INVOKE Sub1, byte_1, byte_1, ADDR byte_1
```

Instead, MASM expands the smaller argument to the size of the declared parameter. In the following code generated by our INVOKE example, the second argument (byte\_1) is expanded into EAX before pushing it on the stack.

- If a doubleword is passed when a pointer was expected, no error is detected. This type of error usually leads to a runtime error when the subroutine tries to use the stack parameter as a pointer.

# Homework Question:

## Exercise:

Write a procedure named **CountMatches** that receives points to two arrays of signed doublewords, and a third parameter that indicates the length of the two arrays. For each element in the first array, if the corresponding in the second array is equal, increment a count. At the end, return a count of the number of matching array elements in EAX. Write a test program that calls your procedure and passes pointers to two different pairs of arrays. Use the **INVOKE** statement to call your procedure and pass stack parameters. Create a **PROTO** declaration for **CountMatches**. Save and restore any registers (other than EAX) changed by your procedure.

## Adapted from:

1. Intel x86 Instruction Set Architecture, Computer Organization and Assembly Languages Yung-Yu Chuang
2. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
3. Assembly Language Programming and organization of the IBM PC by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 9 part b

Chapter 6 “Conditional Processing”

# Outline

- Practice Questions related to chapter 5 and 6

## Example 1:

- Write assembly code that finds the first positive value in an array.

# Example 1:

- Write assembly code that finds the first positive value in an array:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h      ; test sign bit
    pushfd                         ; push flags on stack
    add esi,TYPE array
    popfd                           ; pop flags from stack
    loopnz next                     ; continue loop
    jnz quit                         ; none found
    sub esi,TYPE array              ; ESI points to value
quit:
```

If no value is found then after adding 2 to the ESI it points to sentinel, so we need to skip the next line

ESI was incremented due to which it points to the next value rather than the desired value after the loop terminates. In this step we are subtracting its value by 2 so that it points to the first non negative value in array.

In this code the ESI points to the desired value, There are 2 situations:

1. If positive value is found it points to the desired value.
2. If no value is found it points to the sentinel

## Option 2:

```
INCLUDE Irvine32.inc
.data
array SWORD -3,6,-1,-10,10,30,40,4 ,-40
sentinel SWORD 0
.code
main PROC
mov esi,OFFSET array
mov ecx,LENGTHOF array
next:
movsx eax, word ptr [esi]
or eax, eax
jns quit
add esi,TYPE array
```

```
loop next      ; continue loop
quit:
movsx eax, word PTR [esi]
call DumpRegs
mov ebx, type sword
mov ecx, 1
call DumpMem
call Writeint
exit
main ENDP
END main
```

## Option 3:

```
INCLUDE Irvine32.inc
.data
array SWORD -3,-6,-1,-10,-10,30,40,4 , -40
sentinel SWORD 0
.code
main PROC
mov esi,OFFSET array
mov ecx,LENGTHOF array
next:
cmp word ptr [esi], 0
jge quit
add esi,TYPE array
loop next ; continue loop
quit:
movsx eax, word PTR [esi]
call DumpRegs
mov ebx, type sword
mov ecx, 1
call DumpMem
call WriteInt
exit
main ENDP
END main
```

## Exercise 1:

Create a procedure that returns the sum of all array elements falling within the range j...k (inclusive). Write a test program that calls the procedure twice, passing a pointer to a signed doubleword array, the size of the array, and the values of j and k. Return the sum in the EAX register and preserve all other register values between calls to the procedure. Display both the results.

**Note:**

Do include the documentation of your user-defined procedure

**Hint:**

You can use the **USES** operator for saving registers. Don't use pushad/ popad here as result of procedure will be stored in EAX

**Assume:**

Array values: 30, -40, 20, 65, 80, 45

j = 20 and k = 50 for 1<sup>st</sup> call

j = 35 and k = 90 for 2<sup>nd</sup> call

## Exercise 2:

Create a procedure that saves the positive values of all array elements of an array1 in another array2. Write a test program that calls the procedure, passing a pointer to a signed doubleword array and the size of the array. Preserve all other register values between calls to the procedure.

Let elements of **array1** be **40, -90, -67, 98, 78, -45, 0, 32**.

## Exercise 3:

Implement the following pseudocode in assembly language. Use short-circuit evaluation and assume that A, B, and N are 32-bit signed integers.

while N > 0

if N != 3 AND (N < A OR N > B)

N = N - 2

else

N = N - 1

end while

## Exercise 4:

If AL contains 1 or 3, display "o"; if AL contains 2 or 4, display "e".

## Solution: (option 1)

```
INCLUDE Irvine32.inc
.code
main PROC
    mov eax,0
    CALL ReadDec
    CMP AL, 1
    JE _ODD
    CMP AL,3
    JE _ODD
    CMP AL, 2
    JE _EVEN
    CMP AL,4
    JE _EVEN
    JMP END_CASE
```

```
_ODD:
    MOV AL, 'o'
    JMP DISPLAY
_EVEN:
    MOV AL, 'e'
    JMP DISPLAY
DISPLAY:
    Call Writechar
END_CASE:
    exit
main ENDP
```

## Solution: (option 2)

```
INCLUDE Irvine32.inc
.code
main PROC
    mov eax,0
    CALL Readchar
    CMP AL, '1'
    JE _ODD
    CMP AL, '3'
    JE _ODD
    CMP AL, '2'
    JE _EVEN
    CMP AL,'4'
    JE _EVEN
    JMP END_CASE
_ODD:
    MOV AL, 'o'
    JMP DISPLAY
```

```
_EVEN:
    MOV AL, 'e'
    JMP DISPLAY
DISPLAY:
    Call Writechar
END_CASE:
exit
main ENDP
END main
```

### Note:

If you are calling procedure Readchar then while comparing do insert single quotes, as numbers will be treated as characters.

## Exercise 5:

Write assembly code for the following: Read a character. If it's "A", then execute carriage return. If it's "B", then execute line feed. If it's any other character, then the program ends.

The ASCII code for CR = 0Dh

The ASCII code for LF = 0Ah

## Exercise 6:

The following algorithm may be used to carry out division of two nonnegative numbers by repeated subtraction:

initialize quotient to 0

WHILE dividend >= divisor DO

increment quotient

subtract divisor from dividend

END WHILE

Write an assembly language code to divide AX by BX, and put the quotient in DX.

## Exercise 7:

- **Selectsort** method for sorting an array

To sort an array A of N elements, we proceed as follows:

**Pass 1:** Find the largest element among A[1] ... A[N]. Swap it and A[N]. Because this puts the largest element in position N, we need only sort A[1] ... A[N-1] to finish.

**Pass 2:** Find the largest element among A[1] ... A[N-1]. Swap it and A[N-1]. This places the next-to-largest element in its proper position.

.

.

.

**Pass N-1:** Find the largest element among A[1], A[2]. Swap it and A[2]. At this point A[2]... A[N] are in their proper positions, so A[1] is as well, and the array is sorted.

## Exercise 7: (Cont)

### Selectsort algorithm

i = N

For N-1 times DO

Find the position k of the largest element

among A[1]... A[i]

(\*)Swap A[k] and A[i]

i = i-1

END\_FOR

Step (\*) will be handled by a procedure SWAP.

Write a program that sorts the elements of an array ={60,4,17,45,7} using the selectsort algorithm. Swapping operation should be performed by user-defined procedure named SWAP.

## Exercise 8:

- To sort an array A of N elements by the **bubblesort** method, we proceed as follows:

**Pass 1.** For  $j = 2 \dots N$ , if  $A[j] < A[j-1]$  then swap  $A[j]$  and  $A[j-1]$ . This will place the largest element in position N.

**Pass 2.** For  $j = 2 \dots N-1$ , If  $A[j] < A[j-1]$  then swap  $A[j]$  and  $A[j-1]$ . This will place the second largest element in position N-1.

.

.

.

**Pass N - 1.** If  $A[2] < A[1]$ , then swap  $A[2]$  and  $A[1]$ . At this point the array is sorted.

## Exercise 8(cont):

Write a procedure BUBBLE to sort a byte array by the bubblesort algorithm. The procedure receives the offset address of the array in ESI and the number of elements in EBX. Write a program that takes 10 single-digit numbers as input from user, calls BUBBLE to sort them, and then displays the sorted list on the console window.

