UNIT 3 ASSEMBLY LANGUAGE PROGRAMMING (PART – I)

Structure **Page No.**

3.0 INTRODUCTION

After discussing a few essential directives, program developmental tools and simple programs, let us discuss more about assembly language programs. In this unit, we will start our discussions with simple assembly programs, which fulfil simple tasks such as data transfer, arithmetic operations, and shift operations. A key example here will be about finding the larger of two numbers. Thereafter, we will discuss more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. Finally, we will discuss more complex arithmetic and string operations. You must refer to further readings for more discussions on these programming concepts.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- write assembly programs with simple arithmetic logical and shift operations;
- implement loops;
- use comparisons for implementing various comparison functions;
- write simple assembly programs for code conversion; and
- write simple assembly programs for implementing arrays.

3.2 SIMPLE ASSEMBLY PROGRAMS

As part of this unit, we will discuss writing assembly language programs. We shall start with very simple programs, and later graduate to more complex ones.

3.2.1 Data Transfer

Two most basic data transfer instructions in the 8086 microprocessor are MOV and XCHG. Let us give examples of the use of these instructions.

; **Program 1:** This program shows the difference of MOV and XCHG instructions:

```
DATA  SEGMENT
        VAL    DB      5678H           ; initialize variable VAL
DATA  ENDS

CODE  SEGMENT
                ASSUME        CS:     CODE, DS: DATA
MAINP:          MOV           AX, 1234H     ; AH=12 & AL=34
                XCHG          AH, AL        ; AH=34 & AL=12
                MOV           AX, 1234H     ; AH=12 & AL=34
                MOV           BX, VAL       ; BH=56 & BL=78
                XCHG          AX, BX        ; AX=5678 & BX=1234
                XCHG          AH, BL        ; AH=34, AL=78, BH=12, & BL=56
                MOV           AX, 4C00H     ; Halt using INT 21h
                INT           21H
CODE  ENDS
END   MAINP
```

Discussion:

Just keep on changing values as desired in the program.

; **Program 2:** Program for interchanging the values of two Memory locations
; **input:** Two memory variables of same size: 8-bit for this program

```
DATA SEGMENT
        VALUE1  DB        0Ah             ; Variables
        VALUE2  DB        14h
DATA ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA
        MOV AX, DATA                      ; Initialise data segments
        MOV DS, AX                        ; using AX
        MOV AL, VALUE1                    ; Load Value1 into AL
        XCHG VALUE2,AL                    ; exchange AL with Value2.
        MOV VALUE1,AL                     ; Store Al in Value1
        INT 21h                           ; Return to Operating system
        CODE ENDS
END
```

Discussion:

The question is why cannot we simply use XCHG instruction with two memory variables as operand? To answer the question let us look into some of constraints for the MOV & XCHG instructions:

The MOV instruction has the following constraints and operands:

- CS and IP may never be destination operands in MOV;
- Immediate data value and memory variables may not be moved to segment registers;
- The source and the destination operands should be of the same size;
- Both the operands cannot be memory locations;
- If the source is immediate data, it must not exceed 255 (FFh) for an 8-bit destination or 65,535 (FFFFh) for a 16-bit destination.

The statement MOV AL, VALUE1, copies the VALUE1 that is 0Ah in the AL register:

```
AX : 00  0A  <-----------------  0A    (VALUE1)
     AH AL                       14    (VALUE2)
```

The instruction, XCHG AL, VALUE2 ; exchanges the value of AL with VALUE2

Now AL and VALUE2 contains and values as under:

```
AX :  00  14  <------+           0A    (VALUE1)
                     +---------> 0A    (VALUE2)
```

The statement, MOV VALUE1, AL ;, now puts the value of AL to VALUE1.

Thus the desired exchange is complete

```
AX :  00  14  ---------------->  14    (VALUE1)
                                 0A    (VALUE2)
```

Other statements in the above program have already been discussed in the preceding units.

