8086 Assembly Language Programming

Learning Objectives

After reading this unit you should appreciate the following:

- Instruction set of 8086
- Assembler Directives and Operators
- A Few Machine Level Programs
- Machine coding and Programs
- Programming with an Assembler
- Assembly Language Example Programs

Instruction set of 8086

The 8086/8088 instructions are categorized into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

(i) Data Copy/Transfer Instructions: This type of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) Arithmetic and Logical Instructions: All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) Branch Instructions: These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

(iv) Loop Instructions: If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) Machine Control Instructions: These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) Flag Manipulation Instructions: All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD,STD,CLI,STI, etc. belong to this category of instructions.

(vii) Shift and Rotate Instructions: These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

(viii) String Instructions: These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

Data Copy/Transfer Instructions

MOV: Move This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general or special purpose registers or a memory location and, another register or memory location may act as destination.

However, in case of immediate addressing mode, a segment register cannot be a destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register. The following example instructions explain the fact.

Example 5.1

Load DS with 5000H.

1. `MOV DS, 5000H;` Not permitted (invalid)

   Thus to transfer an immediate data into the segment register, the correct procedure is given below.

2. `MOV AX, 5000H`

   `MOV DS, AX`

   It may be noted, here, that both the source and destination operands cannot be memory locations (Except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.

3. `MOV AX, 5000H;` Immediate
4. `MOV AX, BX;` Register
5. `MOV AX, [SI];` Indirect
6. `MOV AX, [200 OH];` Direct
7. `MOV AX, 50H [BX];` Based relative, 50H Displacement

PUSH: Push to Stack This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address. The examples of these instructions are as follows:

Example 5.2

1. `PUSH AX`
2. `PUSH DS`
3. `PUSH [500OH] ;` Content of location 5000H and 5001 H in DS are pushed onto the stack.

POP: Pop from Stack This instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

The examples of these instructions are as shown:

Example 5.3

```
1. POP  AX
2. POP  DS
3. POP  [5000H]
```

XCHG: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted. The examples are as shown:

Example 5.4

```
1. XCHG      [5000H], AX  ; This instruction exchanges data between AX and a
                          ; memory location [5000H] in the data segment.
2. XCHG      BX           ; This instruction exchanges data between AX and BX.
```

IN: Input the port This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit) which is allowed to carry the port address. The examples are given as shown:

Example 5.5

```
1. IN AL, 030 OH ; This instruction reads data from an 8-bit port whose address
                 ; is 0300H and stores it in AL.
2. IN AX         ; This instruction reads data from a 16-bit port whose
                 ; address is in DX (implicit) and stores it in AX.
```

OUT: Output to the Port This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8-D_{15} while that to an even addressed port is transferred on D_0-D_7. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. The examples are given as shown:

Example 5.6

```
1. OUT       0300H, AL    ; This sends data available in AL to a port whose
                          ; address is 0300H.
2. OUT       AX           ; This sends data available in AX to a port whose
                          ; address is specified implicitly in DX.
```

XLAT: Translate The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique. We will explain this instruction with the aid of the following example.

Suppose, a hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned in AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 7-segment code corresponding to the key pressed is found out and sent to the display port. This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H, and store the 7- segment codes for 0 to Fat the locations 2000H to 200FH sequentially. For executing the XLAT instruction, the code of the pressed key obtained from the keyboard (i.e. the code to be translated) is moved in AL and the base address of the look up table containing the 7-segment codes is kept in BX. After execution of the XLAT instruction, the 7-segment code corresponding to the pressed key is returned in AL, replacing the key code which was in AL prior to the execution of the XLAT instruction. To find out the exact address of the 7-segment code from the base address of look up table, the content of AL is added to BX internally, and the contents of the address pointed to by this new content of BX in DS are transferred to AL. The following sequence of instructions perform the task.

Example 5.7

```
MOV AX, SEG TABLE          ;Address of the segment containing look-up-table
MOV DS, AX                 ; is transferred in DS.
MOV AL, CODE               ; Code of the pressed key is transferred in AL.
MOV BX, OFFSET TABLE       ; Offset of the code look-up-table in BX.
XLAT                       ; Find the equivalent code and store in AL.
```

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is more useful for assembly language rather than for machine language. Suppose, in an assembly language program, a label ADR is used. The instruction LEA BX, ADR loads the offset of the label ADR in BX.

LDS/LES: Load Pointer to DS/ES The instruction , Load DS/ES with pointer, loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The ex

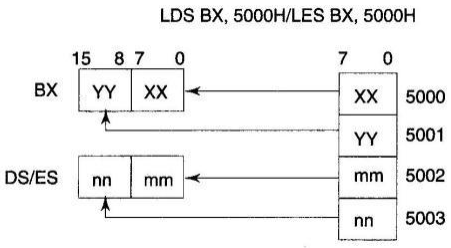


Figure 5.1: LDS/LES Instruction Execution

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except over flow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

Arithmetic Instructions

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The 8086/ 8088 instructions falling under this category are discussed below in significant details. The arithmetic instructions affect all the condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode.

ADD: Add This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected, depending upon the result. The examples of this instruction are given along with the corresponding modes.

Example 5.8

1. `ADD AX, 0100H` Immediate
2. `ADD AX, BX` Register
3. `ADD AX, [SI]` Register indirect
4. `ADD AX, [5000H]` Immediate
5. `ADD [5000H] 0100H` Immediate
6. `ADD 0100H` Destination AX (implicit)

ADC: Add with Carry This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example 5.9

1. `ADD 0100H` Immediate (AX implicit)
2. `ADD AX, BX` Register
3. `ADD AX, [SI]` Register indirect
4. `ADD AX, [5000H]` Direct
5. `ADD [5000H] 0100H` Immediate

INC: Increment This instruction increments the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction. The examples of this instruction are as follows:

Example 5.10

1.	INC	AX	Register
2.	INC	[BX]	Register indirect
3.	INC	[5000H]	Direct

DEC: Decrement The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction. The examples of this instruction are as follows:

Example 5.11

1.	DEC	AX	Register
2.	DEC	[500OH]	Direct

SUB: Subtract The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand can not be an immediate data. All the condition code flags are affected by this instruction. The examples of this instruction along with the addressing modes are as follows:

Example 5.12

1.	SUB	0100H	Immediate [destination AX]
2.	SUB	AX, BX	Register
3.	SUB	AX, [5000H]	Direct
4.	SUB	[5000H], 0100	Immediate

SBB: Subtract with Borrow The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (Condition code) by this instruction. The examples of this instruction are as follows:

Example 5.13

1.	SBB	O100H	Immediate [destination AX]
2.	SBB	AX.BX	Register
3.	SBB	AX, [5000H]	Direct

4. `SBB [5000H], 0100` Immediate

CMP: Compare This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

Example 5.14

1. `CMP BX, 0100H` Immediate
2. `CMP 0100` Immediate [AX implicit]
3. `CMP [5000H],OIOOH` Direct
4. `CMP BX, [SI]` Register indirect
5. `CMP BX, CX` Register

AAA: ASCII Adjust After Addition The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by I, the AF and CF flags are set to I, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Figure 5.2. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

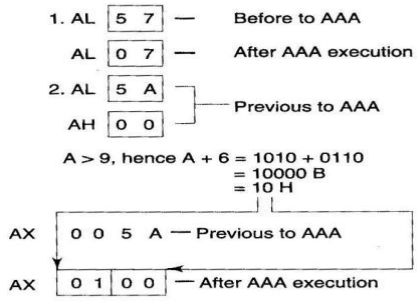


Figure 5.2: ASCII Adjust After Addition Instruction

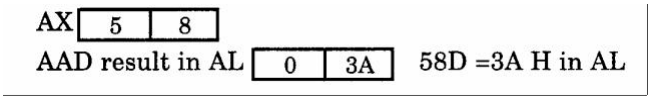
AAS: ASCII Adjust AL After Subtraction AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9.

The procedure is similar to the AAA instruction. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

AAM : ASCII Adjust for Multiplication This instruction, after execution , converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH. The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL = 5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add 6 (0110) to it D + 6 = 13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1 = 6 will be the upper unpacked byte of the result. Thus after the execution, AH = 06 and AL = 03.

AAD: ASCII Adjust for Division Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contain 0508 unpacked BCD for 58 decimal, and DH contain 02H.

Example 5.15



The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Note that 3A is the hexadecimal equivalent of 58 (decimal). Now, instruction DIV DH may be executed. So rather than ASCII adjust for division, it is ASCII adjust before division. All the ASCII adjust instructions are also called as unpacked BCD arithmetic instructions. Now, we will consider the two instructions related to packed BCD arithmetic.

DAA: Decimal Adjust Accumulator This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set , it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL. The examples given below explain the instruction.

Example 5.16

```
(i) AL = 53          CL = 29
    ADD AL, CL       ; AL ← (AL) + (CL)
                     ; AL ← 53 + 29
                     ; AL ← 7C
    DAA              ; AL ← 7C + 06 (as C>9)
                     ; AL ← 82

(ii) AL = 73         CL = 29
    ADD AL,CL        ; AL ← AL + CL
                     ; AL ← 73 + 29
                     ; AL ← 9C
    DAA              ; AL ← 02 and CF = 1
```

```
      + 6
    A 2
  + 6 0
CF = 1  0 2  in AL
```

```
AL = 7 3
     +
CL = 2 9
     9 C
     + 6
     A 2
   + 6 0
CF = 1  0 2  in AL
```

The instruction DAA affects AF, CF, PF, and ZF flags. The OF is undefined.

DAS: Decimal Adjust After Subtraction This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

Example 5.17

```
(i)  AL = 75          BH = 46
     SUB AL,BH        ; AL ← 2 F = (AL) – (BH)
                      ; AF = 1
     DAS              ; AL ← 2 9 (as F> 9, F – 6 = 9)
(ii) AL = 38          CH = 6 1
     SUB AL,CH        ; AL ← D 7   CF = 1 (borrow)
     DAS              ; AL ← 7 7   (as D> 9, D – 6 = 7)
                      ; CF = 1 (borrow)
```

NEG: Negate: The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand, which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

MLIL: Unsigned Multiplication Byte or Word: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The example instructions are as shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' CF and OF both will be set.

Example 5.18

```
1. MUL BH              ; (AX) ← (AL) x (BH)
2. MUL CX              ; (DX) (AX) ← (AX) × (CX)
3. MUL WORD PTR [SI]   ; (DX) (AX) ← (AX) × ([SI])
```

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result

respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. Sign bit and CF fill the unused higher bits of the result, AF are cleared. The example instructions are given as follows:

Example 5.19

```
1. IMUL          BH
2. IMUL          CX
3. IMUL          [SI]
```

CBW: Convert Signed Byte to Word: This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to Double Word: This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

DIV: Unsigned Division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

IDIV: Signed Division: This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by 0 interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation). All the flags are undefined after IDIV instruction.

Logical Instructions

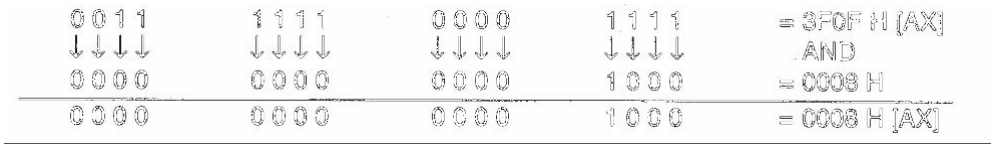
These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR. The instruction for each of these operations is discussed as follows.

AND: Logical AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand. The examples of this instruction are as follows:

Example 5.20

```
1. AND AX, 0008H
2. AND AX, BX
3. AND AX, [5000H]
4. AND [5000H], DX
```

If the content of AX is 3FOFH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.



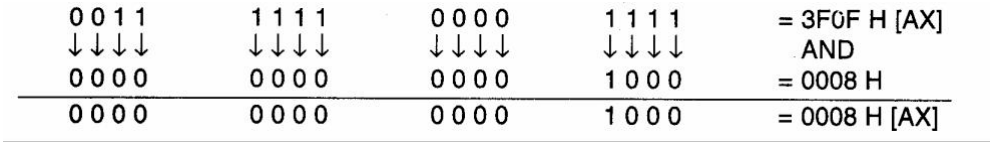
The result 0008H will be in AX.

OR: Logical OR: The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation. The examples are as follows:

Example 5.21

```
1. OR  AX,          0098H
2. OR  R AX,        BX
3. OR  R AX,        [5000H]
4. OR [5000H],      0008H
```

The contents of AX are say 3FOFH, then the first example instruction will be carried out as given below.



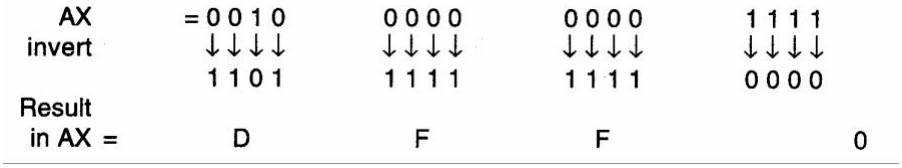
Thus the result 3F9FH will be stored in the AX register.

