HomeWork 2

CSE331 (Lab)

Md Minul Islam Talukdar
1510678042

## Lab-1 Introduction to Assembly Language

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

Steps 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, we should familiarize ourselves with the software before we begin to write any code. Following the in-class instructions regarding the layout of emu8086.

## Microcontrollers vs Microprocessors:

1) A microprocessor is a CPU on a single chip.

2) 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:

1) 8086 is a 16 bit processor, It's ALU, internal registers work with 16 bit binary word.

2) 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.

3) 8086 has a 20 bit address bus which means, it can address up to $2^{20} = 1$ MB memory location.

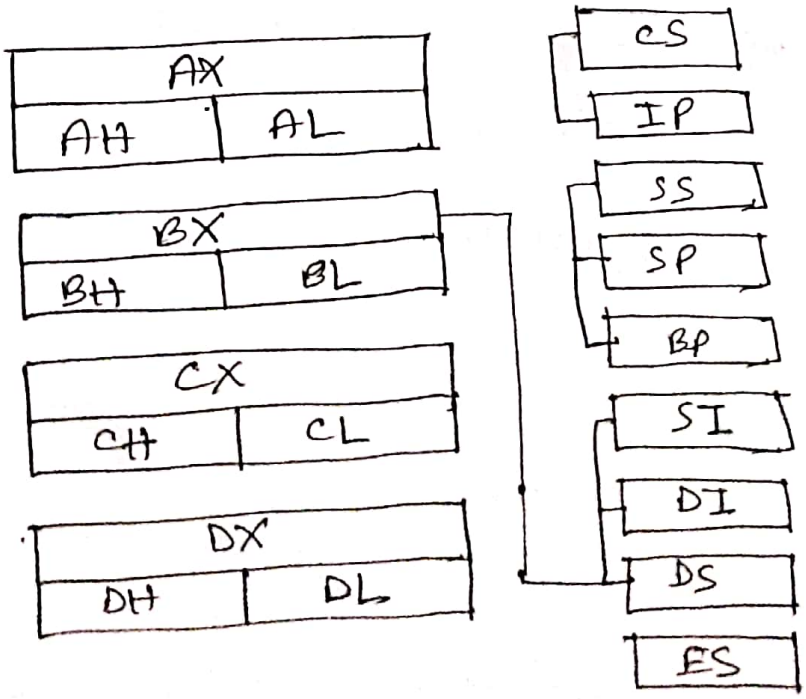
## Registrar - Register - Resistor :

1) 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.

2) The ALU and FPU store intermediate and final results from their calculations in these registers.

3) Processed data goes back to the data cache and then to the main memory from these registers.

## Central Processing Unit (CPU)



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 in detail.

## General Purpose Registers (GPR) :

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

1) AX - The Accumulator registers (AH/AL)
2) BX - The Base Address register (BH/BL)
3) CX - Count register (CH/CL)
4) DX - The Data register (DH/DL)
5) SI - Source Index register.
6) DI - Destination Index register
7) BP - Base Pointer
8) SP - Stack Pointer.

Despite the name of a 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. 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 we 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, we 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 – 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 register, this is never a good idea.

## Special Purpose Registers:

IP – The instruction pointer. Points to the next to the next location of instruction in the memory.

Flags 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 first Assembly Code:

In order to write programs in assembly language, we will need to familiarize ourselves 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.

REG : Any valid register

Memory : referring to a memory location in RAM.

Intermediate: Using direct values.

| Instruction | operands | Description |
|---|---|---|
| MOV | REG, memory<br>memory, REG<br>memory, immediate<br>REG, immediate | Copy Operand 2 to operand 1.<br>Algorithm:<br>operand 1 = operand 2. |
| ADD | REG, memory<br>memory, REG<br>REG, REG<br>memory, immediate<br>REG, immediate | Adds two numbers<br>Algorithm:<br>operand 1 = operand 1 + operand 2 |

# Lab-2: Variable, I/O, Array

Topics to be covered in the class:

1) Creating variables.
2) Creating Arrays.
3) Create Constants.
4) Introduction to INC, DEC, LEA instruction.
5) Learn how to access memory.

## Creating variable:

Syntax for a variable declaration:

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

DB - stands for Define Byte.

