CHAPTER : 01

Assembler is a low-level.

compiler Interpreter is High-Level language

The Assembler used on the behalf of Microsoft is called MASM (Microsoft macro Assemblers

some other well runown assembler under ms windows include TASM (Turbo Assembler), NAM (Netwide Assembler).

- Two most popular linux-based assemblers are GAS (GNU Assembler) and WASM. NASM's syntax is most similar to that of MASM.

Assembly language is the oldest PL. closest resembles to native machine language. require you to understand computer's architecture and operating system.

.exe files, Application files or Directly executable files In LINUX, we cannot exeute it directly. If you want to work on MASM and user of LINUX, then you have to install layer on LINUX called VINE

MASM (.exe) works on LINUX

Before this, install Vine package

(2) If you need to be a C, or C++ developer, you need to understand memory address and instructions worle at a low level

**How Does Assembly Language Relate to Machine Language?**

Assembly language has a one-to-one relationship with machine language. Each assembly language instruction corresponds to a single machine-language instruction. However High-level languages such as Python, C++, and Java have a one-to-many relationship with assembly language and machine.

**Uses of Assembly Language?**

They are short programs stored in a small amount of memory in single-purpose devices

Assembly language is an ideal tool for writing embedded programs

Assembly language permits you to precisely specify a program's executable code.

it permits direct access to computer hardware, and code can be hand optimized for speed.

 Assembly language helps you to gain an overall understanding of the interaction between computer hardware, operating systems, and application programs.

**What Are Assemblers and Linkers?**

An 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. A related utility, called a debugger, lets you to step through a program while it's running and examine registers and memory.

**What Types of Programs Can Be Created Using MASM?**

• 32-Bit Protected Mode: 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode pro

**DIFFERENCE BETWEEN ASSEMBLERS LINKERS AND DEBUGGERS**

Assemblers

A utility program that converts source code program from Assembly to machine language.

Linkers

 A utility program that combines individual file created by an Assembler into singl executable program

Debugger

utility program that let you to step through a program while its running and examises registers and memory.

first real-Address mode OS for user was DOS-( Disk Os). in 1980 from Microsoft.

Nowadays, 16 Bit real address mode is embedded in Microsoft Windows.

can be accessed by CMP or command prompt. (CL1)

Op code.

Instructions written in machine lang.

 Mnemonics

Instruction written in Assembly lang. - such as; ADD, MOVE, SUB,

**Assembly lang. has one-to-one relationship** with machine

language. However, **High-level language have one-to-many relationship** with assembly language & Machine language.

Instruction:  single task is know as instruction.

statement:  multiple instructions or task in single line or that statement.

Single-line comments are written before colon ;

**Binary Integers**

 A computer stores instructions and data in memory as collections of electronic charges. Representing these entities with numbers requires a system geared to the concepts of on and off or true and false. Binary numbers are base 2 numbers, in which each binary digit (called a bit) is either 0 or 1. Bits are numbered sequentially starting at zero on the right side and increasing toward the left. The bit on the left is called the most significant bit (MSB), and the bit on the right is the least significant bit (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure: MSB

***UNSIGNED CONVERSIONS***

**Converting Unsigned Decimal to Binary**

Divide decimal base with binary base ungless remainder is 1 or O

for example

37/2= 18-1=18/2 = 9-0=.....

**Converting Unsigned Binary Integers to Decimal**

dec = (Dn-1 X 2n-1 ) + (Dn-2 X 2n-2) + ... + (D, X 2') + (D0 x 20)

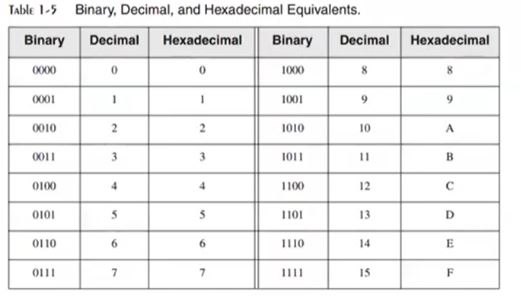
**Converting Unsigned Hexadecimal to Decimal**

dec = (Dn-1 X 16n-1) + ……. + (D0 X 160)

**Converting Unsigned Decimal to Hexadecimal**

To convert an unsigned decimal integer to hexadecimal, repeatedly divide the decimal value by 16 and retain each remainder as a hexadecimal digit. For example, the following table lists the steps when converting decimal 422 to hexadecimal:

**BINARY DECIMAL AND HEXADECIMAL EQUIVALENTS:**



***SIGNED CONVERSIONS***

**Signed Binary Integers**

For x86 processors, the MSB indicates the sign:

0 is positive and I is negative.

**Converting signed Binary into Decimal**

If no is -ve ie; MSB is 1. Take 2's comp of that no. then no will be tve. And then apply same method for unsigned conversion. when result has extracted, put -ve sign before.

• If the highest bit is a 1, the number is stored in two's-complement notation. Create its two's

• If the highest bit is a 0, you can convert it to decimal as if it were an unsigned binary integer.

For example, signed binary 11110000 has a l in the highest bit, indicating that it is a negative integer. First we create its two's complement, and then convert the result to decimal. Here are the steps in the process:

**Converting Signed decimal into Binary**

If no. is negative i.e. -14.  convert 14 into binary ie;  0010 1011 - Because original value was negative - we create 2's compliment of 00101011 i.e; 11010101 which is the result of 43.