NOT: Logical Invert: The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example 5.22

```
NOT   AX
NOT   [5000H]
```

If the content of AX is 200FH, the first example instruction will be executed as shown.



The result DFFOH will be stored in the destination register AX.

XOR: Logical Exclusive OR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

The example instructions are as follows:

Example 5.23

```
1. XOR      AX,0098H
2. XOR      AX, BX
3. XOR      AX, [5000H]
```

If the content of AX is 3FOFH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

AX = 3F0FH =	0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1
XOR	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
0098H =	0 0 0 0	0 0 0 0	1 0 0 1	1 0 0 0
AX = Result =	0 0 1 1	1 1 1 1	1 0 0 1	0 1 1 1
= 3F97H				

TEST: Logical Compare Instruction: The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to I, if the corresponding bits of both operands are I, else the result bit is reset to 0. The result of this anding operation is not available for further use) but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

Example 5.24

```
1. TEST         AX,  BX
2. TEST         [0500],   06H
3. TEST         [BX]    [Dl], CX
```

SHL/SAL: Shift logical/Arithmetic Left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 5.3 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

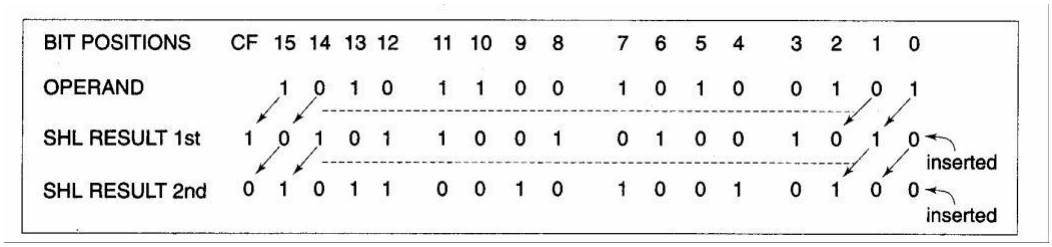


Figure 5.3f

SHR: Shift Logical Right: This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. Figure 5.4 explains execution of this instruction. This instruction shifts the operand through carry flag.

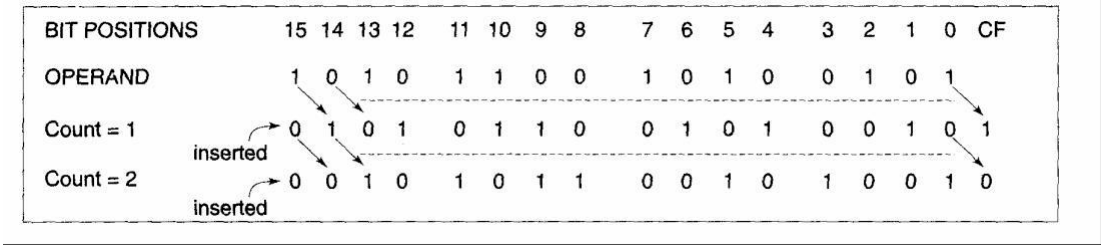
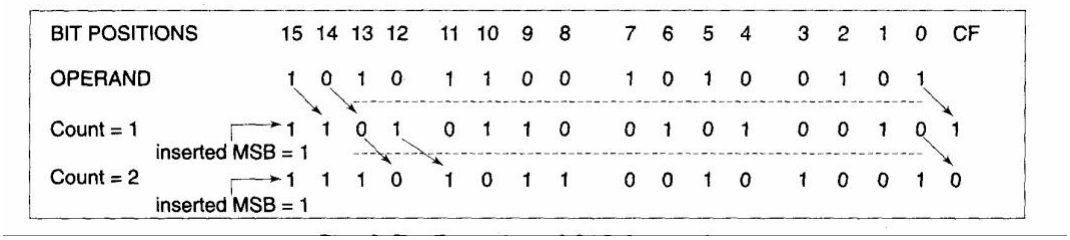


Figure 5.4

SAR: Shift Arithmetic Right: This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. Figure 5.5 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through carry flag.

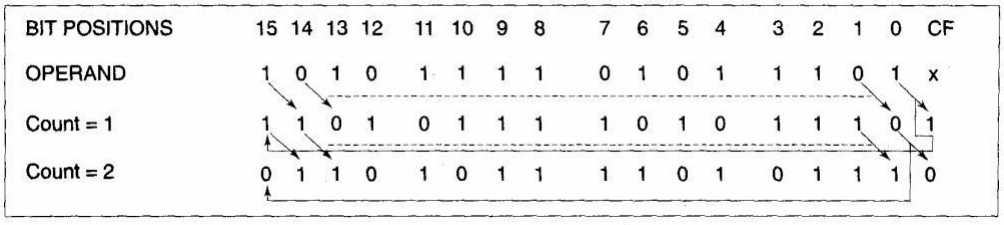


!

Figure 5.5

Immediate operand is not allowed in any of the shift instructions.

ROR: Rotate Right without Carry: This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. Figure 5.6 explains the operation. The destination operand may be a register (except a segment register) or a memory location.



" Figure 5.6

ROL: Rotate Left without Carry: This instruction rotates the content of the destination operand to the left by the specified count (bit- wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged by this rotate operation. The

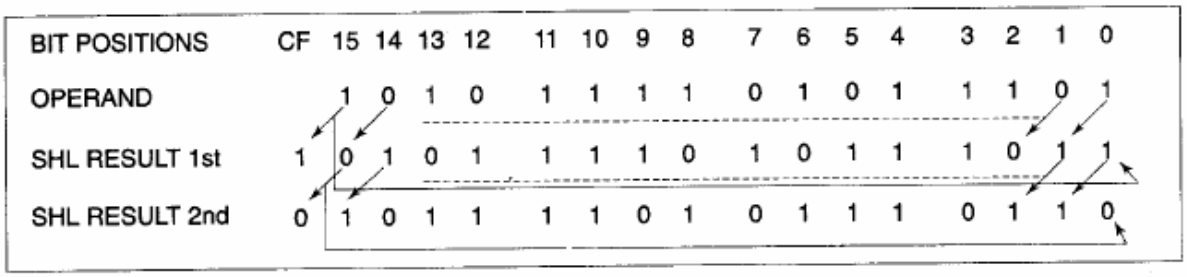


Figure 5.7: Execution of ROL Instruction

RCR: Rotate Right through Carry: This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 5.8 explains this operation.

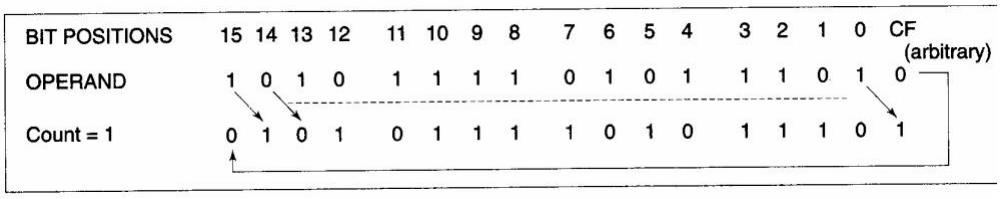


Figure 5.8

RCL: Rotate Left through Carry: This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB , and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left operation. The operand may be a register or a memory location. Figure 5. 9 explains the operation

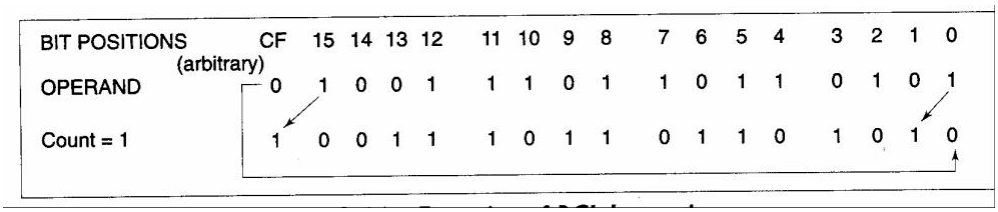


Figure 5.9

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required,

(a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements. The pointers and counters may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the direction flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation , the index registers are updated by two. The counter in both the cases, is decremented by one.

REP: Repeat Instruction Prefix: This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ, i.e. repeat operation while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word: Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (source index) and DS (data segment) contents. The starting address of the destination locations where this string has to be relocated is given by Dl (destination index) and ES (extra segment) contents. The starting address of the source string is 10H*DS+[SI], while the starting address of the destination string is 10H*ES+[DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/ words pointed to by DS: SI pair (source) to the memory location pointed to by ES: Dl pair (destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and Dl depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all the string manipulation instructions. The following string of instructions explain the execution of the MOVS instruction.

Example 5.25

```
MOV AX,5000H        ; Source segment address is 5000h.
MOV DS,AX           ; Load it to DS.

MOV AX,6000H        ; Destination segment address is 6000h.
MOV ES,AX           ; Load it to ES.
MOV CX,0FFH         ; Move length of the string to counter register CX.
MOV SI,1000H        ; Source index address 1000H is moved to SI.
MOV DI,2000H        ; Destination index address 2000H is moved to DI.
CLD                 ; Clear DF, i.e. set autoincrement mode.
REP MOVSB           ; Move 0FFH string bytes from source address to destination .
```

CMPS: Compare String Byte or String Word The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX(counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explain the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the

direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Example 5.26

```
MOV AX,SEG1              ; Segment address of STRING1, i.e. SEG1 is moved to AX.
MOV DS,AX                ; Load it to DS.
MOV AX,SEG2              ; Segment address of STRING2, i.e. SEG2 is moved to AX.
MOV ES,AX                ; Load it to ES.
MOV SI,OFFSET STRING1    ; Offset of STRING1 is moved to SI.
MOV DI,OFFSET STRING2    ; Offset of STRING2 is moved to DI.
MOV CX,010H              ; Length of the string is moved to CX.
CLD                      ; Clear DF, i.e. set autoincrement mode.
REPE CMPSW               ; Compare 010H words of STRING1 and
                         ; STRING2, while they are equal, If a mismatch is found,
                         ; modify the flags and proceed with further execution .
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

SCAS: Scan String Byte or String Word: This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found. The following string of instructions elaborates the use of SCAS instruction.

Example 5.27

```
        MOV AX,SEG       ; Segment address of the string, i.e. SEG is moved to AX.
        MOV ES,AX        ; Load it to ES.
        MOV DI,OFFSET    ; String offset, i.e. OFFSET is moved to DI.
        MOV CX,010H      ; Length of the string is moved to CX.
        MOV AX,WORD      ; The word to be scanned for, i.e. WORD is in AL.
        CLD              ; Clear DF.
REPNE   SCASW            ; Scan the 010H bytes of the string , till a match to
                         ; WORD is found.
```

This string of instructions finds out, if it contains WORD. If the WORD is found in the word string , before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the programme proceeds further.

LODS: Load String Byte or String Word: The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word: The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES : Dl register pair. The Dl is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index D1 are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and Dl are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending

addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, S1 and D1 are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses.

Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes, the CS may or may not be modified. These type of instructions are classified in two types:

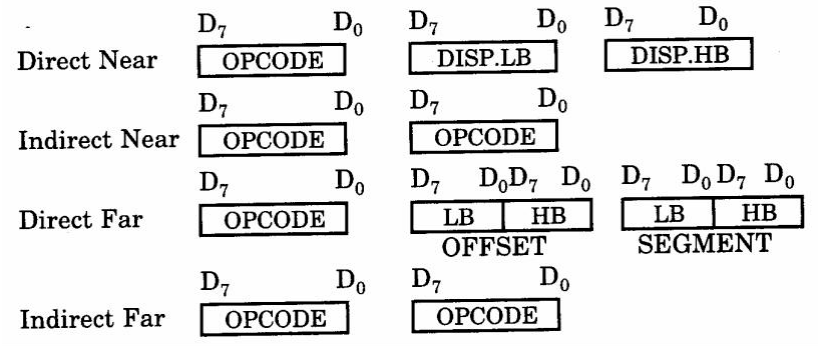
Unconditional Control Transfer (Branch) Instructions: In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

Conditional Control Transfer (Branch) Instructions: In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. Condition code flags replicate the results of the previous operations.

In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Instructions

CALL: Unconditional Call: This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e ± 32K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for them are respectively called as intrasegment and intersegment addressing modes. This instruction comes under unconditional branch instructions and can be described as shown with the coding formats. On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.



RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved

into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure . In case of a FAR procedure, the current

contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from OOH to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (Nx4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine.

For the execution of this instruction, the IF must be enabled.

Example 5.28

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT       20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

Figure 5.10 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

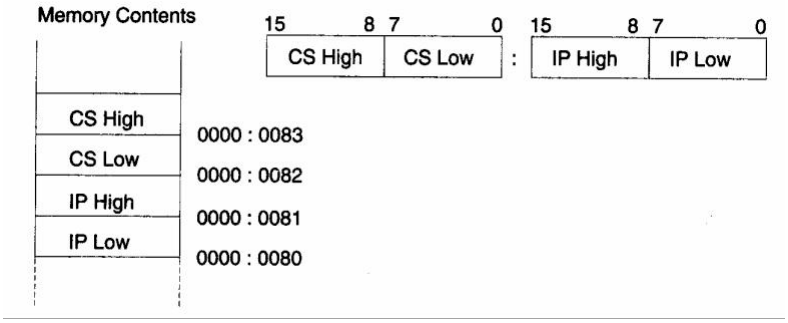
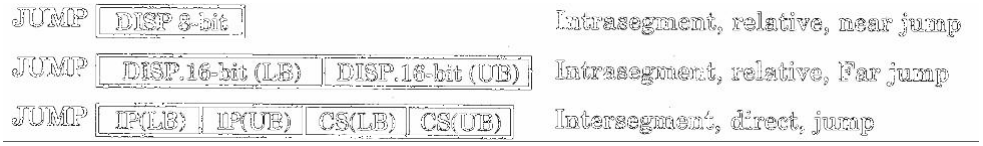


Figure 5.10

INTO: Interrupt on Overflow: This is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0000 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

JMP: Unconditional lump: This instruction unconditionally transfers the control of execution to the specified address using an 8 -bit or 16 -bit displacement (intrasegment relative, short or long) or CS : IP (intersegment direct far). No flags are affected by this instruction. Corresponding to the three methods of specifying jump addresses, the JUMP instruction has the following three formats.



IRET: Return from ISR: When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop Unconditionally: This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMF IF NOT ZERO structure.

Example 5.29

```
        MOV   CX, 0005     ; Number of times in CX
        MOV   BX,0FF7H     ; Data to BX
Label : MOV   AX, CODE1
        OR    BX, AX
        AND   DX, AX
        Loop  Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. Lf CX is already OOH, the execution continues sequentially. No flags are affected by this instruction.