3.2.2 Simple Arithmetic Application

Let us discuss an example that uses simple arithmetic:

```
; Program 3: Find the average of two values stored in
; memory locations named FIRST and SECOND
; and puts the result in the memory location AVGE.

; Input :  Two memory variables stored in memory locations FIRST and SECOND
; REGISTERS                 ; Uses DS, CS, AX, BL
; PORTS                     ; None used
DATA          SEGMENT
        FIRST     DB  90h   ; FIRST number,       90h is a sample value
        SECOND DB  78h      ; SECOND number,      78h is a sample value
        AVGE      DB  ?     ; Store average here
DATA      ENDS
CODE      SEGMENT
        ASSUME CS:CODE, DS:DATA
START:    MOV   AX, DATA           ; Initialise data segment, i.e. set
          MOV   DS, AX             ; Register DS to point to Data Segment
          MOV   AL, FIRST          ; Get first number
          ADD   AL, SECOND         ; Add second to it
          MOV   AH, 00h            ; Clear all of AH register
          ADC   AH,  00h           ; Put carry in LSB of AH
          MOV   BL,  02h           ; Load divisor in BL register
          DIV   BL                 ; Divide AX by BL. Quotient in AL,
                                   ; and remainder in AH
          MOV   AVGE, AL           ; Copy result to memory
CODE      ENDS
        END    START
```

Discussion:

An add instruction cannot add two memory locations directly, so we moved a single value in AL first and added the second value to it.

Please note, on adding the two values, there is a possibility of carry bit. (The values here are being treated as unsigned binary numbers). Now the problem is how to put the carry bit into the AH register such that the AX(AH:AL) reflects the added value. This is done using ADC instruction.

The ADC AH,00h instruction will add the immediate number 00h to the contents of the carry flag and the contents of the AH register. The result will be left in the AH register. Since we had cleared AH to all zeros, before the add, we really are adding 00h + 00h + CF. The result of all this is that the carry flag bit is put in the AH register, which was desired by us.

Finally, to get the average, we divide the sum given in AX by 2. A more general program would require positive and negative numbers. After the division, the 8-bit quotient will be left in the AL register, which can then be copied into the memory location named AVGE.

3.2.3 Application Using Shift Operations

Shift and rotate instructions are useful even for multiplication and division. These operations are not generally available in high-level languages, so assembly language may be an absolute necessity in certain types of applications.

; **Program 4:** Convert the ASCII code to its BCD equivalent. This can be done by simply replacing the bits in the upper four bits of the byte by four zeros. For example, the ASCII '1' is 32h = 0011 0010B. By making the upper four bits as 0 we get 0000 0010 which is 2 in BCD. The number obtained is called unpacked BCD number. The upper four bits of this byte is zero. So the upper four bits can be used to store another BCD digit. The byte thus obtained is called packed BCD number. For example, an unpacked BCD number 59 is 00000101 00001001, that is, 05 09. The packed BCD will be 0101 1001, that is 59.

The algorithm to convert two ASCII digits to packed BCD can be stated as:

Convert first ASCII digit to unpacked BCD.
Convert the second ASCII digit to unpacked BCD.

Decimal	ASCII	BCD
5	00110101	00000101
9	00111001	00001001

Move first BCD to upper four positions in byte.

0101 0000	Using Rotate Instructions

Pack two BCD bits in one byte.

	0101 0000	
	0000 1001	
Pack	0101 1001	Using OR

;The assembly language program for the above can be written in the following manner.

```
; ABSTRACT          Program produces a packed BCD byte from 2 ASCII
                    ; encoded digits. Assume the number as 59.

                    ; The first ASCII digit (5) is loaded in BL.
                    ; The second ASCII digit (9) is loaded in AL.
                    ; The result (packed BCD) is left in AL.
```

```
; REGISTERS         ; Uses CS, AL, BL, CL
; PORTS             ; None used
CODE      SEGMENT
          ASSUME    CS:CODE
START:    MOV  BL,  '5'    ; Load first ASCII digit in BL
          MOV  AL,  '9'    ; Load second ASCII digit in AL
          AND  BL,  0Fh    ; Mask upper 4 bits of first digit
          AND  AL,  0Fh    ; Mask upper 4 bits of second digit
          MOV  CL,  04h    ; Load CL for 4 rotates
          ROL  BL,  CL     ; Rotate BL 4 bit positions
          OR   AL,  BL     ; Combine nibbles, result in AL contains 59
                           ; as packed BCD
CODE      ENDS
          END       START
```

Discussion:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore we need to use the rotate instructions that too by 4 times. Out of the two rotate instructions, ROL and RCL, we have chosen ROL, as it rotates the byte left by one or more positions, on the other hand RCL moves the MSB into the carry flag and brings the original carry flag into the LSB position, which is not what we want.