Adapted from:

1. Intel x86 Instruction Set Architecture, Computer Organization and Assembly Languages Yung-Yu Chuang
2. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)

# Computer Organization and Assembly Language

Week 10 part b

Chapter 7 “Integer Arithmetic”

# Outline

- Practice Questions related to chapter 5, 6 and 7

# Signed Versus Unsigned Multiplication

- In binary multiplication, signed and unsigned numbers must be treated differently.
- For example, suppose we want to multiply the eight-bit numbers 10000000 and 11111111.
- Interpreted as unsigned numbers, they represent 128 and 255; respectively. The product is  $32,640 = 0111111100000000_b$ .
- However, taken as signed numbers, they represent -128 and -1, respectively; and the product is  $128 = 0000000010000000_b$ .
- Because signed and unsigned multiplication lead to different results. There are two multiplication Instructions: MUL for unsigned multiplication and IMUL (integer multiply) for signed multiplication.

## Exercise 1:

Complete the table by filling the missing elements of the table after the execution of the given instructions:

a. Suppose AX contains 1 and BX contains FFFFh:

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL BX					
IMUL BX					

Solution:

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL BX	65535	0000FFFF	0000h	FFFFh	0
IMUL BX	-1	FFFFFFFF	FFFFh	FFFFh	0

b. Suppose AL contains FFh and BL contains FFh:

Instruction	Decimal Product	Hex Product	AH	AL	CF/OF
MUL BL					
IMUL BL					

Solution:

Instruction	Decimal Product	Hex Product	AH	AL	CF/OF
MUL BL	65025	FE01	FEh	01h	1
IMUL BL	1	0001	00h	01h	0

c. Suppose AX contains FFFh:

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL AX					
IMUL AX					

Solution:

Because MSB of AX is 0, both MUL and IMUL give same product.

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL AX	16769025	00FFE001	00FFh	E001h	1
IMUL AX	16769025	00FFE001	00FFh	E001h	1

d. Suppose AX contains 100h and CX contains FFFFh:

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL CX					
IMUL CX					

Solution:

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL CX	16776960	00FFFF00	00FFh	FF00h	1
IMUL CX	-256	FFFFF00	FFFFh	FF00h	0

e. Suppose AL contains 80h and BL contains FFh:

Instruction	Decimal Product	Hex Product	AH	AL	CF/OF
MUL BL					
IMUL BL					

Solution:

Instruction	Decimal Product	Hex Product	AH	AL	CF/OF
MUL BL	32640	7F80	7F	80	1
IMUL BL	128	0080	00	80	1

f. Suppose AL contains 80h and BL contains FFh:

Assuming higher bits are zero.

Instruction	Decimal Product	Hex Product	EDX	EAX	CF/OF
MUL ebx					
IMUL ebx					

Solution:

Instruction	Decimal Product	Hex Product	EDX	EAX	CF/OF
MUL ebx	32640	7F80	00000000h	00007F80h	0
IMUL ebx	32640	7F80	00000000h	00007F80h	0

g. Part a

Write an assembly code to perform the operation  $5h/2h$ . Also specify where will the results be saved.

**Solution:**

```
mov DX, 0  
mov AX, 5h  
mov BX, 2h  
div bx      ; Answer: Quotient in AX = 2, Remainder in DX = 1h
```

g. Part b

Suppose DX contains 0000h, AX contains 0005h and BX contains 0002h.

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX				
IDIV BX				

Solution:

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX	2	1	0002h	0001h
IDIV BX	2	1	0002h	0001h

h. Suppose DX contains 0000h, AX contains 5h and BX contains FFFEh:

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX				
IDIV BX				

Solution:

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX	0	5	0000h	0005h
IDIV BX	-2	1	FFFEh	0001h

i. Part a

Write an assembly code to perform the operation  $-5/2$ . Also specify where will the results be saved

**Solution:**

mov AX, -5

CWD

mov BX, 2

idiv bx ; Answer: Quotient in AX = -2, Remainder in DX = -1

### i. Part b

Suppose DX contains FFFFh, AX contains FFFBh and BX contains 0002:

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX				
IDIV BX				

Solution:

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX	Divide Overflow			
IDIV BX	-2	-1	FFFFh	FFFFh

j. Suppose AX contains 00FBh and BL contains FFh:

Instruction	Decimal Quotient	Decimal Remainder	AL	AH
DIV BL				
IDIV BL				

Solution:

Instruction	Decimal Quotient	Decimal Remainder	AL	AH
DIV BL	0	251	00h	FBh
IDIV BL	Divide overflow			

## Exercise 2:

The time stamp field of a file directory entry uses bits 0 through 4 for seconds, bits 5 through 10 for the minutes, and bits 11 through 15 for the hours. ( The hour number is relative to 8760 hours)

- a) Write instructions that extract the minutes and seconds and copy the values to the byte variables named **bmin** and **bsec**. Assume that the time stamp is in DX register.
  
- a) Write instructions that extract the hours and copy the value to a word variable named **bhours**. Assume that the time stamp is in DX register.

# One possible Solution:

```
mov al, dl  
and al,00011111b  
mov bsec, al  
mov ax, dx  
shr ax, 5  
and ax, 0011111b  
mov bmin, al  
mov ax, dx  
shr ax, 11  
and ax, 000000000011111b  
add ax, 8760  
mov bhours, ax
```



## Exercise 3:

Write a program that takes 8 digit telephone number of a user (local number) in hexadecimal(1 digit corresponds to 1 hex digit), extracts the area code from that number and saves in the variable **areacode** and displays it on the screen.

**Hint:**

Consider a telephone number 35872365

In this telephone number 587 represents area code.

## Exercise 4:

Write assembly instructions that calculate  $EAX * 27$  using binary multiplication.

## Exercise 5:

Create a procedure FACTORIAL that will compute  $N!$  for a positive integer  $N$ . The procedure should receive  $N$  in ECX and return  $N!$  in EAX. Suppose that overflow does not occur. Write a test program that takes  $N$  as input from user, calls the procedure FACTORIAL and displays the output on the console window.

## Exercise 6:

Write instructions to extend the sign of AL to EAX without using movsx instruction. Let value of AL be 8Fh.

Solution:

mov al, 8Fh ; or mov al, -113

SAL EAX, 24 ; or SHL EAX, 24

SAR EAX, 24

## Exercise 7:

Determine which of the following statements are illegal for the code below.  
Also determine where will the output be saved

```
.data
word1  SWORD  4
dword1 SDWORD 4
.code
imul bx, word1, -16          ; BX = word1 * -16
imul dword1, ebx, 2          ; illegal
imul bx, 4, 2                ;illegal
imul ebx, dword1, ecx        ;illegal
imul ebx, bx, 2              ;illegal
imul ebx ,dword1, -16         ; EBX = dword1 * -16
imul ebx, dword1, -2000000000 ; signed overflow!
```

## Exercise 8:

Write a program that prompts the user to enter a character, and on subsequent lines prints its ASCII code in binary, and the number of 1 bits in its ASCII code. (You are only allowed to use writechar and readchar, DumpMem and DumpRegs from Irvine32 library).

### Sample execution:

TYPE A CHARACTER: A

THE ASCII CODE OF A IN BINARY IS 01000001

THE NUMBER OF 1 BITS IS 2

## Exercise 9:

- Perform the operation of  $90876Fh / 23h$  using unsigned division.

