

# EE213 COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE

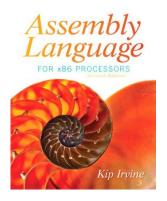
Spring 2018

#### Instructor

Muhammad Danish Khan Lecturer, Department of Computer Science FAST NUCES (Karachi)

m.danish@nu.edu.pk

#### Recommended Text Book (s)



#### Assembly Language

FOR X86 PROCESSORS
Seventh Edition
Kip R. Irvine

#### Reference Text Book



Assembly Language Programming and Organization of the IBM PC

Ytha Yu, Charles Marut

### MARKS DISTRIBUTION

•MID I (6<sup>th</sup> Week) : 15 %

•MID II (12<sup>th</sup> Week): 15 %

•Quizzes/Assignment(s): 20%

•Semester Project : ~ 10%

•Final:  $\sim 40\% - 50\%$ 

### **PREREQUISITES**

Digital Logic Design

Programming experience with some high-level language such C, C ++, Java ...

# **COURSE OBJECTIVE**

- •Covering the basics of computer organization with emphasis on the lower level abstraction of a computer system
- Programming Methodology of low-level languages, the assembly language.
- Accessing computer hardware directly
- •Overview of a user-visible architecture (of Intel 80x86 processors)
- •Intel 80x86 instruction set, assembler directives, macro, etc.
- Device handlers
- How is it possible to interface high-level language and low-level language modules

### TEACHING PLAN

- •Coverage from  $\sim$ **12** chapters from recommended book.
- Additional coverage from reference material.

#### •Mid I:

- Coverage from Ch#1 Ch#5 (~ 5 Chapters)
- Quiz I

#### •Mid II:

- Ch#5 Ch#8 (~ 4 Chapters)
- Quiz II
- Semester Project

#### Finals

- Ch#9, Ch#12, Ch#13, Ch#17, Reference Material
- Quiz III
- Semester Assignment

# ASSEMBLY LANGUAGE (ASM)

- •Machine-dependent, low-level language that uses words instead of binary code to program a specific computer system
  - Assembler is a utility program that converts source code programs from assembly language into machine language.
  - A *linker* is a utility program that combines individual files created by an assembler into a single executable program.
- •Strong correspondence between the language and the architecture's machine code instructions.
- Specific to a particular computer architecture.

### WHY ASSEMBLY LANGUAGE

- •You'll be able to choose better high-level language statements.
- •To learn the costs associated with various high-level constructs.
- Direct hardware manipulation
- Access to specialized processor instructions, or to address critical performance issues
- •For writing the compilers or device drivers, write some code in assembly language.

# COMPUTER ORGANIZATION

- •Computer organization describes how a task is done by the computer.
  - Usually a high-level description of the logic, memory, etc.

#### Computer Architecture VS Computer's organization

- •Computer architecture is abstract model and are those attributes that are visible to programmer
  - instructions sets, no of bits used for data, addressing techniques.
- Computer organization expresses the realization of the architecture
  - how features are implemented like these registers, those data paths or this connection to memory
  - how different components of computer are linked together to meet the requirements.
- Computer architecture comes before computer organization.

#### Assembly Language and the Machine Language

\*Assembly language has a one-to-one relationship with machine language.

#### High Level Language and the Assembly Language

- •High-level languages have a one-to-many relationship with assembly language and machine language.
- \*Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL.

Machine Language	Assembly Language	High-Level Language
Collection of binary numbers	Symbolic form of machine language (I.e. symbolic names are used to represent operations, registers & memory locations	Combines algebraic expressions & symbols taken from English language (ex. C++, java, Pascal, FORTRAN,etc)
Ex. 10100001 00000000 00000000 00000101 00000100 00000000	Ex. MOV AX, A ADD AX, 4 MOV A, AX	$\mathbf{E}\mathbf{x}.$ $\mathbf{A} = \mathbf{A} + 4$

#### The Instruction Set Architecture (ISA)

- •instruction set, in the processor, to carry out basic operations, such as move, add, or multiply.
  - also referred to as machine language.

#### **Assembly Language**

•Above the ISA level, assembly language uses short mnemonics which are easily translated to the ISA level.

