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)

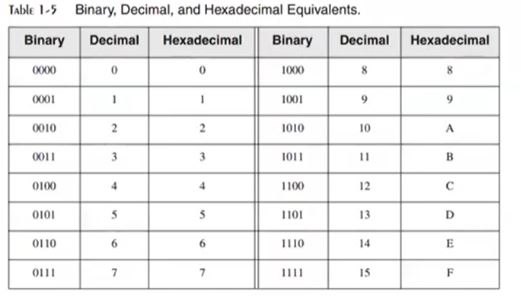
**Converting Unsigned Hexadecimal to Decimal**

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.

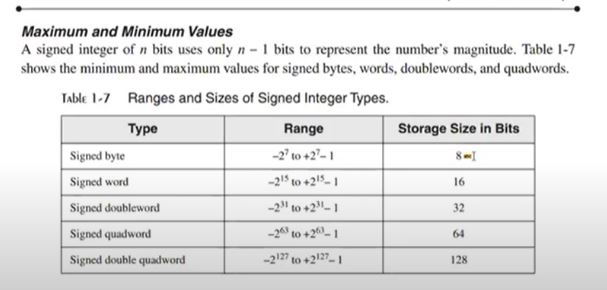
For 2's comp of Hexadecimal no. = subtract 15 from each No.

