Title: Introduction to 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".

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 become the basic x86- architecture of Intel's future processors.

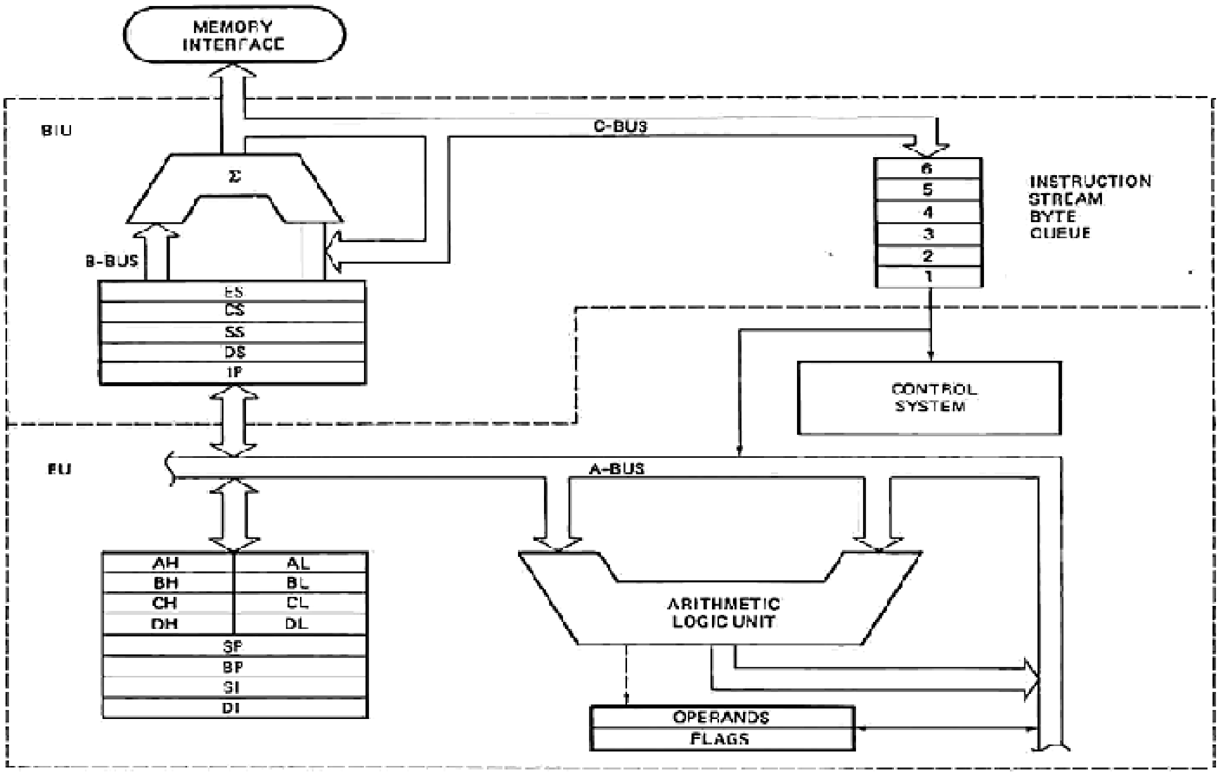


Fig 1: Intel 8086 internal architecture

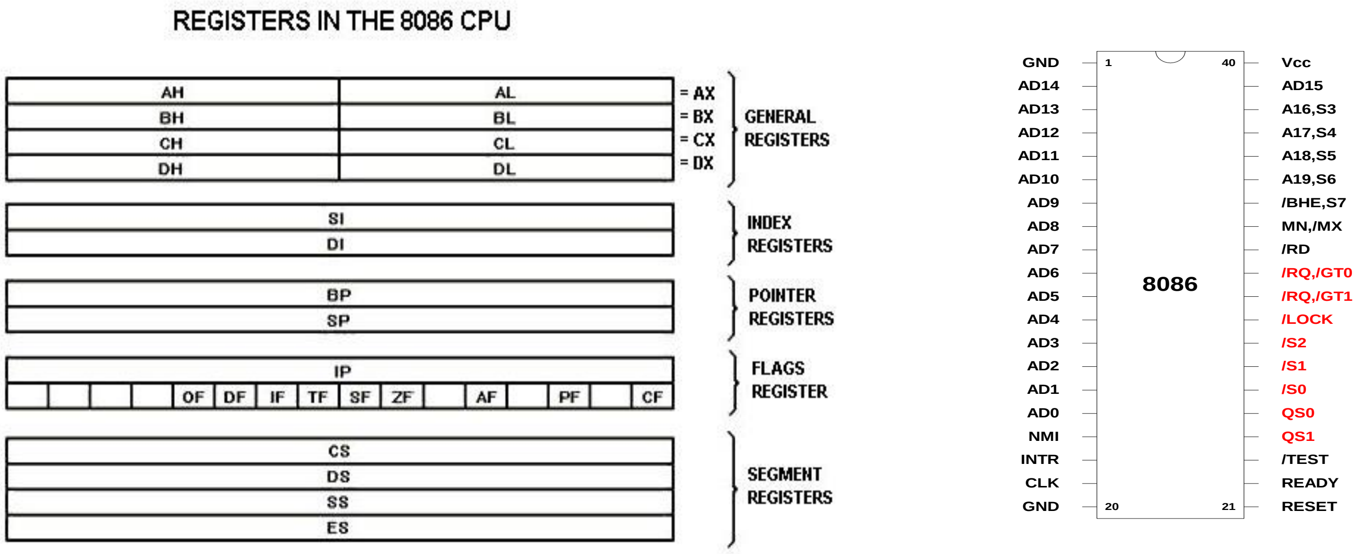


Fig 2: Internal Diagram, Registers and PIN diagram of the 8086 microprocessor

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 = FFFEh

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

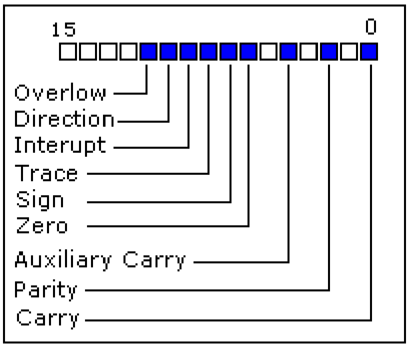
IP - the instruction pointer:

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

$$\begin{array}{r} \text{A4FB0} \\ \underline{+4872} \\ \text{A9882} \end{array}$$

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**

```
; register: move contents of BX to AX
MOV AX,BX
; direct: move contents of the address labeled
; COUNT to AX
MOV AX,COUNT
; immediate: load CX with the value 240
MOV CX,0F0H
; memory: load CX with the value at
; address 240
MOV CX,[0F0H]
```

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

16-bit registers can be pushed (the SP is first decremented by two and then the is value stored at the address in SP) or popped (the value is restored from the memory at SP and then 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 are accessed with IN and OUT instructions. The port address is loaded into DX and the data is read/written to/from AL or AX:

```
MOV DX,372H ; load DX with port address
OUT DX,AL ; output byte in AL to port
; 372 (hex)
IN AX,DX ; input word to AX
```

- **Arithmetic/Logic**

Arithmetic and logic instructions can be performed on byte and 16-bit values. The first operand has to be a register and the result is stored in that register.

```
; increment BX by 4
ADD BX,4
;AX= AX + CX
ADD AX,Cx
; subtract 1 from AL
SUB AL,1
;DX= DX – CX
SUB DX,CX
; increment BX
INC BX
; compare (subtract and set flags
; but without storing result)
CMP AX,54h
; clear AX
XOR AX,AX
```

- **Control Transfer**

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

```
; jump if last result was zero (two values equal)
JZ skip
```

; jump if greater than or equal
JGE notneg
; jump if below
JB smaller
; unconditional jump:
JMP loop