#### **High Level Languages**

 Above the assembly level, their powerful statements are translate into multiple assembly language instructions

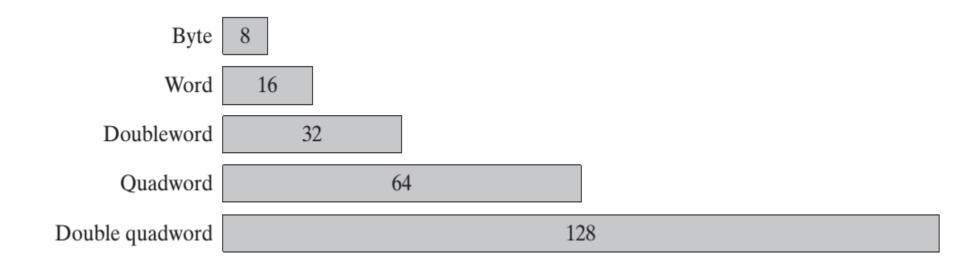
Level 4	High-level language
Level 3	Assembly language
Level 2	Instruction set architecture (ISA)
Level 1	Digital logic

### DATA REPRESENTATION

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0123456789
Hexadecimal	16	0123456789ABCDEF

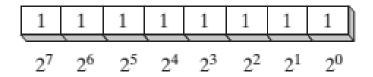
ľ	MSE	3												L	SB	
	1 (	) 1	1	0	0	1	0	1	0	0	1	1	1	0	0	
	15														0	Bit number

### STORAGE SIZES

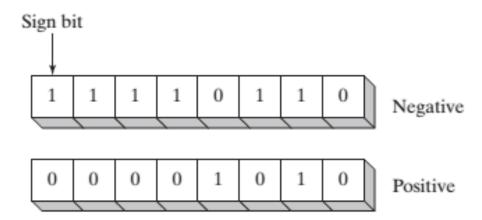


### SIGNED VS UNSIGNED

Unsigned



Signed



# TWO'S-COMPLIMENT REPRESENTATION

•Negative integers use two's-complement representation.

Starting value	0000001
Step 1: Reverse the bits	11111110
Step 2: Add 1 to the value from Step 1	11111110 +00000001
Sum: Two's-complement representation	11111111

### **BOOLEAN EXPRESSIONS**

NOT

X	¬х
F	T
T	F

AND

X	Υ	$\mathbf{X} \wedge \mathbf{Y}$
F	F	F
F	T	F
T	F	F
T	T	T

•OR

X	Υ	$\mathbf{X} \vee \mathbf{Y}$
F	F	F
F	T	T
T	F	T
Т	T	T

### **OUTLINES**

•General Concepts

•32-Bit x86 Processors

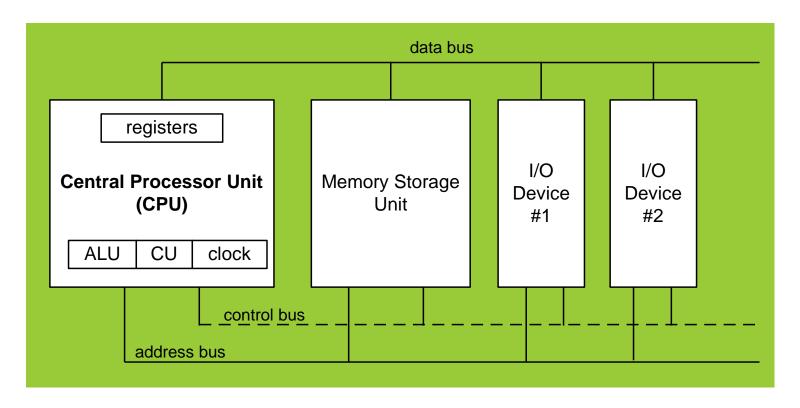
Components of a Typical x86 Computer

### **GENERAL CONCEPTS**

Basic microcomputer design

Instruction execution cycle

### BASIC MICROCOMPUTER DESIGN



Clock synchronizes CPU operations

Control Unit (CU) coordinates sequence of execution steps

ALU performs arithmetic and bitwise processing

•Memory Storage Unit: primary memory

•I/O Devices: Input/Output devices

• Data Bus: moves data between memory, i/o, and registers

•Control Bus: determines where data comes from and goes, and ALU activities

•Address Bus: selects where data comes from or goes to

### INSTRUCTION EXECUTION CYCLE

- •An instruction is not executed all at once, the CPU has to go through the *Instruction Execution Cycle* (a sequence of steps) to execute a machine instruction:
- 1. CPU has to **fetch the instruction** from an area of memory called the instruction queue.
- 2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern.
- 3. If operands are involved, the CPU **fetches the operands** from registers and memory.
- 4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step.
- 5. Finally, if an output operand was part of the instruction, the CPU stores the result of its execution in the operand.