Conditional Branch Instructions

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within -80H to 7FH (or -128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 5.1 next page

.

	Mnemonic	*Displacement*	*Operation*
1.	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1
2.	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0
3.	JS	Label	Transfer execution control to address 'Label' , if SF=1
4.	JNS	Label	Transfer execution control to address 'Label', if SF=0
5.	JO	Label	Transfer execution control to address 'Label', if OF=1
6.	JNO	Label	Transfer execution control to address 'Label', if OF=0
7.	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1
8.	JNP	Label	Transfer execution control to address 'Label', if PF=0
9.	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1
10.	JNB/JAE/JNC	Label	Transfer execution control to address 'Label', if CF=0
11.	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0.
15.	JLE/JNG	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or atleast any one of SF and OF is 1(Botn SF & OF are not 0).

The last four instructions are used in case of decisions based on signed binary number operations, while the remaining instructions can be used for unsigned binary operations. The terms above and below are generally used for unsigned numbers, while the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

```
JCXZ      'Label'     Transfer execution control
                      to address 'Label', if CX=0.
```

The conditional LOOP instructions are given in Table 5.2 with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

Table 5.2: Conditional Loop Instructions

Mnemonic	Displacement	Operation
LOOPZ/LOOPE	Label	Loop through a sequence of instructions from 'Label' while ZF=1 and CX f 0.
LOOPNZ/LOOPENE	Label	Loop through a sequence of instructions from 'Label' while ZF=0 and CX f 0.

The ideas about all these instructions will be more clear with programming practice. This topic is aimed at introducing these instructions to readers. Of course, examples are quoted wherever possible, but the JUMP and the LOOP instructions require a sequence of instructions for explanations and they will be emphasized.

Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

`CLC` - Clear carry flag

`CMC` - Complement carry flag

`STC` - Set carry flag

`CLD` - Clear direction flag

`STD` - Set direction flag

`CLI` - Clear interrupt flag

`STI` - Set interrupt flag

These instructions modify the carry(CF), direction(DF) and interrupt(IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions, are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

`WAIT` - Wait for Test input pin to go low
`HLT` - Halt the processor
`NOP` - No operation
`ESC` - Escape to external device like NDP (numeric co-processor)
`LOCK` - Bus lock instruction prefix.

After executing the HLT instruction, the processor enters the halt state, as explained in Chapter 1. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Student Activity 5.1

Before reading the next section, answer the following question.

1. Bring out the difference between the jump and loop instructions.
2. Which instruction of 8086 can be used for look up table manipulations?

3. What is the difference between the respective shift and rotate

instructions? If your answer is correct, then proceed to the next section.

Assembler Directives and Operators

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he may be able to manage the memory of the system more efficiently. On the other hand, the disadvantages are more prominent. The programming, coding and resource management techniques are tedious. The programmer has to take care of all these functions hence the chances of human errors are more. The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and instruction set.

The assembly language programming is simpler as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement in assembly language over machine language is that the address values and the constants can be identified by labels. If the labels are suggestive, then certainly the program will become more understandable, and each time the programmer will not have to remember the different constants and the addresses at which they are stored, throughout the programs. The labels may help to identify the addresses and constants. Due to this facility, the tedious byte handling and manipulations are got rid of. Similarly, now different logical segments and routines may be assigned with the labels rather than the different addresses. The memory control feature of machine language programming is left unchanged by providing storage define facilities in assembly language programming. The documentation facility which was not possible with machine language programming is now available in assembly language.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks) an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or a variable, logical names of the segments, types of the different routines and modules, end of file, etc. These types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler.

DB: Define Byte The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initialises the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

Example 5.30

```
RANKS  DB OIH, 02H, 03H, 04H
```

This statement directs the assembler to reserve four memory locations for a list named RANKS and initialise them with the above specified four values.

```
MESSAGE DB 'GOOD MORNING
```

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialise those locations by the ASCII equivalent of these characters.

```
VALUE  DB  5OH
```

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialised for the variable named VALUE.

DW: Define Word: The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Example 5.31

```
WORDS  DW   1234H, 4567H, 78ABH, 045CH,
```

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

```
WDATA  DW 5 DUP (6666H)
```

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initialises all the word locations with 6666H.

DQ: Define Quadword: This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

DT: Define Ten Bytes: The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

ASSUME: Assume Logical Segment Name: The ASSUME directive is used to inform the assembler, the names of the logicals segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS : DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialised by the segment address value decided by the operating system for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program, without which a message 'CODE/DATA EMITTED WITHOUT SEGMENT' may be issued by an assembler.

END: END of Program: The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

ENDP: END of Procedure In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a

name, i.e. label. To mark the end of a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitues the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, and the label can then be used in the program in place of that mnemonic. Suppose, a numerical constant appears in a program ten times. If that constant is to be changed at a later time, one will have to make all these ten corrections. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the EQU statement will give the correct and modified code. The examples given below show the syntax.

Example 5.32

```
LABEL      EQU    050 OH
ADDITION   EQU    ADD
```

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD. EXTRN: External and PUBLIC: Public The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULEI from MODULE 2; in MODULE 1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL . The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MODULE 2. Thus the MODULEI and MODULE 2 must have the following declarations.

```
MODULE1        SEGMENT
PUBLIC         FACTORIAL FAR
MODULE1        ENDS
MODULE2        SEGMENT
EXTRN          FACTORIAL FAR
MODULE2        ENDS
```

GROUP: Group the Related Segments: The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes an information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base.

```
PROGRAM GROUP CODE, DATA, STACK
```

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

```
ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.
```

LABEL: Label: The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initialises a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it

assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

```
CONTINUE    LABEL FAR
```

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA          SEGMENT

DATAS DB 5 OH DUP (?)

DATA-LAST LABEL BYTE FAR

DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATALAST and its type will be byte and far.

LENGTH: Byte Length of a Label: This directive is not available in MASM. This is used to refer to the length of a data array or a string.

```
MOV CX. LENGTH ARRAY
```

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

LOCAL The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that module. At a later time, some other module may declare a particular data type LOCAL, which is previously declared LOCAL by another module or modules.

Thus the same label may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local, as shown.

```
LOCAL a, b, DATA, ARRAY, ROUTINE
```

NAME: Logical Name of a Module: The NAME directive is used to assign a name to an assembly language program module. The module, may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

OFFSET: Offset of a Label: When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The segment may also be decided by another operator of similar type, viz, SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

Example 5.33

```
CODE SEGMENT

MOV SI, OFFSET LIST

CODE ENDS

DATA SEGMENT
```

```
LIST DB IOH
DATA ENDS
```

ORG : Origin: The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement. While starting the assembly process for a module, the assembler initialises a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialised to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment. In other words, the location counter will get initialised to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

PROC: Procedure: The PROG directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e. whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

Example 5.34

```
RESULT   PROC     NEAR
ROUTINE PROC     FAR
```

PTR: Pointer: The pointer operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16-bit quantity. In other words, the PTR operator is used to specify the data type - byte or word. The examples of the PTR operator are as follows:

Example 5.35

`MOV AL, BYTE PTR [SI] -`	Moves content of memory location addressed by SI (8-bit) to AL
`INC BYTE PTR [BX]-`	Increments byte contents of memory location addressed by BX
`MOV BX, WORD PTR [2000H]-`	Moves 16-bit content of memory location 2000H to BX, i.e [2000H] to BL [2001H] to BH
`INC WORD PTR [3000H] -`	Increments word contents of memory location 3000H considering contents of 3000H (lower byte) and 3001H (higher byte) as a 16-bit number

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

`JMP WORD PTR [BX]`-NEAR Jump

`JMP WORD PTR [BX]`-FAR Jump

PUBLIC As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be

accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least any one of the other modules of the same program. (Refer to the explanation on EXTRN directive to get the clear idea of PUBLIC.)

SEG: Segment of a Label The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of 'SEG label". The example given below explain the use of SEG operator.

Example 5.36

```
MOV AX, SEG ARRAY     ; This statement moves the segment address of ARRAY in
MOV DS, AX            ; which it is appearing, to register AX and then to DS.
```

SEGMENT: Logical Segment: The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

```
EXE.CODE SEGMENT GLOBAL     ; Start of segment named EXE. CODE,
                            ; that can be accessed by any other module.
EXE.CODE ENDS               ; END of EXE.CODE logical segment.
```

SHORT The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory . Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

```
JMP SHORT LABEL
```

TYPE The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

GLOBAL: The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC    GLOBAL
```

Student Activity 5.2

Before reading the next section, answer the following question.

1. How will you enter the single step mode of 8086?
2. What is LOCK prefix? What is its use?
3. What is REP prefix? What is its use?

If your answer is correct, then proceed to the next section.

A Few Machine Level Programs

A Few Machine Level Programs

In this section, a few machine level programming examples, rather, instruction sequences are presented for comparing the 8086 programming with that of 8085. These programs are ii the form of instruction sequences just like 8085 programs. These may even be hand-coded entered byte by byte and executed on an 8086 based system but due to the complex instruction set of 8086 and its tedious opcode conversion procedure, most of the programmers prefer to use assemblers. However, we will briefly discuss the hand-coding,

Example 5.37

Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

Solution

The flow chart for this problem may be drawn as shown in Figure 5.11

```
MOV   AX, 2000H   ; Initialising DS with value
MOV   DS, AX      ; 2000H.
MOV   AX, [500H]  ; Get first data byte from 0500H offset.
ADD   AX, [600H]  ; Add this to the second byte from 0600H.
MOV   [700H],AX   ; Store AX in 0700H (result).
HLT               ; Stop.
```

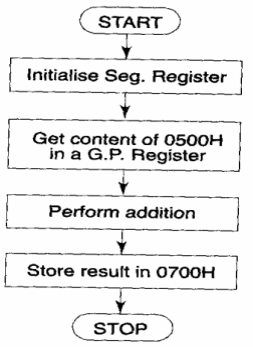


Figure 5.12 Flow chart

The above instruction sequence is quite straight-for -ward. As the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose resistors, say AX. and then the register general purpose registers, say AX, and then the register content is moved to the segment register DS. Thus the data segment register DS contains 2000H. The instruction MOV AX,[500H] signifies that the contents of the particular location, whose offset is specified in the brackets with the segment pointed to by DS as segment register, is to be moved to AX. The MOV [0700], AX instruction moves the contents of the register AX to an offset 0700H in DS (DS = 2000H). Note that the code segment register CS gets automatically loaded by the code segment address of the program whenever it is executed. Actually it is the monitor program that accepts

the CS:IP address of the program and passes it to the corresponding registers at the time of execution. Hence no instructions are required for loading the CS register like DS or SS.

Example 5.38

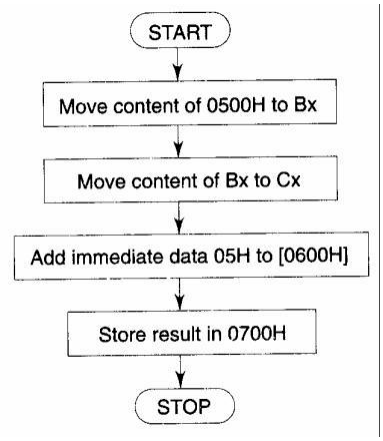
Write a program to move the contents of the memory location 0500H to register BX and also to CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using DS=2000H and offset=0600H. Store the result of the addition in 0700H. Assume that the data is located in the segment specified by the data segment register which contain 2000H.

Solution

The flow chart for the program is shown in Figure 5.12.

After initializing the data segment register the content of location 0500H are moved to the BX register using MOV instruction. The same data is moved also to the CX register. For this data transfer, there may be two options as shown.