## Example(1/3):

- Task:
- Add two integers of any size
- Pass pointers to the addends (ESI, EDI) and sum (EBX),
- ECX indicates the number of doublewords

## Example:(2/3)

```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
sum DWORD 3 dup(?)  
    ; = 0000000122C32B0674BB5736
.code
...
mov  esi,OFFSET op1 ; first operand
mov  edi,OFFSET op2 ; second operand
mov  ebx,OFFSET sum ; sum operand
mov  ecx,2           ; number of doublewords
call Extended_Add
...
```

## Example (3/3)

```
Extended_Add PROC
    Pushad
    clc

    L1:
        mov eax,[esi] ; get the first integer
        adc eax,[edi] ; add the second integer
        pushfd          ; save the Carry flag
        mov [ebx],eax ; store partial sum
        add esi,4      ; advance all 3 pointers
        add edi,4
        add ebx,4
        popfd          ; restore the Carry flag
        loop L1          ; repeat the loop
        adc word ptr [ebx],0 ; add leftover carry

    popad
    ret
Extended_Add ENDP
```

# Alternate Solution

```
.data
op1 BYTE 34h,12h,98h,74h,06h,0A4h,0B2h,0A2h
op2 BYTE 02h,45h,23h,00h,00h,87h,10h,80h
sum BYTE 9 DUP(0)
.code
main PROC
    mov esi,OFFSET op1 ; first operand
    mov edi,OFFSET op2 ; second operand
    mov ebx,OFFSET sum ; sum operand
    mov ecx,LENGTHOF op1 ; number of bytes
    call Extended_Add ; Display the sum
    mov esi,OFFSET sum
    mov ecx,LENGTHOF sum
    call Display_Sum
    call Crlf
    exit
main ENDP
```

# Extended\_Add Procedure

Extended\_Add PROC

; Calculates the sum of two extended integers stored

; as arrays of bytes

; Receives: ESI and EDI point to the two integers

; EBX points to a variable that will hold the sum

; and ECX indicates the number of bytes to be added

; Storage for the sum must be one byte longer than the input operands. ;

Returns: nothing

pushad

clc ; clear the Carry flag

L1:

mov al,[esi] ; get the first integer

```
adc al,[edi]          ; add the second integer
pushfd
mov [ebx],al          ; save the Carry flag
add esi,1             ; store partial sum
add edi,1             ; advance all three pointers
add ebx,1
popfd                ; restore the Carry flag
loop L1               ; repeat the loop
mov byte ptr [ebx],0  ; clear high byte of sum
adc byte ptr [ebx],0  ; add any leftover carry
popad
ret
Extended_Add ENDP
```

# Display\_Sum Procedure

```
Display_Sum PROC
pushad
; point to the last array element
add esi,ecx
sub esi,TYPE BYTE
mov ebx,TYPE BYTE
L1:
    mov al,[esi]      ; get an array byte
    call WriteHexB   ; display it
    sub esi, TYPE BYTE ; point to previous byte
loop L1
popad
ret
Display_Sum ENDP
```

Adapted from:

1. Intel x86 Instruction Set Architecture, Computer Organization and Assembly Languages Yung-Yu Chuang
2. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)

# Computer Organization and Assembly Language

Week 13 part b

Chapter 9 “Strings and Arrays”

# Chapter Overview

- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays

# String Primitive Instructions

- The x86 instruction set has five groups of instructions for processing arrays of bytes, words, and doublewords.
- Each instruction implicitly uses ESI, EDI, or both registers to address memory. String Instructions are designed for array processing.

# String Primitive Instructions

- String primitives execute efficiently because they automatically repeat and modify array indexes.
- The tasks carried out by the string instructions can be performed by using the register indirect addressing mode; however the string instructions have some built-in advantages. For example, **they provide automatic updating of pointer registers and allow memory-memory operations.**

# Direction Flag

- It is a control flag in the CPU Flags register that determines whether automatically repeated instructions will increment or decrement their target addresses
- The Direction flag controls the incrementing or decrementing of ESI and EDI.
  - DF = clear (0) : increment ESI and EDI
  - DF = set (1) : decrement ESI and EDI
  - In the DEBUG display, DF= 0 is symbolized by UP, and DF= 1 by DN

The Direction flag can be explicitly changed using the CLD and STD instructions:

**CLD** ; clear Direction flag(DF = 0)  
**STD** ; set Direction flag(DF = 1)

CLD and STD have no effect on the other flags

# String Primitive Instructions

- MOVSB, MOVSW, and MOVSD
- CMPSB, CMPSW, and CMPSD
- SCASB, SCASW, and SCASD
- STOSB, STOSW, and STOSD
- LODSB, LODSW, and LODSD

# Using a repeat prefix

- If you include a repeat prefix , the instruction repeats, using ECX as a counter. The repeat prefix permits you to process an entire array using a single instruction.

REP	Repeat while ECX > 0
REPZ, REPE	Repeat while ZF = 1 and ECX > 0
REPNZ, REPNE	Repeat while ZF = 0 and ECX > 0

# MOVSB, MOVSW, and MOVSD (1 of 2) (Move string data)

- The MOVSB, MOVSW, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.

```
.data  
source DWORD 0FFFFFFFh  
target DWORD ?  
.code  
mov esi, OFFSET source  
mov edi, OFFSET target  
movsd
```

## MOVSB, MOVSW, and MOVSD (2 of 2)

- ESI and EDI are automatically incremented or decremented:
  - MOVSB increments/decrements by 1
  - MOVSW increments/decrements by 2
  - MOVSD increments/decrements by 4

# Using a repeat prefix

- REP (a repeat prefix) can be inserted just before MOVSB, MOVSW, or MOVSD.
- ECX controls the number of repetitions. To move the entire string first initialize ECX to number of elements in the source string and execute REP MOVSB/MOVSW/ MOVSD.

# Example 1:

Copy 20 doublewords from source to target

```
.data
source DWORD 20 DUP(0FFFFFFFh)
target DWORD 20 DUP(?)

.code
cld                      ; direction = forward
mov ecx, LENGTHOF source ; set REP counter
mov esi, OFFSET source
mov edi, OFFSET target
rep movsd
```

## Example 2:

- Use MOVSD to delete the first element of the following doubleword array. All subsequent array values must be moved one position forward toward the beginning of the array:

array DWORD 1,1,2,3,4,5,6,7,8,9,10

```
.data  
array DWORD 1,1,2,3,4,5,6,7,8,9,10  
.code  
cld  
mov ecx, (LENGTHOF array) - 1  
mov esi, OFFSET array+type array  
mov edi, OFFSET array  
rep movsd
```

## Example 3:

Copy first two characters from string1 to string 2

```
.DATA
STRING1 DB 'HELLO'
STRING2 DB 5 DUP (?)

.CODE
LEA ESI, STRING1      ;ESI points to source string
LEA EDI, STRING2      ;EDI points to destination string
CLD
MOVSB                 ;move first byte
MOVSB                 ; and second byte
```

## Example 4:

- Write instructions to copy STRING1 of the preceding section into STRING2 in reverse order.

The idea is to get ESI pointing to the end of STRING1, EDI to the beginning of STRING2, then move characters as ESI travels to the left across STRING1.

```
LEA ESI, STRING1+4  
LEA EDI, STRING2  
STD  
MOV ECX, 5  
MOVE:  
MOVS  
ADD EDI, 2  
LOOP MOVE
```

Here it is necessary to add 2 to EDI after each MOVS. Because we do this when DF = 1, MOVS automatically decrements both ESI and EDI, and we want to increment EDI.

# CMPSB, CMPSW, and CMPSD (Compare strings)

- The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI.
  - CMPSB compares bytes
  - CMPSW compares words
  - CMPSD compares doublewords
- Repeat prefix often used
  - REPE (REPZ)
  - REPNE (REPNZ)

# Comparing a Pair of Doublewords

If source > target, the code jumps to label L1; otherwise, it jumps to label L2

```
.data
source DWORD 1234h
target DWORD 5678h
.code
mov esi, OFFSET source
mov edi, OFFSET target
cmpsd          ; compare doublewords
ja L1          ; jump if source > target
jmp L2          ; jump if source <= target
```

## Exercise:

- Modify the program in the previous slide by declaring both source and target as WORD variables. Make any other necessary changes.

