American International University- Bangladesh

Faculty of Engineering (EEE)

EEE 4103: Microprocessor and Embedded Systems Laboratory

Title: Introduction to Microprocessor 8086, 8086 instructions and programming with 8086.

Abstract:

In this experiment the main objective is to understand the working principle of MTS-86c& MDA 8086 and familiarize emulator EMU8086 by using a simple program to test its different uses and introduction to segmented memory technology used by Microprocessor 8086.

Introduction:

The microprocessor 8086 can be considered to be the basic processor for the Intel X-86 family. With the knowledge of this 16-bit processor, one can study the further versions of this processor 80386, 80406 and Pentium.

The micro-kit we are using is "MTS-86c" and "MDA 8086". In 8086 there are many instructions. In this laboratory some instructions and mainly use of arrays will be observed. In this experiment the main objective is:

1) To work with advanced 8086 instructions.
2) To learn how to write assembly programs using 8086 instructions and arrays.

Theory and Methodology:

The 8086 Microprocessor

The 8086 is a 16-bit microprocessor chip designed by Intel between early 1976 and mid-1978, when it was released. The 8086 became the basic x86- architecture of Intel's future processors.

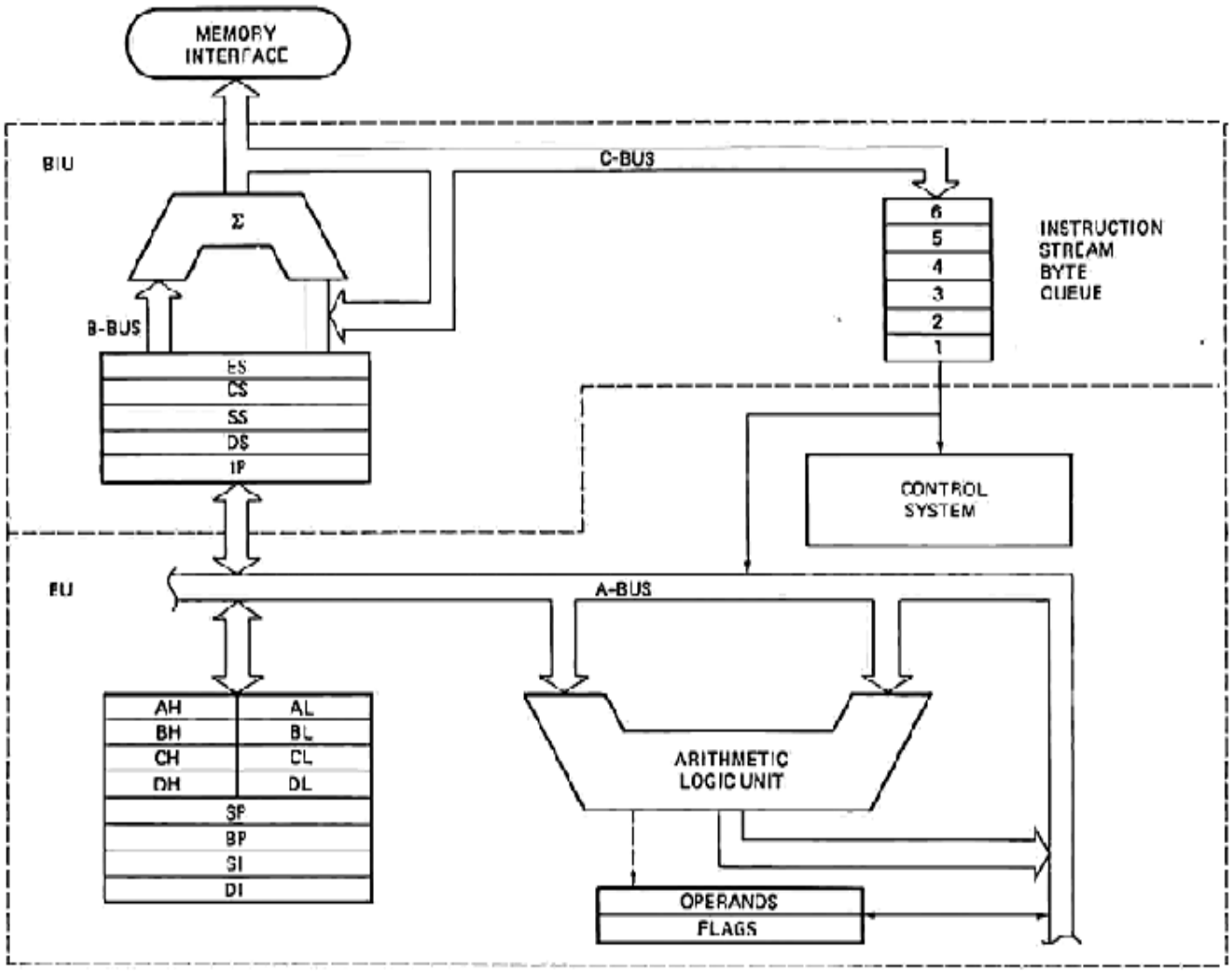


Fig 1: Intel 8086 internal architecture

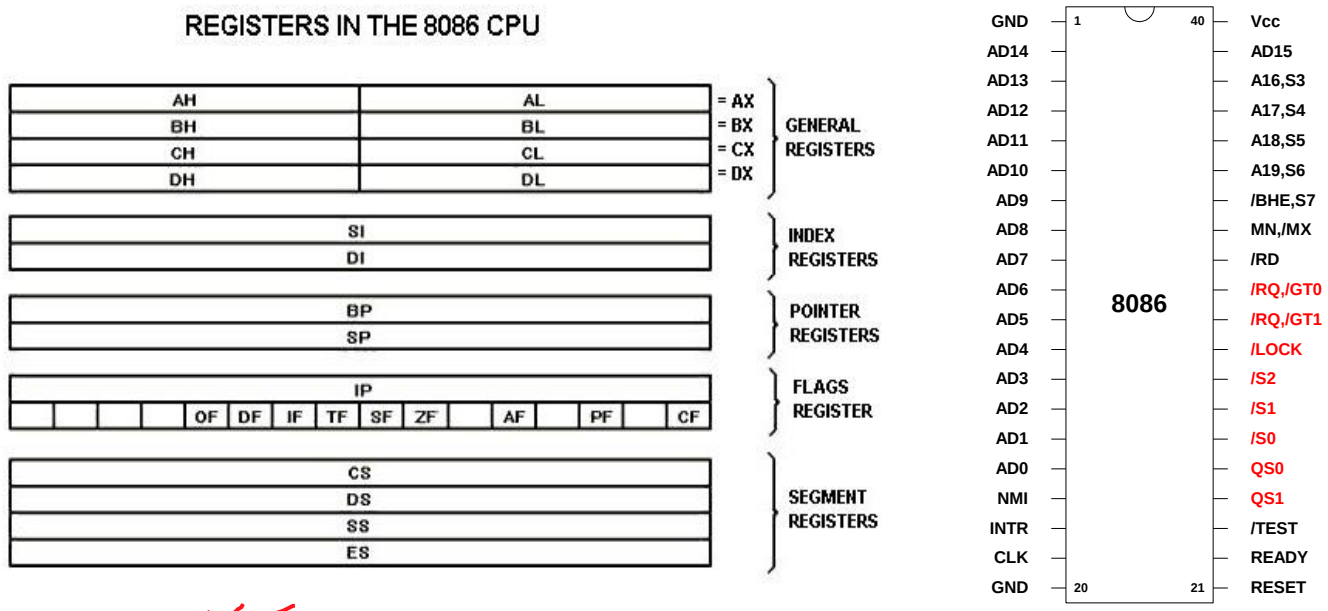


Fig 2: Internal Diagram, Registers and PIN diagram of the 8086 microprocessor

General Purpose Registers