```
MOV AX, 2000H
MOV DS, AX          ; Initialize data segment register.
MOV BX,[0500H]      ; Get contents of 0500H in BX.
MOV CX, BX          ; Copy the same contents in CX.
ADD [0600H],05H     ; Add byte 05H to contents of 0600H.
MOV DX, [0600H]     ; Store the result in DX.
MOV [0700H],DX      ; Store the result in 0700H.
HLT                 ; Stop.
```



, 0

(a) `MOV CX, BX` ; As the contents of BX will be same as 0500H after execution
; of MOV BX,[0500H].

(b) `MOV CX, [050OH]` ; Move directly from 0500H to register CX

The opcode in the first option is only of 2 bytes, while the second option will have 4 bytes of opcode. Thus the second option will require more memory and execution time. Due to these reasons, the first option is preferable.

The immediate data byte 05H is added to content of 0600H using the ADD instruction. The result will be in the destination operand 0600H. This is next stored at the location 0700H. In case of the 8086/8088 instruction set, there is no instruction for the direct transfer of data from the memory source operand to the memory destination operand except, the string instructions. Hence the result of addition which is present at 0600H, should be moved to any one of the general purpose registers, except BX and CX, otherwise the contents of CX and BX will be changed. We have selected DX (we could have selected AX also, because

once DS is initialised to 2000H the contents of AX are no longer useful for this purpose. Thus the transfer of result from 0600H to 0700H is accomplished in two stages using successive MOV instructions, i.e., at

first, the content of 0600H is DX and then the content of DX is moved to 0700H. The program ends with the HLT instruction.

Example 5.39

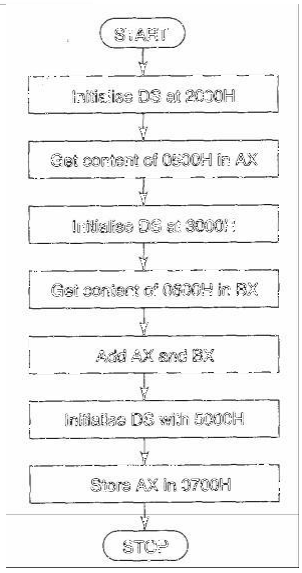
Add the contents of the memory location 2000H:0500H to contents of 3000H:0600H and store the result in 5000H:0700H.

Solution

Unlike the previous example programs, this program refers to the memory locations in different segments, hence, while referring to each location, the data segment will have to be newly initialized with the required value. Figure 5.13 shows the flow chart.

The instruction sequence for the above flow chart is given along with the comments.

```
MOV CX, 2000H      ; Initialize DS at 2000H.
MOV DS, CX
MOV AX, [500H]     ; Get first operand in AX.
MOV CX, 3000H      ; Initialize DS at 3000H.
MOV DS, CX
MOV BX, [0600H]    ; Get second operand in BX.
ADD AX, BX         ; Perform addition.
MOV CX, 5000H      ; Initialize DS at 5000H.
MOV DS, CX
MOV [0700H],AX     ; Store the result of addition in
HLT                ; 0700H and stop.
```



, 0 .*

Actually, the program simply performs the addition of two operands which are located in different memory segments. The program has become lengthy only due to data segment register initialization instructions.

Example 5.40

Move a byte string , 16-bytes long, from the offset 0200H to 0300H in the segment 7000H.

Solution

According to the program statement, a string that is 16-bytes long is available at the offset address 0200H in the segment 7000H. The required program should move this complete string at offset 0300H, in the same segment. Let us emphasize this program in the light of comparison between 8085 and 8086 programming techniques.

An 8085 program to perform this task, is given neglecting the segment addresses.

```
        MVI C, 010H     ; Count for the length of string
        LXIH 0200H      ; Initialization of HL pair for source string
        LXID 0300H      ; Initialization of DE pair for destination
BACK :  MOV A, M        ; Take a byte from source in A.
        STAX D          ; Store contents of A to address pointed to by DE pair.
        INX H           ; Increment source pointer.
        INX D           ; Increment destination pointer.
        DCRC            ; Decrement counter.
        JNZ BACK        ; Continue if counter is not zero.
        HLT             ; Stop if counter is zero.
```

The programmers, with fluent hands on 8085 assembly language programming but starting with 8086, may translate the above 8085 assembly language program listings to 8086 assembly language programs using the analogous or comparable instructions. Of course, this method of programming is not efficient, however, it may help those who are familiar to 8085 programming and wish to start writing programs in 8086 assembly language. The reason for the inefficiency of this method is that the special features and capabilities of 8086 have not been taken into account while preparing the 8086 assembly language program. Now, let us think about how the above program may be transferred to 8086 assembly language using analogous instructions. Note that the segment initialization is to be added. Let us consider that the code and data segment address is 7000H. Consider that the code starts at offset 0000H.

```
        MOV AX, 7000H
        MOV DS, AX          ; Data segment initialization
        MOV SI, 0200H       ; Pointer to source string
        MOV DI, 0300H       ; Pointer to destination string
        MOV CX, 0010H       ; Count for length of string
BACK :  MOV AX, [SI]        ; Take a source byte in AX.
        MOV [DI],AX         ; Move it to destination.
        INC SI              ; Increment source pointer.
        INC DI              ; Increment destination pointer.
        DEC CX              ; Decrement count by 1.
        JNZ BACK            ; Continue if count is not 0.
        HLT                 ; Stop if the count is 0.
```

The above listing has been prepared using the program written in 8085 ALP. Indexed addressing mode is used for string byte accesses and transfer in this case. The functions of all the 8086 instructions and the 8086 addressing modes have already been explained in Unit 2. In this program, all the instructions used are more or less analogous to the 8085 program, and the special software capabilities of 8086 like string instructions and loop instructions have not been considered. The 8086 programs based on 8085 codes are inefficient due to the reason that the full capability of the rich 8086 instruction set and the enhanced architecture of 8086 cannot be fully exploited.

The above program uses the decrement and jump-if-not-zero instructions for checking whether the transfer is complete or not. The 8086 instruction set provides LOOP instructions for this purpose. Using these instructions, the program is modified as shown.

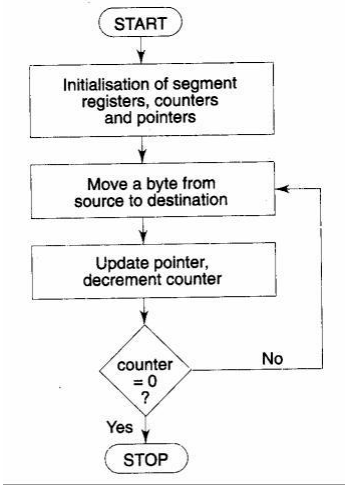
```
        MOV AX,7000H
        MOV DS,AX          ; Data segment initialization
        MOV SI,0200H       ; Source pointer initialization
        MOV DI,0300H       ; Destination pointer initialization
        MOV CX,0010H       ; Counter initialization
BACK :  MOV AX,[SI]        ; Take a byte of string from source
        MOV [DI],AX        ; and then move it to destination
        INC SI             ; Update source pointer
        INC DI             ; Update destination pointer continue
        LOOP BACK          ; till CX=0,[DEC CX and JNZ BACK]
        HLT                ; Stop if CX=0
```

Thus the two instructions bracketed in the comment field are replaced by a single loop instruction which results in the saving of memory and execution time. The loop instruction needs the additional instructions for updating the pointers (for example, INC SI, INC Dl). It does not need counter decrement and check-if-zero instruction.

One more feature of the 8086 instruction set is the string instruction, i.e. MOVSB and MOVSW. Using these instructions one can move a string byte/word from source to destination. The length of the string is specified by the CX register. The SI and Dl point to the source and destination locations. The DS and ES registers should be initialised to source and destination segment addresses respectively. Before the use of string instructions, the program should initialise all these registers properly. Using the string byte instruction the same program may be written as shown.

```
        MOV AX, 7000H
        MOV DS, AX         ; Source segment initialisation
        MOV ES,AX          ; Destination segment initialisation
        MOV CX,0010H       ; Counter initialisation
        MOV SI,0200H       ; Source pointer initialisation
        MOV DI,0300H       ; Destination pointer initialisation
        CLD                ; Clear DF
REP     MOVSB              ; Move the complete string
        HLT                ; Stop
```

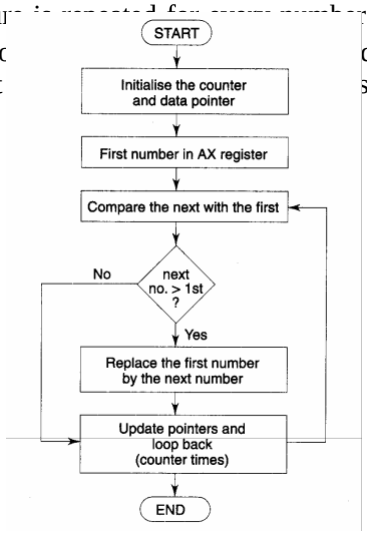
The MOVSB instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. An experienced programmer will thus directly use the string instructions instead of using other options. The flow chart of the final program is presented in Figure 5.14.



, 0 .*

Example 5.41

Find out the largest number from an unordered array of sixteen 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H. Solution The logic for this procedure can be described as follows. The first number of the array is taken in a register, say AX. The second number of the array is then compared with the first one. If the first one is greater than the second one, it is left unchanged. However, if the second one is greater than the first, the second number replaces the first one in the AX register. The procedure is repeated for every number in the array and thus it requires 15 iterations. At the end of 15th iteration the largest number will be in the register AX. This may be represented in terms of the flow chart. A program listing is given below.



Student Activity 5.3

Before reading the next section, answer the following question.

1. Write a program to move the contents of the memory location 0700H to register BX and also to CX. Add immediate byte 10H to the data residing in memory location, whose address is computed using DS=3000H and offset=0200H. Store the result of the addition in 0500H. Assume that the data is located in the segment specified by the data segment register which contain 3000H.
2. Move a byte string, 16-bytes long, from the offset 0500H to 0010H in the segment 75000H.

If your answer is correct, then proceed to the next section.

Machine Coding the Programs

So far we have discussed five programs which were written for handcoding by a programmer. In this section, we will now have a brief look at how these programs can be translate to machine codes. In Appendix, the instruction set along with the Appendix are presented. This Appendix is self-explanatory to handcode most of the instructions. The S, V, W, D, MOD, REG and R/M fields are suitably decided

depending upon the data types, addressing mode and the registers used. The table 3.2 shows the details about how to select these fields.

Most of the instructions either have specific opcodes or they can be decided only by setting the S, V, W, D, REG, MOD and R/M fields suitably but the critical point is the calculation of jump addresses for intrasegment branch instructions. Before starting the coding of jump or call instructions, we will see some easier coding examples.

Example 5.42

MOV BL, CL

For handcoding this instruction, we will have to first note down the following features.

(i) It fits in the register/memory to/from register format.

(ii) It is an 8-bit operation.

(iii) BL is the destination register and CL is the source register.

Now from the feature (i) using the Appendix, the opcode format is given as shown.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7 D_6 D_5	D_4 D_3	D_2 D_1 D_0
1	0	0	0	1	0	d	w	(MOD)	(REG)	(R/M)

If d =1, then transfer of data is to the register shown by the REG field, i.e. the destination is a register (REG). If d = 0, the source is a register shown by the REG field.

It is an 8-bit operation, hence w bit is 0. If it had been a 16-bit operation, the w bit would have been 1.

Refer to Table 2.2 to search the REG to REG addressing in it, i.e. the last column with MOD 11. According to the Appendix when MOD is 11, the R/M field is treated as a REG field. The REG field is used for source register and the R/M field is used for the destination register, if d is 0. If d =1, the REG field is used for destination and the R/M field is used to indicate source. Now the complete machine code of this instruction comes out to be

	code	dw	MOD	REG	R/M
MOVBL,CL	10001000	1	1	001	011=88CB

Note that the register codes are to be found out from the Table 3.1.

Examples 5.43

MOVBX.5000H

From Appendix, the coding format is as shown.

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	D_7 ... D_0	D_7 ... D_0
1 0 1 1 W REG	DATA LOW BYTE	DATA HIGH BYTE

Following the procedure as in Example 1, the code comes out to be (BB 00 50) as shown.

W (R E G)	Data L B	Data H B
1011 1011	0000 0000	01010000
B B	0 0	5 0

Example 5.44

MOV [SI], DL

This instruction belongs to the register to memory format. Hence from the Appendix, and using the already explained procedure , the machine code can be found as shown.

OPCODE	D	W	MOD	REG	R/M
1000	100	1	00	010	100
8	9		1		4

The machine code is 89 14.

Example 5.45

MOV BP[SI], 0005H

OPCODE	W	MOD	R/M
1100 0111		00	000 010
C	7	0	2
0000	0101	0000	0000
0	5	0	0

The machine code of this instruction is C7 02 05 00.

Example 5.46

MOV BP [SI+ 500H], 7293H

OPCODE	W	MOD	R/M	
1100	0111	10	000010	
C	7	8	2	
0000	0000	0000	0101	
0	0	0	5	Displacement 500H
1001	0011	0111	0010	
9	3	7	2	Data 7293 H

The complete machine code comes out to be C7 82 00 05 93 72.

Example 5.47

ADD AX, BX

The machine code is formed as shown by referring to Appendix and using the Tables 3.1 and 3.2 as has been already described.

OPCODE	D W	MOD	REG	R/M
0000	0011	11	000	011

The machine code is 03 C3.

Example 5.48

ADD AX, 5000H

The code formation is explained as follows:

OPCODE	S W	MOD	R/M
1000	0001	00	000000
8	1	0	0
0000	0000	0101	0000
0	0	5	0

The machine code is 81 00 00 50.

If s bit is 0, the 16-bit immediate data is available in the instruction.

Ifs bit is 1, the 16-bit immediate data is the sign extended form of 8-bit immediate data.

For example, if the eight bit data is 11010001, then its sign extended 16-bit version will be 1111111111010001.

Example 5.49

SHRAX

OPCODE	VW	MOD	REG	R/M
1101	0001	11	101	000
D	1		E	8

The instruction code is D1 E8.

Finding out Machine Code for Conditional JUMP (Intrasegment) Instructions: The Appendix shows that, corresponding to each of the conditional jump instructions, the first byte of the opcode is fixed and the jump displacement must be less than or equal to 127(D) bytes and greater than or equal to -128(D). This type of jump is called as short jump. The following conditional forward jump example explains how to find the displacement. The displacement is an 8-bit signed number. If the displacement is positive, it indicates a forward jump, otherwise it indicates a backward jump. The following example is a sequence of instructions rather than a single instruction to elaborate the procedure of the calculation of positive displacement for a forward jump.

Example 5.50