Let us now look at a program that uses RCL instructions. This will make the difference between the instructions clear.

; **Program 5:** Add a byte number from one memory location to a byte from the next memory location and put the sum in the third memory location. Also, save the carry flag in the least significant bit of the fourth memory location.

```
; ABSTRACT          : This program adds 2-8-bit words in the memory locations
;                   : NUM1 and NUM2. The result is stored in the memory
;                   : location RESULT. The carry bit, if any will be stored as
;                   : 0000 0001 in the location CARRY

; ALGORITHM:
;          get NUM1
;          add NUM2 in it
;          put sum into memory location RESULT
;          rotate carry in LSB of byte
;          mask off upper seven bits of byte
;          store the result in the CARRY location.
;
; PORTS             : None used
; PROCEDURES        : None used
; REGISTERS         : Uses CS, DS, AX
;
DATA      SEGMENT
     NUM1      DB   25h    ; First number
     NUM2      DB   80h    ; Second number
     RESULT    DB   ?      ; Put sum here
     CARRY     DB
DATA      ENDS
CODE      SEGMENT
     ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA         ; Initialise data segment
     MOV DS, AX            ; register using AX
     MOV AL, NUM1          ; Load the first number in AL
     ADD AL, NUM2          ; Add 2nd number in AL
```

```
        MOV RESULT, AL              ; Store the result
        RCL AL, 01                  ; Rotate carry into LSB
        AND AL, 00000001B           ; Mask out all but LSB
        MOV CARRY, AL               ; Store the carry result
        MOV  AH, 4CH
        INT 21H
CODE        ENDS
END         START
```

Discussion:

RCL instruction brings the carry into the least significant bit position of the AL register. The AND instruction is used for masking higher order bits, of the carry, now in AL.

In a similar manner we can also write applications using other shift instructions.

3.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP. However, this instruction only sets the flags on comparing two operands (both 8 bits or 16 bits). Compare instruction just subtracts the value of source from destination without storing the result, but setting the flag during the process. Generally only three comparisons are more important. These are:

Result of comparison	Flag(s) affected
Destination < source	Carry flag = 1
Destination = source	Zero flag = 1
Destination > source	Carry = 0, Zero = 0

Let's look at three examples that show how the flags are set when the numbers are compared. In example 1 BL is less than 10, so the carry flag is set. In example 2, the zero flag is set because both operands are equal. In example 3, the destination (BX) is greater than the source, so both the zero and the carry flags are clear.

Example 1:

```
MOV BL, 02h
CMP BL, 10h           ; Carry flag = 1
```

Example 2:

```
MOV AX, F0F0h
MOV DX, F0F0h
CMP AX,DX             ; Zero flag = 1
```

Example 3:

```
MOV BX, 200H
CMP BX, 0             ; Zero and Carry flags = 0
```

In the following section we will discuss an example that uses the flags set by CMP instruction.

Check Your Progress 1

State True or False with respect to 8086/8088 assembly languages. | T | F |

1. In a MOV instruction, the immediate operand value for 8-bit destination cannot exceed F0h.
2. XCHG VALUE1, VALUE2 is a valid instruction.
3. In the example given in section 3.2.2 we can change instruction DIV BL with a shift.
4. A single instruction cannot swap the upper and lower four of a byte register.
5. An unpacked BCD number requires 8 bits of storage, however, two unpacked BCD numbers can be packed in a single byte register.
6. If AL = 05 and BL = 06 then CMP AL, BL instruction will clear the zero and carry flags.

3.3 PROGRAMMING WITH LOOPS AND COMPARISONS

Let us now discuss a few examples which are slightly more advanced than what we have been doing till now. This section deals with more practical examples using loops, comparison and shift instructions.

3.3.1 Simple Program Loops

The loops in assembly can be implemented using:

- Unconditional jump instructions such as JMP, or
- Conditional jump instructions such as JC, JNC, JZ, JNZ etc. and
- Loop instructions.

Let us consider some examples, explaining the use of conditional jumps.

Example 4:

```
        CMP   AX,BX          ; compare instruction: sets flags
        JE    THERE          ; if equal then skip the ADD instruction
        ADD   AX, 02         ; add 02 to AX

THERE:  MOV   CL, 07         ; load 07 to CL
```

In the example above the control of the program will directly transfer to the label THERE if the value stores in AX register is equal to that of the register BX. The same example can be rewritten in the following manner, using different jumps.