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

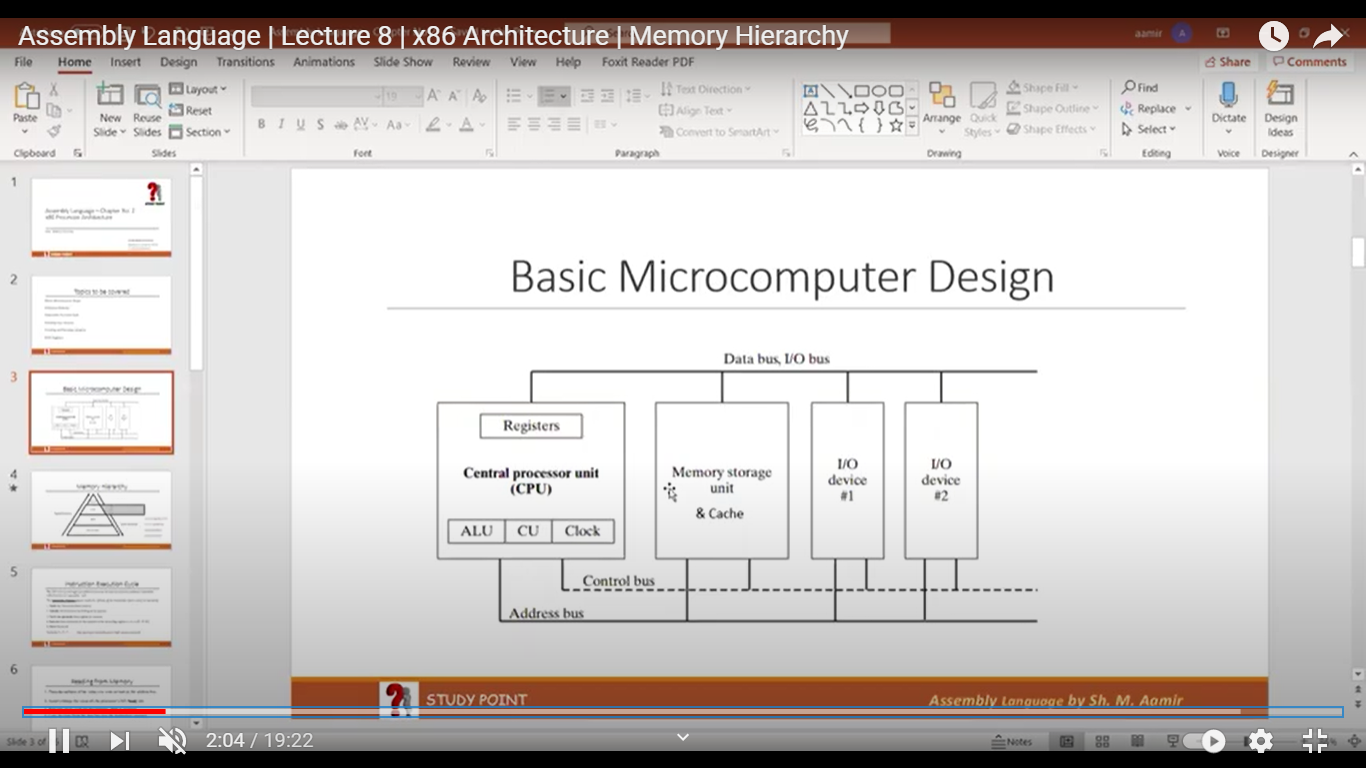
Instruction Execution Cycle

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

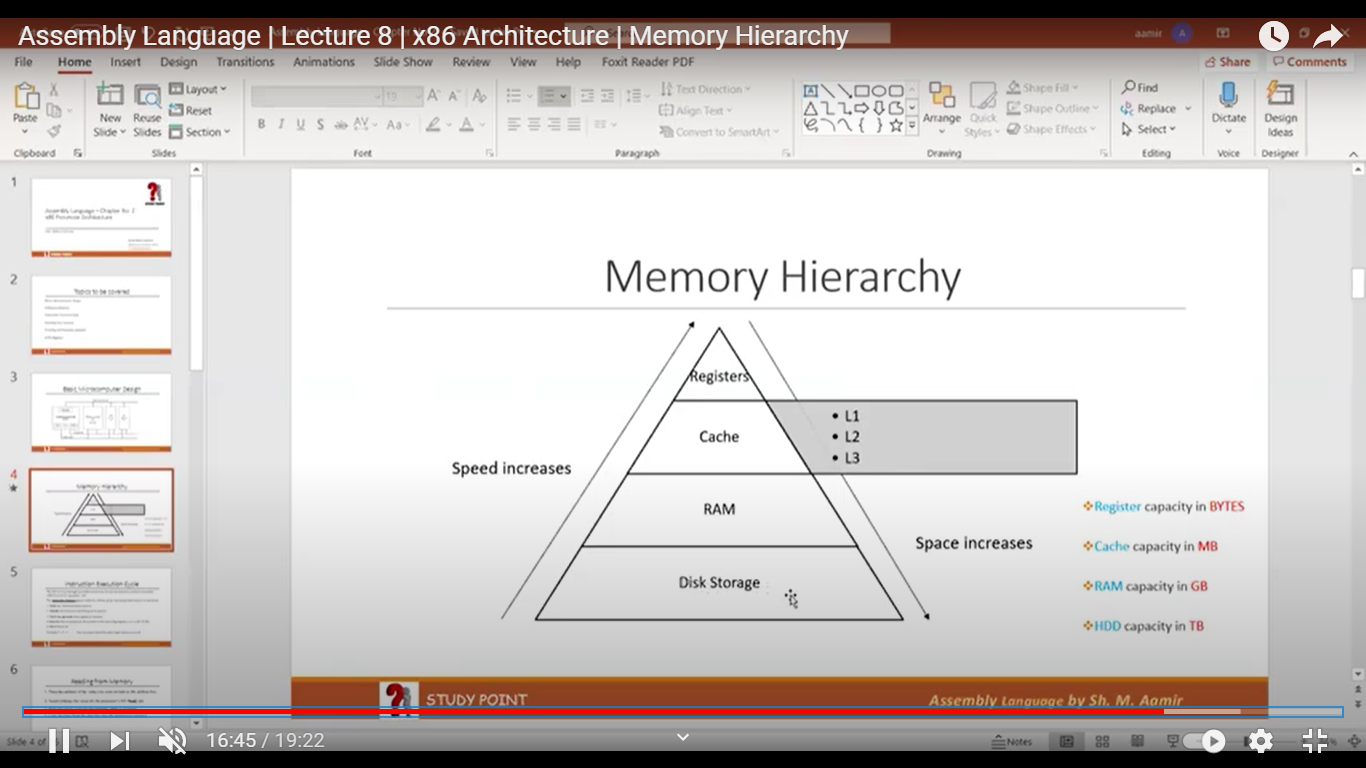
Temporary memory.