```
2000 ,01          XOR AX, BX
2002 ,03          JNZ OK
2004              NOP
2005              NOP
2006 ,7,8,9       ADD BX, 05H
200A  OK : HLT
```

The above sequence shows that the programmer wants a conditional jump to label OK, if the zero flag is not set. For finding out the displacement corresponding to the label OK, subtract the address of the jump instruction (2002H), from the address of label (200AH). The required displacement is 200AH - 2002H = 08H. The 08H is the displacement for the forward jump.

Let us find out the displacement for a backward jump. Consider the following sequence of

instructions. Example 5.51

```
2000, 01, 02      MOV    CL, 05H
2003      Repeat : INC    AX
2004              DEC    CL
2005,2006         JNZ    Repeat
```

For finding out the backward displacement, subtract the address of the label (Repeat) from the address of the jump instruction. Complement the subtraction. The lower byte gives the displacement. In this example, the signed displacement for the JNZ instruction comes out to be (2005H-2003H=02, complement-FDH) The magnitude of the displacement must be less than or equal to 127(D). The MSB of the displacement decides whether it is a forward or backward jump. If it is I, it is a backward jump or else it is a forward jump.

A similar procedure is used to find the displacement for intrasegment short calls.

Finding out Machine Code for Unconditional JUMP Intrasegment: For this instruction there are again two types of jump, i.e. short jump and long jump. The displacement calculation procedures are again the same as given in case of the conditional jump. The only new thing here is that, the displacement may be beyond ±127(D). This type of jump is called as long jump. The method of calculation of the displacement is again similar to that for short jump.

Finding out Machine Code for Intersegment Direct Jump: This type of instruction is used to make a jump directly to the address lying in another segment. The opcode itself specifies the new offset and the segment of jump address, directly.

Example 5.52

JUMP 2000:5000

This instruction implies a jump to a memory location in another code segment with CS = 2000H and Offset =5000H. The code formation is as shown.

Code formation	1110 1010	0000 0000	0101 0000	0000 0000	0010 0000
	Opcode	Offset LB	Offset HB	Seg. LB	Seg. HB

The opcode forms the first byte of this instruction and the successive bytes are formed from the segment and the offset of the jump destination. While specifying the segment and offset, the lower byte (LB) is specified first and then the higher byte (HB) is specified. Finally, the opcode comes out to be EA 00 50 00 20. The procedure of coding the CALL instructions is similar.

Hand Coding a Complete Program After studying the hand-coding procedures of different instructions, let us now tries to code a complete program. We will consider Example 5.53 for complete hand-coding. The program and the code corresponding to it is given. These codes, found using hand-coding, may be entered byte-by-byte into an 8086 based system and executed, provided it supports the memory requirements of the program.

Example 5.53

A Hand-coding Program Example

Addresses	Opcodes	Labels	Mnemonics
2000,01,02	B9 0F 00		MOV CX,0F H
2003,04,05	B8 00 02		MOV AX,2000H
2006,07	8E D8		MOV DS,AX
2008,09,0A	BE 00 05		MOV SI,0500H
200B,0C	89 04		MOV AX,[SI]
200D	46	BACK :	INC SI
200E,0F	3B 04		CMP AX,[SI]
2010,11	77 04		JNC NEXT
2012,13	89 04		MOV AX,[SI]
2014,15	E2 F7	NEXT :	LOOP BACK
2016	F4		HLT

Student Activity 5.4

Before reading the next section, answer the following question.

1. Find out the machine code for following instructions.

(i)	ADC AX, BX	(ii)	OR AX, [0500H]	(iii)	AND CX, [SI]
(iv)	TEST AX,5555H	(v)	MUL [SI+5	(vi)	NEG 50[BP]
(vii)	OUT DX, AX	(viii)	LES DI, [0700H]	(ix)	LEA SI, [BX+500H]
(x)	SHL [BX+2000],CL	(xi)	RET 0200H	(xii)	CALL 7000H
(xiii)	JMP 3000h:2000H	(xiv)	CALL [5000H]	(xv)	DIV [5000H]

2. Describe the procedure for coding the intersegment and intrasegment over machine language.
3. Enlist the advantages of assembly language programming over machine language.

If your answer is correct, then proceed to the next section.

Programming with an Assembler

The procedure of hand -coding 8086 programs is somewhat tedious, hence in general a programmer may find it difficult to get a correct listing of the machine codes. Moreover, the procedure of handcoding is time consuming. This programming procedure is called as machine level programming. The obvious disadvantages of machine level programming are as given:

1. The process is complicated and time consuming.
2. The chances of error being committed are more at the machine level in hand-coding and entering the program byte-by-byte into the system.
3. Debugging a program at the machine level is more difficult.
4. The programs are not understood by everyone and the results are not stored in a user-friendly form.

A program called 'Assembler' is used to convert the mnemonics of instructions along with the data into their equivalent object code modules. These object code modules may further be converted in executable code using the linker and loader programs. This type of programming is called assembly level programming. In assembly language programming, the mnemonics are directly used in the user programs. The assembler performs the task of coding.

The advantages of assembly language over machine language are as given:

1. The programming in assembly language is not so complicated as in machine language because the function of coding is performed by an assembler.
2. The chances of error being committed are less because the mnemonics are used instead of numerical opcodes. It is easier to enter an assembly language program.
3. As the mnemonics are purpose-suggestive the debugging is easier.
4. The constants and address locations can be labeled with suggestive labels hence imparting a more friendly interface to user. Advanced assemblers provide facilities like macros, lists, etc. making the task of programming much easier.

5. The memory control is in the hands of users as in machine language.
6. The results may be stored in a more user-friendly form.
7. The flexibility of programming is more in assembly language programming as compared to machine language because of advanced facilities available with the modern assemblers.

Basically, the assembler is a program that converts an assembly input file also called as source file to an object file that can further be converted into machine codes or an executable file using a linker. The recent versions of the assembler are designed with many facilities like macroassemblers, numerical processor assemblers, procedures, functions and so on.

As far as this book is concerned, we will consider the assembly language programming using MASM (Microsoft Macro Assembler). There are a number of assemblers available like MASM, TASM and DOS assembler . MASM is one of the popular assemblers used along with a LINK program to structure the codes generated by MASM in the form of an executable file. MASM reads the source program as its input and provides an object file. The LINK accepts the object file produced by MASM as input and produces an EXE file.

While writing a program, for an assembler, your first step will be to use a text editor and type the program listing prepared by you. Then check the listing typed by you for any typing mistake and syntax error. Before you quit the editor program, do not forget to save it. Once you save the text file with any name (permissible on operating system), you are free to start the assembly process. A number of text editors are available in the market, e.g. Norton's editor [NE], Turbo C [TC], EDLIN, etc. Throughout this book, the NE is used. Any other free form editor may be used for a better user-friendly environment. Thus for writing a program in assembly language, one will need NE editor, MASM assembler, linker and DEBUG utility of DOS. In the following section, the procedures of opening a file for a program, assembling it, executing it and checking its result are described for beginners.

Entering a Program

In this section, we will explain the procedure for entering a small program on IBM PC with DOS operating system. Consider a program of addition of two bytes, as already discussed for handcoding. The same program is written along with some syntax modifications in it for MASM.

Before starting the process, ensure that all the files namely NE.COM (Norton's Editor), MASM.EXE (Assembler), LINK.EXE (linker), DEBUG.EXE (debugger) are available in the same directory in which you are working. Start the procedure with the following command after you boot the terminal and enter the directory containing all the files mentioned.

```
C> NE
```

You will get display on the screen as in Figure 5.16.

Now type the file name. It will be displayed on the screen. Suppose one types KMB.ASM as file name, the screen display will be as shown in Figure 5.17.

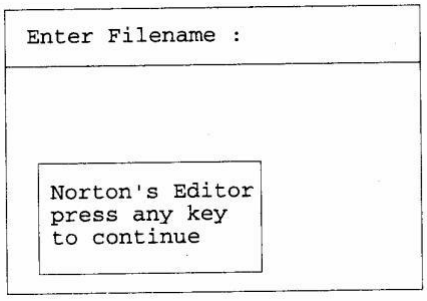


Figure 5.16: Norton's Editor Opening Screen

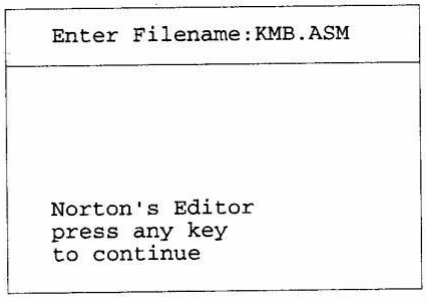


Figure 5.17: Norton's Editor Alternative Opening Screen

Press any of the keys, you will get Figure 5.18 as the screen display.

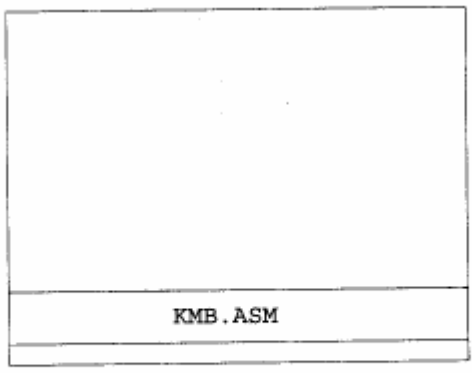


Figure 5.18: Norton's Editor Opens a New KMB.ASM

Note that, for every assembly language program, the extension .ASM must be there Though every time the assembler dose not use the complete name KMB.ASM and uses just KMB to handle the file, the extension shows that it is an assembly language program file. Even if you enter a file name without the .ASM extension, the assembler searches for the file name but with .ASM extension and if it is not available, it issues the message 'File not found'. Once you get the display as in Figure 5.18 you are free to type the program. One may use another type of command line as shown.

```
C> NE KMB.ASM
```

The above command will directly give the display as in Figure 5.18, if KMB.ASM is a newly opened file. Otherwise, if KMB.ASM is already existing in the directory, then it will be opened and the program in it will be displayed. You may modify it and save (command F3-E) it again, if you want the changes to be permanent. Otherwise, simply quit (command F3-Q) it to abandon the changes and exit NE . The entered program in NE looks like in Figure 5.19. We have to just consider that it is an assembly language program to be assembled. Store it with command F3-E. This will generate a new copy of the program in the secondary storage. Then quit the NE with command F3-Q.

Once the above procedure is completed, you may now turn towards the assembling of the above program. Note that all the commands and displays shown in this section are for Norton's Editor. Other editors may require some other commands and their display style may be some what different but the overall procedure is the same.

Note that before quitting the editor program the newly entered or modified file must be saved, otherwise it will be lost and will not be available for further assembling process.

```
ASSUME      CS:CODE, DS:DATA
DATA        SEGMENT
            OPR1 DW 1234 H
            OPR2 DW 0002 H
            RESULT DW 01 H DUP(?)

