Experiment No 6
Shift Rotate and Jump Instructions

Introduction:

In this experiment you will be introduced to the shift and rotate instructions. You will also practice how to control the flow of an assembly language program using the compare instruction, the different jump instructions and the loop instructions.

Objectives:

1- Shift Instructions.
2- Rotate Instructions.
3- Compare Instruction.
4- Jump Instructions.
5- Loop Instructions.

References:

- Lecture notes.

The Shift Operations:

The shift operations are used to multiply or divide a number by another number that is a power of 2 (i.e. $2^n$ or $2^{-n}$). Multiplication by 2 is achieved by a one-bit left shift, while division by 2 is achieved by a one-bit right shift. The Shift Arithmetic Right (SAR) instruction, is used to manipulate signed numbers. The regular Right Shift (SHR) of a signed number affects the sign bit, which could cause numbers to change their sign. The SAR preserves the sign bit by filling the vacated bits with the sign of the number. Shift Arithmetic Left (SAL) is identical in operation to SAR.

The rotate operations are very similar to the shift operations, but the bits are shifted out from one end of a number and fed back into the other end to fill the vacated bits. They are provided to facilitate the shift of long numbers (i.e. numbers of more than 16 bits). They are also used to reposition a certain bit of a number into a desired bit-location. The rotate right or left instructions through the carry flag (RCL and RCR) are similar to the regular rotate instructions (ROL and ROR), but the carry flag is considered as a part of the number. Hence, before the rotate operation, the carry flag bit is appended to the number as the least significant bit in the case of RCL, or as the most significant bit in the case of RCR.
### Table 6.1: Summary of the Shift and Rotate Instructions of the 8086 Microprocessor

#### 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 6.2.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>CMP AX, BX</td>
<td>If (AX = BX) then ZF ← 1 and CF ← 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (AX &lt; BX) then ZF ← 0 and CF ← 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (AX &gt; BX) then ZF ← 0 and CF ← 0</td>
</tr>
</tbody>
</table>

#### 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 6.3. Table 6.4 lists the possible addressing modes used with the jump instructions. Whereas, Table 6.5 gives examples on the use of such instructions.
<table>
<thead>
<tr>
<th>Type</th>
<th>Instruction</th>
<th>Meaning (jump if)</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional</td>
<td>JMP</td>
<td>unconditional</td>
<td>None</td>
</tr>
<tr>
<td>Comparisons</td>
<td>JA</td>
<td>above (not below or equal) CF = 0 and ZF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAE</td>
<td>above or equal (not below) CF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JB</td>
<td>below (not above or equal) CF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JBE</td>
<td>below or equal (not above) CF = 1 or ZF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JE</td>
<td>equal ( zero) ZF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JNE</td>
<td>not equal (not zero) ZF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JG</td>
<td>greater (not lower or equal) ZF = 0 and SF = OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JGE</td>
<td>greater or equal (not lower) SF = OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JL</td>
<td>lower (not greater or equal) (SF xor OF) = 1 i.e. SF ≠ OF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JLE</td>
<td>lower or equal (not greater) (SF xor OF or ZF) = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JCXZ</td>
<td>loop CX register is zero (CF or ZF) = 0</td>
<td></td>
</tr>
<tr>
<td>Carry</td>
<td>JC</td>
<td>Carry CF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JNC</td>
<td>no carry CF = 0</td>
<td></td>
</tr>
<tr>
<td>Overflow</td>
<td>JNO</td>
<td>no overflow OF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JO</td>
<td>overflow OF = 1</td>
<td></td>
</tr>
<tr>
<td>Parity Test</td>
<td>JNP</td>
<td>no parity (parity odd) PF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JP</td>
<td>parity (parity even) PF = 1</td>
<td></td>
</tr>
<tr>
<td>Sign Bit</td>
<td>JNS</td>
<td>no sign SF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JS</td>
<td>sign SF = 1</td>
<td></td>
</tr>
<tr>
<td>Zero Flag</td>
<td>JZ</td>
<td>zero ZF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JNZ</td>
<td>non-zero ZF = 0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.3:** Jump Instructions of the 8086 Microprocessor

<table>
<thead>
<tr>
<th>Label Pointer</th>
<th>Range</th>
<th>Addressing Mode</th>
<th>Specified By</th>
<th>Encoded As</th>
<th>Directive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>+127/-128 bytes IP ↔ IP + Offset</td>
<td>Immediate Word</td>
<td>Differentially</td>
<td>SHORT PTR</td>
<td></td>
</tr>
<tr>
<td>Near</td>
<td>Intra-segment IP ↔ Address</td>
<td>Immediate Word</td>
<td>Differentially</td>
<td>NEAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Word</td>
<td>Absolute address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory Word</td>
<td>Absolute address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Far</td>
<td>Inter-segment IP ↔ Address CS ↔ Segment</td>
<td>Immediate Double Word</td>
<td>Absolute address</td>
<td>FAR PTR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory Double Word</td>
<td>Absolute address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Differentially = Difference between current and next address.