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



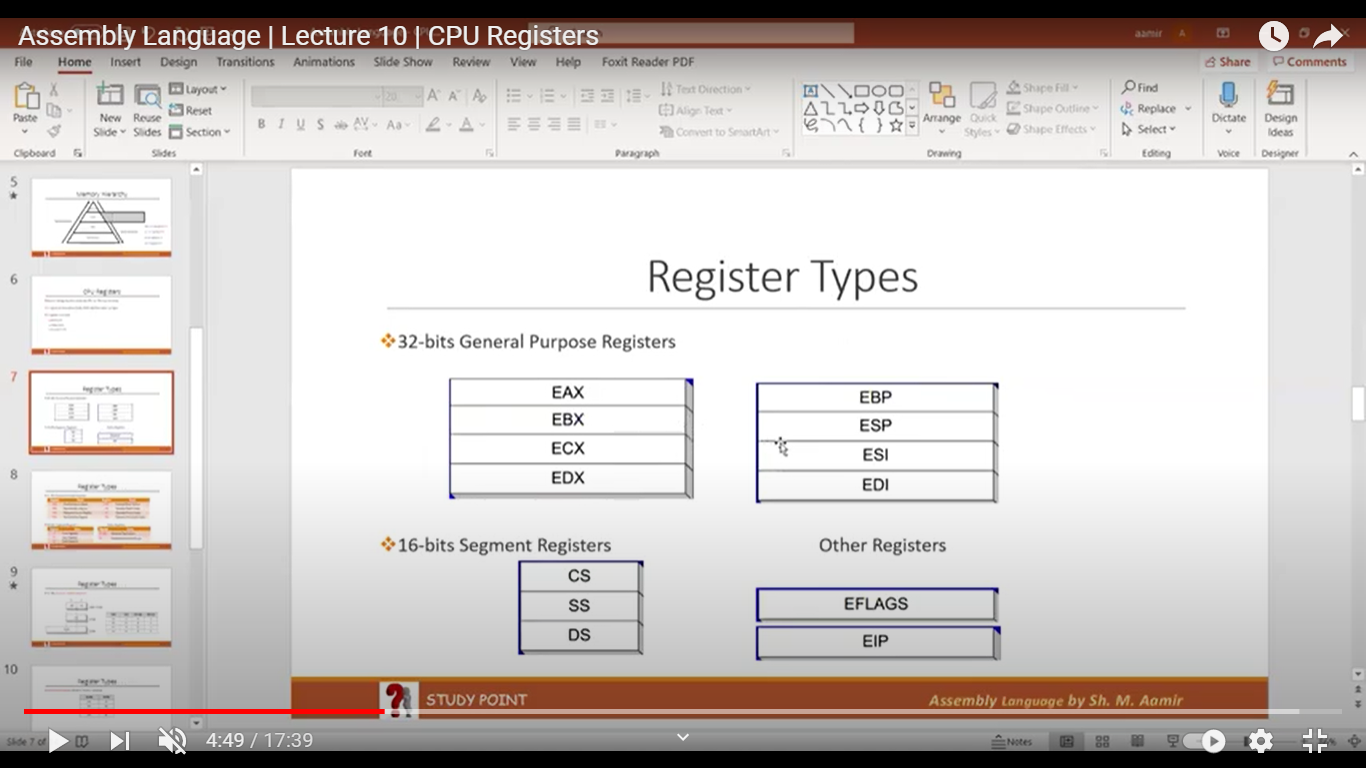
**CPU Registers**

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.