# Comparing Arrays

Use a REPE (repeat while equal) prefix to compare corresponding elements of two arrays.

```
.data  
COUNT =10  
source DWORD COUNT DUP(?)  
target DWORD COUNT DUP(?)  
.code  
mov ecx, COUNT          ; repetition count  
mov esi, OFFSET source  
mov edi, OFFSET target  
cld                      ; direction = forward  
repe cmpsd               ; repeat while equal
```

# Example: Comparing Two Strings (1 of 3)

This program compares two strings (source and destination). It displays a message indicating whether the lexical value of the source string is less than the destination string.

```
.data  
source BYTE "MARTIN "  
dest  BYTE "MARTINEZ"  
str1 BYTE "Source is smaller",0dh,0ah,0  
str2 BYTE "Source is not smaller",0dh,0ah,0
```

Screen  
output:

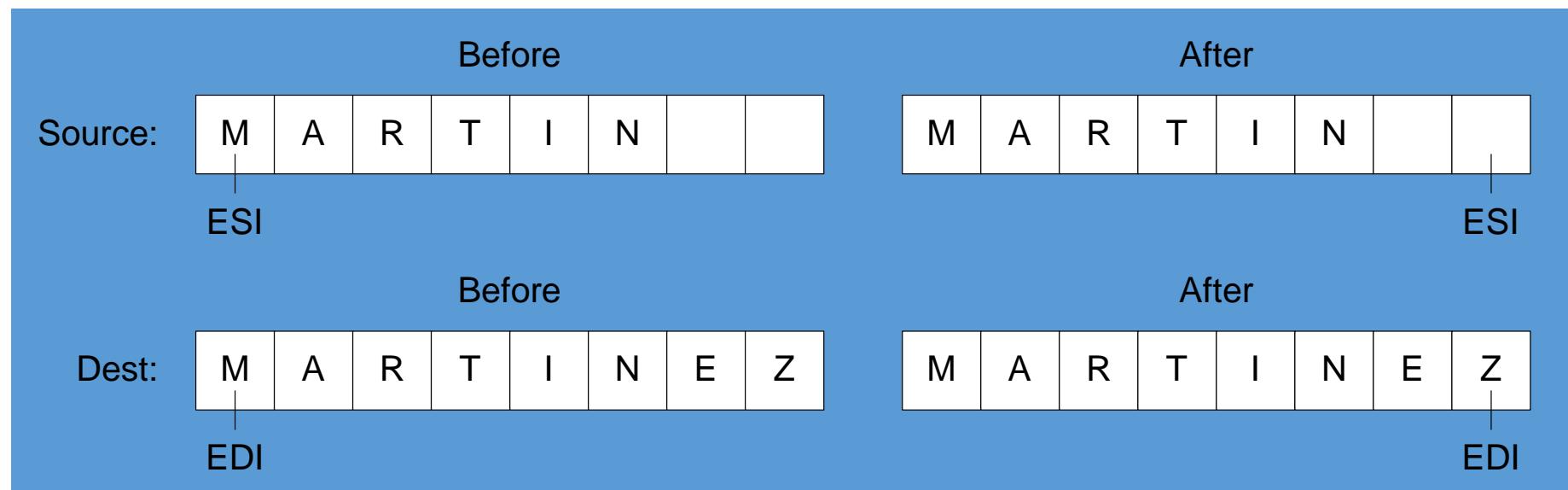
Source is smaller

## Example: Comparing Two Strings (2 of 3)

```
.code
main PROC
    cld                      ; direction = forward
    mov esi, OFFSET source
    mov edi, OFFSET dest
    mov ecx, LENGTHOF source
    repe cmpsb
    jb source_smaller
    mov edx, OFFSET str2      ; "source is not smaller"
    jmp done
source_smaller:
    mov edx, OFFSET str1      ; "source is smaller"
done:
    call WriteString
    exit
main ENDP
END main
```

## Example: Comparing Two Strings (3 of 3)

- The following diagram shows the final values of ESI and EDI after comparing the strings:



# SCASB, SCASW, and SCASD(Scan String)

- The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI.
- Useful types of searches:
  - Search for a specific element in a long string or array.
  - Search for the first element that does not match a given value.

# SCASB Example

Search for the letter 'F' in a string named `alpha`:

```
.data  
alpha BYTE "ABCDEFGHI",0  
.code  
mov edi, OFFSET alpha  
mov al, 'F' ; search for 'F'  
mov ecx, LENGTHOF alpha  
cld  
repne scasb ; repeat while not equal  
jnz quit  
dec edi ; EDI points to 'F'
```

What is the purpose of the JNZ instruction?

It is added after the loop to test for the possibility that the loop stopped because ECX = 0 and character in AL was not found

# STOSB, STOSW, and STOSD(Store string data)

- The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI.
- When used with the REP prefix, these instructions are **useful for filling all elements of a string or array with a single value**
- Example: fill an array with OFFH

```
.data  
Count = 100  
string1 BYTE Count DUP(?)  
.code  
mov al,OFFH           ; value to be stored  
mov edi, OFFSET string1 ; EDI points to target  
mov ecx, Count        ; character count  
cld                  ; direction = forward  
rep stosb            ; fill with contents of AL
```

## LODSB, LODSW, and LODSD (Load Accumulator from string)

- LODSB, LODSW, and LODSD load a byte or word from memory at ESI into AL/AX/EAX, respectively.
- The REP prefix is rarely used with LODS because each new value loaded into the accumulator overwrites its previous contents. Instead, LODS is used to load a single value.

# Array Multiplication Example

Multiply each element of a doubleword array by a constant value.

# Example:

Write a program that converts each unpacked binary-coded decimal byte belonging to an array into an ASCII decimal byte and copies it to a new array.

```
.data
array BYTE 1,2,3,4,5,6,7,8,9
dest BYTE (LENGTHOF array) DUP(?)

.code
mov esi,OFFSET array
mov edi,OFFSET dest
mov ecx,LENGTHOF array
cld

L1: lodsb          ; load into AL
    or al,30h      ; convert to ASCII
    stosb          ; store into memory
loop L1
```

Adapted from:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
2. Assembly Language Programming and organization of the IBM PC  
by Ytha Yu Charles Marut

# Computer Organization and Assembly Language

Week 14

Chapter 9 “Strings and Arrays”

# Chapter Overview

- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays

# Selected String Procedures

# Selected String Procedures

The following string procedures may be found in the Irvine32 and Irvine16 libraries:

- `Str_compare` Procedure
- `Str_length` Procedure
- `Str_copy` Procedure
- `Str_trim` Procedure
- `Str_ucase` Procedure

# Str\_compare Procedure

- Compares *string1* to *string2*, setting the Carry and Zero flags accordingly
- Prototype:

```
Str_compare PROTO,  
    string1:PTR BYTE,    ; pointer to string  
    string2:PTR BYTE      ; pointer to string
```

Relation	Carry Flag	Zero Flag	Branch if True
string1 < string2	1	0	JB
string1 == string2	0	1	JE
string1 > string2	0	0	JA

# Str\_compare Source Code

```
Str_compare PROC USES eax edx esi edi,  
            string1:PTR BYTE, string2:PTR BYTE  
    mov esi,string1  
    mov edi,string2  
L1:   mov al,[esi]  
          mov dl,[edi]  
          cmp al,0           ; end of string1?  
          jne L2             ; no  
          cmp dl,0           ; yes: end of string2?  
          jne L2             ; no  
          jmp L3             ; yes, exit with ZF = 1  
L2:   inc esi             ; point to next  
          inc edi  
          cmp al,dl          ; chars equal?  
          je L1              ; yes: continue loop  
L3:   ret  
Str_compare ENDP
```

# Str\_length Procedure

- Calculates the length of a null-terminated string and returns the length of the string(excluding the null byte) in the EAX register.
- Prototype:

```
Str_length PROTO,  
    pString:PTR BYTE           ; pointer to string
```

Example:

```
.data  
myString BYTE "abcdefg",0  
.code  
    INVOKE Str_length,  
        ADDR myString  
; EAX = 7
```

# Str\_length Source Code

```
Str_length PROC USES edi,  
    pString:PTR BYTE           ; pointer to string  
  
    mov edi,pString  
    mov eax,0                  ; character count  
  
L1:  
    cmp byte ptr [edi],0       ; end of string?  
    je L2                      ; yes: quit  
    inc edi                    ; no: point to next  
    inc eax                    ; add 1 to count  
    jmp L1  
  
L2: ret  
Str_length ENDP
```

# Str\_copy Procedure

- Copies a null-terminated string from a source location to a target location.
- Prototype:

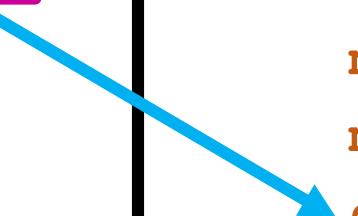
```
Str_copy PROTO,  
    source:PTR BYTE,    ; pointer to string  
    target:PTR BYTE     ; pointer to string
```

See the [CopyStr.asm](#) program for a working example.