Example 5:

```
        CMP   AX, BX         ; compare instruction: sets flags
        JNE   FIX            ; if not equal do addition
        JMP   THERE          ; if equal skip next instruction
FIX:    ADD   AX, 02         ; add 02 to AX
```

```
THERE:  MOV   CL, 07
```

The above code is not efficient, but suggest that there are many ways through which a conditional jump can be implemented. Select the most optimum way.

Example 6:

```
        CMP    DX, 00          ; checks if DX is zero.
        JE     Label1          ; if yes, jump to Label1 i.e. if ZF=1

Label1:----                    ; control comes here if DX=0
```

Example 7:

```
        MOV    AL, 10          ; moves 10 to AL
        CMP    AL, 20          ; checks if AL < 20 i.e. CF=1
        JL     Lab1            ; carry flag = 1 then jump to Lab1

        Lab1: ------           ; control comes here if condition is satisfied
```

LOOPING

```
; Program 6: Assume a constant inflation factor that is added to a series of prices
; stored in the memory. The program copies the new price over the old price. It is
; assumed that price data is available in BCD form.

; The algorithm:

;Repeat
;        Read a price from the array
;        Add inflation factor
;        Adjust result to correct BCD
;        Put result back in array
;        Until all prices are inflated

; REGISTERS: Uses DS, CS, AX, BX, CX
; PORTS      : Not used
ARRAYS       SEGMENT
             PRICE        DB      36h, 55h, 27h, 42h, 38h, 41h, 29h, 39h
ARRAYS       ENDS
CODE         SEGMENT
             ASSUME CS:CODE, DS:ARRAYS
START:       MOV   AX, ARRAYS  ; Initialize data segment
             MOV   DS, AX      ; register using AX
             LEA   BX, PRICES  ; initialize pointer to base of array
             MOV   CX, 0008h   ; Initialise counter to 8 as array have 8
                               ; values.
DO_NEXT:     MOV   AL, [BX]    ; Copy a price to AL. BX is addressed in
                               ; indirect mode.
             ADD   AL, 0Ah     ; Add inflation factor
             DAA               ; Make sure that result is BCD
             MOV   [BX], AL    ; Copy result back to the memory
             INC   BX          ; increment BX to make it point to next price
             DEC   CX          ; Decrement counter register
             JNZ   DO_NEXT     : If not last, (last would be when CX will
                               ; become 0) Loop back to DO_NEXT
             MOV   AH, 4CH     ; Return to DOS
             INT   21H
CODE         ENDS
             END   START
```

Discussion:

Please note the use of instruction: LEA BX,PRICES: It will load the BX register with the offset of the array PRICES in the data segment. [BX] is an indirection through BX and contains the value stored at that element of array. PRICES. BX is incremented to point to the next element of the array. CX register acts as a loop counter and is decremented by one to keep a check of the bounds of the array. Once the CX register becomes zero, zero flag is set to 1. The JNZ instruction keeps track of the value of CX, and the loop terminates when zero flag is 1 because JNZ does not loop back. The same program can be written using the LOOP instruction, in such case, DEC CX and JNZ DO_NEXT instructions are replaced by LOOP DO_NEXT instruction. LOOP decrements the value of CX and jumps to the given label, only if CX is not equal to zero.

Let us demonstrate the use of LOOP instruction, with the help of following program:

; **Program 7:** This following program prints the alphabets (A-Z)

; Register used : AX, CX, DX

```
CODE  SEGMENT
        ASSUME : CS:CODE.
MAINP:  MOV   CX, 1AH      ; 26 in decimal  = 1A in  hexadecimal Counter.
        MOV   DL, 41H      ; Loading DL with ASCII hexadecimal of  A.
NEXTC:  MOV   AH, 02H      ; display result character in DL
        INT   21H          ;  DOS interrupt
        INC   DL           ; Increment  DL for next char
        LOOP  NEXTC        ;  Repeat until CX=0.(loop automatically decrements
                           ;  CS and checks whether it is zero or not)
        MOV   AX, 4C00H    ;  Exit DOS
        INT   21H          ;  DOS Call
CODE ENDS
END MAINP
```

Let us now discuss a slightly more complex looping program.

