Experiment 7

Control Instructions

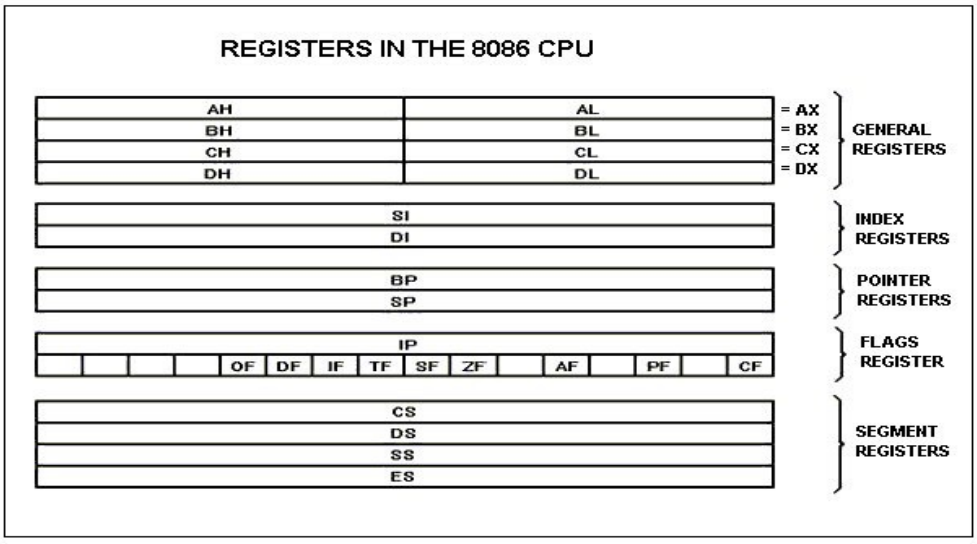
Introduction:

In this experiment you will be introduced to the 8086 Processor's Architecture and Assembly program's control structure. You will practice how to control the flow of an assembly language program using the compare instruction, the different jump instructions and the loop instructions.

After this lab you will be able to:

- Work easily with different registers of 8086 processor
- Understand what is the Flag register and how flags are effected
- Use Compare Instruction.
- Use Jump Instructions.
- Use Loop Instructions.

Inside the 8086 Processor:



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. 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

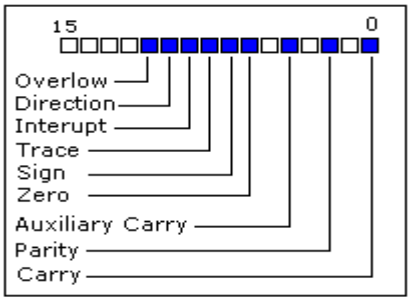
IP - the instruction pointer:

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 one bits in result, and to 0 when there is odd number of one 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 to 0. (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 1 the 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).

Note: You can see flag register from view->flags in emulator window.

Control Structure

Compare instruction:

The compare instruction is used to compare two numbers. At most one of these numbers may reside in memory. The compare instruction subtracts its source operand from its destination operand and sets the value of the status flags according to the subtraction result. The result of the subtraction is not stored anywhere. The flags are set as indicated in Table 5. 1.

Table 5.1: The Compare Instruction of the 8086 Microprocessor

Instruction	Example	Meaning
CMP	CMP AX, BX	If (AX = BX) then ZF ← 1 and CF ← 0
		If (AX < BX) then ZF ← 0 and CF ← 1
		If (AX > BX) then ZF ← 0 and CF ← 0

Jump Instructions:

The jump instructions are used to transfer the flow of the process to the indicated operator. When the jump address is within the same segment, the jump is called intra-segment jump. When this address is outside the current segment, the jump is called inter-segment jump. An overview of all the jump instructions is given in Table 5. 2.