# Str\_copy Source Code

```
Str_copy PROC USES eax ecx esi edi,  
          source:PTR BYTE,           ; source string  
          target:PTR BYTE           ; target string  
  
          INVOKE Str_length,source ; EAX = length source  
          mov ecx,eax             ; REP count  
          inc ecx                 ; add 1 for null byte  
          mov esi,source  
          mov edi,target  
          cld                     ; direction = up  
          rep movsb                ; copy the string  
          ret  
  
Str_copy ENDP
```

DF =0



# Str\_trim Procedure

- The Str\_trim procedure removes all occurrences of a selected trailing character from a null-terminated string.
- Prototype:

```
Str_trim PROTO,  
    pString:PTR BYTE,          ; points to string  
    char:BYTE                  ; char to remove
```

Example:

```
.data  
myString BYTE "Hello##",0  
.code  
    INVOKE Str_trim, ADDR myString, '#'  
  
myString = "Hello"
```

# Str\_trim Procedure

- Str\_trim checks a number of possible cases (shown here with # as the trailing character):
  - The string is empty.
  - The string contains other characters followed by one or more trailing characters, as in "Hello##".
  - The string contains only one character, the trailing character, as in "#"
  - The string contains no trailing character, as in "Hello" or "H".
  - The string contains one or more trailing characters followed by one or more nontrailing characters, as in "#H" or "###Hello".

# Testing the Str\_trim Procedure

String Definition	EDI, When SCASB Stops	Zero Flag	ECX	Position to Store the Null
str BYTE "Hello##", 0		0	>0	
str BYTE "#", 0		1	0	
str BYTE "Hello", 0		0	>0	
str BYTE "H", 0		0	0	
str BYTE "#H", 0		0	>0	

# Str\_trim Source Code

```
; Str_trim  
; Remove all occurrences of a given delimiter  
; character from the end of a string.  
; Returns: nothing  
-----  
----
```

```
Str_trim PROC USES eax ecx edi,  
pString:PTR BYTE,           ; points to string  
char: BYTE                 ; character to remove  
mov edi,pString            ; prepare to call  
Str_length  
    INVOKE Str_length, edi ; returns the length in EAX  
    cmp eax, 0              ; is the length equal to zero?  
    je L3                  ; yes: exit now  
    mov ecx, eax            ; no: ECX = string length  
    dec eax  
    add edi,eax             ; point to last character
```

# Str\_trim Source Code

```
L1: mov al,[edi]          ; get a character
    cmp al,char          ; is it the delimiter?
    jne L2                ; no: insert null byte
    dec edi               ; yes: keep backing up
    loop L1               ; until beginning reached
L2: mov BYTE PTR [edi+1],0 ; insert a null byte
L3: ret
Stmr_trim ENDP
```

# Str\_ucase Procedure

- The Str\_ucase procedure converts a string to all uppercase characters. It returns no value.
- Prototype:

```
Str_ucase PROTO,  
    pString:PTR BYTE           ; pointer to string
```

Example:

```
.data  
myString BYTE "Hello",0  
.code  
    INVOKE Str_ucase,  
        ADDR myString
```

# Str\_ucase Source Code

```
Str_ucase PROC USES eax esi,  
    pString:PTR BYTE  
    mov esi,pString  
  
L1: mov al,[esi]           ; get char  
    cmp al,0             ; end of string?  
    je L3               ; yes: quit  
    cmp al,'a'           ; below "a"?  
    jb L2               ; above "z"?  
    cmp al,'z'  
    ja L2               ; above "z"?  
    and BYTE PTR [esi],11011111b ; convert the char  
  
L2: inc esi              ; next char  
    jmp L1  
  
L3: ret  
Str_ucase ENDP
```

# Two-Dimensional Arrays

**Memory is one-dimensional. The elements of a two-dimensional array must be stored sequentially. The High-level languages select one of two methods of arranging the rows and columns in memory: row-major order and column-major , as shown in Figure.**

---

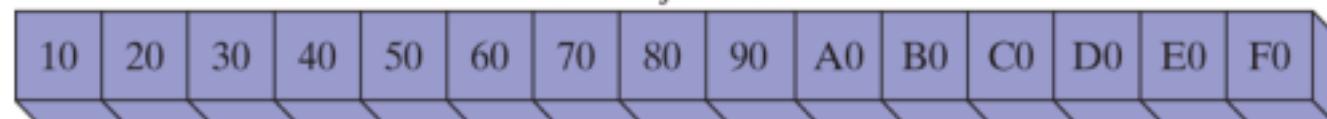
**Figure Row-major and column-major ordering.**

---

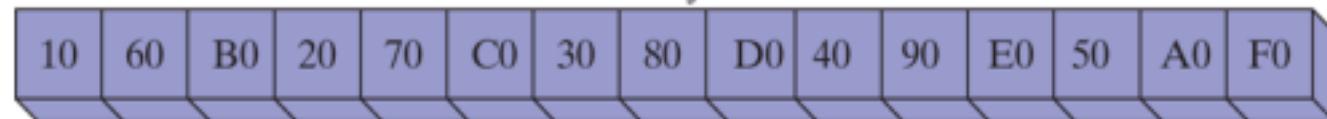
Logical arrangement:

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

Row-major order



Column-major order



# Two-Dimensional Arrays

- Base-Index Operands
- Base-Index Displacement

# Base-Index Operand

- A **base-index** operand adds the values of two registers (called base and index), producing an **effective address**. Any two 32-bit general-purpose registers may be used. (*Note: esp is not a general-purpose register*)
- **Syntax:**  
[base + index]
- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name) and two-dimensional arrays.

# Two-Dimensional Table Example

Imagine a table with three rows and five columns. The data can be arranged in any format on the page:

```
table BYTE 10h, 20h, 30h, 40h, 50h  
Rowsize = ($ - table)  
        BYTE 60h, 70h, 80h, 90h, 0A0h  
        BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h  
NumCols = 5
```

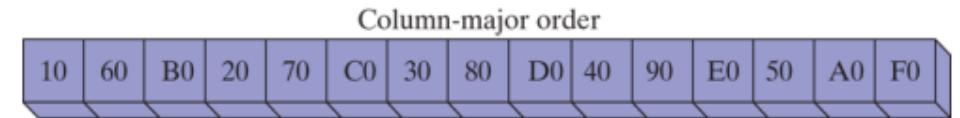
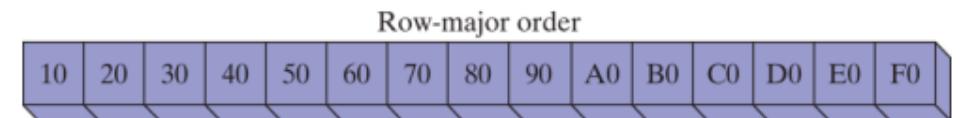
Alternative format:

```
table BYTE 10h,20h,30h,40h,50h,60h,70h,  
        80h,90h,0A0h,  
        0B0h,0C0h,0D0h,  
        0E0h,0F0h  
NumCols = 5
```

Figure Row-major and column-major ordering.