DW - stands for Define word.

i) 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).

ii) 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:

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.

name EQU <any expression>

For example:

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

## Creating Arrays:

Arrays 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, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

1) We can access the value of any element in array using square brackets, for example:

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

2) We can also use any of the memory index registers BX, SI, DI, BP, for example:

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

3) If we need to declare a large array we 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 (9)

is an alternative way of declaring:

C DB 9, 9, 9, 9.

one more example:

d DB 5 DUP (1,2)

is an alternative way of declaring:

d BB 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 locations.

| | | |
|---|---|---|
| [BX+SI] | [SI] | [BX+SI+d8] |
| [BX+DI] | [DI] | [BX+DI+d8] |
| [BP+SI] | d16 (variable) | [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] |

1) 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.

2) Displacement can be inside or outside of the [] symbols, assembler generates the same machine code for both ways.

Instructions:

| Instructions: | Operands | Description |
|---|---|---|
| INC | REG<br>MEM | Increment.<br>Algorithm:<br>operand = operand + 1<br>Example:<br>MOV AL, 4; INC AL;<br>AL = 5. RET |
| DEC | REG<br>MEM | Decrement.<br>Algorithm:<br>operand = operand - 1<br>Example:<br>MOV AL, 86<br>DEC AL; AL = 85<br>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 is a way of handling the overloading of the "mov" instruction. Allow.

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.

1. mov si, offset variable.

2. mov si, [variable].

The square brackets aren't necessary, but they make 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)" loads the 32-bit register with the address zero extended to 32 bits.

# Lab-3 - Print and I/O

In 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!!!$"

ENDS
 Code Segment
    ASSUME DS: DATA CS: CODE

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

ENDS
END START
```

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 variables.

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

MESSAGE is the variable name given to a Data Type (Size) that is DB, DB stands for Define Byte and is of one byte (8 bits). In Assembly language programs, variables are defined by Data size not its Type.

Next Line DATA ENDS :

DATA ENDS is the end point of the Data SEGMENT in a program.

Next Line CODE SEGMENT:

CODE SEGMENT is the starting point of the code segment in a program and CODE is the

given to this SEGMENT and SEGMENT is the keyword for defining segments.

Next Line - ASSUME DS: DATA CS: CODE :

There are different registers 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 code Segment register.

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 cannot move DATA directly to DS due to MOV commands restriction, hence we move DATA to AX and then from AX to DS.

```
Next Line - LEA DX, MESSAGE
MOV AH, 9
INT 21 H
```

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.

```
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 interrupts are found in INT 21H which is also called DOS interrupt.

## Next Line - CODE ENDS

CODE ENDS is the End point of the code Segment in a program.

## Last Line - 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 Segment.

## Assembly Example 1 - Print 2 strings

```
.MODEL SMALL
.STACK 100H
.DATA
    STRING_1 DB 'I hate CSE331 $'
    STRING_2 DB 'But I Love Kacchi!!! $'
.CODE
 MAIN PROC
 MOV AX, @DATA
 MOV DS, AX                    ; initialize DS
 LEA DX, STRING_1
 MOV AH, 9                     ; load and display
 INT 21H                         the STRING_1
```

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

MOV DL, 0AH          ; line feed
  INT 21H

LEA DX, STRING_2     ; load and display the
MOV AH, 9              STRING_2
  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
    INT 21H            ; return control to DOS
    MAIN ENDP
END MAIN
```

## Assembly Example 7 – 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 \"Sum of first and second digit: $\"
  VALUE_1 DB ?
  VALUE_2 DB ?
.CODE
  MAIN PROC
     MOV AX, @DATA          ; initialize DS
     MOV DS, AX
   . LEA DX, PROMPT_1
     MOV AH, 9              ; load and display the PROMPT_1
     INT 21H

     MOV AH, 1              ; read a character
     INT 21H

     SUB AL, 30H
     MOV VALUE_1, AL        ; save First digit in VALUE_1
                            in ASCII code
    MOV AH, 2
    MOV DL, 0DH             ; carriage return
    INT 21H
```

```
MOV DL, 0AH
INT 21H          ; line feed

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

MOV AH, 1
INT 21H          ; read a character

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

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

MOV DL, 0AH
INT 21H          ; line feed

LEA DX, PROMPT_3
MOV AH, 9        ; load and display PROMPT_3
INT 21H

MOV AL, VALUE_1  ; add first and second digit
ADD AL, VALUE_2

ADD AL, 30H      ; convert ASCII to Decimal code

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

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