#### program counter

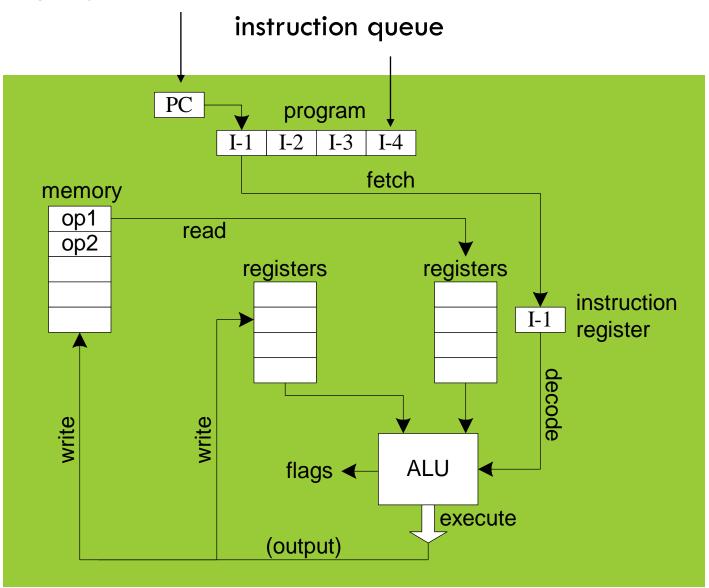
**Fetch** 

**Decode** 

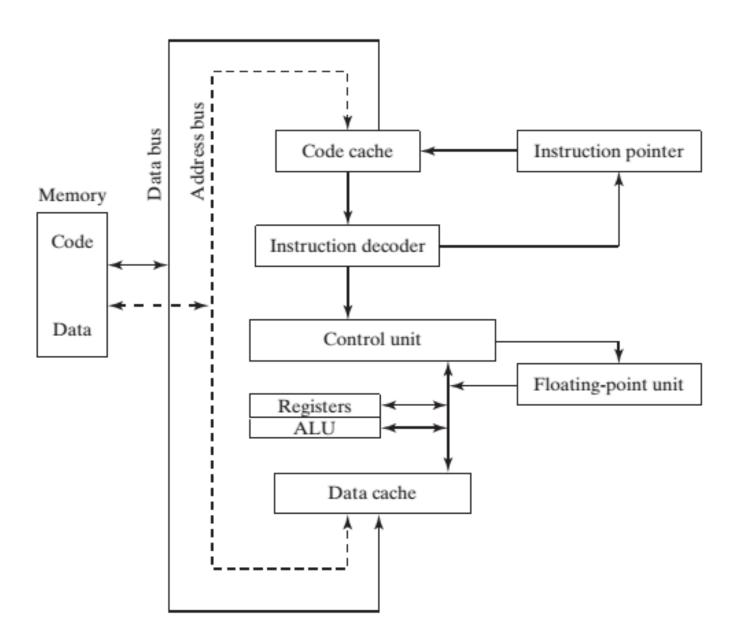
Fetch operands

Execute

Store output



INSTRUCTION EXECUTION CYCLE



# 32-BIT X86 PROCESSORS

Modes of Operation

Basic Execution Environment

\*x86 Memory Management

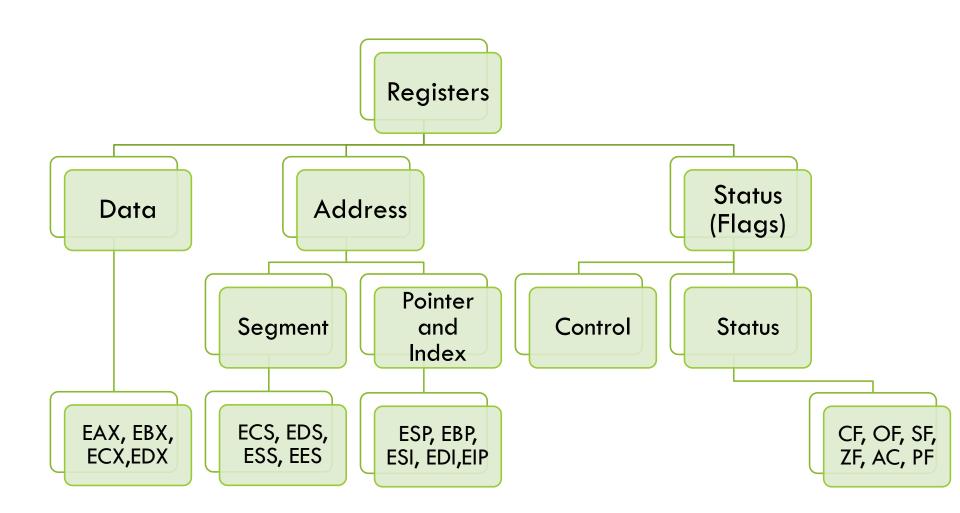
### MODES OF OPERATION

- The CPU privilege levels.
- Protected Mode: all instructions are available.
- Real-address Mode: implements programming environment of Intel8086 processor
- System Management Mode: provides OS for power management, system security, diagnostics etc.
- Virtual-8086 mode
  - hybrid of Protected
  - each program has its own 8086 computer

### BASIC EXECUTION ENVIRONMENT

#### **Basic Program Execution Registers**

- •Registers are high-speed named storage locations directly inside the CPU.
- •The registers are classified according to the functions they perform:
  - Data/General Purpose Registers hold data for an operation
  - Address registers hold the address of an instruction or data
  - Status register keeps the current status of the processor.



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

- 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.

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)	8	8	_
EAX	AX	АН	AL	AH	AL	8 bits + 8 bits
EBX	BX	ВН	BL			o ons i o ons
ECX	CX	СН	CL		AV	
EDX	DX	DH	DL		AX	16 bits
				-		
				EAX		32 bits

#### •Segment Registers:

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

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

#### Pointer and Index Register (ESP, EBP, ESI, EDI)

- •The registers ESP, EBP, ESI, and EDI normally point to (contain the offset addresses of) memory locations.
- •Unlike segment registers, the pointers and index registers can be used in arithmetic and other operations.
  - **ESP** (**Extended Stack Pointer**): this register is used in conjunction with ESS for accessing the stack segment.
  - EBP (Extended Base Pointer): these registers are used to access data on the stack and other segments.
  - **ESI (Extended Source Index):** these registers are used to point to memory locations in the data segment addressed by EDS.