Logical arrangement:

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0



# Example : Procedure for finding sum of all elements in a row

```
; calc_row_sum  
; Calculates the sum of a row in a byte matrix.  
; Receives: EBX = table offset, EAX = row index,  
; ECX = row size, in bytes.  
; Returns: EAX holds the sum.  
calc_row_sum PROC USES ebx ecx edx esi  
    mul ecx          ; row index * row size  
    add ebx, eax     ; row offset  
    mov eax, 0        ; accumulator  
    mov esi, 0        ; column index  
    L1: movzx edx, BYTE PTR[ebx + esi]      ; get a byte  
        add eax, edx           ; add to accumulator  
        inc esi                ; next byte in row  
    loop L1  
    ret  
calc_row_sum ENDP
```

# Base-Index-Displacement Operand

- A **base-index-displacement** operand adds base and index registers to a constant, producing an **effective address**. Any two 32-bit general-purpose register can be used.
- **Displacement can be the name of a variable or a constant expression.** In 32-bit mode, any general-purpose 32-bit registers may be used for the base and index. Base-index-displacement operands are well suited to processing two-dimensional arrays. **The displacement can be an array name, the base operand can hold the row offset, and the index operand can hold the column offset.**
- Common formats:

*[ base + index + displacement ]*

*displacement [ base + index ]*

# Two-Dimensional Table Example

Imagine a table with three rows and five columns. The data can be arranged in any format on the page:

```
table BYTE 10h, 20h, 30h, 40h, 50h  
Rowsize = ($ - tableB)  
        BYTE 60h, 70h, 80h, 90h, 0A0h  
        BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h  
NumCols = 5
```

Alternative format:

```
table BYTE 10h,20h,30h,40h,50h,60h,70h,  
        80h,90h,0A0h,  
        0B0h,0C0h,0D0h,  
        0E0h,0F0h  
NumCols = 5
```

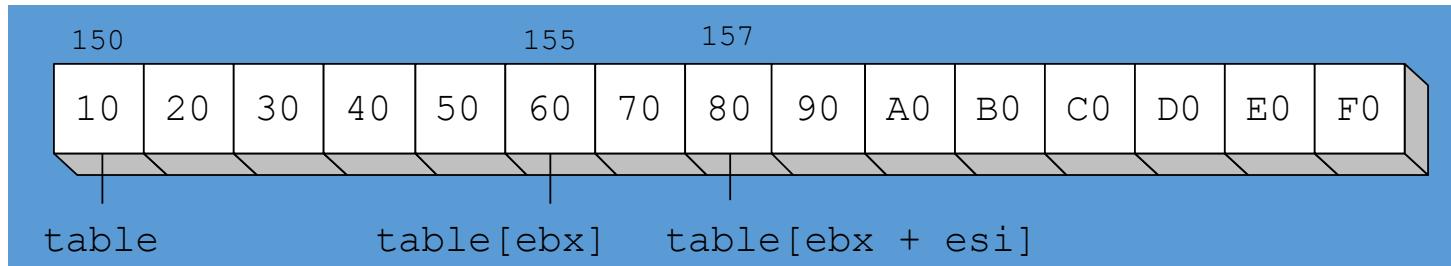
# Two-Dimensional Table Example

The following code loads the table element stored in row 1, column 2:

```
RowNumber = 1  
ColumnNumber = 2
```

```
mov ebx, NumCols * RowNumber  
mov esi, ColumnNumber  
mov al, table[ebx + esi]  
;mov al, table[ebx + esi * type table]
```

Scale factors



# Homework Question:

# Averaging test scores

Suppose a class of five students is given four exams. The results are recorded as follows:

	Test 1	Test 2	Test 3	Test 4
MARY ALLEN	67	45	98	33
SCOTT BAYLIS	70	56	87	44
GEORGE FRANK	82	72	89	40
BETH HARRIS	80	67	95	50
SAM WONG	78	76	92	60

I write a program to find the class average on each exam. To do this, we sum the entries in each column and divide by 5.

## Algorithm

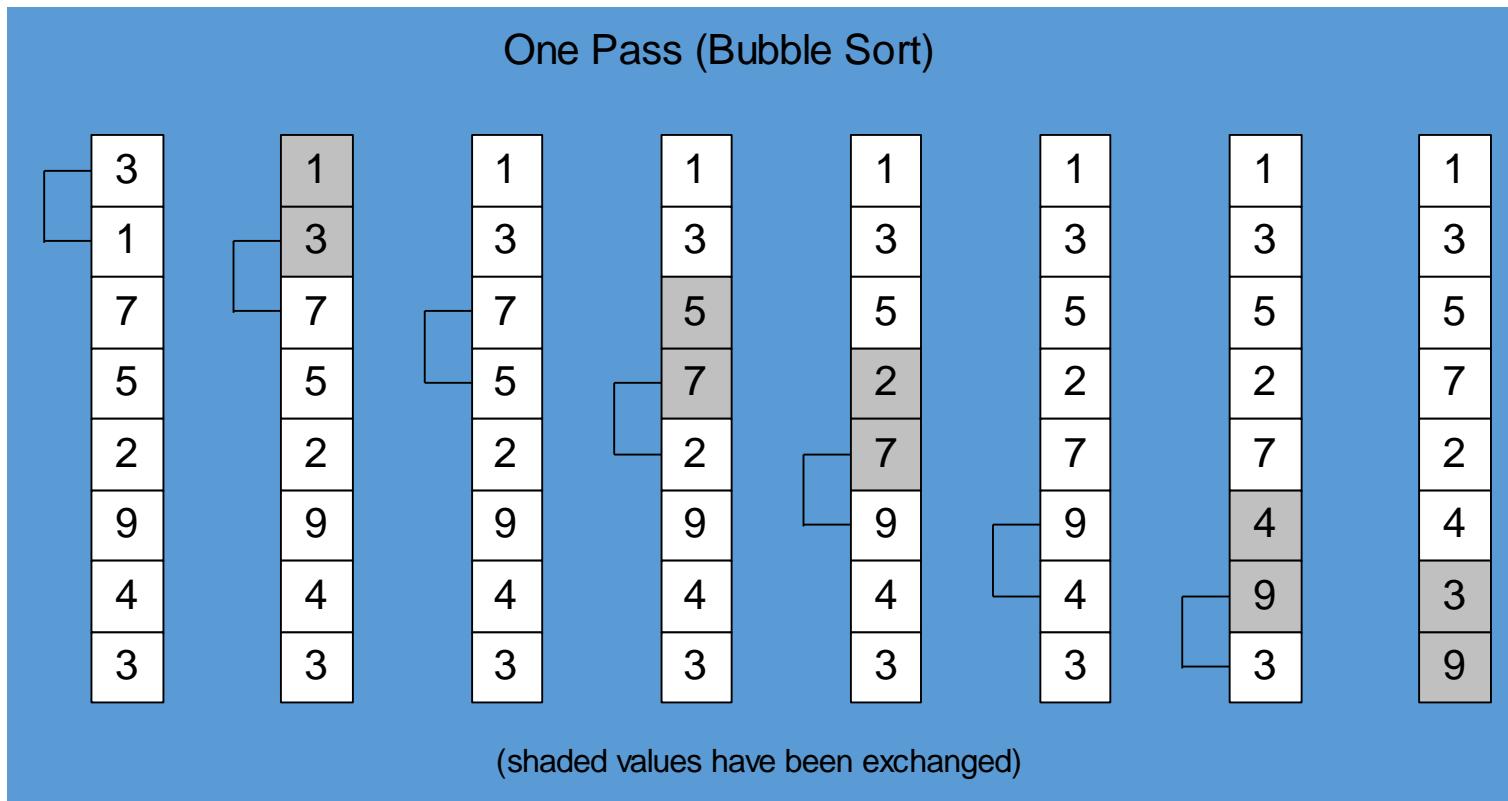
1.  $j = 4$
2. REPEAT
3.   sum the scores in column  $j$
4.   divide sum by 5 to get the average in column  $j$
4.    $j = j - 1$
5. UNTIL  $j = 0$

# Searching and Sorting Integer Arrays

- Bubble Sort
  - A simple sorting algorithm that works well for small arrays
- Binary Search
  - A simple searching algorithm that works well for large arrays of values that have been placed in either ascending or descending order

# Bubble Sort

Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



# Bubble Sort Pseudocode

N = array size, cx1 = outer loop counter, cx2 = inner loop counter:

```
cx1 = N - 1
while( cx1 > 0 )
{
    esi = addr(array)
    cx2 = cx1
    while( cx2 > 0 )
    {
        if( array[esi] < array[esi+4] )
            exchange( array[esi], array[esi+4] )
        add esi,4
        dec cx2
    }
    dec cx1
}
```

# Bubble Sort Implementation

```
BubbleSort PROC USES eax ecx esi,  
    pArray:PTR DWORD,Count:DWORD  
    mov  ecx,Count  
    dec  ecx          ; decrement count by 1  
L1: push ecx          ; save outer loop count  
    mov  esi,pArray   ; point to first value  
L2: mov  eax,[esi]    ; get array value  
    cmp  [esi+4],eax  ; compare a pair of values  
    jge  L3          ; if [esi] <= [edi] , skip  
    xchg eax,[esi+4]  ; else exchange the pair  
    mov  [esi],eax  
L3: add  esi,4        ; move both pointers forward  
    loop L2          ; inner loop  
    pop  ecx          ; retrieve outer loop count  
    loop L1          ; else repeat outer loop  
L4: ret  
BubbleSort ENDP
```

# Binary Search

- Searching algorithm, well-suited to large ordered data sets
- Divide and conquer strategy
- Each "guess" divides the list in half
- Classified as an  $O(\log n)$  algorithm:
  - As the number of array elements increases by a factor of  $n$ , the average search time increases by a factor of  $\log n$ .