**Converting Signed Decimal to Hexadecimal**

1. Convert the absolute value of the decimal integer to hexadecimal.

2. If the decimal integer was negative, create the two's complement of the hexadecimal number

**Converting Signed Hexadecimal to Decimal**

 1. If the hexadecimal integer is negative, create its two's complement; otherwise, retain the integer as is.

 2. Using the integer from the previous step, convert it to decimal. If the original value was negative, attach a minus sign to the beginning of the decimal integer.

**Hexadecimal 2's comp:**

Check 6A3D and 9503 is the of -ve. we took MSD 6 and a and convert 4 into binary no. - O110 and 1110 respectively so, 6A30 is the and 9503 is -ve.

If MSB is from 0-7, it will be +ve

If MSB is from 8-F, it will be -ve.

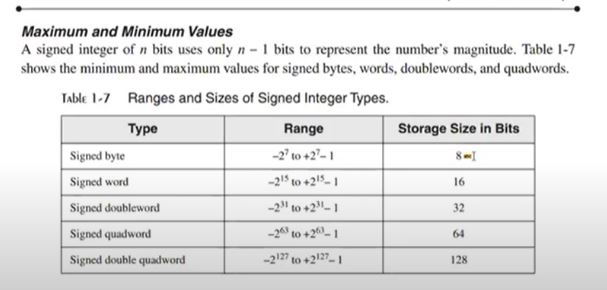
For 2's comp of Hexadecimal no. = subtract 15 from each No.

Then add 1 in LSB

 Number will be 2's compliment

**MAXIMUM AND MINIMUM VALUES**

**Ranges and size of signed integer types**



**Binary Addition.**

1+1=10

sometimes carry is generated out of highest bit position in this case we have to complete a supe and Ignore other values. That value will be registered in a carry flag register theans it will be unabled.

Integer storage sizes:

**Binary Subtraction.**

A simpler way to approach binary subtraction is to reverse the sign of the value being subtracted, and then add the two values.

Basic storge unit for all data is byte -8 bits

word (2 bytes)

double word (4 bytes)

quad word (8 bytes)

Double quadword (16 bytes)

Large Measurements

A number of large measurements are used when referring to both memory and disk space:

• One kilobyte is equal to 210, or 1024 bytes.

One megabyte (1 MByte) is equal to 220, or 1,048,576 bytes.

• One gigabyte (1 GByte) is equal to 230, or 1024', or 1,073,741,824 bytes.

• One terabyte (1 TByte) is equal to 240, or 1024", or 1,099,511,627,776 bytes.

• One petabyte is equal to 250, or 1,125,899,906,842,624 bytes.

• One exabyte is equal to 200 or 1,152,921,504,606,846,976 bytes.

• One zettabyte is equal to 270 bytes.

• One yottabyte is equal to 280 bytes.