Table 5.2: Jump Instructions of the 8086 Microprocessor

Type	Instruction		Meaning (jump if)	Condition
Unconditional	JMP		unconditional	None
Comparisons	JA	jnbe	above (not below or equal)	CF = 0 and ZF = 0
	JAE	jnb	above or equal (not below)	CF = 0
	JB	jnae	below (not above or equal)	CF = 1
	JBE	jna	below or equal (not above)	CF = 1 or ZF = 1
	JE	jz	equal (zero)	ZF = 1
	JNE	jnz	not equal (not zero)	ZF = 0
	JG	jnle	greater (not lower or equal)	ZF = 0 and SF = OF
	JGE	jnl	greater or equal (not lower)	SF = OF
	JL	jnge	lower (not greater or equal)	(SF xor OF) = 1 i.e. SF ≠ OF
	JLE	jng	lower or equal (not greater)	(SF xor OF or ZF) = 1
	JCXZ	loop	CX register is zero	(CF or ZF) = 0
Carry	JC		Carry	CF = 1
	JNC		no carry	CF = 0
Overflow	JNO		no overflow	OF = 0
	JO		overflow	OF =1
Parity Test	JNP	jpo	no parity (parity odd)	PF = 0
	JP	jpe	parity (parity even)	PF = 1
Sign Bit	JNS		no sign	SF = 0
	JS		sign	SF = 1
Zero Flag	JZ		zero	ZF = 1
	JNZ		non-zero	ZF = 0

The LOOP Instructions:

The LOOP instruction is a combination of a DEC and JNZ instructions. It causes execution to branch to the address associated with the LOOP instruction. The branching occurs a number of times equal to the number stored in the CX register. All LOOP instructions are summarized in Table 6. 6.

Instructions	Example	Meaning
LOOP	LOOP LABEL1	If (CX≠0) then IP Offset Label1
LOOPE LOOPZ	LOOPE Label1	If (CX≠0 and ZF =0) then IP Offset Label1
LOOPNE LOOPNZ	LOOPNZ Label 1	If (CX≠0 and ZF =0then IP Offset Label1

Table 6. 6: Summary of the LOOP Instructions.

The Loop Program Structure, Repeat-Until and While-Do:

Like the conditional and unconditional jump instructions which can be used to simulate the IF-Then-Else structure of any programming language, the Loop instructions can be used to simulate the Repeat-Until

and While-Do loops. These are used as shown in the following

Structure	Repeat-Until	Do
Code	; Repeat until CX = 0 - MOV CX, COUNT Again: - - - - - LOOP Again	(CX ≠ 0) Do - MOV CX, COUNT Again: JZ Next - - - LOOP Again Next: -

ASSEMBLER PROGRAM

```
mov bx, 0  ; total step counter

mov cx, 5
k1: add bx, 1
   mov al, '1'
   mov ah, 0eh
   int 10h
   push cx
   mov cx, 5
     k2: add bx, 1
     mov al, '2'
     mov ah, 0eh
     int 10h
     push cx
        mov cx, 5
        k3: add bx, 1
        mov al, '3'
        mov ah, 0eh
        int 10h
        loop k3    ; internal in internal loop.
     pop  cx
     loop  k2      ; internal loop.
   pop cx
loop k1            ; external loop.

; wait any key...
mov ah, 1
int 21h
```

OBSERVATIONS

By using single stepping observe the contents of registers AX, BX after execution of each instruction

EXERCISE

1. Three numbers are entered by users. Arranged these numbers in registers AX, BX, CX respectively in ascending numerical order with AX containing the smallest, BX the next largest and CX the largest of all.

2. The user types any of the characters A, B, ..., Z and it and the next character in the ASCII code definition are displayed on the screen followed by a new line. Then the sequence of events is repeated. Thus, if the user types the letters S,T, R,A,N,G,E the resulting display should be:

ST
TU
RS
AB
NO

GH
EF

3. Asequence of numbers can be generated by starting with a whole number n, n>0, and then applying the ru :
    a. If n=1 then stop
    b. If n is even, then the next number in the sequence, m, is n/2
    c. If n is odd then the next number in the sequence , m, is (3*n) +1

    and then applying these rules to generate the next number in the sequence from m, and so on. Thus, the sequence beginning with n =5 is 5,16,8,4,2,1.

4. Write a program which, given an initial n in AX, 1<n<20D, generates the appropriate sequence and stores it in memory.