CSE331L→1

Introduction to Assembly Language.

Introduction:

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

Steps required to run an assembly programe.

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 familiarize 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 associated 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. It's 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 16 bits or 8 bits at a time.
* 8086 has a 20 bit address bus which

means, it can address up to 2^{20} = 1 MB memory locatio

Registrar - Register - Resistor

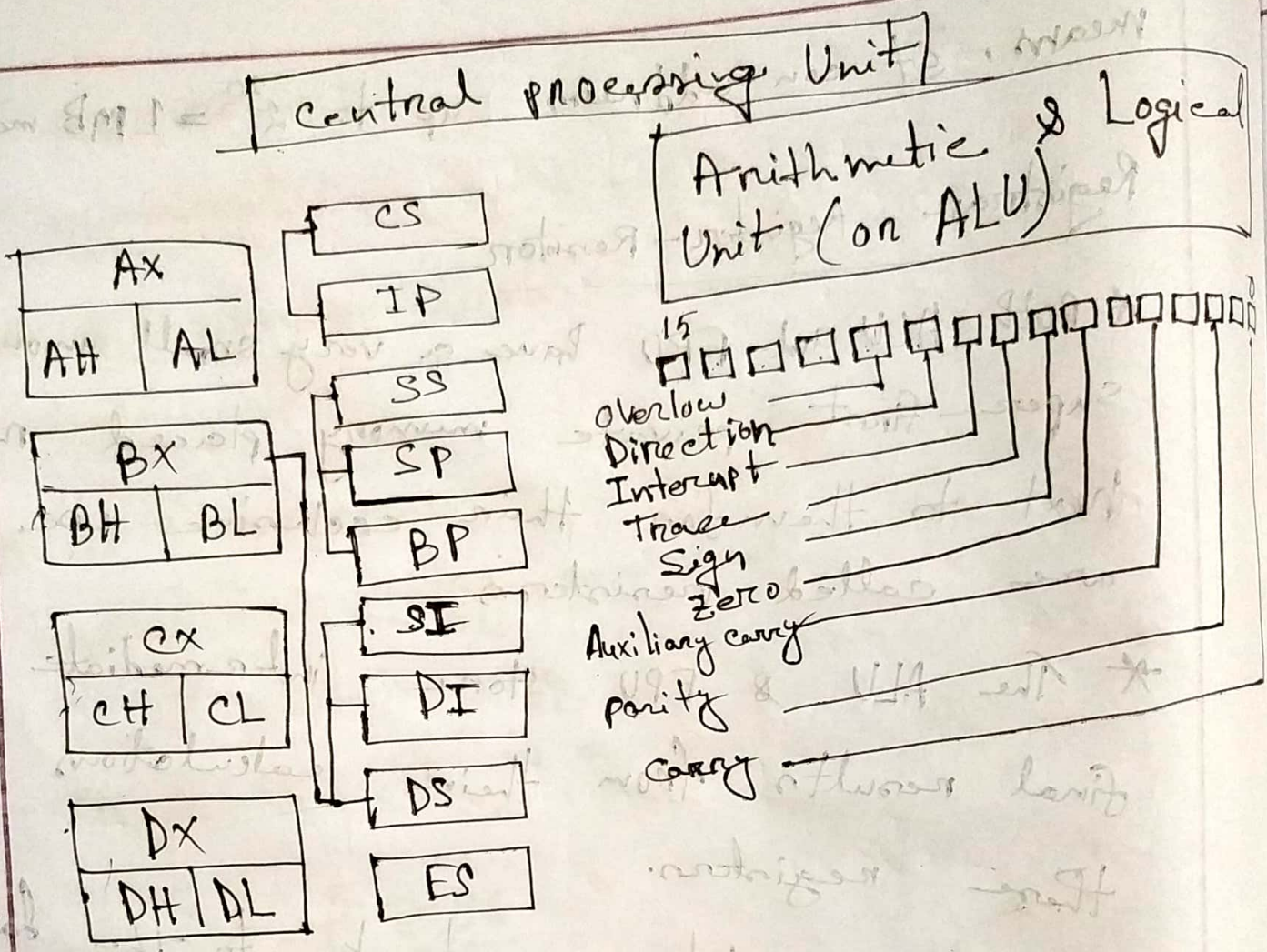
* Both ALU and FPU have a very small amount of super-fast private memory placed right next to them for their exclusive use. These are called registers.

* The ALU & FPU store intermediate and final results from their calculations in these registers.

* processed data goes back to the data cache and then to main memory from these registers.