CHAPTER 2

Topics to be covered

Memory Hierarchy

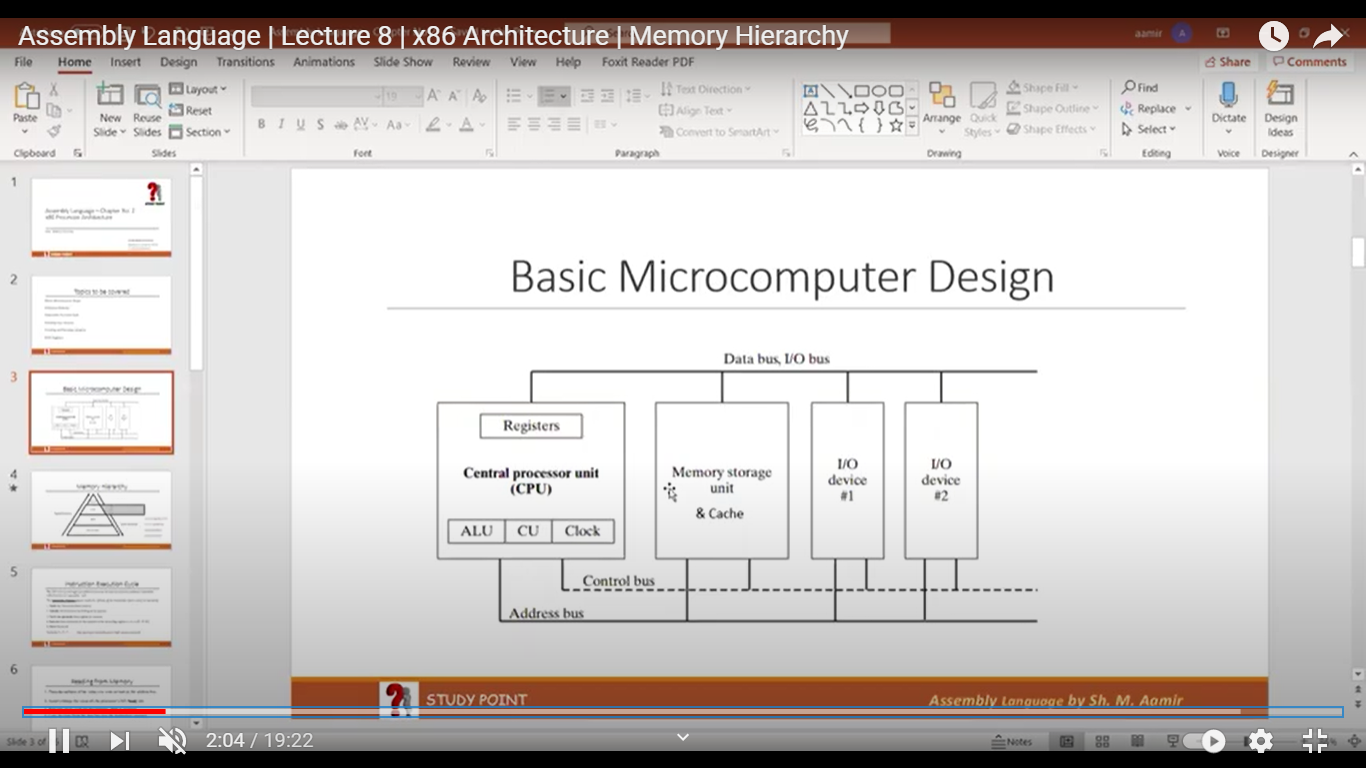
Instruction Execution Cycle

Reading from Memory

Loading and Executing a program

CPU Registers

Basic Microcomputer Design



**Registers**

Local memory of cpu or Internal memory of cup (Memory unit)

control unit

used to control all activities, components within computer

**clock**

measures speed of cpu (in Hertz)

Generate pulses & synchronize all components (at same level)

**cache**

faster and small in size

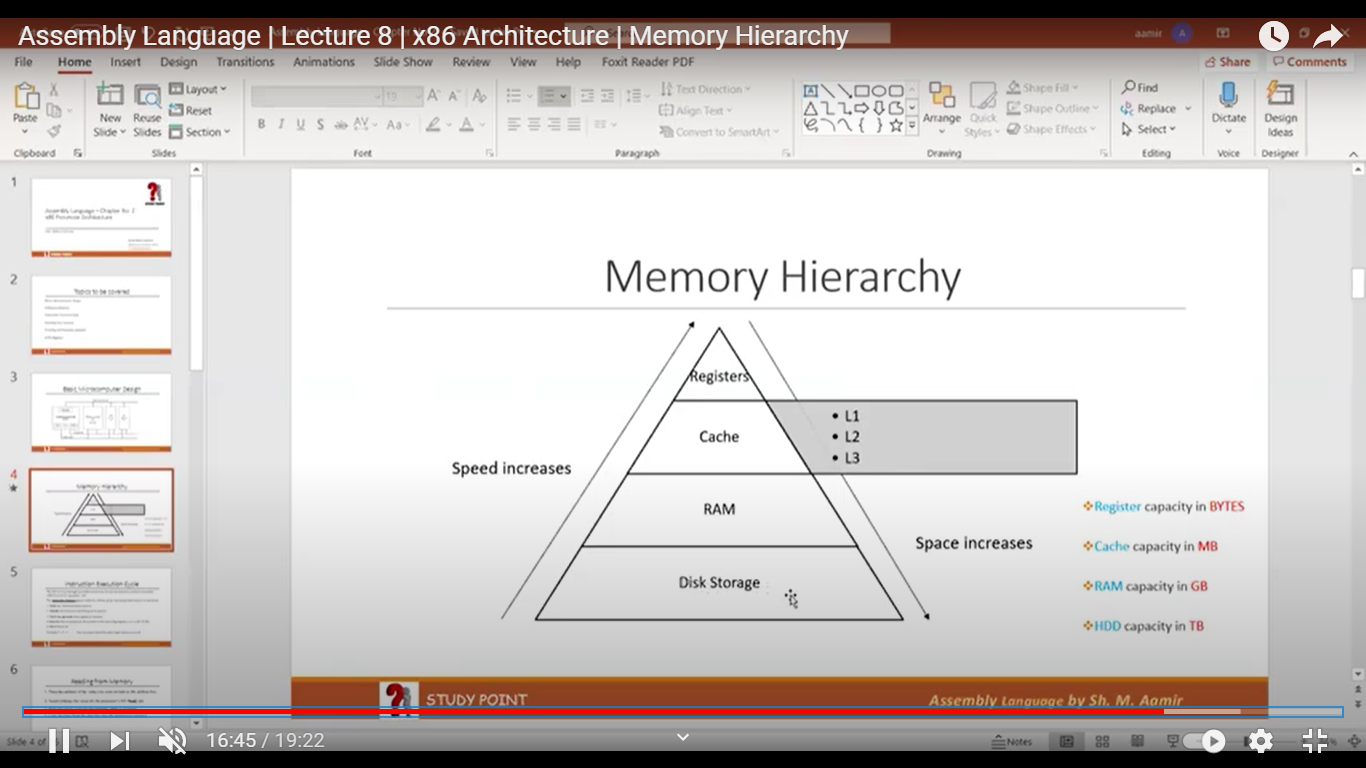
Temporary memory.