DATA        ENDS
CODE        SEGMENT
START       MOV AX, DATA
            MOV DS, AX
            MOV AX, OPR 1
            MOV BX, OPR 2
            CLC
            ADD AX, BX
            MOV DI, OFFSET RESULT
            MOV (DI), AX
            MOV AH, 4CH
            INT 21 H
CODE        ENDS
            END START
```

KBM.ASM

Figure 5.19

Assembling a Program

Microsoft Assembler MASM is one of the easy to use and popular assemblers. All the references and discussions in this section are related to the MASM. As already discussed, the main task of any assembler program is to accept the text assembly language program file as input and prepare an object file. The text assembly language program file is prepared by using any of the editor programs. The MASM accepts the file names only with the extension .ASM. Even if a filename without any extension is given as input, it provides .ASM extension to it. For example, to assemble the program in Figure 5.19, one may enter the following command options-

```
A> MASM KMB
```

or

```
C ——>A> MASM KMB. ASM
```

If any of the command option is entered as above, the programmer gets the display on the screen, as shown in Figure 5.20.

```
C←A>MASM KMB

Microsoft @ Macro Assembler Version 5.10
Copyright (c) Microsoft Corp.1981, 1989.
All Rights Reserved



Object filename[.OBJ]:
List filename[NUL.LST]:
Cross Reference[NUL.CRF]:
```

Figure 5.20

Also another command option, available in MASM that does not need any filename in the command line, is given along with the corresponding result display in Figure 5.21.

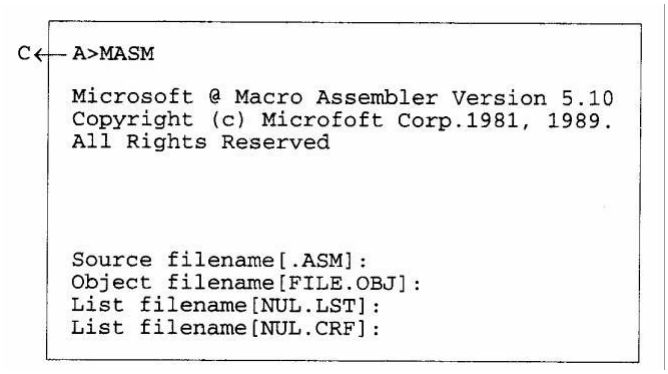


Figure 5.21

If you do not enter the filename to be assembled at the command line as show in Figure 5.20, then you may enter it as a source filename as shown in Figure 5.21. The source filename is to be typed in the source filename line with or without the extension .ASM. The valid filename entry is accepted with a pressure of enter key. On the next line, the expected .OBJ filename is to be entered. The object file of the assembly language program is created with the .OBJ extension. The .OBJ file is created with the entered name and .OBJ extension the coded filename is entered for the .OBJ file before pressing enter key, the new .OBJ file is created with the same name as source file and extension.OBJ. The .OBJ file contains the coded object modules of the program to be assembled. On the next line, a filename is entered expected listing file of the source file, in the same way as the object filename was entered. The listing file is automatically generated in the assembly process. The listing file is identified by the entered or source filename and an extension .LST. The listing file contains the total offset map of the source file including labels, offset addresses, opcodes, memory allotment for different labels and directives and relocation information. The cross reference filename is also entered in the same way as discussed for listing file. The cross reference file is used for debugging the source program and contains the statistical information size of the file in bytes, number of labels, list of labels, routines to be called, etc. about the source program. After the cross-reference file name is entered, the assembly process starts. If the program contains syntax errors, they are displayed using error code number and the corresponding line number at which they appear. After these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully. The successful assembly process may generate the .OBJ, .LST and .CRF files. Those may further be used by the linker programmer to link the object modules and generate an executable (.EXE) file from a .OBJ file. All the above said files may not be generated during the assembling of every program. The generation of some of them may be suppressed using the specific command line options of MASM. The files generated by the MASM are further used by the program LINK.EXE to generate an executable file of the source program.

Linking a Program

The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. The other supporting information may be obtained from the files generated by MASM. The linker program is invoked using the following options.

```
C ──> A> LINK
```

or

```
C ──> A> LINK KMB.OBJ
```

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first option may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The other option also generates the similar display, but will not ask for the .OBJ filename, as it is already specified at the command line. If no filenames are entered for these files, by default, the source filename is considered with the different extensions. The procedure of entering the filenames in LINK is also similar to that in MASM. The LINK command display is as shown in Figure 5.22.

The option input 'Libraries' in the display of Figure 5.22 expects any special library name of which the functions were used by the source program. The output of the LINK program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced versions of MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options that cannot be detailed here for the obvious reasons. For further details users may refer to "Technical reference and Users' Manual-MASM, Version 5".

```
C ←——A>LINK

Microsoft @ Overlay Linker Version. 3.64
Copyright(c)Microsoft Corp. 1983-88. All Rights Reserved.


Object Module[.OBJ]:
Run file[.EXE]:
List file[NUL.MAP]:
Libraries[LIB]:
```

Figure 5.22: Link Command Screen Display

Using DEBUG

DEBUG.COM is a DOS utility that facilitates the debugging and trouble-shooting of assembly language programs. In case of personal computers, all the processor resources and memory resource management functions are carried out by the operating systems. Hence,
users have very little control over the computer hardware at lower levels. The DEBUG utility enables you to have the control of these resources up to some extent. In the simplest, rather, crude words, the DEBUG enables you to use the personal computer as a low level microprocessor kit.

The DEBUG command at DOS prompt invokes this facility. A '_' (dash) display signals the successful invoke operation of DEBUG, that is further used as DEBUG prompt for debugging commands. The following command line, DEBUG prompt and the DEBUG command character display explain the DEBUG command entry procedure, as in Figure 5.23.

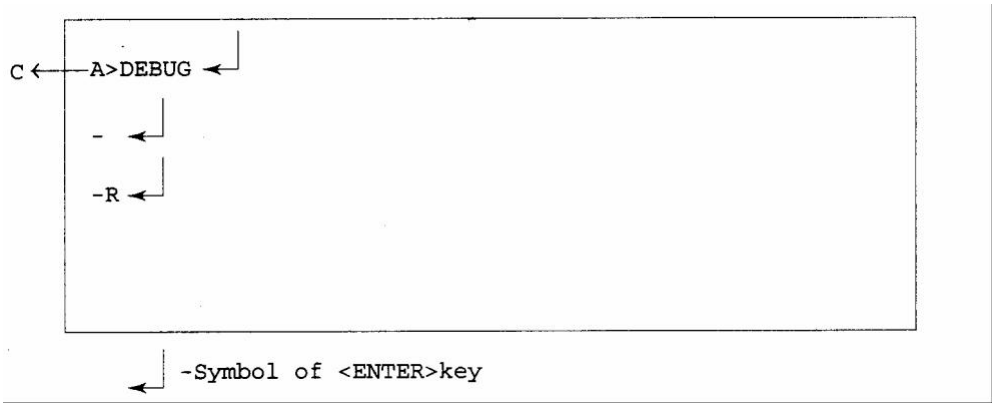


Figure: 5:23 :DEBUG Command Line and Prompt

A valid command is accepted using the enter key. The list of generally used valid commands of DEBUG is given in Table 5.3 along with their respective syntax.

The program DEBUG may be used either to debug a source program or to observe the results of execution of an .EXE file with the help of the .LST file and the above commands. The .LST file shows the offset address allotments for result variables of a program in the particular segment. After execution of the program, the offset address of the result variables may be observed using the d command. The results available in the registers may be observed using the r command. Thus the DEBUG offers a reasonably good platform for trouble-shooting executing and observing the results of the assembly language programs. Here one should note that the DEBUG is able to trouble-shoot only .EXE files.

Table 5.3 Debug Commands

COMMAND CHARACTER	*Format / Formats*	*Functions*
-R	<ENTER>	Display all Registers and flags
-R	reg<ENTER> old contents:New contents	Display specified register contents and modify with the entered new contents.
-D	<ENTER>	Display 128 memory locations of RAM starting from the current display pointer.
-D	SEG:OFFSET1 OFFSET2<ENTER>	Display memory contents in SEG from OFFSET1 to OFFSET2.
-E	<ENTER>	Enter Hex data at current display pointer SEG:OFFSET.
-E	SEG:OFFSET1 <ENTER>	Enter data at SEG:OFFSET1 byte by byte. The memory pointer is to be incremented by space key, data entry is to be completed by pressing the <ENTER> key.
-f	SEG:OFFSET1 OFFSET2 BYTE <ENTER>	Fill the memory area starting from SEG:OFFSET1 to OFFSET2 by the byte BYTE.
-f	SEG:OFFSET1 OFFSET2 BYTE1,BYTE2,BYTE3<ENTER>	Fill the memory area as above with the sequence BYTE1, BYTE2, BYTE3, etc.
-a	<ENTER>	Assemble from the current CS:IP.
-a	SEG:OFFSET <ENTER>	Assemble the entered instruction from SEG:OFFSET address.
-u	<ENTER>	Unassemble from the current CS:IP.
-u	SEG:OFFSET <ENTER>	Unassemble from the address SEG:OFFSET.
-g	<ENTER>	Execute from current CS:IP. By modifying CS and IP using R command this can be used for any address.
-g	=OFFSET <ENTER>	Execute from OFFSET in the current CS.
-s	SEG:OFFSET1 to OFFSET2 BYTE/BYTES <ENTER>	Searches a BYTE or string of BYTES separated by ',' in the memory block SEG:OFFSET1 to OFFSET2, and displays all the offsets at which the byte or string of bytes is found.
-q	<ENTER>	Quit the DEBUG and return to DOS
-T	SEG:OFFSET <ENTER>	Trace the program execution by single stepping starting from the address SEG:OFFSET.
-m	SEG:OFFSET1 OFFSET2 NB <ENTER>	Move NB bytes from OFFSET1 to OFFSET2 in segment SEG
-c	SEG:OFFSET OFFSET2 NB <ENTER>	Copy NB bytes from OFFSET1 to OFFSET2 in segment SEG.
-n	FILENAME.EXE <ENTER>	Set filename pointer to FILENAME
-l	<ENTER>	Load the file FILENAME.EXE as set

- Note that, changing the case of the command letters does not change the command option.
- The entered nubers are consiered as hexadecimal

Student Activity 5.5

Before reading the next section, answer the following question.

1. Write an ALP to convert a four digit hexadecimal number to decimal number.

2. Write an ALP to covert a four digit octal number to decimal number.

3. Write an ALP to fund out ASCSII codes of alphanumeric characters from a look up

table. If your answer is correct, then proceed to the next section.

%), $

In the previous chapter, we studied the complete instruction set of 8086/88, the assembler directives and pseudo-ops. In the previous sections, we have described the procedure of entering an assembly language program into a computer and coding it, i.e. preparing an EXE file from the source code. In this section, we will study some programs which elucidate the use of instructions, directives and some other facilities that are available in assembly language programming. After studying these programs, it is expected that one should have a clear idea about the use of different directives and pseudo-ops, their syntaxes besides understanding the logic of each program. If one writes an assembly language program and tries to code it, in the first attempt, the chances are rare that there is no error. Rather, errors in programming depend upon the skill of the Programmer. A lot of practice is needed to obtain skills in assembly language programming as compared to high level languages. So, to obtain a command on assembly language one should write and execute a number of assembly programs, besides studying the example programs given in this text.

Before starting the explanation of the written programs, we have to explain one more important point, that is about the DOS function calls available under INT 21H instruction. DOS is an operating system, which is a program that stands between a bare computer system hardware and a user. It acts as a user interface with the available computer hardware resources. It also manages the hardware resources of the computer system. In the Disk Operating System, the hardware resources of the computer system like memory, keyboard, CRT display, hard disk, and floppy disk drives can be handled with the help of the instruction INT 21H. The routines required to refer to these resources are written as interrupt service routines for 21H interrupt. Under this interrupt, the specific resource is selected depending upon the value in AH register. For example, if AH contains 09H, then CRT display is to be used for displaying a message or if, AH contain OAH, then the keyboard is to be accessed. These interrupts are called 'function calls' and the value in AH is called 'function value'. In short, the purpose of 'function calls' under INT 21 His to be decided by the 'function value' that is in AH. Some function values also control the software operations of the machine. The list of the function values available under INT 21 H, their corresponding functions, the required parameters and the returns are given in tabulated form in the Appendix-B. Note that there are a number of interrupt functions in DOS, but INT 21H is used more frequently. The readers may find other interrupts of DOS and BIOS from the respective technical references.

Here, a few example programs are presented. Starting from very simple programs, the chapter concludes with more complex programs.

Program 5.1

Write a program for addition of two numbers.

Solution

The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program.