~~control~~

Inside the CPU: Get to know the various registers:



Registers are basically the CPU's own internal memory. 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 it's 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 register, it is 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 purpose registers (AX, BX, CX, DX) are made of two separate 8-bit registers, for example if AX = 0011000000111001b, then AH = 00110000b and AL = 00111001b. Therefore, when you modify any of 8-bit registers 16-bit registers

are also updated, and vice versa. The same in 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 purposes 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 - general points at the segment where variables are defined.

ES - extra Segment register, it is up to a coder to define its user.

SS - points at the segment containing the stack.

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

Special perpose 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 Assemly code

In order to write programs in assembly language, you will need to familiarize yourself with most, if not all, of the 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 it's description, The operands heading refers to the type of operands that

can be used with the instruction along with their proper order:

* REG: Any valid register.
* Memory: Referring to a memory location in RAM
* Immediate; Using direct values.

Instruction	operand	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. * 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

Lab-2

Variables, I/O, Array

Creating variable:

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 be start with a letter. It's ~~a variable~~

possible to declare unnamed variables by not specifying the name (this variable will have an adress but no name).

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

Creating constants:

constants 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 constants EQU directive is used:

name EQU <any expression>

For example:

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

Creating Arrays

Arrays can be seen as chain of variables. A text is a example of a byte array, each character is presented as an ASCII code value (0-255).

Here are some array ~~function~~ definition examples:

a DB 48h, 65h, 6Ch, 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 register BX, SI, DI, BP,

for example:

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

* If you need to declare a large array you can use DUP operation.

The syntax for DUP:

number DUP (value(s))

number - number of duplicates to make (any constant value).

value - expression that DUP will duplicate.

for examples:

c DB 5 DUP(1)

is an alternative way of declaring:

c DB 1, 1, 1, 1, 1

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 registers inside [ ] symbols, we can get different memory location.

[BX + SI]	[SI]	[BX + SI + d8]
[BX + DI]	[DI]	[BX + DI + d8]
[BP + SI]	d16 (variable offset only)	[BX + SI + d8]
[BP + DI]	[BX]	[BP + DI + d8]
[SI + d8]	[BX + SI + d16]	[SI + d16]
	[BX + DI + d16]	[DI + d16]
	[BP + SI + d16]	[BP + d16]
	[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 [] symbols, assembler generates the same machine code for both ways.
* Displacement is a signed value, so it can be both positive or negative.

Instructions

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
```

OFFSET:

"offset" is an assembler directive in x86 assembly language. It actually means "address" and in 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 mad it much clear while loading the contents rather than the adress.

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

Lab-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.
4. Extra Segment.

print: Hello world in Assembly Language;

```
Data Segment
        MESSAGE DB "HELLO WORLD!!!$"
ENDS
CODE SEGMENT
    ASSUME DS: Data CS: CODE
START:
        MOV AX, DATA
        MOV DS, AX
        LEA DX, MESSAGE
        MOV AH, 9
        INT 21 H
        MOV AH, 4CH
```

INT 21H

ENDS
END START

Now from these one is compulsory i.e code segment if at all you dont need variable for your program if you need variable(s) for your program you will need two segments ie code segments and Data Segments.

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 keywond for defining Segments, where we can declare our variables.

Next Line - MESSAGE DB "HELLO WORLD !!! $"

MESSAGE is the variable name given a Data type (size) that is DB. 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 hold number or numerical value. The string is given in double quotes. $ is used as NULL character in c programming so that compiler can understand where to stop.

Next-Line - DATA ENDS

Data ENDS is the End point of the 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 the ~~string of program.~~ 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 keyword for defining Segments, 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 register and CODE is the name given to ~~the~~ Code Segment Register (SS, ES

are used in the same way as CS, DS)

Next Line - START:

START is the label used to show the starting point 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 keyword to move the second element into the first element. But we can not move DATA Directly to DS due to MOV commands restrictions, Hence we move data to AX and then from AX to DS. AX is the first and most important register in ALU unit. This part is

Also called INITIALIZATION OF DATA SEGMENT and it is important so that the Data elements or variables in the DATA segment are made accessible. Other segment are not needed to be initialized only assuming in intiale.

Next Line - LEA DX, MESSAGE

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.

To do input and output in Assembly Language we use Interrupts. Standard Input and Standard Output related Interrupts are found. INT 21 H

which also called as DOS interrupt. It wonks with the value of AH register, If the value is 9 or 09h or 09H, that means print the string whos Address in Loaded in DX register.

Next Line - MOV AH, 4CH

The avobe two line code is used to dos or exit to operating system. Standard Input and Standard Output related Interrupts are found In INT 21H which in also called as DOS interrupt. It wonks with the value of AH register, If the value in 4CH, that means Return to Operating system or DOS which in 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 give to the code segment.

Last Line - END START

END START in the ~~start~~ end of the level used to show the ending point of the code which is written in the code segment.

Execution of program explaination - Hello world

First save the program with helloworld.asm filename. No space is allowed in the name of the program file and extension as asm (dot asm because it's an Assembly language program). The written program has to be compiled and Run by clicking on the RUN button on the top.

The program with no error will only run and could show you ~~and~~ the desire output.