8086 CPU has 8 general purpose registers; each register has its own name:
AX - the accumulator register (divided into AH / AL):

1. Generates shortest machine code
2. Arithmetic, logic and data transfer
3. One number must be in AL or AX
4. Multiplication & Division
5. Input & Output

BX - the base address register (divided into BH / BL).
CX - the count register (divided into CH / CL):

1. Iterative code segments using the LOOP instruction
2. Repetitive operations on strings with the REP command
3. Count (in CL) of bits to shift and rotate

DX - the data register (divided into DH / DL):

1. DX:AX concatenated into 32-bit register for some MUL and DIV operations
2. Specifying ports in some IN and OUT operations

SI - source index register:

1. Can be used for pointer addressing of data
2. Used as source in some string processing instructions
3. Offset address relative to DS

DI - destination index register:

1. Can be used for pointer addressing of data
2. Used as destination in some string processing instructions
3. Offset address relative to ES

BP - base pointer:

1. Primarily used to access parameters passed via the stack
2. Offset address relative to SS

SP - stack pointer:

1. Always points to top item on the stack
2. Offset address relative to SS
3. Always points to word (byte at even address)
4. An empty stack will had SP = FFFFh

Segment Registers

CS - points at the segment containing the current program.
DS - generally points at 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 purpose - pointing at accessible blocks of memory.

Segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address 12345h(hexadecimal), we could set the DS = 1230h and SI = 0045h. This way we can access much more memory than with a single register, which is limited to 16 bit values.
The CPU makes a calculation of the physical address by multiplying the segment register by 10h and adding the general purpose register to it (1230h * 10h + 45h = 12345h):

```
 12300
+  0045
-------
 12345
```

The address formed with 2 registers is called an effective address.
By default BX, SI and DI registers work with DS segment register; BP and SP work with SS segment register.Other general purpose registers cannot form an effective address.Also, although BX can form an effective address, BH and BL cannot.

Special Purpose Registers

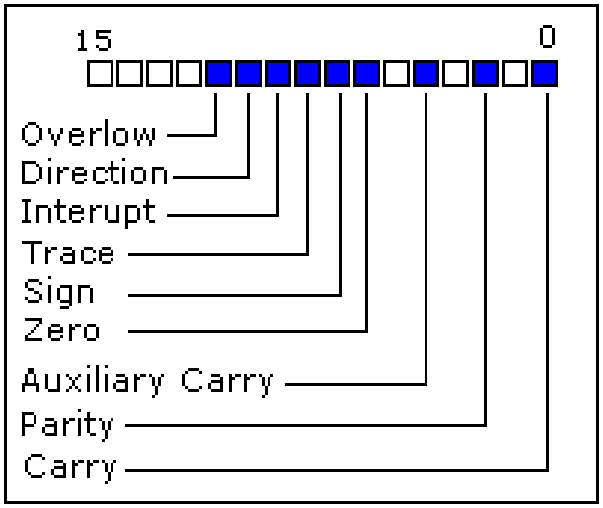
IP - the instruction pointer:

1. Always points to next instruction to be executed
2. Offset address relative to CS

IP register always works together with CS segment register and it points to currently executing instruction.

Flags Register

Flags Register - determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.Generally you cannot access these registers directly.



1. Carry Flag (CF) - this flag is set to 1 when there is an unsigned overflow. For example when you add bytes 255 + 1 (result is not in range 0...255). When there is no overflow this flag is set to 0.
2. Parity Flag (PF) - this flag is set to 1 when there is even number of '1'bits in result, and to 0 when there is odd number of '1' bits.
3. Auxiliary Flag (AF) - set to 1 when there is an unsigned overflow for low nibble (4 bits).
4. Zero Flag (ZF) - set to 1 when result is zero. For non-zero result this flag is set to 0.
5. Sign Flag (SF) - set to 1 when result is negative. When result is positive it is set to0. (This flag takes the value of the most significant bit.)
6. Trap Flag (TF) - Used for on-chip debugging.
7. Interrupt enable Flag (IF) - when this flag is set to 1 CPU reacts to interrupts from external devices.
8. Direction Flag (DF) - this flag is used by some instructions to process data chains, when this flag is set to 0 - the processing is done forward, when this flag is set to 1the processing is done backward.
9. Overflow Flag (OF) - set to 1 when there is a signed overflow. For example, when you add bytes 100 + 50 (result is not in range -128...127).

8086 Segmented Memory

Microprocessor 8086 consists of 9 address registers CS,DS,SS,ES,SI,DI,SP,BP,IP. Address registers store address of instruction and data in memory. These values are used by the processor to access memory locations.8086 assigns a 20 bit physical address to its memory locations. Thus it is possible to address 2^{20} = 1 megabyte of memory. The physical addresses are represented as:

00000h
00001h
00002h
……
……
FFFFFh

Segmented memory is the direct consequence of using 20 bit address in a 16 bit processor. The address are too big to fit in a 16 bit register or memory word. The 8086 gets around this problem by portioning its memory into segments.

A memory segment is a block of 2^16 (64K) consecutive memory bytes. Each segment is defined by a segment number. A segment number is 16 bit, so the highest segment number is FFFFh.

Within a segment, a memory location is specified by giving an offset. The offset is the number of byte beginning from the segment. With a 64KB segment , the offset can be given as a 16 bit number. The first byte in a segment has offset 0. The last offset in a segment is FFFFh.

A memory location may be specified by providing a segment number and an offset, written in the form **segment:offset .** This is known as logical address. For example, 1234:FF67 means offset number FF67 of segment 1234. The 20 bit physical address can be calculated by multiplying the segment number with 10h and then adding the offset with the result. For example the physical address for A4FB:4872 is

A4FB0
+4872
A9822

Segment 0 starts at 0000:0000=00000h and ends at 0000:FFFF=0FFFFh. Segment 1 starts at 0001:0000=00010h and ends at 0001:FFFF=1000Fh. So there is a lot of overlapping between segments. Because segments may overlap, the segment:offset form of an address is not unique, that is the same physical address can be represented in different segment:offset combinations. For example, 1256Ah=1256:000A, that is physical address 1256Ah can be represented as offset 000A of segment 1256. Again the same physical address 1256Ah can be represented as offset 016A of segment 1240 as

1256Ah=1240:016A.