Divided in levels: 1, 2 and 3

close to RAM (space more speed low)

At first, CPU finds data in register - After that it sees further in cache if data found in cache then it is called to be a Cache hit and if it is not found, we called it cache miss

Memory Hierarchy



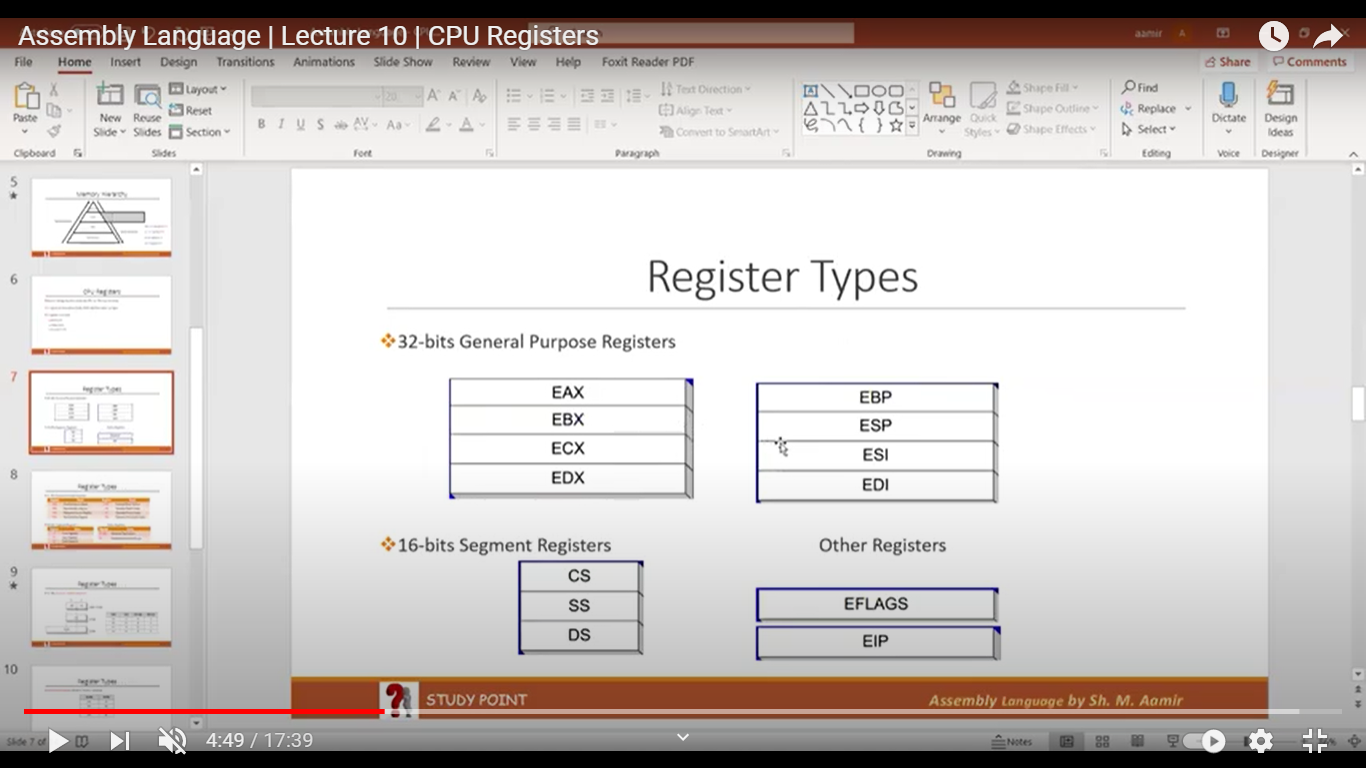
**CPU Registers**

Named storage location inside the CPU or CPU local memory

A register is faster than Cache, RAM and than other storages

A register may hold:

* An instruction
* Storage address
* Or any kind of data



**Register Types ...**

**General Purpose register (32 bit)**

**Segment Registers (16-bits)**

Segments are specific areas defined in a program for containing data, code and stack.