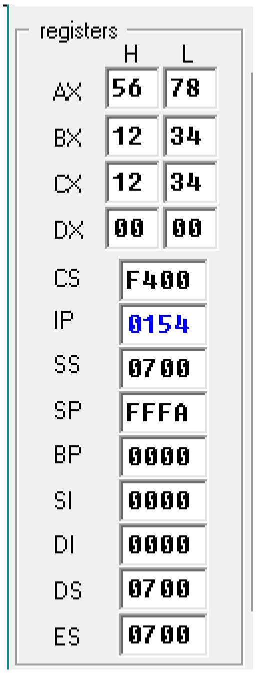
Equipment:

- Microprocessor 8086 Trainer Board (MTS-86c)
- EMU8086 [ver.408 (32 bit WINOS compatible)]
- PC having Intel Microprocessor

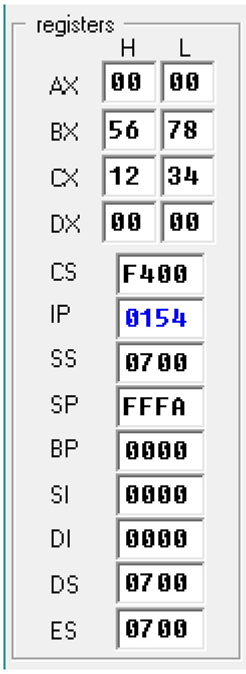
Lab Task:

- **Exchange program**

```
org 100h

mov ax,1234h
mov bx,5678h

mov cx,ax
mov ax,bx
mov bx,cx

ret
```

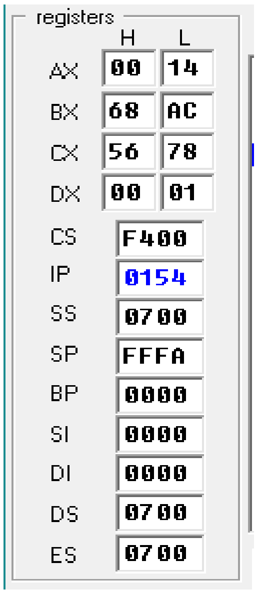


```
org 100h

mov bx,1234h
mov cx,5678h

xchg bx,cx

ret
```



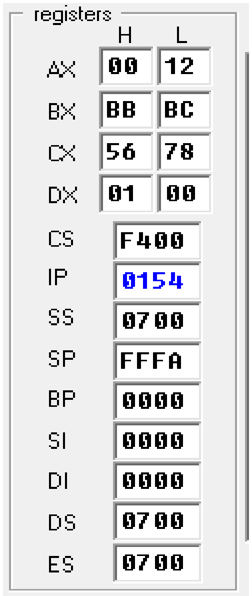
- **<u>Addition Program</u>**

```
org 100h

mov bx,1234h
mov cx,5678h

add bx,cx

mov al,13h
mov dl,01h

add al,dl

ret
```



