CSE 331 L-1 : Introduction to Assembly Language.

Introduction:

In this session, you will be introduced to assembly language programming and to the emu8086 emulator software. emu 8086 will be used as both an editor and as an assembler for all your assembly language programming.

Step required to run an assembly program:

1. Write the necessary assembly source code
2. Save the assembly source code
3. Compile/Assemble source code to create machine code
4. Emulate/Run the machine code.

First; familirize yourself with the software before you begin to write any code. Follow the in-class instructions regarding the layout of emu8086.

Microcontrollers VS. Microprocessors

- A microprocessor is a CPU on a single chip
- If a microprocessor, its associate support

Circuitry, memory, and peripheral I/O components are implemented on a single chip, it is a microcontroller.

features of 8086

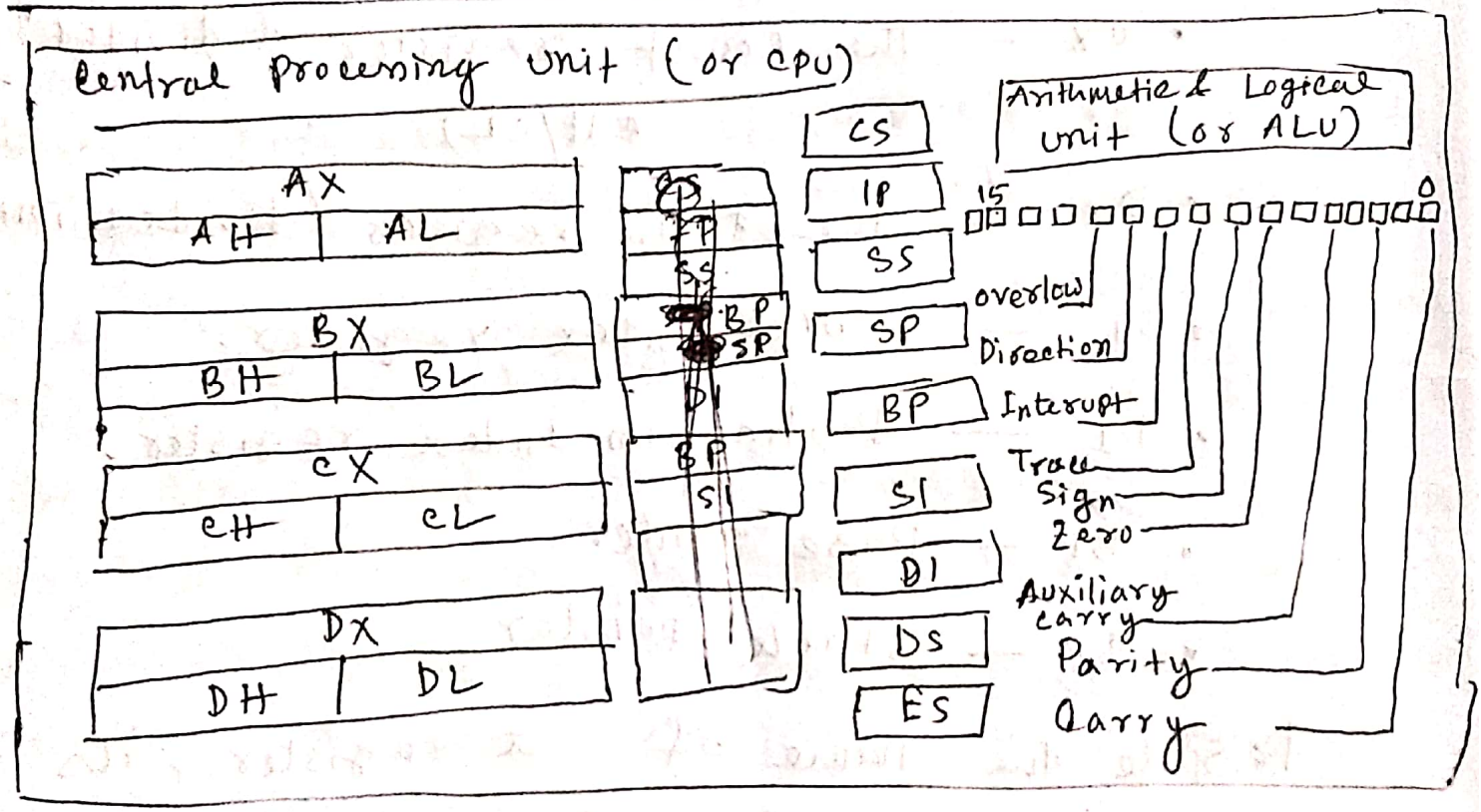
- 8086 is a 16 bit processor, Its ALU, internal registers work with 16 bit binary word.
- 8086 has a 16 bit data bus. It can read or write data to a memory/port either 16bits or 8 bits at a time.
- 8086 has 20 bit address bus which means it can address up to 2^{20} = 1 MB memory location.