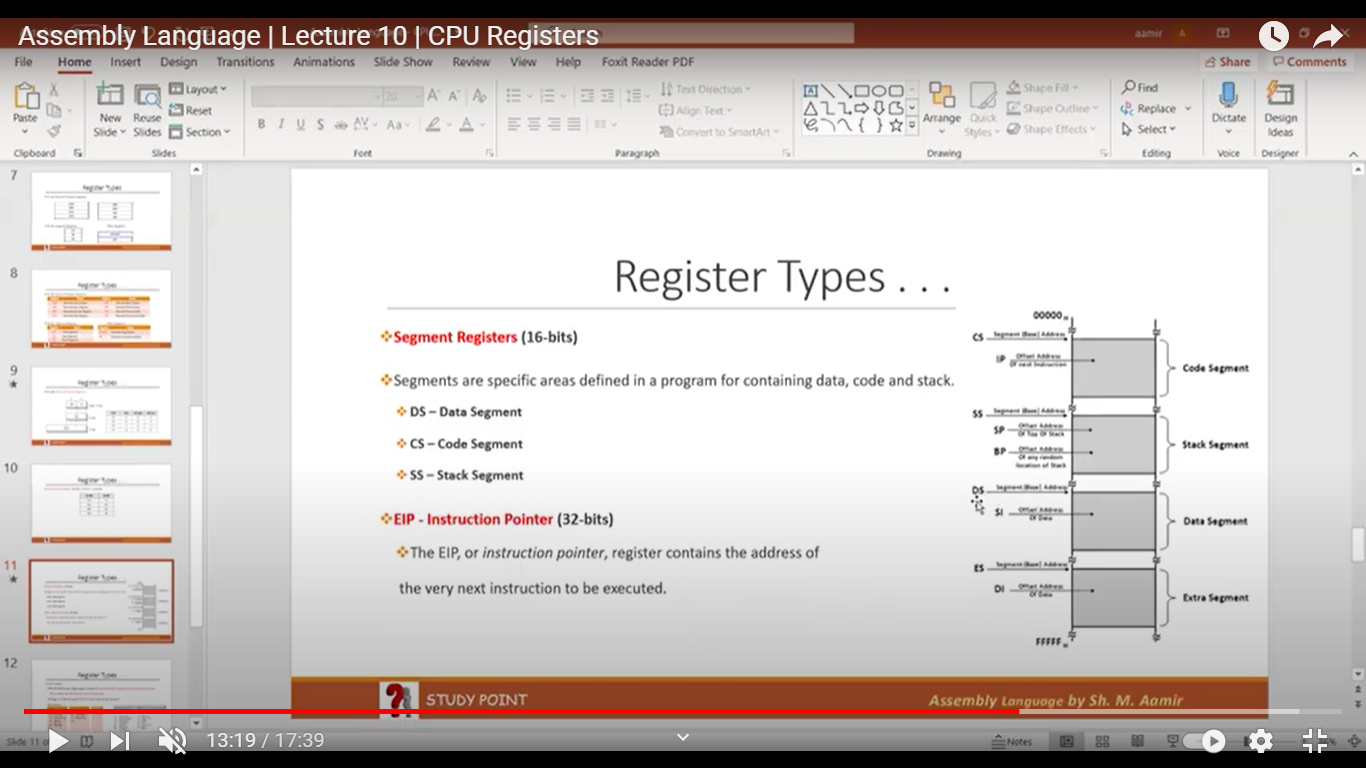
DS-Data Segment

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

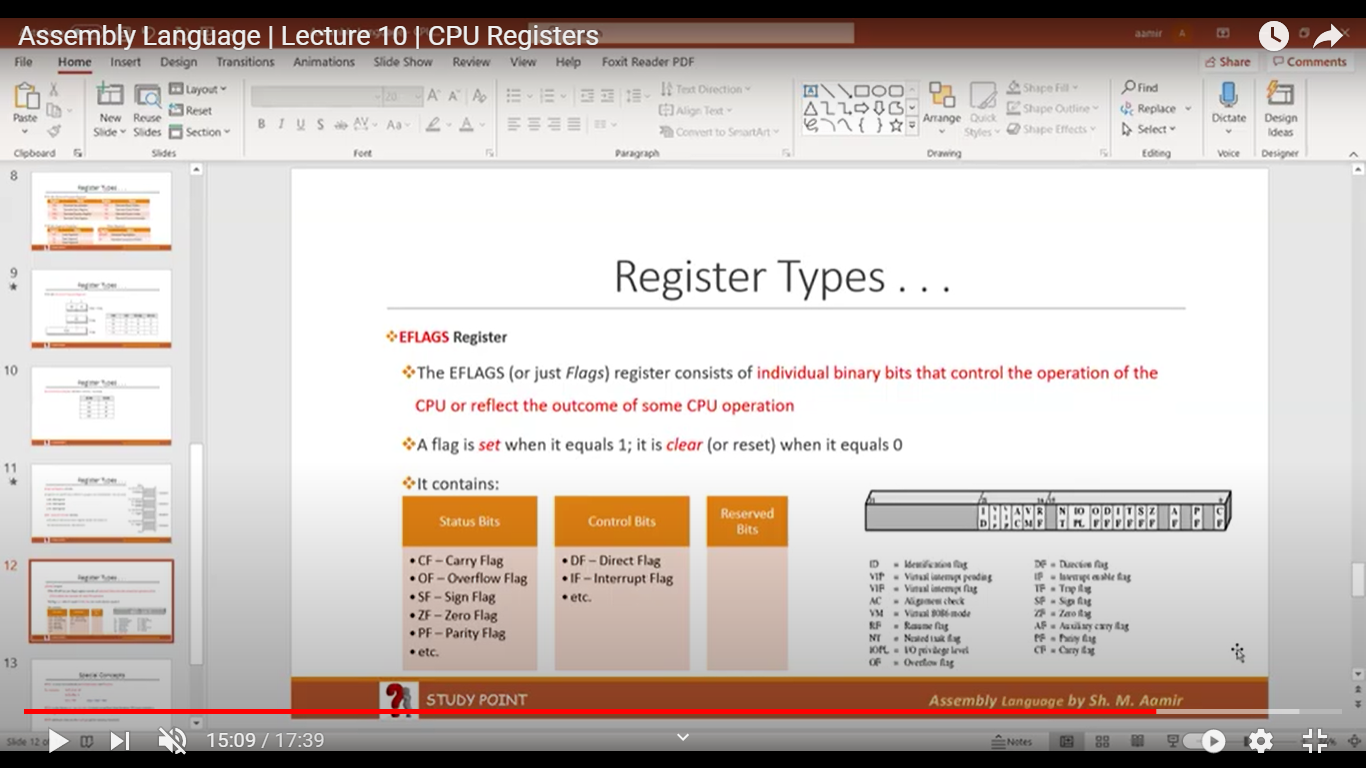
    OF - Overflow

     SF -sign flag