```
ASSUME CS:CODE,DS:DATA
DATA      SEGMENT
OPR1      DW 1234H                ; 1st operand
OPR2      DW 0002H                ; 2nd operand
RESULT    DW 01 DUP(?)            ; A word of memory reserved for result
DATA      ENDS
CODE      SEGMENT
START:    MOV AX,DATA             ; Initialize data segment
          MOV DS,AX               ;
          MOV AX,OPR1             ; Take 1st operand in AX.
          MOV BX,OPR2             ; Take 2nd operand in BX.
          CLC                     ; Clear previous carry if any.
          ADD AX,BX               ; Add BX to AX.
          MOV DI,OFFSET RESULT    ; Take offset of RESULT in DI.
          MOV [DI],AX             ; Store the result at memory address in DI.
          MOV AH,4CH              ; Return to DOS prompt.
          INT 21H
CODE      ENDS                    ; CODE segment ends.
          END START               ; Program ends
```

6 *7

- (. % $

The first step in writing an assembly language program is to define and study the problem. After studying the problem, decide the logical modules required for the program. From the statement of the program one may guess that some data may be required for the program or the result of the program that is to be stored in memory. Hence the program will need a logical space called DATA segment. Invariably CODE segment is a part of a program containing the actual instruction sequence to be executed. If the stack facility is to be used in the program, it will require the STACK segment. The EXTRA segment may be used as an additional destination data segment. Note that the use of all these logical segments is not compulsory except for the CODE segment. Some programs may require DATA and CODE segments, while the others may also contain STACK and EXTRA. For example, Program 5.1 requires only DATA and CODE segment.

The first line of the program containing 'ASSUME' directive declares that the label CODE is to be used as a logical name for CODE segment and the label DATA is to be used for DATA segment. These labels CODE and DATA are reserved by MASM for these purposes only.

They should not be used as general labels. Once this statement is written in the program, CODE refers to the code segment and DATA refers to data segment throughout the program If you want to change it in a program you will have to write another ASSUME statement the program.

The second statement DATA SEGMENT marks the starting of a logical data space DATA Inside DATA segment, OPRI is the first operand. The directive DW defines OPRI as a word operand of value 1234H and OPR2 as a word operand of value 0002H. The third DW directive reserves OIH words of memory for storing the result of the program and leaves it undefined due to the directive DUP(?). The statement DATA ENDS marks the end of the DATA segment. Thus the logical space DATA contains OPRI, OPR2 and RESULT. These labels OPRI, OPR2 and RESULT will be allotted physical memory locations whenever the logical SEGMENT DATA is allocated memory or loaded in the memory of a computer as explained in the previous topic of relocation. The assembler calculates that the above data segment requires 6 bytes, i.e. 2 bytes each for OPRI, OPR2 and RESULT.

The code segment in the above program starts with the statement CODE SEGMENT. The code segment as already explained is a logical segment space containing the instructions. The label STARTS marks the starting point of the execution sequence. The ASSUME statement just informs the assembler that label

CODE is used for the code segment and the label DATA is used for the DATA segment. It does not actually put the address corresponding to CODE in code segment register (CS) and address corresponding to DATA in the data segment register (DS) . This procedure of putting the actual segment address values into the corresponding segment registers is called as segment register initialisation. A programmer has to carry out these initializations for DS, SS and ES using instructions, while the CS automatically initialised by the loader at the time of loading the EXE file into the memory for actual execution. The first two instructions in the program are for data segment initialization. Note that, no segment register in 8086 can be loaded with immediate segment address value, instead the address value should be first loaded into any one of the general purpose registers and then the contents of the general purpose register can be transferred to any of the segment registers DS, ES and SS. Also one should note that CS cannot be loaded at all. Its contents can be changed by using a long jump instruction, a call instruction or an interrupt instruction. For each of the segments DS, ES and SS, the programmer will have to carry out initialization if they are used in the program, while CS is automatically initialized by the loader program at the time of loading and execution. Then the two instructions move the two operands OPRI and OPR2 in AX and BX respectively. Carry is cleared before addition operation (optional in this program). The ADD instruction will add BX into AX and store the result in AX. The instruction used to store the result in RESULT uses different addressing mode than that used for taking OPRI into AX. The indexed addressing mode is used to store the result of addition in memory locations labeled RESULT. The instruction MOVDI, OFFSET RESULT stores the offset of the label RESULT into Dl register. Next instruction stores the result available in AX into the address pointed to by Dl, i.e address of the RESULT. A lot has been already discussed about the function calls under INT 21H. The function value 4CH is for returning to the DOS prompt. If instead of these one writes HLT instruction there will not be any difference in program execution except that the computer will hang as the processor goes to HLT state, and the user will not be able to examine the result. In that case, for further operation, one will have to reset the computer and boot it again. To avoid this resetting of the computer every time you run the program and enable one to check the result, it is better to use the function call 4CH at the end of each program so that after executing the program, the computer returns back to DOS prompt. The statement CODE ENDS marks the end of the CODE segment. The statement END START marks the end of the procedure that started with the label START. At the end of each file, the END statement is a must.

Until now we have discussed every line of Program 5.1 in significant details. As we have already said, the program contains two logical segments CODE and DATA, but it is not at all necessary that all the programs must contain the two segments. A programmer may use a single segment to cover up data as well as instructions. The following listings explain the fact.

```
ASSUME CS:CODE
          CODE        SEGMENT
          OPR1        DW   1234H
          OPR2        DW   0002H
          RESULT      DW   01 DUP(?)
START :   MOV AX,CODE
          MOV DS,AX
          MOV AX,OPR1
          MOV BX,OPR2
          CLC
          ADD AX,BX
          MOV DI,OFFSET RESULT
          MOV [DI],AX
          MOV AH,4CH
          INT 21H
CODE      ENDS
          END START
```

```
6  *7    , 7* 3 ,      /  *7
```

For this program, we have discussed all clauses and clones in details. For all the following programs, we will not explain the common things like forming segments using directives and operators, etc. Instead, just the logic of the program will be explained. The use of proper syntax of the 8086/8088 assembler MASM is self explanatory. The comments may help the reader in getting the ideas regarding the logic of the program.

Assemble the above written program using MASM after entering it into the computer using the procedure explained earlier. Once you get the EXE file as the output of the LINK program, Your listing is ready for execution. The Program 5.1 is prepared in the form of EXE file with the name KMB.EXE in the directory. Next, it can be executed with the command line as given below.

```
C> KMB
```

This execution method will store the result of the program in memory but will not display it on the screen. To display the result on the screen the programmer will have to use DOS function calls. This will make the programs too lengthy. Hence, another method to check the results is to run the program under the control of DEBUG. To run the program under the control of debug and to observe the results one must prepare the LST file, that gives information about memory allotment to different labels and variables of the program while assembling it. The LST file can be displayed on the screen using NE-Norton's Editor.

Program 5.2

Write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers).

Solution

In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                          ; Data segment starts
NUMLIST DB 52H,23H,—                  ; List of byte numbers
COUNT EQU 100D                        ; Number of bytes to be added
RESULT DW 01H DUP(?)                  ; One word is reserved for result.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts at relative
ORG 200H                              ; address 0200h in code segment.
START:      MOV AX,DATA               ; Initialize data segment.
            MOV DS,AX
            MOV CX,COUNT              ; Number of bytes to be added in CX.
            XOR AX,AX                 ; Clear AX and CF.
            XOR BX,BX                 ; Clear BH for converting the byte to word
            MOV SI,OFFSET NUMLIST     ; Point to the first number in the list.
AGAIN:      MOV BL,[SI]               ; Take the first number in BL, BH is zero
            ADD AX,BX                 ; Add AX with BX.
            INC SI                    ; Increment pointer to the byte list.
            DEC CX                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are added,point to result
            MOV DI,OFFSET RESULT      ; destination and store it.
            MOV [DI],AX
            MOV AH,4CH                ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

6 *7

The use of statement ORG 200H in this program is not compulsory. We have used this statement here just to explain the way to use it. It will not affect the result at all. Whenever the program is loaded into the memory whatever is the address assigned for CODE, the executable code starts at the offset address 0200H due to the above statement. Similar to DW, the directive DB reserves space for the list of 8 -bit numbers in the series. The procedure for entering the program, coding and execution has already been explained. The result of addition will be stored in the memory locations allotted to the label RESULT.

Program 5.3

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution

Compare the ith number of the series with the (i+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (i+1)th number is greater. If the ith number is greater than (i+1)th, leave it in AX (any register may be used). Otherwise, load the (i+1)th number in AX, replacing the ith number in AX. The procedure is repeated tilt all the members in the array have been compared.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                          ; Data segment starts
LIST DB 52H,23H,56H,45H,——            ; List of byte numbers
COUNT EQU 0F                          ; Number of bytes in the list
LARGEST DB 01H DUP(?)                 ; One byte is reserved for the largest number.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts.
START:      MOV AX,DATA               ; Initialize data segment.
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT              ; Number of bytes in CL.
            MOV AL,[SI]               ; Take the first number in AL
AGAIN:      CMP AL,[SI+1]             ; and compare it with the next number.
            JNL NEXT
            MOV AL,[SI+1]
NEXT:       INC SI                    ; Increment pointer to the byte list.
            DEC CL                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are compared,point to result
            MOV SI,OFFSET LARGEST     ; destination and store it.
            MOV [SI],AL
            MOV AH,4CH                ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

6 *7

Program 5.4

Modify the Program 5.3 for a series of words.

Solution

The logic is similar to the previous program written for a series of byte numbers. The program is directly written as follows without any comment leaving it to the reader to find out the use of each instruction and directive used.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 1234H,2354H,0056H,045AH,——
COUNT EQU 0F
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT
            MOV AX,[SI]
AGAIN:      CMP AX,[SI+2]
            JNL NEXT
            MOV AX,[SI+2]
NEXT:       INC SI
            INC SI
            DEC CL
            JNZ AGAIN
            MOV SI,OFFSET LARGEST
            MOV [SI],AX
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END     START
```

6 *7

Program 5.5

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Solution

The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number. If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2357H,0A579H,0C322H,0C91EH,0C000H,0957H
COUNT EQU 006H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI, OFFSET LIST
AGAIN:      MOV AX,[SI]

            ROR AX,01
            JC ODD
            INC BX
            JMP NEXT
ODD:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

6 *7

Program 5.6

Write a program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Solution

Take the ith number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. If CF is 1, the number is negative; otherwise, it is positive.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2579H,0A500H,0C009H,0159H,0B900H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI,OFFSET LIST
AGAIN:      MOV AX,[SI]
            SHL AX,01
            JC NEG
            INC BX
            JMP NEXT
NEG:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

6 *7 "

The logic of Program 5.6 is similar to that of Program 5.5, hence comments are not given in Program 5.6 except for a few important ones.

Program 5.7

Write a program to move a string of data words from offset 2000H to offset 3000H the length of the string is OFH.

Solution

To write this program, we will use an important facility, available in the 8086 instruction set, i.e. move string byte/word instruction. We will also study the flexibility imparted by this instructions to the 8086 assembly language program. Let us first write the Program 5.7 for 8085, assuming that the string is available at location 2000H and is to

```
            LXI H , 2000H
            LXI D , 3000H
            MVI C , 0FH
AGAIN :     MOV A , M
            STAX D
            INX  H
            INX  D
            DCR  C
            JNZ  AGAIN
            HLT
```

%'% 6 *7 6 *7 $

Now assuming DS is suitably set, let us write the sequence for 8086. At first using the index registers, the program can be written as given.

```
        MOV  SI ,2000H
        MOV  DI ,3000H
        MOV  CX ,0FH
AGAIN : MOV  AX ,[SI]
        MOV  [DI],AX
        INC  SI
        INC  DI
        DEC  CX
        JNZ  AGAIN
        HLT
```

%'%" 6 *7 6 *7 $

Comparing the above listings for 8085 and 8086 we may infer that every instruction in 8085 listing is replaced by an equivalent instruction of 8086. The above 8086 listing is absolutely correct but it is not efficient. Let us try to write the listings for the same purpose using the string instruction. Due to the assembler directives and the syntax, one may feel that the program is lengthy, though it eliminates four instructions for a MOVSW instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV SI,SOURCESTRT
            MOV DI,DESTSTRT
            MOV CX,COUNT
            CLD
REP         MOVSW
            MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

6 *7 $

An 8086 Program listing for Program 5.7 using String instruction

Compare the above two 8086 listings. Both contain ten instructions. However, in case of the second program, the instruction for the initialisation of segment register and DOS interrupt are additional while the first one neither contains initialisation of any segment registers nor does it contain the DOS interrupt instruction. We can say that the first program uses 9 instructions, while the second one uses only 5 for implementing the same algorithm. This program and the related discussions are aimed at explaining the importance of the string instructions and the method to use them.