There are several advantages of working with the segmented memory. First of all, after initializing the 16 bit segment registers, the 8086 has to deal with only 16 bit effective addresses. That is 8086 has to store and manipulate only 16 bit address components as both segment and offset are 16 bits.

8086 Instruction and Assembly language

8086 instruction set consists of the following instructions:

- **Data Transfer**

¤ MOV AX, BX; register: move contents of BX to AX

¤ COUNT to AX
MOV AX, COUNT; direct: move contents of the address labeled
¤ MOV CX, 0F0H; immediate: load CX with the value 240

¤ MOV CX, [0F0H]; memory: load CX with the value ataddress 240

¤ MOV [BX], AL; register indirect: move contents of ALto memory location in BX

16-bit registers can be pushed (the SP is first decremented by two and then the is valuestored at the address in SP) or popped (the value is restored from the memory at SP andthen SP is incremented by 2). For example:

¤ PUSH AX ; push contents of AX
¤ POP BX ; restore into B

- **I/O Operations**

The 8086 has separate I/O and memory address spaces. Values in the I/O space areaccessed with IN and OUT instructions. The port address is loaded into DX and the datais read/written to/from AL or AX:
¤ MOV DX,372H ; load DX with port address
¤ OUT DX,AL ; output byte in AL to port372 (hex)
¤ IN AX,DX ; input word to AX

- **Arithmetic/Logic**

Arithmetic and logic instructions can be performed on byte and 16-bit values. The firstoperand has to be a register and the result is stored in that register.
¤ ADD BX, 4; increment BX by 4

¤ ADD AX, CX; AX= AX + CX

¤ SUB AL, 1; subtract 1 from AL

¤ SUB DX, CX;DX= DX – CX

¤ INC BX; increment BX

¤ CMP AX, 54h; compare (subtract and set flags but without storing result)

¤ XOR AX, AX; clear AX

- **Control Transfer**

Conditional jumps transfer control to another address depending on the values of theflags in the flag register. Conditional jumps are restricted to a range of -128 to +127 bytesfrom the next instruction while unconditional jumps can be to any point.

¤ JZ skip; jump to label defined as ‘skip’ if last result was zero (two values equal)

¤ JGE notneg; jump to label defined as‘notneg’ if greater than or equal

¤ JB smaller; jump to label defined as ‘smaller’ if below

¤ JMP loop; unconditional jump to a label defined by ‘loop’

*all jump instructions jump to a level defined by a label. Label can be any name; used to define a specific location of code as needed by the programmer.

Important tips

There are some things to note about Intel assembly language syntax:

- The order of the operands is *destinatio*n, *source*
- Semicolons begin a comment
- The suffix ’H’ is used to indicate a hexadecimal constant, if the constant begins with a letter it must be prefixed with a zero to distinguish it from a label
- The suffix ’B’ indicates a binary constant
- Square brackets indicate indirect addressing or direct addressing to memory (with a constant)
- The size of the transfer (byte or word) is determined by the size of the *register*

Sample Programs

1. **Exchange program**

Table: Sample Code of Value Exchange Program in General Register.

Source Code
code segment
assume cs:code, ds:code
mov ax, 1234h
mov bx, 5678h
mov cx, ax
mov ax, bx
mov bx, cx
hlt
code ends
end

Source Code
code segment
assume cs:code, ds:code
mov bx, 1234h
mov cx, 5678h
xchg bx, cx
hlt
code ends
end

2. Addition program

Source Code
org 100h
code segment
assume cs:code, ds:code
mov bx, 1234h
mov cx, 5678h
add bx, cx
mov al, 13h
mov dl, 01h
add al, dl
hlt
code ends
end

3. Subtraction program

Source Code
org 100h
code segment
assume cs:code, ds:code
mov bx, 1234h
mov cx, 5678h
sub bx, cx
mov al, 13h
mov dh, 01h
sub al, dh
hlt
code ends
end

Some common instructions used :

inc **Increment by 1**
Syntax: inc op
op: register or memory
Action: op = op + 1

dec **Decrement by 1**
Syntax: dec op

op: register or memory
Action: op = op - 1

mul **Unsigned multiply**
Syntax: mul op8
mul op16
op8: 8-bit register or memory
op16: 16-bit register or memory
Action: If operand is op8, unsigned AX = AL * op8
If operand is op16, unsigned DX::AX = AX * op16

div **Unsigned divide**
Syntax: div op8
div op16
op8: 8-bit register or memory
op16: 16-bit register or memory
Action: If operand is op8, unsigned AL = AX / op8 and AH = AX % op8
If operand is op16, unsigned AX = DX::AX / op16 and DX = DX::AX % op16

LOOP B1

The LOOP instruction is a combination of a decrement of CX and a conditional jump. In the 8086, LOOP decrements CX and if CX is not equal to zero, it jumps to the address indicated by the label B1. If CX becomes a 0, the next sequential instruction executes.