; **Program 8:** This program compares a pair of characters entered through keyboard.
; Registers used: AX, BX, CX, DX

```
DATA SEGMENT
             XX  DB  ?
             YY  DB  ?
DATA ENDS

CODE SEGMENT
        ASSUME    CS:  CODE,   DS: DATA
MAINP:       MOV  AX,  DATA         ; initialize data
             MOV  DS,  AX           ; segment using AX
             MOV  CX,  03H          ; set counter to 3.
NEXTP:       MOV  AH,  01H          ; Waiting for user to enter a char.
             INT   21H
             MOV  XX, AL            ; store the 1st input character in XX
             MOV  AH,  01H          ; waiting for user to enter second
             INT    21H             ;          character.
             MOV  YY, AL            ; store the character to YY
             MOV  BH, XX            ; load first character in BH
             MOV  BL, YY            ; load second character in BL
             CMP  BH,  BL           ; compare the characters
             JNE  NOT_EQUAL         ;
```

```
EQUAL:        MOV  AH, 02H          ; if characters  are equal then control
              MOV  DL, 'Y'          ;  will execute this block and
              INT  21H              ; display 'Y'
              JMP  CONTINUE         ;  Jump to continue loop.

NOT_EQUAL: MOV  AH, 02H             ; if characters  are not equal then
                                    control
              MOV  DL,  'N''        ;  will execute this block and
              INT  21 H             ;  display 'N'

CONTINUE :    LOOP NEXT P           ; Get the next character
              MOV AH, 4C H          ; Exit to DOS
              INT  21 H
CODE ENDS
END MAINP
```

Discussion:

This program will be executed, at least 3 times.

3.3.2 Find the Largest and the Smallest Array Values

Let us now put together whatever we have done in the preceding sections and write down a program to find the largest and the smallest numbers from a given array. This program uses the JGE (jump greater than or equal to) instruction, because we have assumed the array values as signed. We have not used the JAE instruction, which works correctly for unsigned numbers.

; **Program 9:** Initialise the **smallest** and the **largest** variables as the first number in
; the array. They are then compared with the other array values one by one. If the
; value happens to be smaller than the assumed smallest number or larger than the
; assumed largest value, the **smallest** and the **largest** variables are changed with the
; new values respectively. Let us use register DI to point the current array value and
; LOOP instruction for looping.

```
DATA          SEGMENT
              ARRAY       DW    -1, 2000, -4000, 32767, 500,0
              LARGE       DW    ?
              SMALL       DW    ?
DATA          ENDS
END.

CODE          SEGMENT
              MOV   AX,DATA
              MOV   DS,AX                   ; Initialize DS
              MOV   DI, OFFSET ARRAY        ; DI points to the array
              MOV   AX, [DI]                ; AX contains the first element
              MOV   DX, AX                  ; initialize large in DX register
              MOV   BX, AX                  ; initialize small in BX register
              MOV   CX, 6                   ; initialize loop counter
A1:           MOV   AX, [DI]                ; get next array value
              CMP   AX, BX                  ; Is the new value smaller?
              JGE   A2                      ; If greater then (not smaller) jump to
                                            ; A2, to check larger than large in DX
              MOV   BX, AX                  ; Otherwise it is smaller so move it to
                                            ; the smallest value  (BX register)
              JMP   A3                      ; as it is small, thus no need
                                            ; to compare it with the large so jump
```

```
                                    ; to A3 to continue or terminate loop.
A2:         CMP   AX, DX            ; [DI] = large
            JLE   A3                ; if less than it implies not large so
                                    ; jump to A3
                                    ; to continue or terminate
            MOV   DX, AX            ; otherwise it is larger value, so move
                                    ; it to DX that store the large value
A3:         ADD   DI, 2             ; DI now points to next number
            LOOP  A1                ; repeat the loop until CX = 0
            MOV   LARGE, DX
            MOV   SMALL, BX         ; move the large and small in the
                                    ; memory locations
            MOV   AX, 4C00h
            INT   21h               ; halt, return to DOS
CODE        ENDS
```

Discussion:

Since the data is word type that is equal to 2 bytes and memory organisation is byte wise, to point to next array value DI is incremented by 2.