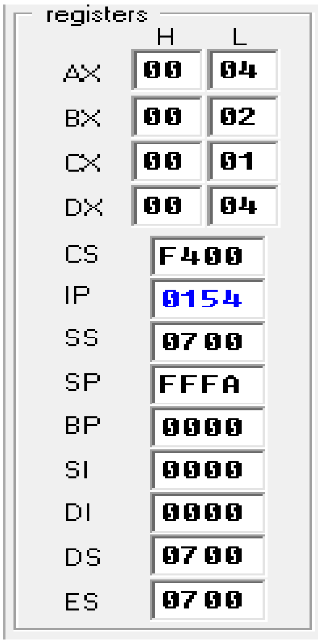
- **Subtraction Program**

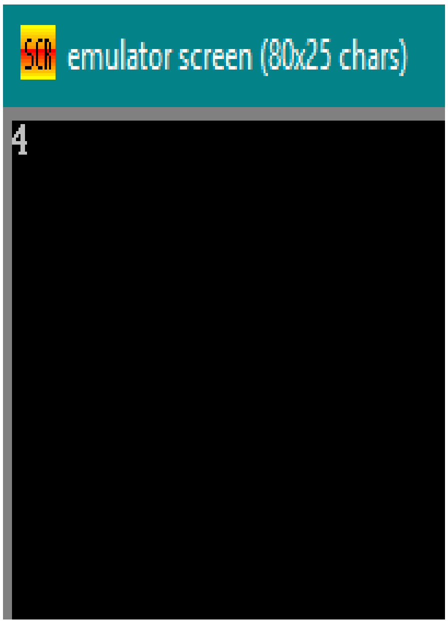
```
org 100h

mov bx,1234h
mov cx,5678h

sub bx,cx

mov al,13h
mov dh,01h

sub al,dh

ret
```



- **DX= AX + BX - CX**

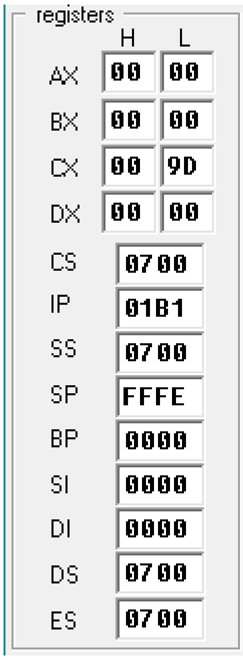
```
org 100h

include "emu8086.inc"
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS

mov dx,0000h
mov ax,0003h
mov bx,0002h
mov cx,0001h

add dx,ax
add dx,bx

sub dx,cx

mov ax,dx
CALL PRINT_NUM

ret
```

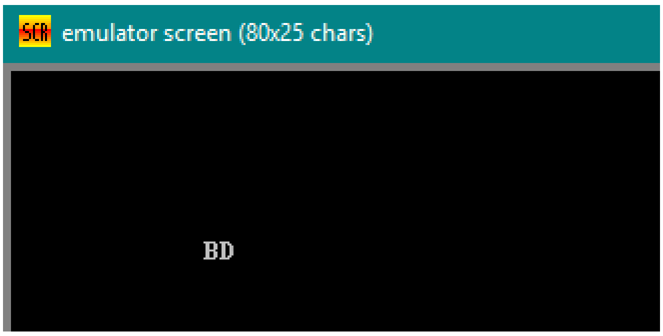




- **<u>Two charaters at column#12 and row#7</u>**

```
org 100h

include "emu8086.inc"
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS

GOTOXY 12,7

PUTC 66
PUTC 68

call PRINT_NUM
```





Precautions:

The procedure was followed carefully when entering input in MTS-86c board. Pen drive wasn't used in the computers.

Lab Task:

1. All the example code is mentioned above in emulator EMU8086 and MTS-86c and all general register values was noted.
2. The program for **<u>DX= AX + BX - CX</u>** is mentioned above and the result is shown on emulator screen of DX.
3. A program which display two charaters at column#12 and row#7 is mentioned above with emulator screen.

Report:

1. All codes are attached above which are mentioned in appendix A.

2. What is the advantage of having overlapping segments in 8086 memory system?
   Ans- It reduces the internal fragment. That is why we do not need to worry about the segment, we need to consider only the offset.

3. We know that,
   Physical address=segment*10+offset

   For segment 1256h,

1256Ah	=	1256h*10 + offset
Offset	=	1256Ah - 1256h * 10h
	=	000Ah

   Segment : offset = 1256h : 000Ah

   For segment 1240h,

1256Ah	=	1240h * 10 + offset
Offset	=	1256Ah - 1240h * 10h
	=	016Ah

   So, Segment : offset = 1256h : 016Ah

Conclusion and Discussion:

After completing the lab, we have learnt about the internal structure of 8086 microprocessor and different kinds of register. We have also learned how registers value get affected during assembly operations. It was explained how an assembly code is converted into object code and how it works. We have also learn how the emu 8086 software works and how the MTS-86 works.