Registerar - Register - Resistor

- Both ALU & FPU have a very small amount of super-fast private memory placed right next to them for their exclusive use. These are called registers.
- The ALU & FPU store intermediate and final result from their calculations in these registers

- Processed data goes back to the data Cache and then to the main memory from these registers.

Inside the CPU: Get to know the Various Registers



Registers are basically the CPUs own internal mem They are used, among other purposes, to store temporary data while performing calculations. Let's look at each one in detail.

General Purpose Registers (GPR)

The 8086 CPU has 8 general purpose registers each register has its own name:

- AX - The Accumulator register (divided into AH / AL).
- BX - the Base Address register (divided into BH / BL)
- CX - The count register (divided into CH / CL).
- DX - The Data register (divided into DH / DL)
- SI - Source Index register.
- DI - Destination Index register.
- BP - Base pointer
- SP - Stack pointer.

Despite the name of a register, its the programmer who determines the usage for each general-purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bits.

4 General-porpose registers (AX, BX, CX, DX) are made of two separates 8-bit registers for example if AX = 0011000001110011b, then AH = 00110000b and AL = 01110011b. Therefore.

When you modify any of the 8- bit registers 16 bit registers are also updated and vice- versa. The same is for other 3 registers, "H" is for high and "L" is for low part.

Since registers are located inside the CPU, they are much faster than a memory, accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purpose which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

Segment Registers:

CS - points at the segment containing the current program.

DS - generally points at the segment where variables are defined.

Es — extra segment register ; it's up to a coder to define its usage.

Ss — points at the segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea, The segment registers have a Very special porpose — pointing at accessible blocks of memory. This will be discussed further in upcoming classes.

Special purpose Registers:

- Ip - The Instruction pointer. points to the next location of instruction in the memory.
- Flag Register - Determines the current state of the microprocessor modified automatically by the cpu after some mathematical operations, determines certain types of results and determines how to transfer control of a program.

Writing your first Assembly code

In order to write programs in assembly language you will need to familiarize yourself with most, if not all, of the instructions in the 8086 instruction set. This class will introduce two instructions and will serve as the basis for your first assembly program.

The following table shows the instruction name the syntax of its use, and its description. The operands heading refers to the type of operands that can be used with the instruction along their order.

- REG: Any Valid register.
- Memory: Referring to a memory location in RAM.
- Immediate: using direct Values.

Instruction	operands	Description
MOV	REG, memory memory, REG REG, REG memory, immediate REG, immediate.	Copy operand2 to operand1 • The MOV instruction cannot set the value of the CS and IP registers. • Copy value of one segment register to another segment register (should copy to general register first) • Copy an immediate value to segment register (should copy to general register first) Algorithm: operand1 = operand2
ADD	~~REG~~, memory memory, REG REG, REG memory, immediate REG, immediate.	Adds two numbers Algorithm: operand1 = operand1 + operand2

CSE 331L - 2 - Variables, I/O, Array

Topics to be Covered in this class:

- Creating variables
- Creating Arrays
- create Constants
- Introduction to INC, DEC, LEA instruction
- Learn how to access Memory.

Creating Variables

Syntax for a variable declaration:

```
name DB value
name DW value
```

DB - Stands for Define Byte

DW - Stands for Define word.

- name : can be any letter or digit combination; though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

- Value – can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal) or ? symbol for variables that are not initialized.

Creating Constants

Constant are just like variables, but they exist only until your program is compiled (assembled) After definition of a constant its value cannot be changed. To define constant EQU directive is used.

name EQU <any expression>

for example

```
K EQU 5
MOV AX, K
```

Creating Arrays

Array can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0–255)

Here are some array definition examples:

```
a DB 48h, 65h, 6ch, 6fh, 00h

b DB 'Hello', 0
```

- You can access the value of any element in array using square brackets, for example,

```
Mov AL, a[3]
```

- You can also use any of the memory index registers BE, SI, DI, BP, for Example:

```
Mov SI, 3
Mov AL, a[SI]
```

- If you need to declare a large array you can use DUP operator.

The Syntax for DUP:

Number DUP (value(s))

number - Number of duplicates to make (any Constant value)

value - expression that DUP will duplicate.

for example:

c DB 5 DUP (0)

is an alternative way of declaring

c DB 0,0,0,0,0

One more example

d DB 5 DUP (1,2)

is an alternative way of declaring

d DB 1,2,1,2,1,2,1,2,1,2

Memory Access:

To access memory, we can use these four registers, BX, SI, DI, BP. Combining these register inside [] symbol, we can get different memory locations.