3.3.3 Character Coded Data

The input output takes place in the form of ASCII data. These ASCII characters are entered as a string of data. For example, to get two numbers from console, we may enter the numbers as:

Enter first number	1234
Enter second number	3210
The sum is	4444

As each digit is input, we would store its ASCII code in a memory byte. After the first number was input the number would be stored as follows:

The number is entered as:

31	32	33	34	hexadecimal storage
1	2	3	4	ASCII digits

Each of these numbers will be input as equivalent ASCII digits and need to be converted either to digit string to a 16-bit binary value that can be used for computation or the ASCII digits themselves can be added which can be followed by instruction that adjust the sum to binary. Let us use the conversion operation to perform these calculations here.

Another important data format is packed decimal numbers (packed BCD). A packed BCD contains two decimal digits per byte. Packed BCD format has the following advantages:

- The BCD numbers allow accurate calculations for almost any number of significant digits.
- Conversion of packed BCD numbers to ASCII (and vice versa) is relatively fast.
- An implicit decimal point may be used for keeping track of its position in a separate variable.

The instructions DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) are used for adjusting the result of an addition of subtraction operation on packed decimal numbers. However, no such instruction exists for multiplication and division. For the cases of multiplication and division the number must be unpacked. First, multiplied or divided and packed again. The instruction DAA and DAS has already been explained in unit 1.

3.3.4 Code Conversion

The conversion of data from one form to another is needed. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to binary form. Many ASCII to BCD and other conversion examples have been given earlier in unit 2.

Program 10:

```
;       This program converts an ASCII input to equivalent hex digit that it represents.
;       Thus, valid ASCII digits are 0 to 9, A to F and the program assumes that the
;       ASCII digit is read from a location in memory called ASCII. The hex result is
;       left in the AL. Since the program converts only one digit number the AL is
;       sufficient for the results. The result in AL is made FF if the character in ASCII
;       is not the proper hex digit.
; ALGORITHM
;       IF number <30h THEN error
;       ELSE
;       IF number <3Ah THEN Subtract 30h (it's a number 0-9)
;       ELSE (number is >39h)
;       IF number <41h THEN error (number in range 3Ah-40h which is not a valid
;        A-F character range)
;       ELSE
;       IF number <47h THEN Subtract 37h for letter A-F 41-46 (Please note
;       that 41h – 37h = Ah)
;       ELSE            ERROR
;
; PORTS           : None used
; PROCEDURES      : None
; REGISTERS       : Uses CS, DS, AX,
;
DATA        SEGMENT
            ASCII  DB 39h         ; Any experimental data
DATA        ENDS
CODE        SEGMENT
      ASSUME CS:CODE, DS:DATA
START:      MOV   AX, DATA        ; initialise data segment
            MOV   DS, AX          ; Register using AX
            MOV   AL, ASCII       ; Get the ASCII digits of the number
                                  ; start the conversion
            CMP   AL, 30h         ; If the ASCII digit is below 30h then it is not
            JB    ERROR           ; a proper Hex digit
            CMP   AL, 3Ah         ; compare it to 3Ah
            JB    NUMBER          ; If greater then possibly a letter between A-F
            CMP   AL, 41h         ; This step will be done if equal to or above
                                  ; 3Ah
            JB    ERROR           ; Between 3Ah and 40h is error
            CMP   AL, 46h
            JA    ERROR           ; The ASCII is out of 0-9 and A-F range
            SUB   AL, 37h         ; It's a letter in the range A-F so convert
            JMP   CONVERTED
NUMBER:     SUB   AL, 30h         ; it is a number in the range 0-9 so convert
            JMP   CONVERTED
```

```
ERROR:      MOV  AL, 0FFh     ; You can also display some message here
CONVERTED:  MOV  AX, 4C00h
            INT  21h          ; the hex result is in AL
CODE        ENDS
            END  START
```

Discussions:

The above program demonstrates a single hex digit represented by an ASCII character. The above programs can be extended to take more ASCII values and convert them into a 16-bit binary number.

☞ Check Your Progress 2

1. Write the code sequence in assembly for performing following operation:

   Z = ((A – B) / 10 * C) * * 2

   ....

   ....

2. Write an assembly code sequence for adding an array of binary numbers.

   ....

   ....

3. An assembly program is to be written for inputting two 4 digits decimal numbers from console, adding them up and putting back the results. Will you prefer packed BCD addition for such numbers? Why?

   ....

   ....

4. How can we implement nested loops, for example,

   For (i = 1 to 10, step 1)
   { for (j = 1 to, step 1)
   add 1 to AX}

   in assembly language?

   ....

   ....