**Table 6.4:** Jump Instructions and Addressing Modes
### Table 6.5: Examples of Jump Instructions of the 8086 Microprocessor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>JMP FAR PTR [BX]</td>
<td>IP ← [BX], CS ← [BX+2]</td>
</tr>
<tr>
<td>JNZ</td>
<td>JNZ END</td>
<td>If (ZF=0) Then IP ← Offset of END</td>
</tr>
<tr>
<td>JE</td>
<td>JE FIRST</td>
<td>If (ZF=1) Then IP ← Offset of FIRST</td>
</tr>
<tr>
<td>JC</td>
<td>JC SECOND</td>
<td>If (CF=1) Then IP ← Offset of SECOND</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>LOOP Label1</td>
<td>If (CX≠0) then IP ← Offset Label1</td>
</tr>
<tr>
<td>LOOPE</td>
<td>LOOPE Label1</td>
<td>If (CX≠0 and ZF = 0) then IP ← Offset Label1</td>
</tr>
<tr>
<td>LOOPNZ</td>
<td>LOOPNZ Label1</td>
<td>If (CX≠0 and ZF = 0) then IP ← Offset Label1</td>
</tr>
</tbody>
</table>

### Table 6.6: Summary of the LOOP Instructions.

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

#### B. Like the condionnal and unconditionnal 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 (Table 6.7).

<table>
<thead>
<tr>
<th>Structure</th>
<th>Repeat-Until</th>
<th>While-Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>; Repeat until CX = 0</td>
<td>; While (CX ≠ 0) Do</td>
</tr>
<tr>
<td></td>
<td>Again: MOV CX, COUNT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Again: JZ Next</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Again: LOOP Again</td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.7: The Loop Program Structure.
Pre Lab Work:

1. Complete program 6.1, according to the given guidelines.
2. Check on some values and see if it is working properly.
3. Comment on the program, trying to understand how conversion is done.
4. Write program 6.2 and make sure it contains no errors.
5. Do the modifications given in the guidelines. This will be program 6.3.
6. Bring your work to the lab.

Lab Work:

1- Show program 6.1 and your comments to your lab instructor.
2- Run program 6.2 using CodeView.
3- See what is the effect of such program on NUM1.
4- Run program 6.3, and see the effect on NUM1 after displaying NUM2.
5- Modify program 6.2, using program 6.1, such that you enter an 8-bit binary number from the keyboard, and invert swap the high and the low nibbles of the number, and finally display it. Call this program 6.4.
6- Show all your work to the instructor, and submit it at the end of the lab session.

Lab Assignment:

Write a program that prompts the user to enter an 8 bit binary number, between 0 and 255. The program then inverts all bits according to the figure below. This program is useful in Signal Processing for the calculation of the Fast Fourier Transform (FFT). The operation is called Decimation (in time or frequency), and the bit manipulation is called bit shuffling i.e. rearrangement of bits.

![Bit Shuffling](image)

Bit \((i) \leftrightarrow Bit (7- i)\) for \(i = 0 – 7\)

**Figure 6.1:** Bit Shuffling
TITLE “Experiment 6, Program 1”
; This program adds 2 binary numbers and prints the result in binary format

.MODEL SMALL
.STACK 200
.DATA
    CRLF DB 0DH, 0AH, '$'
    PROMPT1 DB 'Enter the first 8-bit binary number: ', '$'
    PROMPT2 DB 'Enter the second 8-bit binary number: ', '$'
    PROMPT3 DB 'The sum of the two numbers in binary is: ', '$'
    NUM1 DW ?
    NUM2 DW ?

.CODE
.STARTUP
; DISPLAY PROMPT1
; READ AND CONVERT THE FIRST NUMBER
    CALL READ

    MOV NUM1,BX               ; READ FROM STACK

    MOV DX, OFFSET CRLF
    MOV AH, 09H
    INT 21H
; DISPLAY PROMPT2

; READ AND CONVERT THE SECOND NUMBER
    CALL READ
    MOV NUM2,BX               ; READ FROM STACK

    MOV BX, NUM1
    ADD BX, NUM2              ; ADD THE TWO NUMBERS
; DISPLAY PROMPT3
    CALL RESULT

.EXIT

;******************************************************************************
; PROC READ A NUMBER AND CONVERT IT TO BINARY
READ  PROC  NEAR
    MOV BX, 0000
    MOV CX, 0008
    MOV AH, 01H
    L1:    INT 21H
    SUB AL, 30H
    SHL BL, 1
    OR BL, AL
    LOOP L1
    XOR BH, BH
    RET

READ ENDP
END
; PROC. DISPLAY RESULT
RESULT PROC NEAR
    MOV CX, 0008   ; DISPLAYING
    CLC
NEXXX:    RCL BL, 1
    JNC BIT_0
    MOV DL, '1'
    MOV AH, 02H
    INT 21H
    JMP LAST
BIT_0:  MOV DL, '0'
    MOV AH, 02H
    INT 21H
LAST:     LOOP NEXT
    RET
RESULT ENDP

;******************************************************************************
; TITLE “PROGRAM 2 EXPERIMENT 6”
; This program shows how to manipulate a two-digit numbers

.MODEL SMALL
.STACK 200
.DATA
    NUM1 DB ?
    NUM2 DB ?

.CODE
.STARTUP
    ; READ NUMBER NUM1
    MOV AL, NUM1
    AND AL, 0FH
    MOV CL, 04H
    SHL AL,CL
    MOV BL, AL
    MOV AL, NUM1
    AND AL, 0F0H
    MOV CL, 04H
    SHR AL, CL
    OR BL, AL
    MOV NUM2, BL
    ; DISPLAY NUMBER NUM1
    ; CONVERT NUM2 TO BINARY
    ; DISPLAY NUMBER NUM2
.EXIT
END