[BX+SI]	[SI]	[BX+SI+d8]
[BX+DI]	[DI]	[BX+DI+d8]
[BP+SI]	d16 (variable offset only	[BP+SI+d8]
[BP+DI]	[BX]	[BP+DI+d8]
[SI+d8]	[BX+SI+d16]	[SI+d16]
[DI+d8]	[BX+DI+d16]	[DI+d16]
[BP+d8]	[BP+SI+d16]	[BP+d16]
[BX+d8]	[BP+DI+d16]	[BX+d16]

- Displacement can be an immediate value or offset of a variable, or even both. If there are several values, assembler evaluates all values and calculates a single immediate value
- Displacement can be inside or outside of the [] symbol, assembler generates the same machine code for both ways.
- Displacement is a signed value, so it can be both positive or negative.

instruction	operands	Description
INC	REG MEM	Increment: Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL ; AL = 5 RET
DEC	REG MEM	Decrement: Algorithm: operand = operand - 1 Example: MOV AL, 86 DEC AL; AL = 85 RET

LEA	REG, MEM	Load Effective Address
		Algorithm
		REG = Address of memory (offset)
		Example:
		mov BX, 35h
		MOV DI, 12h
		LEA SI, [BX+DI]

Declaring Array:

Array Name db size DUP (?)

value 'initialize.

arr1 db 50 dup (5, 10, 12)

Index values:

```
Mov bx, offset arr0
mov [bx], 6 ;; inc bx
mov [bx+1], 10
mov [bx+2], 9
```

Off set:

'offset' is an assembler directive in x86 assembly language. It actually means 'address' and is a way of handling the overloading of the 'mov' instruction. Allow me to illustrate the usage-

1. Mov si, offset Variable
2. mov si, Variable.

The first line loads SI with the address of variable. The second line loads SI with the value stored at the address of variable.

As a matter of style, when I wrote X86 assembler I would write it this way.

1. mov si, offset variable

2. mov si, [variable]

The square brackets aren't necessary but they made it much clearer while loading the contents rather than the address.

LEA is an instruction that load "offset variable" while adjusting the address between 16 and 32 bits as necessary. 'LEA (16 bit-register) ;(32 bit address) ;; loads the lower 16 bits of the address into the register and "LEA ;(32-bit register), ;(16-bit address) load the 32-bit register with the address zero extended to 32 bits.

CSE 331 L-3 Print and I/O

In this Assembly Language programming, A single program is divided into four segments which are-

1. Data segment
2. code Segment
3. Stack segment and
4. Extra Segment.

Print: Hello world in Assembly language.

```
Data Segment
    MESSAGE DB "HELLO WORLD!! $"
DATA ENDS
Code Segment
    ASSUME DS: DATA CS: CODE

START:
        MOV AX, DATA
        MOV DS, AX
        LEA DX, MESSAGE
        MOV AH, 9
        INT 21H
        MOV AH, 4CH
        INT 21H
```

ENDS

END START

Now, from these one is compulsory i.e. Code Segment If at all you don't need variables for your Program. If you need variables for your program you will need two segment i.e. Code segment and Data Segment.

First Line: DATA SEGMENT

DATA SEGMENT is the starting point of the DATA Segment in a program and DATA is the name given to this segment and SEGMENT is the keyword for defining Segments, where we can declare our variable.

Next Lines: MESSAGE DB "HELLO WORLD.!!!$"

MESSAGE is the variable name given to a Data Type (Size) that is DB. Stands for define Byte and is of one byte (8 bits)

In Assembly language programs, variable are defined by Data Size not its type. Character need one Byte so to store character or string we need DB only that don't mean DB can't number or numerical value. The string is given in double quotes. $ is used as NULL character in a program so that compiler can understand where to STOP.

Next Line : DATA ENDS

DATA END S is the End point of Data Segment in a program. we can write just ENDS But to differentiate the end of which Segment it is of which we have to write the same name given to the Data Segment.

Next Line - CODE SEGMENT

CODE SEGMENT is the starting point of the code Segment in a program and CODE is the name given to this Segment and Segment is the keyword for defining segme

Where we can write the coding of the Program.

Next Line — ASSUME DS: DATA CS: CODE

In this Assembly Language programming there are Different Register present for Different purpose. so we have to assume DATA is the name given to DATA segment and CODE is the name given to code segment register (SS, ES and used in the same way as CS, DS)

Next Line :- START

START is the label used to show the starting poin of the code which is written in the code segment: is used to define a label as in c programming.