3.4 PROGRAMMING FOR ARITHMETIC AND STRING OPERATIONS

Let us discuss some more advanced features of assembly language programming in this section. Some of these features give assembly an edge over the high level language programming as far as efficiency is concerned. One such instruction is for string processing. The object code generated after compiling the HLL program containing string instruction is much longer than the same program written in assembly language. Let us discuss this in more detail in the next subsection:

3.4.1 String Processing

Let us write a program for comparing two strings. Consider the following piece of code, which has been written in C to compare two strings. Let us assume that 'str1' and 'str2' are two strings, initialised by some values and 'ind' is the index for these character strings:

```
for (ind = 0; ( (ind <9) and (str1[ind] = = str2[ind]) ), ind + +)
```

The intermediate code in assembly language generated by a non-optimising compiler for the above piece may look like:

```
        MOV     IND, 00                 ; ind : = 0
L3:     CMP     IND, 08                 ; ind < 9
        JG      L1                      ; not so; skip
        LEA     AX, STR1                ; offset of str1 in AX register
        MOV     BX, IND                 ; it uses a register for indexing into
                                        ; the array
        LEA     CX, STR2                ; str2 in CX
        MOV     DL, BYTE PTR CX[BX]
        CMP     DL, BYTE PTR AX[BX]             ; str1[ind] = str2[ind]
        JNE     L1                              ; no, skip
        MOV     IND, BX
        ADD     IND, 01
L2:     JMP     L3                              ; loop back
L1:
```

What we find in the above code: a large code that could have been improved further, if the 8086 string instructions would have been used.

```
; Program 11: Matching two strings of same length stored in memory locations.
; REGISTERS : Uses CS, DS, ES, AX, DX, CX, SI, DI

DATA        SEGMENT
            PASSWORD    DB    'FAILSAFE'     ; source string
            DESTSTR     DB    'FEELSAFE'     ; destination string
            MESSAGE     DB    'String are equal $'
DATA        ENDS
CODE        SEGMENT
        ASSUME  CS:CODE, DS:DATA, ES:DATA
            MOV   AX, DATA
            MOV   DS, AX                  ; Initialise data segment register
            MOV   ES, AX                  ; Initialise extra segment register
; as destination string is considered to be in extra segment. Please note that ES is also
; initialised to the same segment as of DS.
            LEA   SI, PASSWORD            ; Load source pointer
            LEA   DI, DESTSTR             ; Load destination pointer
            MOV   CX, 08                  ; Load counter with string length
            CLD                     ; Clear direction flag so that comparison is
                                    ; done in forward direction.

            REPE  CMPSB             ; Compare the two string byte by byte
            JNE   NOTEQUAL          ; If not equal, jump to NOTEQUAL
            MOV   AH, 09            ; else display message
            MOV   DX, OFFSET MESSAGE ;
            INT   21h               ; display the message
NOTEQUAL:MOV      AX, 4C00h         ; interrupt function to halt
            INT   21h
CODE        ENDS
            END
```

Discussion:

In the above program the instruction CMPSB compares the two strings, pointed by SI in Data Segment and DI register in extra data segment. The strings are compared byte by byte and then the pointers SI and DI are incremented to next byte. Please note the last letter B in the instruction indicates a byte. If it is W, that is if instruction is CMPSW, then comparison is done word by word and SI and DI are incremented by 2, that is to next word. The REPE prefix in front of the instruction tells the 8086 to decrement the CX register by one, and continue to execute the CMPSB instruction, until the counter (CX) becomes zero. Thus, the code size is substantially reduced.

Similarly, you can write efficient programs for moving one string to another, using MOVS, and scanning a string for a character using SCAS.

3.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

Use of delay loops

A very useful application of assembly is to produce delay loops. Such loops are used for waiting for some time prior to execution of next instruction.

But how to find the time for the delay? The rate at which the instructions are executed is determined by the clock frequency. Each instruction takes a certain number of clock cycles to execute. This, multiplied by the clock frequency of the microprocessor, gives the actual time of execution of a instruction. For example, MOV instruction takes four clock cycles. This instruction when run on a microprocessor with a 4Mhz clock takes 4/4, i.e. 1 microsecond. NOP is an instruction that is used to produce the delay, without affecting the actual running of the program.