       ZF - Zero flag

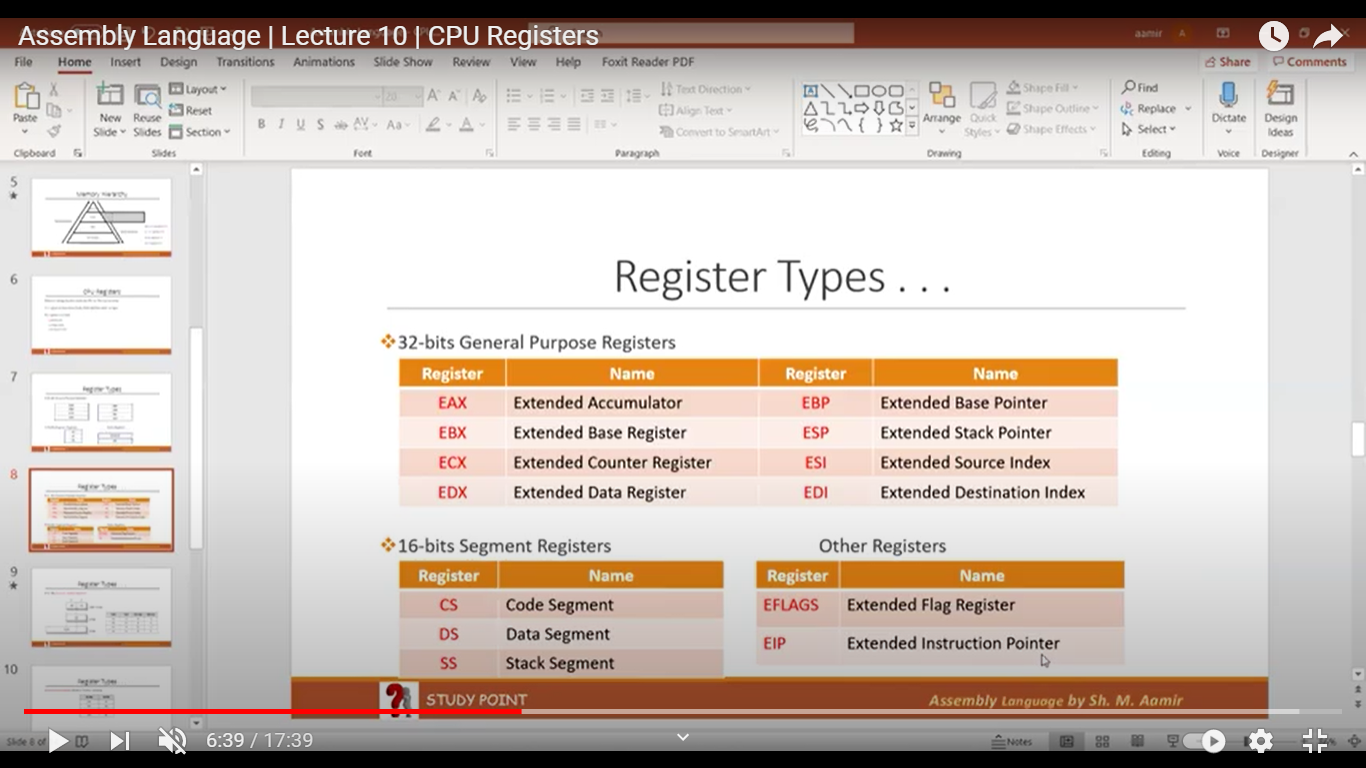
       PF - Parity flag

       DF - Direct flag

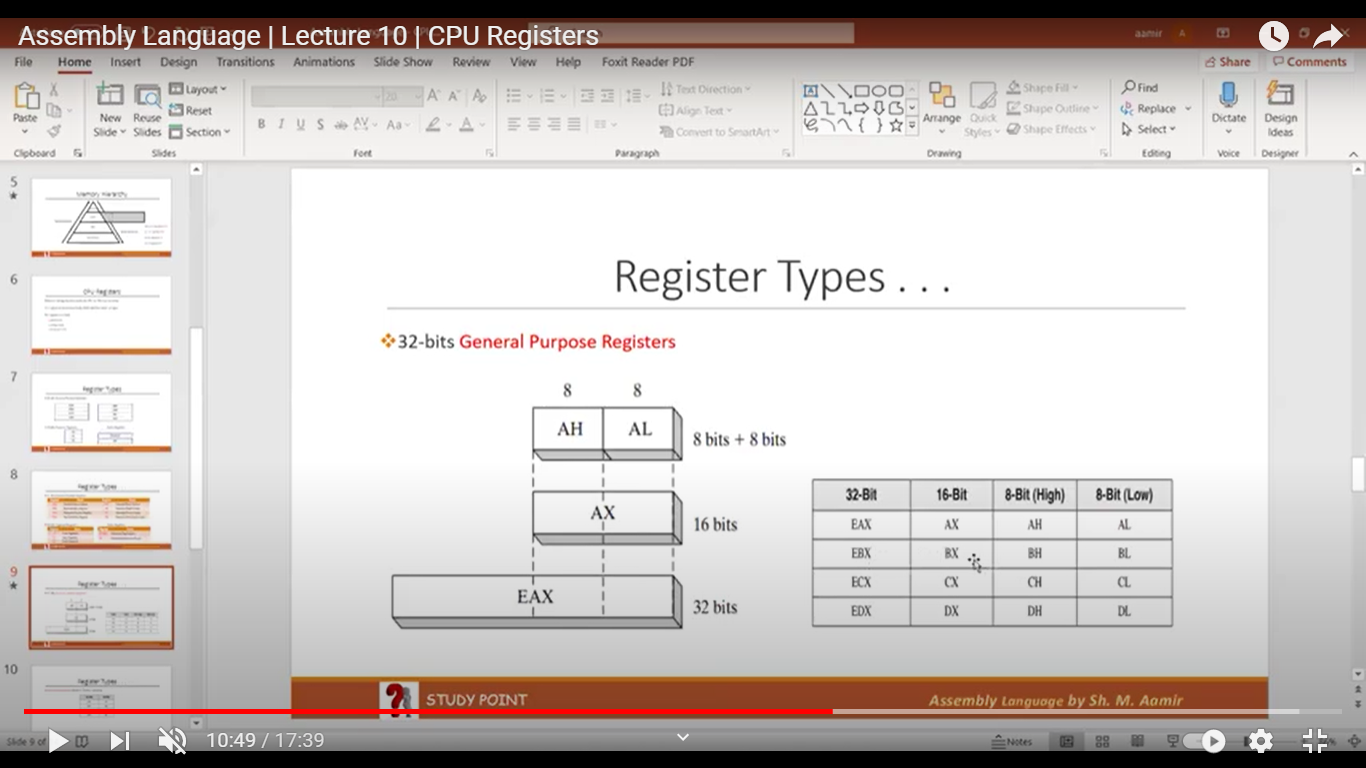


32-bit General Purpose register:

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

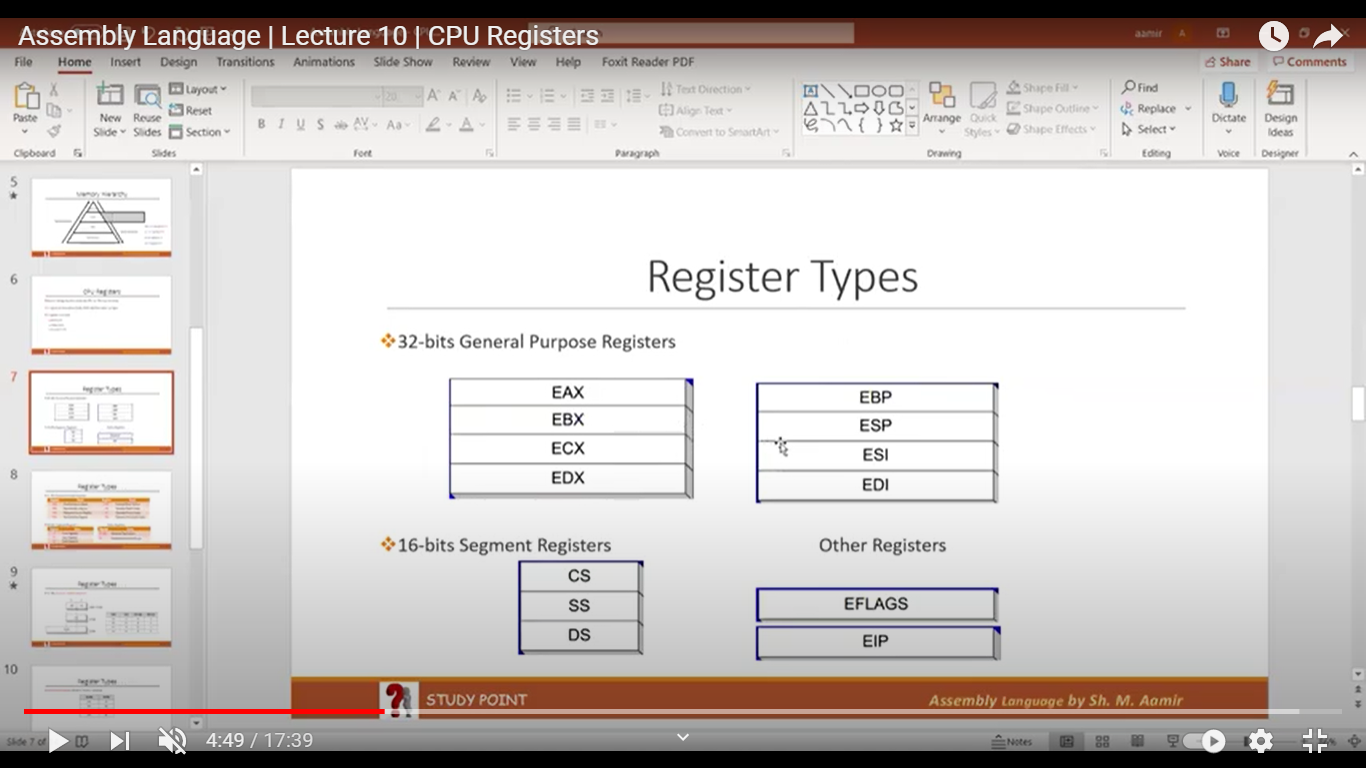
in memory 2's comp form

-

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 |
|  |  |
|  |  |
|  |  |

**Table 3-3 Legacy Data Directives.**

Directive Usage

DB 8-bit integer

DW 16-bit integer

DD 32-bit integer or real

DQ 64-bit integer or real

DT define 80-bit (10-byte) integer

**Defining Data**

• Intrinsic Data Types

• Data Definition Statement

• Defining BYTE and SBYTE Data

• Defining WORD and SWORD Data

• Defining DWORD and SDWORD Data

• Defining QWORD Data

• Defining TBYTE Data

• Defining Real Number Data

• Little Endian Order

• Adding Variables to the AddSub Program

• Declaring Uninitialized Data

Intrinsic Data Types (1 of 2)

• BYTE, SBYTE

• 8-bit unsigned integer; 8-bit signed integer

• WORD, SWORD

. 16-bit unsigned & signed integer

• DWORD, SDWORD

• 32-bit unsigned & signed integer

• QWORD

• 64-bit integer

• TBYTE

• 80-bit integer

**Intrinsic Data Types (2 of 2)**

REAL4

• 4-byte IEEE short real

• REAL8

• 8-byte IEEE long real

• REAL 10

• 10-byte IEEE extended real

**Data Definition Statement**

•

A data definition statement sets aside storage in memory for a variable May optionally assign a name (label) to the data

Syntax: [name] directive initializer (initializer] ...

E.G; value11 BYTE 10

• All initializers become binary data in memory

**Defining BYTE and SBYTE Data,**

Each of the following defines a single byte of storage:

value1 BYTE 'A’ ; character constant

value2 BYTE O smallest unsigned byte

value3 BYTE 255 ; largest unsigned byte

value4 SBYTE -128 ; smallest signed byte

value5 SBYTE +127 ; largest signed by te

value6 BYTE ? ; uninitialized byte

• MASM does not prevent you from initializing a BYTE with a

negative value, but it's considered poor style. • If you declare a SBYTE variable, the Microsoft debugger will,

automatically display its value in decimal with a leading sign.

7th slide missing

**Defining Strings (1 of 3)**

• A string is implemented as an array of characters

. For convenience, it is usually enclosed in quotation marks

• It often will be null-terminated • Examples:

stri BYTE "Enter your name. 0 str2 BYTE 'Error: halting program', 0 str3 BYTE 'A', 'E','I', '0', 'U' greeting BYTE "Welcome to the Encryption Demo program

BYTE "created by Kip Irvine.",0

**Defining Strings (3 of 3)**

• End-of-line character sequence:

• ODh = carriage return • 0Ah = line feed

stri BYTE "Enter your name: ", ODh,0Ah

BYTE "Enter your address: ",0

newLine BYTE ODh,0Ah,0

Idea: Define all strings used by your program in the same area of the data segment.