Next Line :- MOV AX, DATA

MOV DS, AX

After Assuming DATA and CODE Segment still it is compulsory to initialize Data Segment to DS register. Mov is a keyWor

to Mov the second element into the first element. But we can not move a DATA Directly to DS due to MOV commands restriction, hence we move DATA to AX and then from AX to DS. AX is the first and Most important register in the ALU unit This part is also called INITIALIZATION OF DATA SEGMENT and it is important so that the Data elements or variable in the DATA Segment are made accessible other segments are not needed to be initialized, only assuming is inhale.

Next line :- LEA DX, MESSAGE

MOV AH, 9

INT 21H

The above three-line code is used to print the string inside the MESSAGE variable. LEA stands for Load Effective Address which is used to Assign Address of variable to DX register (The ~~first~~ ~~to~~ same can be written like this also MOV DX, OFFSET MESSAGE both mean the same). To do input and output in Assembly

Language we use Interrupts. Standard Input and standerd output related Interrupt are found in INT 21H which is also called as DOS interrupt. It works with the value of AH register. if the value is 09 or 09h or 09H (all means the same) that means "PRINT the string whos Address is loaded in Dx register.

Next Line - MOV AH, 4CH

INT 21H

The above two-line code is used to exit to dos or exit to operating System. Standard input and Standard output related Interrupt are found in INT 21H which is also called as DOS interrupt. It works with the value of AH register. If the value is 4ch, that means Return to Operating System or Dos which is the END of the Program.

Next-Line - CODE ENDS

CODE ENDS is the END point of the code Segment in a program. we can write just

ENDS But to differentiate the end of which segment it is of which we have to write the same name given to the code segment.

Last Line — ~~End~~ END START

END START is the end of the label used to show the ending point of the Code which is written in the code segmen

Execution of program explanation - Hello wor

First save the program which Hello world .asm filename; No Space is allowed in the name of the program file and extensio as .asm (dot asm because IT's an Assembly language Program). The written program has to be compiled and Run by ~~extensi~~ clickin on the RUN button on the top. The program with No Errors will only run and Could show You the desired output. Just see the Screenshots below.

Note - In this Assembly Language programming we have .com format and EXE format. We are Learning in EXE format only which simple then .com format to understand and write. We can write the program in lower or upper case, but i prepare upper case. (this program is executed on emu8086 emulator software)

Now Try this

```
DATA SEGMENT
MESSAGE DB "Hello world$"
START:
    MOV AX, DATA
    MOV DS, AX
    LEA DX, MESSAGE
    MOV AH, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
END START
```

Assembly Example 1 – Print 2 strings.

```
.MODEL SMALL
.STACK 100H

.DATA
    STRING_1 DB 'I hate LSE BSI $'
    STRING_2 DB 'BUT I love Kacchi!!!$'

.CODE
.MAIN PROC
MOV AX, @DATA ; initialize DS
MOV DS, AX

LEA DX, STRING_1 ; load & display
MOV AH, 9          the STRING_1
INT 21H

MOV AH, 2
MOV DL, 0DH ; carriage return
INT 21H
MOV DL, 0AH ; line feed
INT 21H
```

```
LEA DX, STRING_2 ; load & display the STRING_2
MOV AH, 9
INT 21H

MOV AH, 4CH ; return control to DOS
INT 21H

MAIN ENDP
END MAIN
```

Assembly Example 2 – Read a string and print it

```
.MODEL SMALL
.STACK 100H

.DATA
    MSG_1 EQU 'Enter the Character : $'
    MSG_2 EQU 0DH, 0AH, 'The given character is : $'
    PROMPT_1 DB MSG_1
    PROMPT_2 DB MSG_2

.CODE
MAIN PROC
MOV AX, @DATA ; initialize DS
```

```
MOV DS, AX
LEA DX, PROMPT_1 ; load and display PROMPT_1
MOV AH, 9
INT 21H
MOV AH, 1 ; read a character
INT 21H
MOV BL, AL ; save the given character into BL
LEA DX, PROMPT_2 ; load and display PROMPT_2
MOV AH, 9
INT 21H
MOV AH, 2 ; display the character
MOV DL, BL
INT 21H
MOV AH, 4CH ; return control to DOS
INT 21H
MAIN ENDP
END MAIN
```