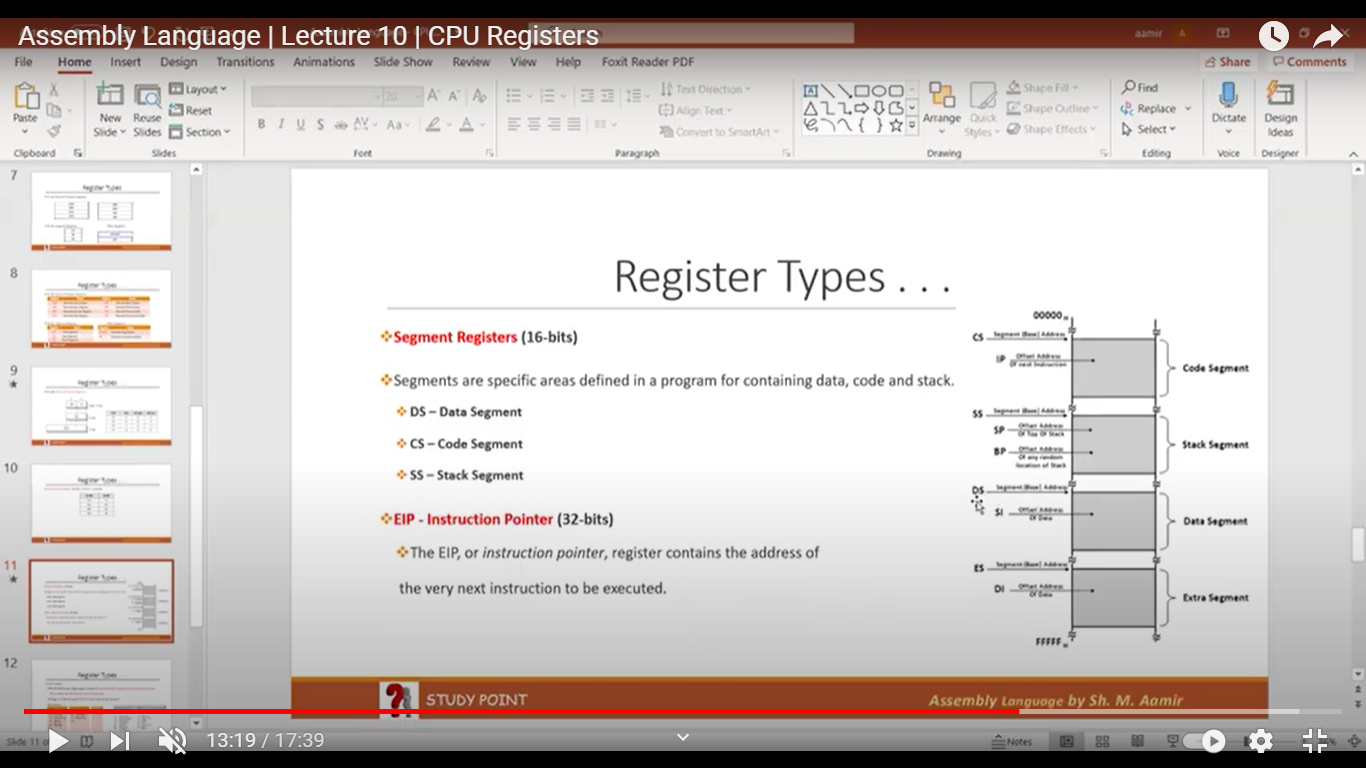
DS-Data Segment

CS - Code Segment

SS - Stack Segment

**EIP - Instruction Pointer (32-bits)**

The EIP, or instruction pointer, register contains the address of the very next instruction to be executed.



OTHER REG

**EFLAGS Register**

The EFLAGS (or just Flags) 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

It contains:

control Bits

      IF- Interrupt flag

       DF- Direct flag

Reserved Bits

Status Bits

     CF - Carry Flag

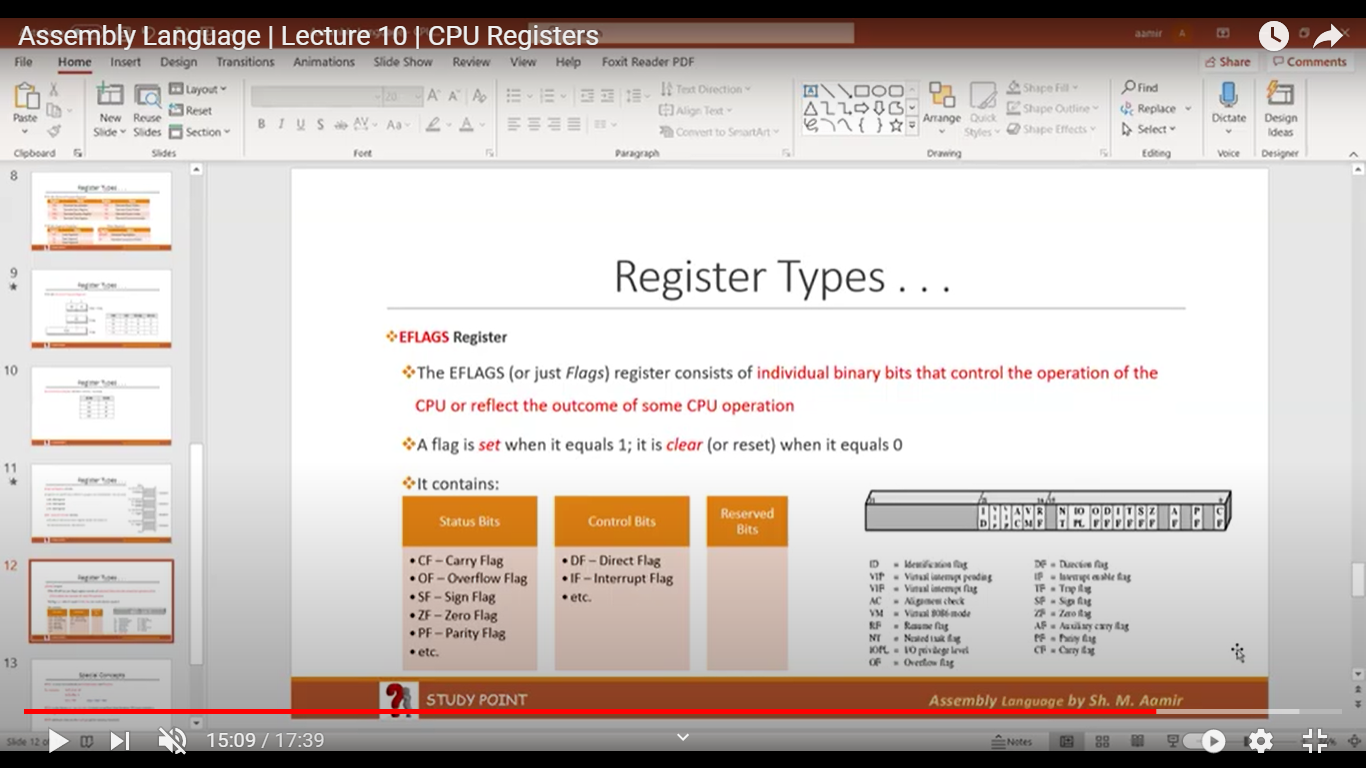
    OF - Overflow

     SF -sign flag

       ZF - Zero flag

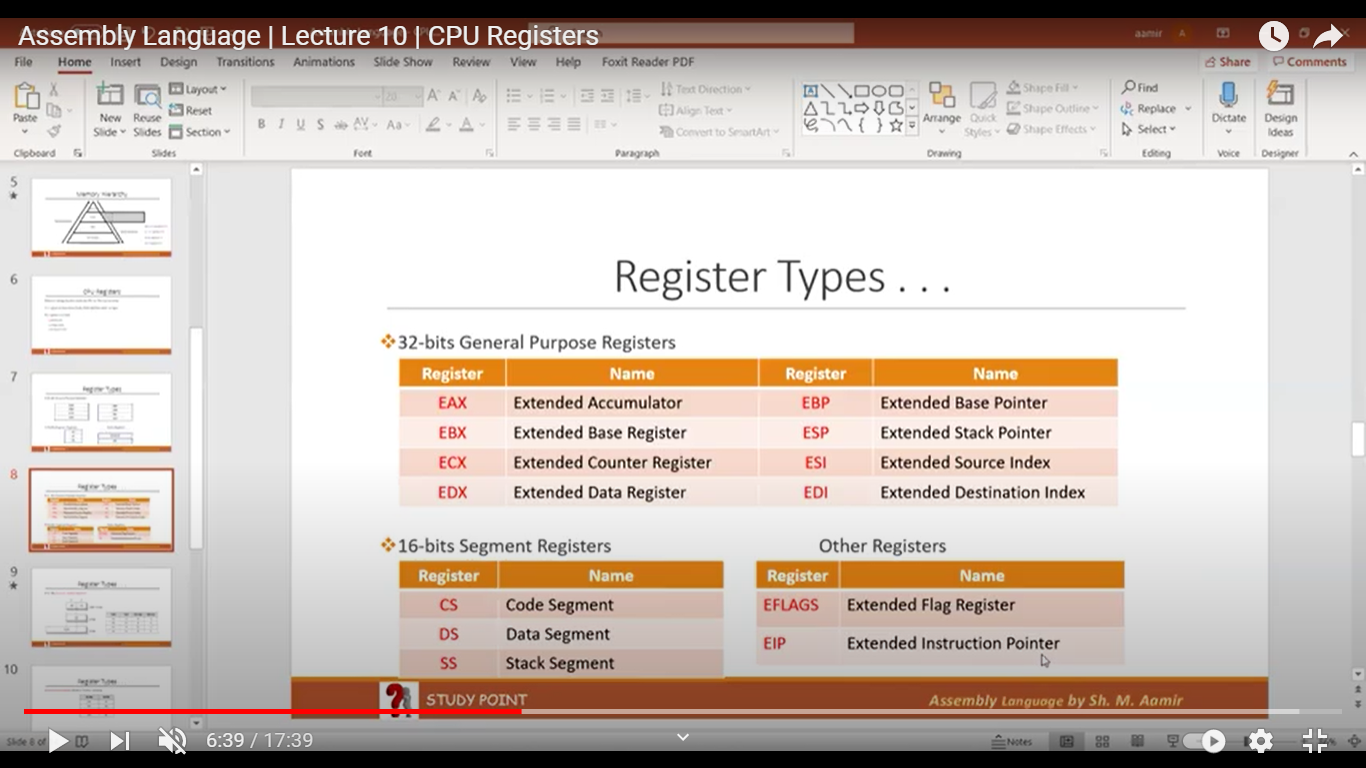
       PF - Parity flag

       DF - Direct flag

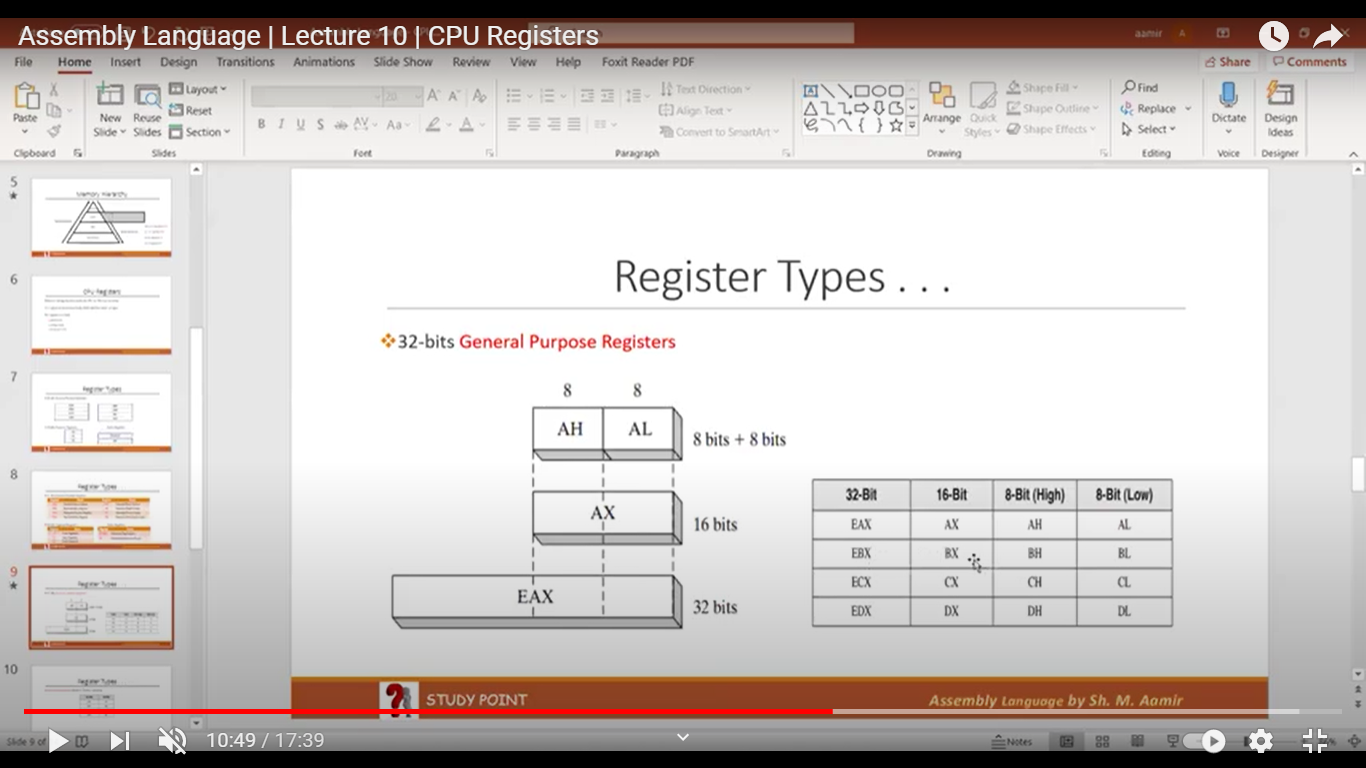


32-bit General Purpose register:

16-bits segments Register:



AH, AL, EAX, AX



Special Concepts

EAX is used automatically in Multiplication and Division