Sample Program

4. Summation of a series.
[1+2+3+4+……+N] = BX
The value of N is stored in CX.

Source code
code segment
assume cs:code, ds:code
xor bx, bx
mov cx, 9
start:
add bx, cx
loop start
hlt
code ends
end

The 8086 processors lets the user to access the memory in different ways. The 8086 memory addressing modes provide flexible access to memory, allowing one to easily access variables, arrays, records, pointers, and other complex data types.

8086 Register Addressing Modes

Most 8086 instructions can operate on the 8086's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 8086 mov (move) instruction:

```
mov       destination, source
```

This instruction copies the data from the source operand to the destination operand. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 8086 mov instructions:

```
mov     ax, bx  ;Copies the value from BX into AX
mov     dl, al    ;Copies the value from AL into DL
 mov     si, dx   ;Copies the value from DX into SI
 mov     sp, bp  ;Copies the value from BP into SP
mov     dh, cl   ;Copies the value from CL into DH
          mov     ax, ax  ;Yes, this is legal!
```

The registers are the best place to keep often used variables.
In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, one may not specify cs as the destination operand; second, only one of the operands can be a segment register. Data cannot be moved from one **segment register** to another with a single mov instruction. To copy the value of cs to ds, you'd have to use some sequence like:

```
mov     ax, cs
mov     ds, ax
```

Segment registers should not be programmed as data registers to hold arbitrary values. They should only contain segment addresses.

8086 Memory Addressing Modes

Direct Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the displacement-only (or direct) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory location 8088h. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h. The direct addressing mode is perfect for accessing simple variables.

By default, all direct values provide offsets into the data segment. If it is needed to provide an offset into a different segment, a segment override prefix must be used before the address. For example, to access location 1234h in the extra segment (es) an instruction of the following form has to be used.

```
mov ax,es:[1234h]
mov ds,ax
```

Likewise, to access this location in the code segment, the instruction mov ax, cs:[1234h]has to be used. The ds: prefix in the previous examples is not a segment override.

Register Indirect Addressing Mode

The 8086 CPUs let the users to access memory indirectly through a register using the register **indirect addressing modes**. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the **ds** segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

For example if DS contains 1000h and BX contains 2345h, Instruction mov al,[bx] will move the content of memory location 12345h into al. Similarly instruction mov ax,[bx] will move the content of memory location 12345h and 12346h into AX.

Indexed Addressing Mode

In indexed addressing mode the 20 bit physical address is calculated as:

Physical address= [Value at the segment register]*10+offset+displacement.
The offset value can be stored at any index register, either SI or DI. The indexed addressing modes use the following syntax:

```
mov     al, disp[si]
mov     al, disp[di]
```

For example if DS contains 1000h and si contains 2345h and a equ 2,
Instruction mov al,a[si] will move the content of memory location 12347h into al.
Similarly instruction mov ax,a[si] will move the content of memory location 12347h and 12348h into AX.

Based Addressing Mode

Based addressing mode is similar to indexed addressing mode. The only difference is, here BX and BP register is used to hold offset value. The based addressing modes use the following syntax:

```
mov     al, disp[bx]
mov     al, disp[bp]
```

Based Indexed Addressing Mode

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

In based indexed addressing mode the 20 bit physical address is calculated as:

Physical address= [Value at the segment register]*10+base register value + index register value + displacement.

Pre-Lab Homework:

- Study the theory and methodology part before attending the lab.
- Consult the book "Microprocessors and Micro-Computer based System Design", Second edition – by Dr. M. Rafiquzzaman.

Equipment:

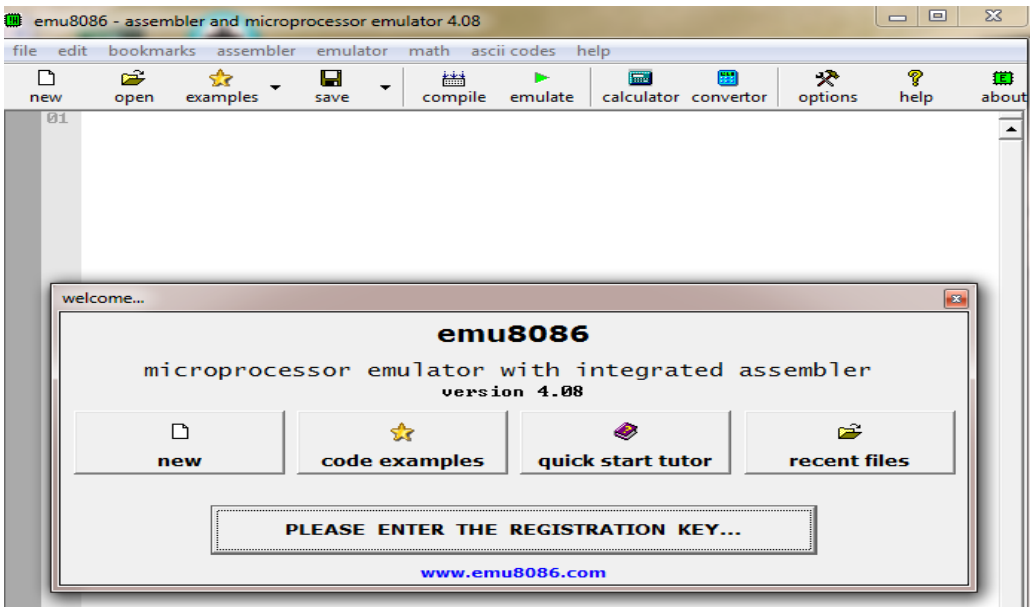
1) EMU8086 [ver.408 (32 bit WINOS compatible)]
2) PC having Intel Microprocessor

Precautions: The assembly language is case sensitive so while writing the code make sure the functions are correctly typed, else while emulating in the software, errors may get displayed in emulator assembler window.

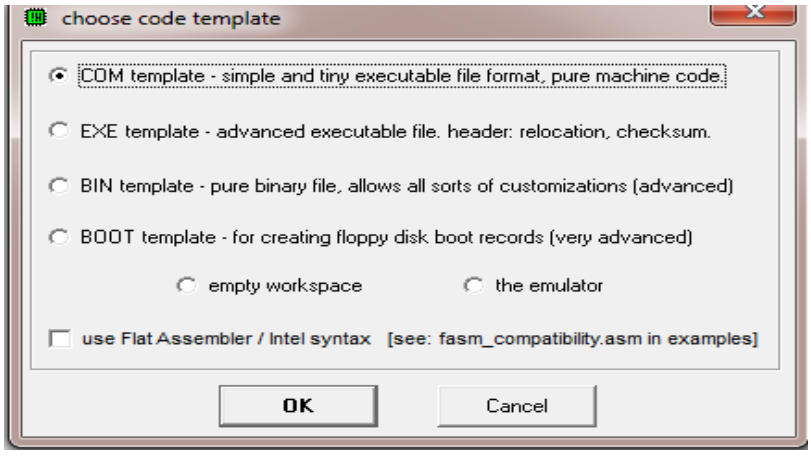
Lab Procedure:

1. ***Familiarize with emulator EMU8086***

✓ Emulator EMU8086 software is used for assembly programming. Open the EMU8086 from start menu.