# Sample Binary Search Estimates

Array Size (n)	Maximum Number of Comparisons: $(\log_2 n) + 1$
64	7
1,024	11
65,536	17
1,048,576	21
4,294,967,296	33

# Binary Search Pseudocode

```
int BinSearch( int values[],
               const int searchVal, int count )
{
    int first = 0;
    int last = count - 1;
    while( first <= last )
    {
        int mid = (last + first) / 2;
        if( values[mid] < searchVal )
            first = mid + 1;
        else if( values[mid] > searchVal )
            last = mid - 1;
        else
            return mid;      // success
    }
    return -1;                  // not found
}
```

# Binary Search Implementation (1 of 3)

```
BinarySearch PROC uses ebx edx esi edi,  
    pArray:PTR DWORD,           ; pointer to array  
    Count:DWORD,                ; array size  
    searchVal:DWORD            ; search value  
  
LOCAL first:DWORD,             ; first position  
     last:DWORD,                ; last position  
     mid:DWORD                  ; midpoint  
     mov  first,0                ; first = 0  
     mov  eax,Count              ; last = (count - 1)  
     dec  eax  
     mov  last,eax  
     mov  edi,searchVal         ; EDI = searchVal  
     mov  ebx,pArray             ; EBX points to the array  
L1:  
     mov  eax,first  
     cmp  eax,last  
     jg   L5                     ; exit search
```

# Binary Search Implementation (2 of 3)

```
; mid = (last + first) / 2
    mov  eax,last
    add  eax,first
    shr  eax,1
    mov  mid,eax

; EDX = values[mid]
    mov  esi,mid
    shl  esi,2          ; scale mid value by 4
    mov  edx,[ebx+esi]   ; EDX = values[mid]

; if ( EDX < searchval(EDI) )
;     first = mid + 1;
    cmp  edx,edi
    jge  L2
    mov  eax,mid          ; first = mid + 1
    inc  eax
    mov  first,eax
    jmp  L4                ; continue the loop
```

base-index  
addressing

# Binary Search Implementation (3 of 3)

```
; else if( EDX > searchVal(EDI) )
;   last = mid - 1;
L2: cmp  edx,edi          ; (could be removed)
    jle  L3
    mov  eax,mid           ; last = mid - 1
    dec  eax
    mov  last,eax
    jmp  L4                ; continue the loop

; else return mid
L3: mov  eax,mid           ; value found
    jmp  L9                ; return (mid)

L4: jmp  L1                ; continue the loop
L5: mov  eax,-1            ; search failed
L9: ret
BinarySearch ENDP
```

# Summary

- String primitives are optimized for efficiency
- Strings and arrays are essentially the same
- Keep code inside loops simple
- Use base-index operands with two-dimensional arrays
- Avoid the bubble sort for large arrays
- Use binary search for large sequentially ordered arrays

Adapted from:

1. Assembly Language for x86 Processors by Kip R. Irvine (7<sup>th</sup> Edition)
2. Assembly Language Programming and organization of the IBM PC  
by Ytha Yu Charles Marut

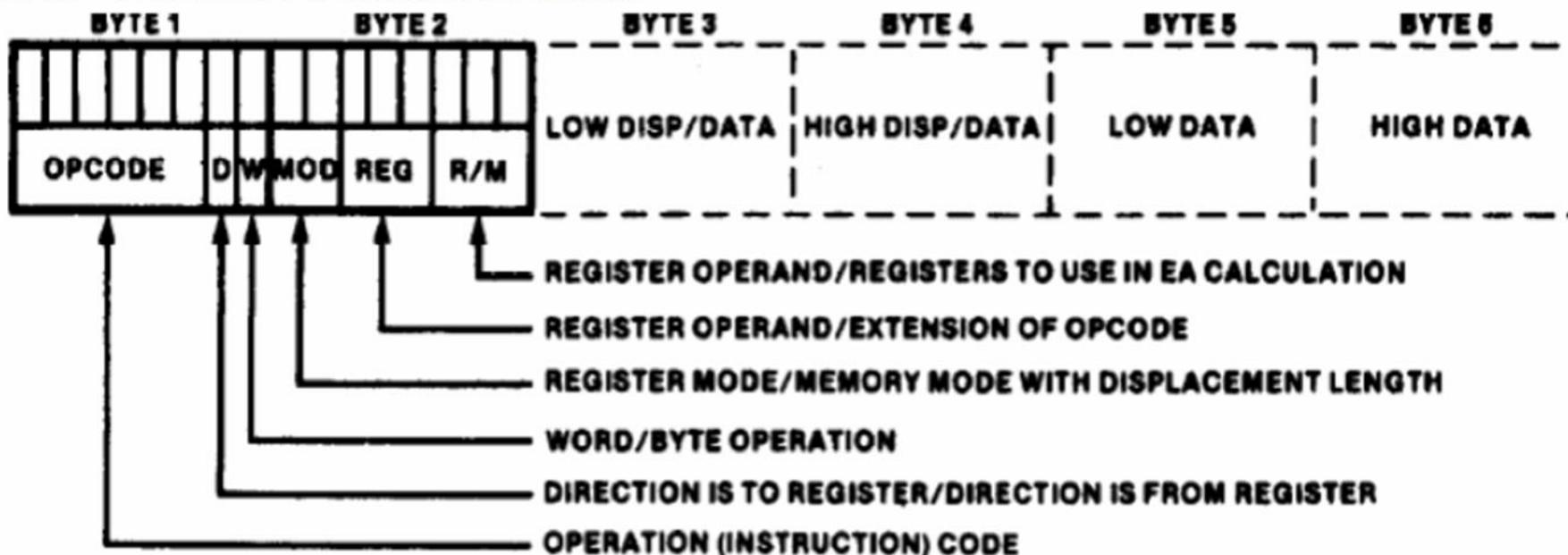
# Machine Language

## Week 15

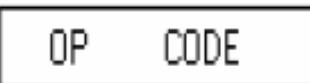
byte	7	6	5	4	3	2	1	0
1	opcode				d	w		
2	mod	reg	r/m					
3	[optional]							
4	[optional]							
5	[optional]							
6	[optional]							

# Instruction Format

## 8086-General Instruction Format



One byte instruction - implied operand(s)



One byte instruction - register mode



REG - Register

MOD - Mode

R/M - Register or memory

DISP - Displacement

DATA - Immediate data

Register to register



Register to/from memory with no displacement



Register to/from memory with displacement



(If 16-bit displacement is used)

Immediate operand to register



(If 16-bit data are used)

Immediate operand to memory with 16-bit displacement



(If 16-bit data are used)

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

\*Except when R/M = 110, then 16-bit displacement follows

(a)

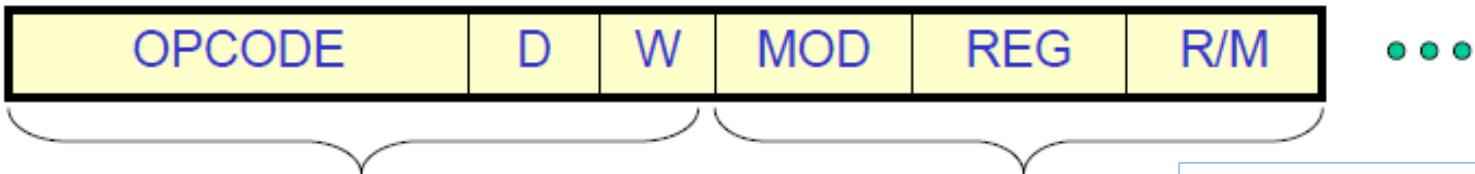
Segment Override	
00	ES
01	CS
10	SS
11	DS