**Using the DUP Operator**

• Use DUP to allocate (create space for) an array or

string. Syntax: counter DUP ( argument) • Counter and argument must be constants or constant

expressions

var1 BYTE 20 DUP (0) var2 BYTE 20 DUP (?) var3 BYTE 4 DUP ("STACK") var4 BYTE 10,3 DUP(0), 20

; 20 bytes, all equal to zero ; 20 bytes, uninitialized ; 20 bytes: "STACKSTACKSTACKSTACK" ; 5 bytes

**Defining DWORD and SDWORD Data**

Storage definitions for signed and unsigned 32-bit integers:

vali DWORD 12345678h val2 SDWORD -2147483648 val3 DWORD 20 DUP (?) val4 SDWORD -3,-2,-1,0,1

; unsigned ; signed ; unsigned array ; signed array

12 slide missing

**Symbolic Constants**

• Equal-Sign Directive

• Calculating the Sizes of Arrays and Strings

• EQU Directive

• TEXTEQU Directive

14 Slide missing

**Calculating the Size of a Byte Array**

current location counter: $

• subtract address of list

• difference is the number of bytes

list BYTE 10,20,30,40 ListSize = ($ - list)

16 missing

**Calculating the Size of a Doubleword Array**

Divide total number of bytes by 4 (the size of a doubleword)

list DWORD 1,2,3,4 ListSize = ($ - list) / 4

18 missing

**TEXTEQU Directive**

• Define a symbol as either an integer or text expression, • Called a text macro • Can be redefined

continueMsg TEXTEQU <"Do you wish to continue (Y/N)?"> rowSize = 5 .data prompt1 BYTE continueMsg count TEXTEQU (rowsize \* 2) ; evaluates the expression setupAL TEXTEQU <mov al, count>

.code setupAL

i generates: "mov al, 10"

**4.1.2 Operand Types** Chapter 3 introduced x86 instruction formats:

[label:) mnemonic (operands] ; comment Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

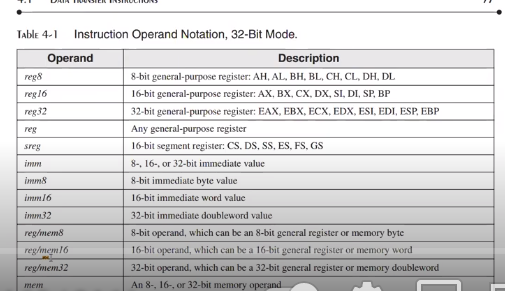
mnemonic mnemonic (destination) mnemonic (destination) [source]

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



21 missing

**4.1.4 MOV Instruction** The MOV instruction copies data from a source operand to a destination operand. Known as a data transfer instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

MOV destination, source

The destination operand's contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

dest = source;

In nearly all assembly language instructions, the left-hand operand is the destination and the righthand operand is the source, MOV is very flexible in its use of operands, as long as the following rules are observed: • Both operands must be the same size.

Both operands cannot be memory operands. • The instruction pointer register (IP. EIP, or RIP) cannot be a destination operand

23 missing

**MOV Instruction**

MOV instruction is used to copy data from source operand to destination operand Syntax:

MOV Destination\_Operand, Source\_Operand Example:

MOV EAX, 12345678H Rules:

Both operands must be of same sizes Both operands must not be memory variables EIP/IP/immediate value can't be used as a destination operand

**List of standard MOV Instructions**

MOV REG, REG

(register to register)

MOV REG, MEM

(memory variable to register)

• MOV REG, IMM

(immediate value to register)

• MOV MEM, REG

(register to memory variable)

MOV MEM, IMM

(immediate value to memory variable)

**MOV with Zero Extension Instruction**

**(MOVZX)**

MOVZX instruction copies the source operand value (with zero extension) into destination operand if source operand size is smaller than destination operand size.

MOVZX is only used with un-signed data

Syntax:

MOVZX Destination Operand, Source Operand Like

MOVZX reg32, reg/mem8

MOVZX reg32, reg/mem16

MOVZX reg16, reg/mem8

**MOV with Sign Bit Extension Instruction**

**(MOVSX)**

MOVSX instruction copies the source operand value (with sign bit extension) into destination operand if source operand size is smaller than destination operand size.

Calibri ( - 19 -A A MOVSXI B IUEA- d data - MSB is O if the value is +ve • MSB is if the value is -ve • Negative number is represented in memory as two's compliment of original positive number

Syntax

MOVSX Destination\_Operand, Source\_Operand Like

MOVSX reg32, reg/mem8 MOVSX reg32, reg/mem16

MOVSX reg16, reg/mem8

**Data related Operators and Directives**

Operators and directives are not executable instructions but they are interpreted by Assembler

Some Operators are :

OFFSET PTR TYPE LENGTHOF SIZEOF

**OFFSET OPERATOR**

OFFSET operator returns the offset address of data label

OFFSET presents the distance of data label from the beginning of data

segment

A variable named myByte.

Offset

Data segment:

myByte

30 missing

**TYPE OPERATOR**

The TYPE operator returns the size in no. of bytes, of a single element of a variable.