Program 5.8

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Solution

There exist a large number of sorting algorithms. The algorithm used here is called bubble sorting. The method of sorting is explained as follows. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series (n-1) times. After (n-1) iterations, you will get the largest number at the end of the series, where n is the length of the series. Again start from the first address of the series. Repeat the same procedure right from the first element to the last element. After (n -2) iterations you will get the second highest number at the last but one place in the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

53 , 25 , 19, 02		n = 4
25 , 53 , 19, 02		1st operation
25 , 19 , 53 , 02		2nd operation
25 , 19 , 02 , 53		3rd operation
largest no.	⇒	4 – 1 = 3 operations
19 , 25 , 02 , 53		1st operation
19 , 02 , 25 , 53		2nd operation

2nd largest number => 4-2=2 operations

02, 19, 25, 53 1st operation

3rd largest number => 4 -3 = 1 operations

Instead of taking a variable count for the external loop in the program like (n - 1), (n – 2), (n-3),...., etc. It is better to take the count (n- 1) all the time for simplicity. The resulting program is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 53H,25H,19H,02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DX,COUNT-1
AGAIN0:     MOV CX,DX
            MOV SI,OFFSET LIST
AGAIN1:     MOV AX,[SI]
            CMP AX,[SI+2]
            JL  PR1
            XCHG [SI+2],AX
            XCHG [SI],AX
PR1:        ADD SI,02
            LOOP AGAIN1
            DEC DX
            JNZ AGAIN0
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

6 *7 %

With a similar approach, the reader may write a program to arrange the string in descending order. Just instead of the JL instruction in the above program, one will have to use a JG instruction.

Program 5.9

Write a program to perform a one byte BCD addition.

Solution

It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

```
  92
+ 59
------
  EB
```

Actual result after addition considering hex. operands

```
        1011
     +  0110        As 0BH (LSD of addition) > 09, add 06 to it.
     -------
       10001        Least significant nibble of result (neglect the auxiliary carry)
                    → AF is set to 1
```

0110 is added to most significant nibble of the result if it is greater than 9 or AF is set.

```
                          1      Carry from previous digit (AF)
           E    →      1110
                +      0110
                -----------
CF is set to 1         0101      next significant nibble of result

Result CF       Most significant      Least significant digit
   1                  5                        1
```

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 92H
            OPR2 EQU 52H
RESULT DB 02 DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS,AX
            MOV BL,OPR1
            XOR AL,AL
            MOV AL,OPR2
            ADD AL,BL
            DAA
            MOV RESULT,AL
            JNC MSB0
            INC [RESULT+1]
MSB0:       MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

6 *7 &

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after SUB instruction. The reader may try to write a program for BCD subtraction for practice.

Program 5.10

Write a program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

Solution

Here we have directly given the routine for Program 5.10.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 98H
```

```
                OPR2 EQU 49H
                SUM DW 01 DUP(00)
                SUBT DW 01 DUP(00)
                PROD DW 01 DUP(00)
                DIVS DW 01 DUP(00)
DATA            ENDS
CODE            SEGMENT
START:          MOV AX,DATA
                MOV DS,AX
                MOV BL,OPR2
                XOR AL,AL
                MOV AL,OPR1
                ADD AL,BL
                DAA
                MOV BYTE PTR SUM,AL
                JNC MSB0
                INC [SUM+1]
MSB0:           XOR AL,AL
                MOV AL,OPR1
                SUB AL,BL
                DAS
                MOV BYTE PTR SUBT,AL
                JNB MSB1
                INC [SUBT+1]
MSB1:           XOR AL,AL
                MOV AL,OPR1
                MUL BL
                MOV WORD PTR PROD,AX
                XOR AH,AH
                MOV AL,OPR1
                DIV BL
                MOV WORD PTR DIVS,AX
                MOV AH,4CH
                INT 21H
CODE            ENDS
                END START
```

6 *7 '

Program 5.11

Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Solution

The given string is scanned for the given byte. If it is found in the string, the zero flag is set; else, it is reset. Use of the SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found out. Note that, in this program, the code segment is written before the data segment.

```
ASSUME CS:CODE,DS:DATA
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV CX,COUNT
            MOV DI,OFFSET STRING
            MOV BL,00H
            MOV AL,BYTE1
SCAN1:      NOP
            SCASB [DI]
            JZ XXX
            INC BL
            LOOP SCAN1
XXX:        MOV AH,4CH
            INT 21H
            CODE ENDS
DATA SEGMENT
BYTE1 EQU 25H
COUNT EQU 06H
STRING DB 12H,13H,20H,20H,25H,21H
DATA ENDS
            END START
```

6 *7

Program 5.12

Write a program to convert the BCD numbers 0 to 9 to their equivalent sever segment codes using the look-up table technique. Assume the codes [7-seg] are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

Solution

Refer to the explanation of the XLAT instruction. The statement of the program itself gives the explanation about the logic of the program.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            CODELIST DB 34,45,56,45,23,12,19,24,21,00
            CHAR EQU 05
            CODEC    DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV BX,OFFSET CODELIST
            MOV AL, CHAR
            XLAT
            MOV BYTE PTR CODEC,AL
```

```
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

6 *7

Program 5.13

Decide whether the parity of a given number is even or odd. If parity is even set DL to 00; else, set DL to 1. The given number may be a multibyte

number. Solution

The simplest algorithm to check the parity of a multibyte number is to go on adding the parity byte by byte with OOH. The result of the addition reflects the parity of that byte of the multibyte number. Adding the parities of all the bytes of the number, one will obtain the over all parity of the number.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            NUM DD 335A379BH
            BYTE_COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DH,BYTE_COUNT
            XOR AL,AL
            MOV CL,00
            MOV SI,OFFSET NUM
NEXT_BYTE:  ADD AL,[SI]
            JP EVENP
            INC CL
EVENP:      INC SI
            MOV AL,00
            DEC DH
            JNZ NEXT_BYTE
            MOV DL,00
            RCR CL,1
            JNC CLEAR
            INC DL
CLEAR:      MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

6 *7

The contents of CL are incremented depending upon either the parity for that byte is even or odd. If LSB of CL is I, after testing for all bytes, it means the parity of the multibyte number is odd otherwise it is even and DL is modified correspondingly.

Program 5.14

Write a program for the addition of two 3 x 3 matrices . The matrices are stored in the form of lists (row wise). Store the result of addition in the third list.

Solution

In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

$$[A] \quad + \quad [B] \quad = \quad [A+B]$$

The matrix A is stored in the memory at an offset MAT1, as given:

a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{32}, a_{33}, etc.

A total of 3 x3= 9 additions are to be done. The assembly language program is written as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            DIM EQU 09H
            MAT1 DB 01,02,03,04,05,06,07,08,09
            MAT2 DB 01,02,03,04,05,06,07,08,09
            RMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,DIM
            MOV SI,OFFSET MAT1
            MOV DI,OFFSET MAT2
            MOV BX,OFFSET RMAT3
NEXT:       XOR AX,AX
            MOV AL,[SI]
            ADD AL, [DI]
            MOV WORD PTR [BX],AX
            INC SI
            INC DI
            ADD BX,02
            LOOP NEXT
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

6 *7

Program 5.15

Write a program to find out the product of two matrices. Store the result in the third matrix. The matrices are specified as in the Program 5.14.

Solution

The multiplication of matrices is carried out as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{12}b_{12} + a_{22}b_{12} + a_{13}b_{32} & a_{13}b_{13} + a_{12}b_{23} + a_{13}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

The listings to carry out the above operation is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
ROCOL EQU 03H
MAT1 DB 05H,09H,0AH,03H,02H,07H,03H,00H,09H
MAT2 DB 09H,07H,02H,01H,0H,0DH,7H,06H,02H
PMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CH,ROCOL
            MOV BX,OFFSET PMAT3
            MOV SI,OFFSET MAT1
NEXTROW:    MOV DI,OFFSET MAT2
            MOV CL,ROCOL
NEXTCOL:    MOV DL,ROCOL
            MOV BP,0000H
            MOV AX,0000H
            SAHF
```

6 *7

Program 5.16

Write a program to add two multibyte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

Solution

This program is simila xcept for the addition instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BYTES EQU 08H
NUM1 DB 05,5AH,6CH,55H,66H,77H,34H,12H
NUM2 DB 04,56H,04H,57H,32H,12H,19H,13H
NUM3 DB 0AH DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,BYTES
            MOV SI,OFFSET NUM1
            MOV DI,OFFSET NUM2
            MOV BX,OFFSET NUM3
            XOR AX,AX
NEXTBYTE:   MOV AL,[SI]
            ADC AL,[DI]
```

```
                MOV BYTE PTR[BX],AL
                INC SI
                INC DI
                INC BX
                DEC CX
                JNZ NEXTBYTE
                JNC NCARRY
                MOV BYTE PTR[BX],01
NCARRY:         MOV AH,4CH
                INT 21H
CODE ENDS
                END START
```

6 *7 "

Student Activity 5.6

Answer the following question.

1. Write an ALP to perform a sixteen-bit increment operation using 8-bit instructions.
2. Write an ALP to find out average of a given string of data bytes neglecting fractions.
3. Write an ALP to find out decimal addition of sixteen four digit decimal numbers.
4. Write an ALP to convert a four digit decimal number to its binary equivalent.

This unit is aimed at introducing the readers with the instruction set of 8086/88 and the most commonly used assembler directives and operators. To start with, the available instruction formats in 8086/88 instruction set are explained in details. Further, the addressing modes available in 8086/88 are discussed in significant details with necessary examples. The 8086/88 instructions can be broadly categorized in six types depending upon their functions, namely data transfer instructions, arithmetic instructions and logical instructions, shift and rotate instructions, string manipulation instructions, control transfer instructions and processor control instructions. All these instruction types have been discussed before proceeding with the assembler directives and operators. The coding information details of all these instructions may be obtained from the Intel Appendix. The necessary assembler directives have been discussed later with their possible syntax, functions and examples. Most of these directives are available in Microsoft MASM. The detailed discussion on every assembler directive and operator is out of scope of this book.

This unit starts with simple programs written for hand-coding. Further, hand-coding procedures of a few example instructions are studied. Advantages of the assembly language over machine language are then presented in brief. With an overview of the assembler operation, we have initiated the discussion of assembly language programming. The procedures of writing, entering and coding the assembly language programs are then discussed in brief. Further, DOS debugging program - DEBUG, a basic tool for trouble-shooting the assembly language programs, is discussed briefly with an emphasis on the most useful commands. Then we have presented various examples of 8086/8088 programs those highlight the actual use and the syntax of the instructions and directives. Finally, a few programs those enable the programmer to access the computer system resources using DOS function calls are discussed. We do not claim that each program presented here is the most efficient one, rather it just suggests a way to implement the algorithm asked in the problem. There may be a number of

alternate algorithms and program listings to implement a logic but amongst them a programmer should choose one which requires minimum storage, execution time and complexity.

/ 0

I. State True or False

1. Assembly language depends on the CPU architecture of the machine for which it was designed.
2. An assembly language for solving a problem is very efficient compared to the same program written in a High Level Language.
3. Writing application programs in assembly language is difficult compared to writing in high level language.
4. There are many assemblers available for the IBM PC.
5. In immediate type addressing the time required to fetch the desired data is zero.

II. Fill in the blanks

1. Assembly language use _____ to represent operation codes.
2. Assembly language use _____ to represent operands.
3. One assembly language instruction become _____machine for which it was designed.
4. A translator which translates an assembly language program to machine language is called _____.
5. Early IBM PCs used _____ chip as CUP.

I. State True or False

1. True
2. True
3. True
4. False
5. False

II. Fill in the blanks

1. mnemonics
2. symbols
3. one

4. compiler
5. Intel 8088/8086

I. State True or False

1. Direct addressing mode is used when speed is important.
2. Registers are normally directly addressed as the number of registers in CPU is small.
3. A disadvantage of immediate addressing is negative operands cannot be used.
4. Indirect addressing through a register requires 3 memory accesses to fetch data.
5. Subroutines are self-contained programs which can be assembled separately.
6. A subroutine may call itself.
7. A subroutine can be independently debugged.
8. When a program call a subroutine the return address is stored in bottom of stack.

II. Fill in the blanks

1. Assembly language operand can be a _____. List all correct answers::

(a) number (b) character constant
(c) variable name (d) an absolute address
(e) symbolic address (f) mnemonic code

2. Directives in an assembly language are required to _____
(Pick as many as relevant).

(a) read a program (b) specify values to be stored
(c) allocate space in memory (d) find execution time
(e) find error in program

3. Data registers are used to stone _____.

(a) instructions (b) operands
(c) operators (d) operators

4. ALU stands for _____.

(a) All Logic Unit (b) Arithmetic Logic Unit
(c) Another Legal Unit (d) All-purpose Language Utility

5. The number of bits in an address bus equals the number of bits in _____.

(a) memory (b) data register

(c) address register (d) flags resister

6. The operation code of an assembly language program is represented by a _____.

(a) code (b) mnemonic

(c) instruction (d) number

7. At the end of the sequence of instructions

MOV AX, 0F289h

MOV BL, AH

BL will store _____.

(a) 89h (b) F2h

(c) 28h (d) F9h

" 0

1. What is an assembler?
2. What is a linker?
3. Write an ALP to change an already available ascending order byte string to descending order.
4. Write an ALP to convert a given sixteen bit binary number to its GRAY equivalent.
5. Write an ALP to find out transpose of a 3x3 matrix.
6. Write an ALP to find out cube of an 8-bit hexadecimal number.