ECX is also known as Loop Counter. It means to perform loop iteration, ECX must contains a non-negative value (more than 0)

ESP addresses data on the stack (a system memory structure)

weight of a bit = base raised to power base

(weight is considered whose bit is on)

-ve data 2s compliment is considered as +ve data

+ve data 2s compliment is considered as -ve data

If signed \_ve value has to represent in memory then we have to represent in

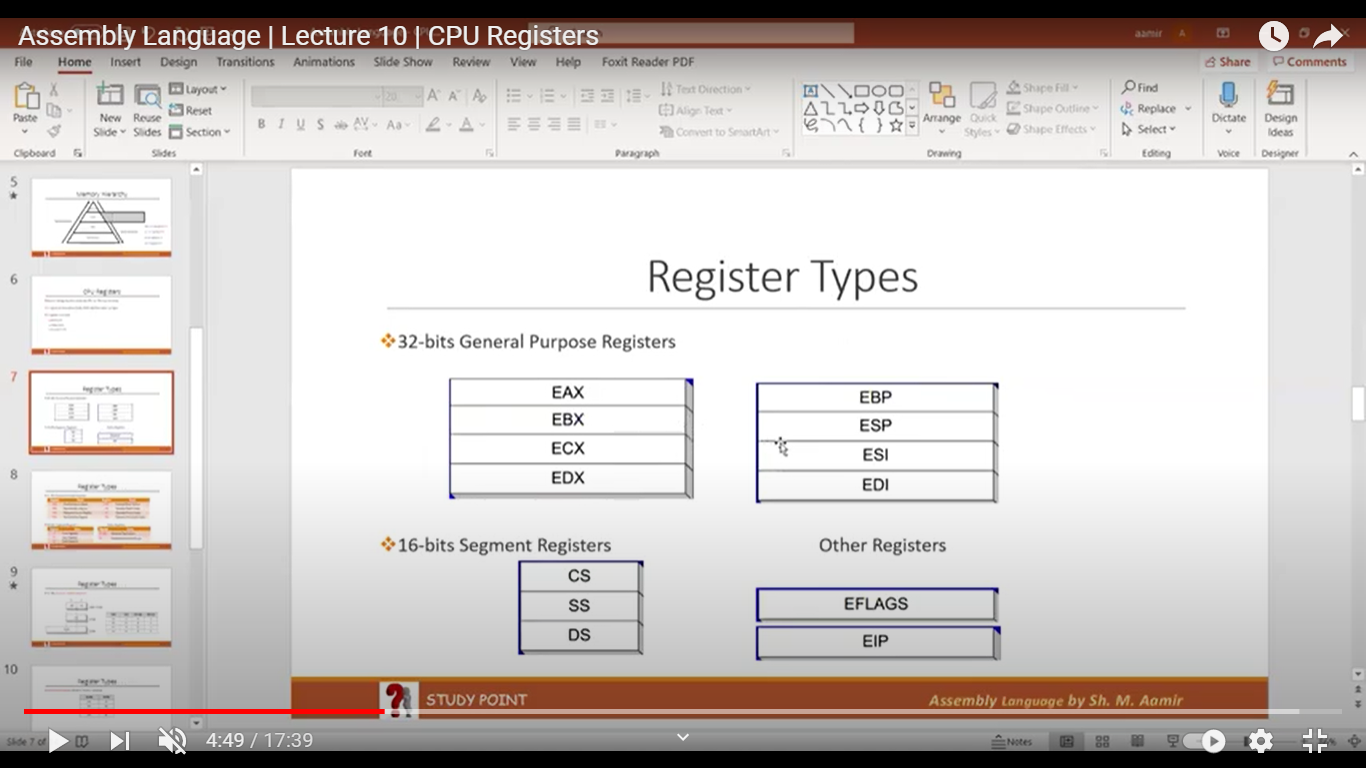
in memory 2's comp form

-

A bit's weight is doubled from its previous digit.