REG field is used to identify the register for the first operand

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

MOD = 11			EFFECTIVE ADDRESS CALCULATION						
R/M	W = 0	W = 1	R/M	MOD = 00		MOD = 01		MOD = 10	
000	AL	AX	000	(BX) + (SI)		(BX) + (SI) + D8		(BX) + (SI) + D16	
001	CL	CX	001	(BX) + (DI)		(BX) + (DI) + D8		(BX) + (DI) + D16	
010	DL	DX	010	(BP) + (SI)		(BP) + (SI) + D8		(BP) + (SI) + D16	
011	BL	BX	011	(BP) + (DI)		(BP) + (DI) + D8		(BP) + (DI) + D16	
100	AH	SP	100	(SI)		(SI) + D8		(SI) + D16	
101	CH	BP	101	(DI)		(DI) + D8		(DI) + D16	
110	DH	SI	110	DIRECT ADDRESS		(BP) + D8		(BP) + D16	
111	BH	DI	111	(BX)		(BX) + D8		(BX) + D16	

# Converting Assembly Language Instructions to Machine Code



An instruction can be coded with 1 to 6 bytes

(1 bit) Direction. 1 = Register is Destination, 0 = Register is source.

## Byte 1 contains three kinds of information:

- Opcode field (6 bits) specifies the operation such as add, subtract, or move
- Register Direction Bit (D bit)
  - Tells the register operand in REG field in byte 2 is source or destination operand
    - 1: Data flow to the REG field from R/M
    - 0: Data flow from the REG field to the R/M
- Data Size Bit (W bit)
  - Specifies whether the operation will be performed on 8-bit or 16-bit data
    - 0: 8 bits
    - 1: 16 bits

## Byte 2 has three fields:

- Mode field (MOD) – 2 bits
- Register field (REG) - 3 bits
- Register/memory field (R/M field) – 2 bits

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

\*Except when R/M = 110, then 16-bit displacement follows

(a)

Segment Override	
00	ES
01	CS
10	SS
11	DS

REG field is used to identify the register for the first operand

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

MOD = 11			EFFECTIVE ADDRESS CALCULATION						
R/M	W = 0	W = 1	R/M	MOD = 00		MOD = 01		MOD = 10	
000	AL	AX	000	(BX) + (SI)		(BX) + (SI) + D8		(BX) + (SI) + D16	
001	CL	CX	001	(BX) + (DI)		(BX) + (DI) + D8		(BX) + (DI) + D16	
010	DL	DX	010	(BP) + (SI)		(BP) + (SI) + D8		(BP) + (SI) + D16	
011	BL	BX	011	(BP) + (DI)		(BP) + (DI) + D8		(BP) + (DI) + D16	
100	AH	SP	100	(SI)		(SI) + D8		(SI) + D16	
101	CH	BP	101	(DI)		(DI) + D8		(DI) + D16	
110	DH	SI	110	DIRECT ADDRESS		(BP) + D8		(BP) + D16	
111	BH	DI	111	(BX)		(BX) + D8		(BX) + D16	

## Examples

---

- MOV BL,AL
  - Opcode for MOV = 100010
  - We'll encode AL so
    - D = 0 (AL source operand)
  - W bit = 0 (8-bits)
  - MOD = 11 (register mode)
  - REG = 000 (code for AL)
  - R/M = 011
- D=0 when Data moving from a register

OPCODE	D	W	MOD	REG	R/M
100010	0	0	11	000	011

MOV BL,AL => 10001000 11000011 = 88 C3h

ADD AX,[SI] => 00000011 00000100 = 03 04 h

ADD [BX][DI] + 1234h, AX => 00000001 10000001 \_\_\_\_ h

=> 01 81 34 12 h

# In some cases, S, V and Z are used before w

The S, V, Z fields of the opcode in specific instructions

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W=1
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

- SR (2-bit segment register identifier field)—used in instructions to specify a segment register

The segment register identifiers

Register	SR
ES	00
CS	01
SS	10
DS	11

## **Example 1 : MOV CH, BL**

This instruction transfers 8 bit content of BL

### **Into CH**

The 6 bit Opcode for this instruction is  $100010_2$  D bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand.

D=0 indicates BL is a source operand.

W=0 byte operation

In byte 2, since the second operand is a register MOD field is  $11_2$ .

The R/M field = 101 (CH)

Register (REG) field = 011 (BL)

Hence the machine code for MOV CH, BL is

10001000 11 011 101

Byte 1              Byte2  
= 88DD

## **Example 2 : SUB Bx, (DI)**

This instruction subtracts the 16 bit content of memory location addressed by DI and DS from Bx. The 6 bit Opcode for SUB is  $001010_2$ .

D=1 so that REG field of byte 2 is the destination operand. W=1 indicates 16 bit operation.

MOD = 00

REG = 011

R/M = 101

The machine code is   0010    1011    0001    1101  
                2             B             1             D

**2B1D<sub>16</sub>**

### Example 3 : Code for MOV 1234 (BP), DX

Here we have specify DX using REG field, the D bit must be 0, indicating the DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. 1234 [BP] is specified using MOD value of 10 and R/M value of 110 and a displacement of 1234H. The 4 byte code for this instruction would be 89 96 34 12H.

<b>Opcode</b>	<b>D</b>	<b>W</b>	<b>MOD</b>	<b>REG</b>	<b>R/M</b>	<b>LB displacement</b>	<b>HB displacement</b>
100010	0	1	10	010	110	34H	12H

### Example 4 : Code for MOV DS : 2345 [BP], DX

Here we have to specify DX using REG field. The D bit must be 0, indicating that Dx is the source register. The REG field must be 010 to indicate DX register. The w bit must be 1 to indicate it is a word operation. 2345 [BP] is specified with MOD=10 and R/M = 110 and displacement = 2345 H.

Whenever BP is used to generate the Effective Address (EA), the default segment would be SS. In this example, we want the segment register to be DS, we have to provide the segment override prefix byte (SOP byte) to start with. The SOP byte is 001 SR 110, where SR value is provided as per table shown below.

<b>SR</b>	<b>Segment register</b>
00	ES
01	CS
10	SS
11	DS

To specify DS register, the SOP byte would be  $001\ 11\ 110 = 3E\ H$ . Thus the 5 byte code for this instruction would be  $3E\ 89\ 96\ 45\ 23\ H$ .

<b>SOP</b>	<b>Opcode</b>	<b>D</b>	<b>W</b>	<b>MOD</b>	<b>REG</b>	<b>R/M</b>	<b>LB disp.</b>	<b>HD disp.</b>
3EH	1000 10	0	1	10	010	110	45	23

Suppose we want to code  $MOV\ SS : 2345\ (BP),\ DX$ . This generates only a 4 byte code, without SOP byte, as SS is already the default segment register in this case.

**Example 5 :**

Give the instruction template and generate code for the instruction ADD OFABE [BX], [DI], DX (code for ADD instruction is 000000)

ADD OFABE [BX] [DI], DX

Here we have to specify DX using REG field. The bit D is 0, indicating that DX is the source register. The REG field must be 010 to indicate DX register. The w must be 1 to indicate it is a word operation. FABE (BX + DI) is specified using MOD value of 10 and R/M value of 001 (from the summary table). The 4 byte code for this instruction would be

Opcode	D	W	MOD	REG	R/M	16 bit disp.	=01 91 BE FAH
000000	0	1	10	010	001	BEH	FAH

**Example 6 :**

Give the instruction template and generate the code for the instruction MOV AX, [BX]

(Code for MOV instruction is 100010)

AX destination register with D=1 and code for AX is 000 [BX] is specified using 00 Mode and R/M value 111

It is a word operation

Opcode	D	W	Mod	REG	R/M	=8B 07H
100010	1	1	00	000	111	