Assembly Example 3 :- Read a string from user and display this string in a new line.

```
.MODEL SMALL
.STACK 100H

.CODE
MAIN PROC
    MOV AH, 1          ; read a character
    INT 21H
    MOV BL, AL         ; save input character into BL
    MOV AH, 2          ; carriage return
    MOV DL, 0DH
    INT 21H
    MOV DL, 0AH        ; line feed
    INT 21H
    MOV AH, 2          ; display the character stored in BL
    MOV DL, BL
    INT 21H
    MOV AH, 4CH        ; return control to DOS
    INT 21H
END MAIN
```

Assembly Example 4 - Read a string with gaps and print it

```
.MODEL SMALL
.STACK 64
.DATA
        STRING DB ?,
        SYM DB '$'
INPUT_M DB 0ah, 0dh, 0AH, 0DH, 'Enter-h
  'Input', 0DH, 0AH, '$'
OUTPUT_M DB 0ah, 0dh, 0AH, 0DH, 'The outp
        is ', 0DH, 0AH, '$'

.CODE
MAIN PROC
        MOV AX, @DATA
        MOV DS, AX
        MOV DX, OFFSET INPUT_M ; lea dx, input
        MOV AH, 09
        INT 21H
        LEA SI, STRING
```

```
INPUT:  MOV AH, 01
        INT 21H
        MOV [SI], AL

        INC SI
        CMP AL, 0DH
        JNZ INPUT
        ;MOV AL, SYM
        MOV [SI], '$'
OUTPUT: LEA DX, OUTPUT MSG
        MOV AH, 9
        INT 21H
        MOV DL, 0AH
        MOV AH, 02H
        INT 21H
        MOV DX, OFFSET STRING
        MOV AH, 09H
        INT 21H
        MOV AH, 4CH
        INT 21H
MAIN ENDP
 END MAIN
```

Assembly Example 5 — Printing string using MOV instruction

```
.MODEL SMALL
.STACK
.DATA
    MSG1 DB 'HI!!!  Roman Jage.DB'

.CODE
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET MSG1 ; LEA dx, msg1
    MOV ah, 09h
    int 21h
    mov ah, 4ch
    int 21h

END
```

Assembly Example 6 — Print Digit from 0-9

```
.MODEL SMALL
.STACK 100H

.Data
    PROMT DB 'The counting from 0 to 9 is:$'
```

code

```
CODE
    MAIN PROC
    MOV AX, @DATA   ; initialize DS
    MOV DS, AX
    LEA DX, PROMPT  ; load and print PROMPT
    MOV AH, 9
    INT 21H
    MOV CX, 10      ; initialize CX
    MOV AH, 2       ; set output function
    MOV DL, 48      ; set DL with 0
@LOOP:              ; loop label
    INT 21H         ; print character
    INC DL          ; increment DL to next
                    ; ASCII character
    DEC CX          ; decrement CX
    JNZ @LOOP       ; JUMP to label @LOOP
                    ; if CX is 0
    MOV AH, 4CH     ; return control to DOS
    INT 21H
```

```
MAIN    ENDP
        END     MAIN
```

Assembly Example 4 - Sum of two integers

```
.MODEL SMALL
.STACK 100H

.DATA
    PROMPT_1 DB 'Enter the first digit: $'
    PROMPT_2 DB 'Enter the second digit: $'
    PROMPT_3 DB 'The sum of first and second digits: $'

    VALUE_1 DB ?
    VALUE_2 DB ?

.CODE
MAIN PROC
    MOV AX, @DATA       ; initialize DS
    MOV DS, AX
    LEA DX, PROMPT_1    ; load and disp the PROMPT_1
    MOV AH, 9
    INT 21H
```

```
MOV AH, 1          ; read a character
INT 21H

SUB AL, 30H        ; save first digit in
                   ; VALUE_1 in ASCII cod

MOV VALUE_1 ;AL

MOV AH, 2          ; carriage return
MOV DL, 0DH
INT 21H

MOV DL, 0AH        ; line feed
INT 21H

LEA DX, PROMPT_2   ; load and display
                   ; PROMPT_2
MOV AH, 9
INT 21H

MOV AH, 1          ; read a character.
INT 21H

SUB AL, 30H        ; save second digit in
                   ; VALUE_2 in ASCII cod
MOV VALUE_2 AL
```

```
MOV AH, 2      ; carriage return
MOV DL, 0DH
INT 21H

MOV DL, 0AH    ; Line feed
INT 21H

LEA DX, PROMPT_3 ; Load and display the
                   PROMPT-3
MOV AH, 9
INT 21H

MOV AL, VALUE-1  ; add first and second
                   digit
MOV AL, VALUE-2

ADD AL, 30H      ; Convert ASCII to Decimal
                   Code

MOV AH, 2        ; Display the character
MOV DL, AL
INT 21H

MOV AH, 4CH      ; return control to DOS
INT 21H
END MAIN.
```