  - **EDI (Extended Destination Index):** these registers perform the same functions as ESI, these are used to access memory location addressed by EES.

#### •Extended Instruction Pointer (EIP):

- •The memory registers covered so far are for data access.
- •To access instructions, x86 uses the ECS and EIP registers.
- •The ECS contain the number(base address) of next instruction to be executed and EIP contains the offset.
  - EIP is updated every time an instruction is executed so that it will point to the next instruction.

#### FLAGS Register

- •The purpose of the FLAGS register is to indicate the status of the microprocessor. It does this by the setting of individual bits called *flags*.
  - A flag is set when it equals 1; it is clear (or reset) when it equals 0.

- •There are two kinds of flags: control flags and status flags.
- The control flags enable or disable certain operations of the processor.
  - for example, if the IF (interrupt flag) is cleared (set to 0), inputs from the keyboard are ignored by the processor.
- •The status flags reflect the result of an instruction executed by the processor.
- •For example, when a subtraction operation results in a 0, the **ZF** (zero flag) is set to 1 (true).
- •Carry Flag: set when the result of an *unsigned* arithmetic operation is too large to fit into the destination.
- •The **Overflow** flag (OF) is set when the result of a *signed* arithmetic operation is too large or too small to fit into the destination.

•The **Sign** flag (SF) is set when the result of an arithmetic or logical operation generates a negative result.

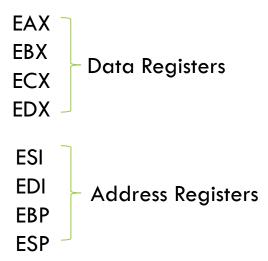
•The **Zero** flag (ZF) is set when the result of an arithmetic or logical operation generates a result of zero.

- •The **Auxiliary Carry** flag (AC) is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.
- •The **Parity** flag (PF) is set if the least-significant byte in the result contains an even number of 1 bits.

### •In x86 processors, there are:

- eight general-purpose registers
- six segment registers
- a processor status flags register (EFLAGS)
- an instruction pointer (EIP).

