- you can also use any of the memory index registers 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 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, 9
```

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

Assembly Sample 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          ;read a character
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
MOV DL, BL         in BL
INT 21H

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

MAIN ENDP
END MAIN
```

```
OUTPUT: LEA DX, OUTPUT_M   ; load and display
                           PROMPT_2
        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
```

Printing string using MOV instruction

```
.Model small;
.STACK
.DATA
    MSG1 DB 'kI!!! kemon lage :D '$'
.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
```

Displacement is a signed value, so it can be both positive and 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]

```
    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, 4CH
 INT 21H
 MAIN ENDP
END MAIN
```

DECLARING Arrays

Array Name db size DUP(?)

value initialized

arr1 db 50 dup(5,10,12)

Index values:

```
mov bx, offset arr.0
mov [bx], 6 ; inc bx
mov [bx+1], 10
mov [bx+9], 9
```

Sum of two integers

```
.MODEL SMALL
.STACK 100H

.DATA
    PROMPT_1 DB 'Enter the first digit: $'
    PROMPT-2 DB 'Enter the 2nd . : $'
    PROMPT-3 DB '  .   .  3rd  . : $'

    VALUE_1 DB ?
    VALUE_2 DB ?

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

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. Extra segment
4. stack 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 AX, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
  ENDS
END START
```

Now from these one is compulsory. Code segment if at all you don't need variable

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 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]
```

for your program will need two segments. Code segment and Data segments.

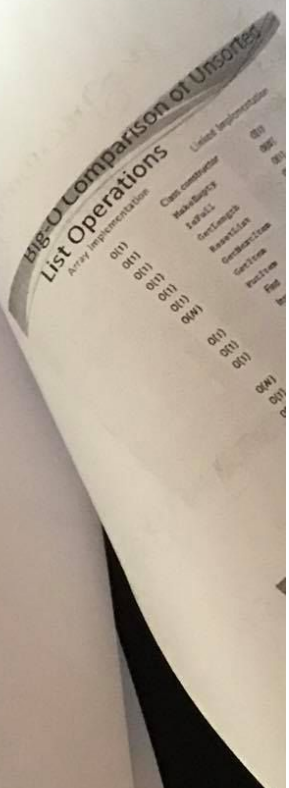
```
DATA SEGMENT
    MESSAGE DB "HELLO WORLD$"
START:
    MOV AX, DATA
    MOV DS, AX
    LEA DX, MESSAGE
    MOV AH, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
END START
```

ASSEMBLY Example; Print 2 strings

```
.MODE SMALL
.STACK 100H

.DATA
  STRING_1 DB 'I hate CSE 331 $'
  STRING_2 DB 'But I Love kacchi!!! $'

.CODE
MAIN PROC
    MOV AX, @DATA          ; Initialize DS
    LEA DX, STRING_1       ; Load & Display the
                              STRING_1
    MOV AH, 9
    INT 21H
```



4 general-purpose registers (AX, BX, CX, DX) are made of two separates 8 bit registers for example if AX = 0011000000111001b, then AH = 00110000b and AL = 00111001b. Therefore when you modify any of the 8 bit registers 16 bits 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 purposes which limit their use as variables, but they are still an excellent place temporary data of calculations.

Instruction	Operands	Description
MOV	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Copy Operand2 to Oper • The MOV instruction can 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

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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, '$'
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV DX, OFFSET INPUT_M ; lea dx, input_m
    MOV AH, 09
    INT 21H
    LEA SI, STRING
INPUT: MOV AH, 01
    INT 21H
    MOV [SI], AL
    INC SI
    CMP AL, 0DH
    JNZ INPUT
    ; MOV AL, SYM
    MOV [SI], '$'
```

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.

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 purpose, 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.
- Flags Register - Determines the current state of the microprocessor. Modified automatically by the CPU after some mathmetical operations determines certain types of results and determines how to transfer control of a program.

Introduction to Assembly Language.

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

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

Writing Your First Assembly Codes

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 with their proper order.

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

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 its own name

- AX - The accumulator register (divided into AH/AL)
- BX - The Base Address register (" " BH/BL)
- CX - The Count Register (divided into CH/CL)
- DX - The Data " (divided into DH/DL)
- SI - Source Index Register.
- DI - Destination Index " .
- BP - Base Pointer.
- SP - Stack Pointer.

Despite the name of a register, it's 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.

or 8bits at a time.

- 8086 has a 20bit address bus which means, it can address up to 2^20 = 1 MB memory location

Registrar - Register - Resistor

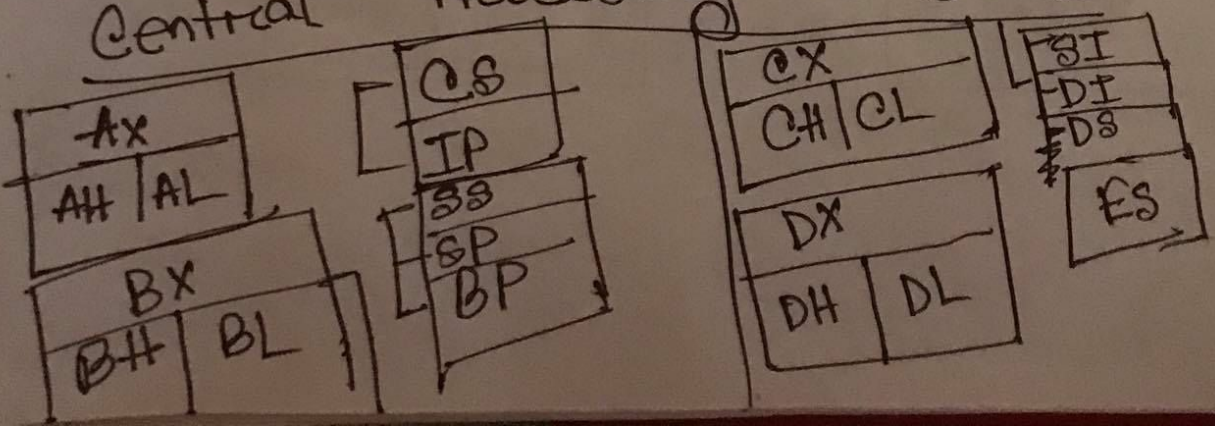
- Both ALU & FPU have a very small amount of super fast private memory placed right next to them for their exclusive use. These are called registers.

- The ALU & FPU store intermediate and final results 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.

Central Processing Unit (or CPU)



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

MOV DL, 0AH
INT 21H       ; line feed

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

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

MAIN ENDP
END MAIN
```

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
```

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 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 [ ] symbols, assembler generates the same machine code for both ways.

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  ; 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:
  INT 21H       ; Loop label
                ; print character
  Int DL        ; increament DL to
                  next ASCII character
  DEC CX        ; decrement CX
  JNZ @LOOP     ; jump to label @LOOP is
                  CX is 0
MOV AH, 4CH ; return control to DOS
MAIN ENDP
  END MAIN
```

Variables, I/O, Array

Topics to be covered in this class:

- Creating variables
- Creating Arrays
- Crate Constants
- Introduction to INC, DEC, LEA instruction
- 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.

- 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 numbering system (hexadecimal, binary or decimal) or "?" symbol for variables that are not initialized