```
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 Ex-1 - print 2 string

```
.MODEL SMALL
.STACK 100H

.DATA
    STRING_1  DB 'I Hate CSE331 $'
    STRING_2  DB 'But I Love Karachi !!! $'

.CODE
    MAIN PROC
```

```
MOV AX, @DATA
MOV DS, AX

LEA DX, STRING_1
MOV AH, 9
INT 21H

MOV AH, 2
MOV DL, 0DH
INT 21H

MOV DL, 0AH
INT 21H

LEA DX, STRING_2
MOV AH, 9
INT 21H

MOV AH, 4CH
INT 21H

MAIN ENDP
END MAIN
```

Assembly Ex-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
    MOV DS, AX

  LEA DX, PROMPT_1
  MOV AH, 9
  INT 21H

  MOV AH, 1
  INT 21H

  MOV BL, AL

  LEA DX, PROMPT_2
  MOV AH, 9
  INT 21H
```

```
MOV  AH,2
MOV  DL,BL
INT  21H

MOV AH,4CH
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
       INT 21H

       MOV BL,AL

       MOV AH,2
       MOV DL,0DH
        INT 21H
```

```
MOV AH,2
MOV DL,BL
INT 21H

MOV AH,4CH
INT 21H

MAIN ENDP
END MAIN
```

Assembly Ex-4- Read a String with gaps and print it.

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

.CODE
MAIN PROC
        MOV AX,@DATA
        MOV DS,AX
        MOV DX,OFFSET INPUT_M
        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[SI],'$'

OUTPUT: LEA  DX, OUTPUT_M
        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 EX-5- Printing string using MOV instruction

```
.MODEL Small
.DATA
    MSG1 DB 'KI !!! Kemon lage :D $'
.CODE
    MOV AX, @DATA
    MOV dx, OFFSET MSG1
    MOV ah, 09h
    int 21h

    mov ah, 4Ch
    int 21h

END
```

Assembly EX-6 - Print Digit from 0-9

```
.MODEL SMALL
 .STACK 100H

 .DATA
     PROMPT DB 'The counting from 0 to 9 is:$'
```

```
.CODE
  MAIN PROC
    MOV AX, @DATA

    MOV DS, AX

    LEA Dx, PROMPT
    MOV AH, 9
    INT 21H

    MOV cx, 10

    MOV AH, 2
    MOV DL, 48

    @LOOP:
      INT 21H

      INC DL
      DEC cx
      JNC @LOOP

     MOV AH, 9CH
     INT 21H
  MAIN ENDP
END MAIN
```

Assembly Example 7 - Sum of two integers

```
.MODEL SMALL
  .STACK 100H
  .DATA
    PROMT_1 DB \'Enter 1st digit: $\'
    PROMT_2 DB \'Enter 2nd digit: $\'
    PROMT_3 DB \'sum of 1st and 2nd digit: $\'

    VALUE_1 DB ?
    VALUE_2 DB ?

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

        LEA  DX, PROMPT_1
        MOV  AH, 9
        INT  21H

        MOV AH,1
        INT 21H

        SUB AL, 30H
        MOV VALUE_1, AL
```

```
MOV AH, 2
MOV DL, 0DH
INT 21H

MOV DL, 0AH
INT 21H

LEA DX, PROMPT_2
MOV AH, 9
INT 21H

MOV AH, 1
INT 21H

SUB AL, 30H
MOV VALUE_2, AL

MOV AH, 2
MOV DL, 0DH
INT 21H

MOV DL, 0AH
INT 21H

LEA DX, PROMPT_3
MOV AH, 9
INT 21H
```

```
MOV AL, VALUE_1
ADD AL, VALUE-2

ADD AL, 30H

MOV AH, 2
MOV DL, AL
INT 21H

MOV AH, 9CH
INT 21H
MAIN ENDP
END MAIN
```