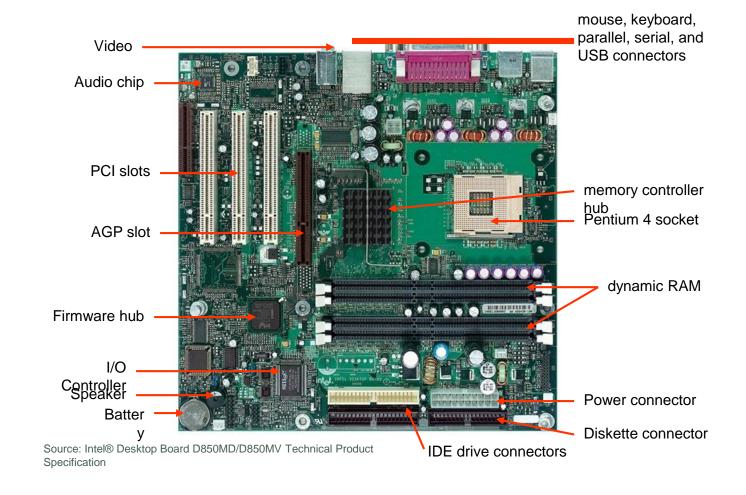
•General Purpose Registers: These are primarily used for arithmetic and data movement.



# 2.4 COMPONENTS OF A TYPICAL X86 COMPUTER

#### **Motherboard**

- •Heart of microcomputer, having following components:
  - CPU Sockets
  - Memory Slots (or RAM slots)
  - BIOS (basic input-output system) computer chips.
  - CMOS (Complementary metal-oxide-semiconductor) RAM.
  - Connectors for mass-storage devices
  - USB connectors for external devices
  - Keyboard and mouse ports
  - PCI (Peripheral Component Interconnect) bus connectors for sound cards, graphics cards, etc.



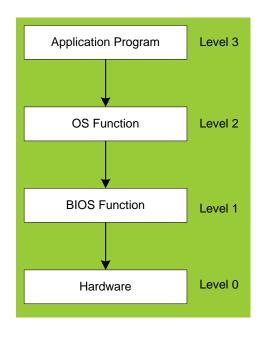
#### **Memory**

- Several basic types of memory are used in Intel-based systems:
  - **Read Only Memory (ROM)**: is permanently burned into a chip and cannot be erased.
  - Erasable Programmable Read-Only Memory (EPROM): can be erased slowly with ultraviolet light and reprogrammed
  - Dynamic Random-Access Memory (DRAM): it is where programs and data are kept when a program is running. Must be refreshed constantly.
  - Static RAM (SRAM): expensive; used for cache memory; no refresh required.
  - Video RAM (VRAM): holds video data. optimized for constant video refresh.
  - CMOS (complimentary metal oxide semiconductor) RAM: stores system setup information.

## 2.5 INPUT—OUTPUT SYSTEM

- •I/O need not be accomplished by directly accessing hardware.
- •I/O is available at different access levels:
  - High Level Language Functions: HLL contain function to perform I/O
  - Operating system: Programmers can call operating system functions from a library known as the API (application programming interface).
  - **BIOS:** The basic input—output system is a collection of low-level subroutines that communicate directly with hardware devices. Drivers that communicate directly with devices
- •Device Drivers Device drivers are programs that permit the operating system to communicate directly with hardware devices and the system BIOS.

## DISPLAYING A STRING OF CHARACTERS

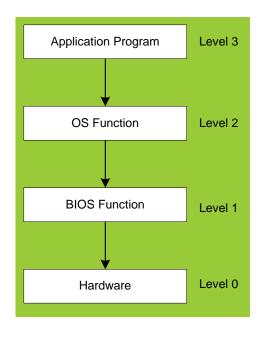


When a HLL program displays a string of characters, the following steps take place:

Level 3: The library function of HLL calls an operating system function, passing a string pointer.

Level 2: The operating system function uses a loop to call a BIOS subroutine, passing it the ASCII code and color of each character. The operating system calls another BIOS subroutine to advance the cursor to the next position on the screen.

## DISPLAYING A STRING OF CHARACTERS



Level 1: The BIOS subroutine receives a character, maps it to a particular system font, and sends the character to a hardware port attached to the video controller card.

Level 0: The video controller card generates timed hardware signals to the video display that control the raster scanning and displaying of pixels.

## SUMMARY

Assembly Language and its Objective(s)

Computer Organization VS Computer Architecture

High Level, Low Level, Machine Language

Instruction Set Architecture

Data Representation

Storage Sizes

Signed VS Unsigned Values

Two's Complement

**Boolean Expressions** 

## **SUMMARY**

Central Processing Unit (CPU)

Arithmetic Logic Unit (ALU)

Instruction execution cycle

Real mode and Protected mode

Base vs Offset Address

Registers

Motherboard components

Memory types

Input/Output and access levels