In case of signed bits data bits weight will be half

So in this case, signed MSB is ignored & weight will be a half



CHAPTER : 03

**INTEGER CONSTANT**

Basic Elements of Assembly Language,

* Integer constants
* Character and string constants
* Reserved words and identifiers
* Directives and instructions
* Labels
* Mnemonics and Operands
* Comments
* Integer Constant

Syntax:

          [[+, -}] digits (radix]

* binary, decimal, hexadecimal, or octal digits
* Common radix characters!
* h-hexadecimal
* d-decimal (Default)
* b-binary

Examples: 300, 6Ah, 42, 1101b, + 25, -25.

Hexadecimal beginning with letter: OASh

**Character and String Constant**

* Enclose character in single or double quotes .

Examples:

            'A', "A"

           ASCII character = 1 byte

* Enclose strings in single or double quotes.

Examples:

           "Aamir", 'Aamir

**Reserved Words**

* Reserved words are predefined words which have some special meanings
* Reserved word cannot be used as an identifier
* e.g Mnemonics, directives etc.

**Identifiers**

* Identifiers are name for a variable, procedure, or label etc
* 1-247 characters, including digits
* Space not allowed in an identifier
* Not case sensitive
* First character must be a letter, \_, @?, or $

**Directives**

* Commands that are executed by the assembler during compile time
  + Not part of the Intel Instruction Set
  + Used to declare code, data sections, select memory model, declare procedures, etc.
* Not case sensitive

**Instructions**

* Commands that are executed by the CPU during running time
* Use of Intel Instruction Set
* An instruction contains:
* Label         (Optional)
* Mnemonic     (Required)
* Operand     (depends on the instruction)
* Comment    (Optional)

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

**Labels**

* Act as place markers
* marks the address (offset) of code and data
* Follow identifier rules
* Data label

must be unique

example: myArray (not followed by colon)

* Code label

target of jump and loop instructions

example: L1: (followed by colon)

**Mnemonics and Operands**

* Instruction Mnemonics

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

* Operands

Constant / constant expression

Register

memory (data label)

* Constants and constant expressions are often called immediate values

**Comments**

* Comments are good!

To increase the readability

explain the program's purpose

tricky coding techniques

application-specific explanations

* Single-line comments

begin with semicolon (;)

* Multi-line comments

begin with COMMENT directive and a programmer chosen character.

end with the same programmer-chosen character

**Comments examples**

* For Single Line Comment

; This is my single line comment

* For Multiline Comment

Comment $

This is my first line.

This is my second line.

This is my third line.

**Instruction Format Examples**

* No operands

STCI ; set carry flag

* One operand

INC EAX ; Register

INC Mybyte ; Memory variable

* Two operands

ADD EBX, ECX ; Register, Register

SUB Mybyte, 25 ; Memory, Constant

ADD EAX, 36 \* 25 ; Register, Constant-expression

*3.2 Example: Adding and Subtracting Integers*

*3.2.1 The AddTwo Program*

 Let's revisit the AddTwo program we showed at the beginning of this chapter and add the necessary declarations to make it a fully operational program. Remember, the line numbers are not really part of the program:

1: Addto.am - adds two 32-bit integers

2: Chapter 3 example

3:

4: 386

5: model flat,stdcall

6: .stack 4096

7: Exit Process PROTO, dwExitCode: DWORD

8:

9: .code

10: main PROC

11: moveax. 5 ; move 5 to the eax register

12: add a x, 6 ; add 6 to the eax register

13:

14: INVOKE Exit Process, o

15; main ENDP

16; END main

Line 4 contains the 386 directive, which identifies this as a 32-bit program that can access 32-bit registers and addresses. Line 5 selects the program's memory model (fonts), and identifies the calling convention named ideal for procedures. We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how stdcall works.) Line 6 sets aside 4096 bytes of storage for the intime stack, which for

*3.4 Defining Data*

*3.4.1 Intrinsic Data Types*

The assembler recognizes a basic set of intrinsic data types, which describe types in terms of their size by te word, doubleword, and so on. whether they are signed and whether they are integers or reals There's a fair amount of overlap in these types--for example, the DWORD type (32-bit, unsigned integer) is interchangeable with the SDWORD type (32-bit signed inte ger). You might say that programmers use SDWORD to communicate to readers that a value will contain a sign, but there is no enforcement by the assembler. The assembler only evaluates the sizes of operands. So, for example, you can only assign variables of type DWORD, SDWORD, or REALA to a 32-bit integer Table 3-2 contains a list of all the intrinsic data types. The notation IEEE in some of the table entries refers to standard real number formats published by the IEEE Computer Society

3.4.2 Data Definition Statement

A data definition statement sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types (Table 3-2). A data definition has the following syntax:

[name) directive initializer I. Initializer)...

Intrinsic Data Types.

|  |  |
| --- | --- |
| **BYTE** | 8-bit unsigned integer stands for byte |
| **SBYTE** | 8 hit signed integer. S stands for signed |
| **WORD** | 16-bit unsigned integer. |
| **SWORD** | 16-bit signed integer. |
| **DWORD** | 32-bit unsigned integer. D stands for double |
| **SDWORD** | 32-bit signed integer. SD stands for signed double |
| **FWORD** | 48-bit integer (Far pointer in protected mode) |
| **QWORD** | 6-bitnego. Q stands for qual |
| **TBYTE** | 80 hit (10 byte) integer T stands for Ten-byte |
|  |  |
|  |  |
|  |  |