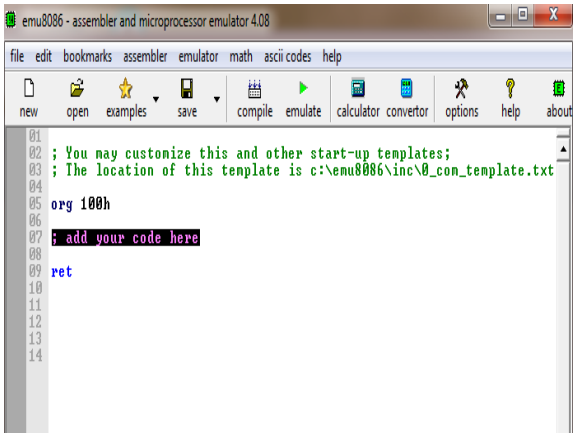


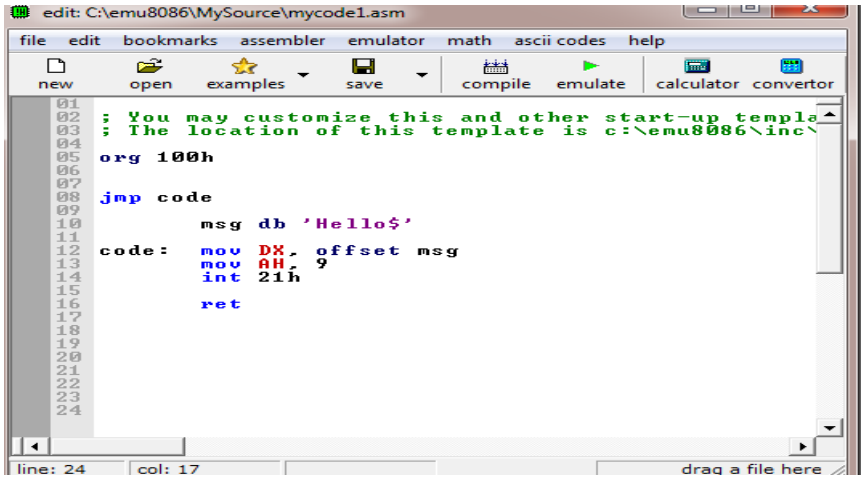
✓ Select new tab



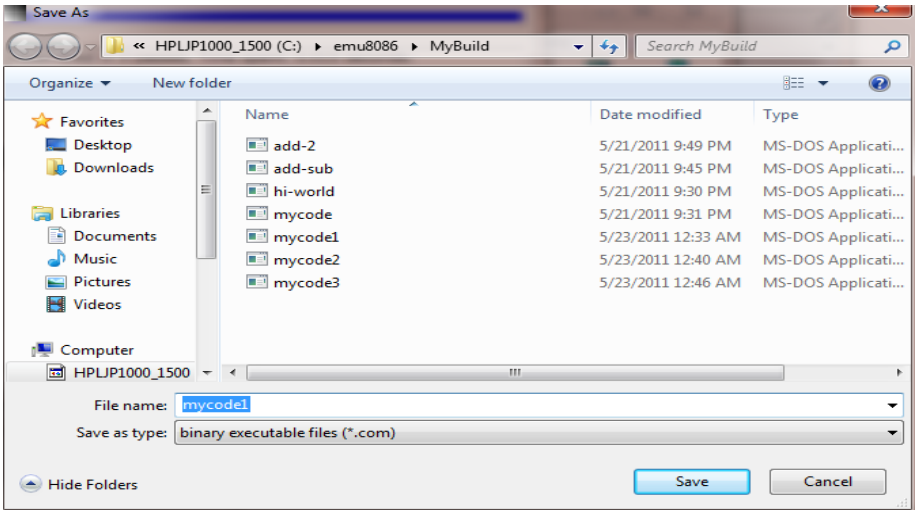
✓ Select 1st option and press ok.

✓ Write your code in the highlighted area.

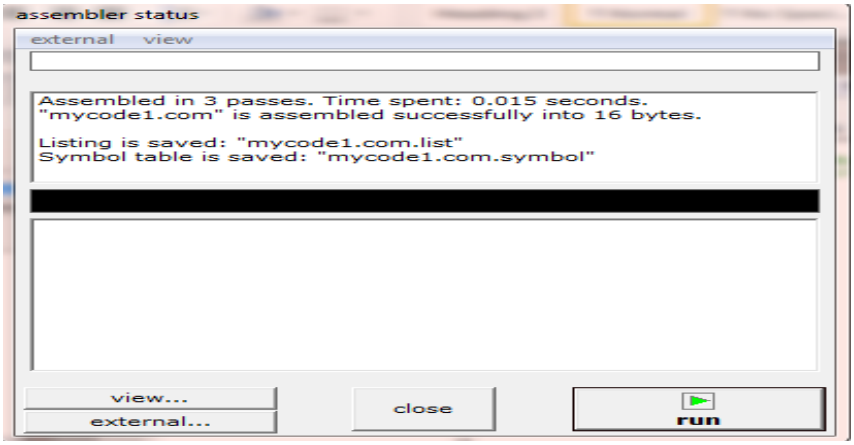




✓ Compile your code.