.data

X BYTE? Y'WORD? Z DWORD? ARR WORD 10, 20, 30, 40

.code

MOV AL, TYPE X MOV AL, TYPE Y MOV AL, TYPE Z MOV AL, TYPE ARR

; AL = 1 ; AL = 2 ; AL = 4 ; AL = 2

32 missing

**SIZEOF OPERATOR**

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

In other words, total size in bytes of an array

.data

ARRAY DWORD

10, 20, 30, 40, 50

.code

MOV AL, SIZEOF ARRAY

; AL = LENGTHOF \* TYPE ; AL = 5 \* 4 = 20

**4.4 Indirect Addressing** Direct addressing is rarely used for array processing because it is impractical to use constant off sets to address more than a few array elements. Instead, we use a register as a pointer (called indirect addressing) and manipulate the register's value. When an operand uses indirect address ing. It is called an indirect operand.

**4.4.1 Indirect Operands**

Protected Mode An indirect operand can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) surrounded by brackets. The register is assumed to contain the address of some data. In the next example, ESI contains the offset of byte Val. The MOV instruction uses the indirect operand as the source, the offset in ESI is dereferenced, and a byte is moved to AL:

.data byteval BYTE 10h .code mov esi,OFFSET byteval mov a1, fesil

AL

**4.4.3 Indexed Operands** An indexed operand adds a constant to a register to generate an effective address. Any of the 32-bit general purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

constant [regl Iconstant reg

**4.4.4 Pointers** A variable containing the address of another variable is called a pointer. Pointers are a great tool for manipulating arrays and data structures because the address they hold can be modified at runtime You might use a system call to allocate (reserve) a block of memory, for example, and save the address of that block in a variable. A pointer's size is affected by the processor's current mode (32-bit or 64-bit). In the following 32-bit code example, ptr contains the offset of array:

.data arrayb byte 10h, 20h, 30h, 40h

ptr dword arrays Optionally, you can declare ptr with the OFFSET operator to make the relationship clearer:

ptrB dword OFFSET arrays

JMP & LOOP Instructions

**JMP instruction**

The JMP instruction causes an unconditional transfer to a destination label, identified by a code label that is translated by the assembler into an offset.

Address of destination label is placed in EIP register Syntax:

JMP LABEL\_NAME Example:

disp:

mov eax, 100 call writedec jmp disp

**JMP & LOOP Instructions**

LOOP instruction

LOOP instruction is used to execute a statement or set of statements for a specific number of times (no. of iteration value is placed in ECX register before using LOOP instruction). ECX is automatically used as loop counter Syntax:

LOOP LABEL\_NAME Working:

ECX is decremented by 1 automatically when control approaches to LOOP instruction Compare the value of ECX with 0 (zero) if the value of ECX 0 then control is transferred to destination label and iteration is performed again Otherwise loop will be terminated and control is transferred to the next statement of LOOP instruction

CHAPTER NO. 04

**Chapter Overview**

• Stack Operations • Defining and Using Procedures • Linking to an External Library • The Irvine32 Library • 64-Bit Assembly Programming

**Stack Operations**

• Runtime Stack • PUSH Operation • POP Operation • PUSH and POP Instructions • Using PUSH and POP • Example: Reversing a String • Related Instructions

**Runtime Stack**

• Managed by the CPU, using two registers

• SS (stack segment)

• ESP (stack pointer) \*

**POP Operation**

• Copies value at stack[ESP Jinto a register or variable, • Adds n to ESP, where n is either 2 or 4.

value of n depends on the attribute of the operand receiving the data

**PUSH and POP Instructions**

• PUSH syntax

• PUSH r/m16 • PUSH /m32 • PUSH imm32 POP syntax • POP r/

m16 • POP r/

m32

**Using PUSH and POP**

Save and restore registers when they contain important values, PUSH and POP instructions occur in the opposite order.

; push registers

push esi push ecx push obx

; display some memory

movesi, OFFSET dwordval mov ecx, LENGTHOF dwordval mov ebx, TYPE dwordval call Dump Mem

; restore registers

pop ebx popecx pop esi

**Example: Reversing a String**

• Use a loop with indexed addressing • Push each character on the stack

Start at the beginning of the string, pop the stack in reverse

order, insert each character back into the string • Source code

**. Q: Why must each character be put in EAX before it is pushed?**

Because only word (16-bit) or doubleword (32-bit) values can be pushed on the stack.

Related Instructions

• PUSHFD and POPFD

• push and pop the EFLAGS register • PUSHAD pushes the 32-bit general-purpose registers on the stack • order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI POPAD pops the same registers off the stack in reverse order • PUSHA and POPA do the same for 16-bit registers

**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

**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.

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

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

**Local and Global Labels**

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

Procedure Parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

ArraySum PROC

mov esi, 0 mov bax, mov ecx, LENGTHOF myarray

; array index

set the sum to zero set number of elements

Li: add eax, myArray (esi]

add esi, 4 loop L1

; add each integer to sum ; point to next integer ; repeat for array size

store the sum

mov theSum, eax

ret ArraySum ENDP

What if you wanted to calculate the sum of two or three arrays within the same program?

**USES Operator** • Lists the registers that will be preserved

ArraySum PROC USES esi ecx

mov bax,0 eto

; set the sum to zero

MASM generates the code shown in gold:

ArraySum PROC

push esi push ecx

pop ecx popesi

ret ArraySum ENDP

**When not to push a register**

The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4)