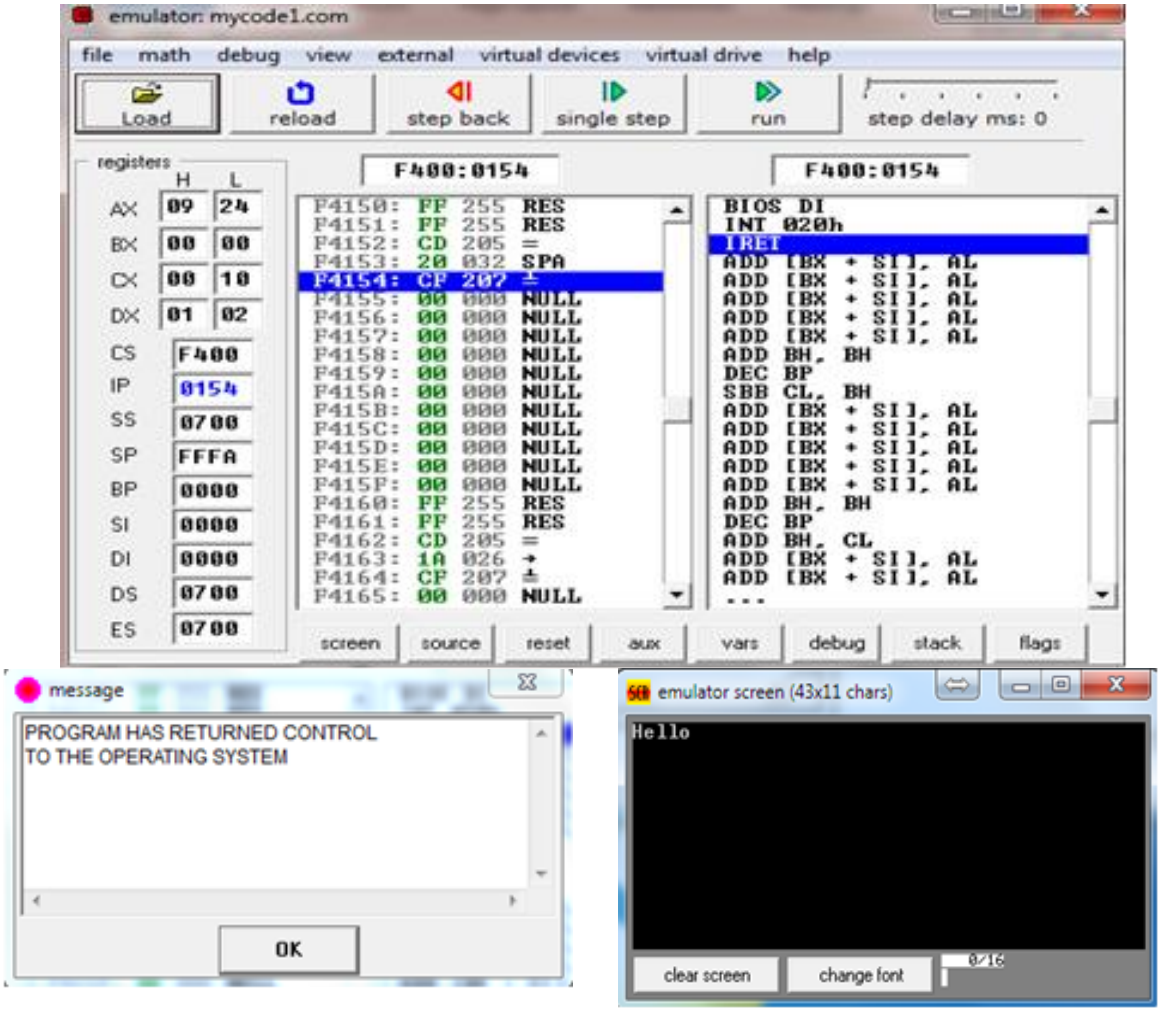


✓ Click on save.



✓ Click on run button.

✓ Following window will appear.



Lab Task:

1. Implement all the example codes given above in emulator EMU8086 and take note of all general register values.
2. Write the assembly language program for DX= AX + BX – CX
   Show the result on emulator screen of DX.

3. Write a program which display two charaters at column#12 and row#7 at emulator screen.

Hints for 3rd lab task :

Single step run button executes the code in step wise and shows the the values of registers. Run button executes the program at once. Load button use the load the program. Reload is used to load again your program.

To display a number on screen, you have to add followings in your program:

```
org  100h  ;

include
"emu8086.inc"

GOTOXY 12,7
PUTC 65
PUTC 'B'

ret
```

Ouput will be **AB**

You can select the column and row by using the following command:

GOTOXY 10, 5

4. Write the assembly code for the following sequence 1+3+5+7…..+N. Where N = 5 using loop. the code and result is given below (done in emulator):

```
org 100h

mov ax,0h
mov bx,1h
mov cx, 5h

l1:
   add ax,bx
   inc bx
   inc bx
   loop l1

ret
```

Questions for Report writing:

1. Include all codes' list file printout following lab report writing template mentioned in appendix A.
2. What is the advantage of having overlapping segments in 8086 memory system?
3. For a memory location with physical address 1256Ah, Calculate the address in segment:offset form for segments 1256h and 1240h.
4. What are the different data addressing modes available in 8086? Briefly explain each of them with examples.
5. Write a code for finding the value of 6!

Discussion:

In this laboratory our main focus would be to teach the basic idea about assembly language by the help of emulator and 8086 board. Student will observe the results of both simulated and implemented program.

Conclusion:

While writing each programs, its important to understand the purpose behind using each functions and the associated values. It is very important to understand each line of code, so that an individual can solve a given a task or problem using the common functions that were learnt from this experiment.

References:

1. "Microprocessors and Micro-Computer based System Design", Second edition – by Dr. M. Rafiquzzaman
2. EMU8086 Manual