Time delay of 1 ms on a microprocessor having a clock frequency of 5 MHz would require:

$$1 \text{ clock cycle} = \frac{1}{5\text{MHz}}$$

$$= \frac{1}{5\times10^{6}} \text{ Seconds}$$

Thus, a 1-millisecond delay will require:

$$= \frac{1\times10^{-3}}{\left(\dfrac{1}{5\times10^{6}}\right)} \text{ clock cycles}$$

$$= 5000 \text{ clock cycles.}$$

The following program segment can be used to produce the delay, with the counter value correctly initialised.

```
            MOV     CX, N          ; 4 clock cycles  N will vary depending on
                                   ; the amount of delay required

DELAY:-             NOP            ; 3 cycles
                    NOP            ; 3 cycles
                    LOOP DELAY     ; 17 or 5
```

LOOP instruction takes 17 clock cycles when the condition is true and 5 clock cycles otherwise. The condition will be true, 'N' number of times and false only once, when the control comes out of the loop.

To calculate 'N':

Total clock cycles = clock cycles for MOV + N(2*NOP clock cycles + 17) – 12 (when CX = 0)

$$5000 = 4 + N(6 + 17) - 12$$
$$N = 5000/23 = 218 = 0DAh$$

Therefore, the counter, CX, should be initialized by 0DAh, in order to get the delay of 1 millisecond.

Use of array in assembly

Let us write a program to add two 5-byte numbers stored in an array. For example, two numbers in hex can be:

	20	11	01	10	FF
	FF	40	30	20	10
1	1F	51	31	31	1F
Carry					

Let us also assume that the numbers are represented as the lowest significant byte first and put in memory in two arrays. The result is stored in the third array SUM. The SUM also contains the carry out information, thus would be 1 byte longer than number arrays.

```
; Program 12: Add two five-byte numbers using arrays
; ALGORITHM:
;           Make count = LEN
;           Clear the carry flag
;           Load address of NUM1
;           REPEAT
;               Put byte from NUM1 in accumulator
;               Add byte from NUM2 to accumulator + carry
;               Store result in SUM
;               Decrement count
;               Increment to next address
;           UNTIL count = 0
;           Rotate carry into LSB of accumulator
;           Mask all but LSB of accumulator
;           Store carry result, address pointer in correct position.
; PORTS               : None used
; PROCEDURES          : None used
; REGISTERS           : Uses CS, DS, AX, CX, BX, DX

DATA    SEGMENT
        NUM1    DB    0FFh,  10h   ,01h   ,11h   ,20h
        NUM2    DB    10h,   20h,   30h,   40h   ,0FFh
        SUM     DB    6DUP(0)
DATA    ENDS
        LEN     EQU   05h      ; constant for length of the array

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV     AX, DATA       ; initialise data segment
        MOV     DS, AX         ; using AX register
        MOV     SI, 00         ; load displacement of 1st number.
                               ; SI is being used as index register
        MOV     CX, 0000       ; clear counter
        MOV     CL, LEN        ; set up count to designed length
        CLC                    ; clear carry. Ready for addition
AGAIN:  MOV     AL, NUM1[SI]   ; get a byte from NUM1
        ADC     AL, NUM2[SI]   ; add to byte from NUM2 with carry
```

```
          MOV     SUM[SI], AL  ; store in SUM array
          INC     SI
          LOOP    AGAIN        ; continue until no more bytes
          RCL     AL, 01h      ; move carry into bit 0 of AL
          AND     AL, 01h      ; mask all but the 0th bit of AL
          MOV     SUM[SI], AL  ; put carry into 6th byte
FINISH:   MOV     AX, 4C00h
          INT     21h
CODE      ENDS
          END     START
```

```
;Program 13: A good example of code conversion: Write a program to convert a
; 4-digit BCD number into its binary equivalent. The BCD number is stored as a
; word in memory location called BCD. The result is to be stored in location HEX.
; ALGORITHM:
;         Let us assume the BCD number as 4567
;         Put the BCD number into 4, 16bit registers
;         Extract the first digit (4 in this case)
;         by masking out the other three digits. Since, its place value is 1000.
;         So Multiply by 3E8h (that is 1000 in hexadecimal) to get 4000 = 0FA0h
;         Extract the second digit (5)
;         by masking out the other three digits.
;         Multiply by 64h (100)
;         Add to first digit and get 4500 = 1194h
;         Extract the third digit (6)
;         by masking out the other three digits (0060)
;         Multiply by 0Ah (10)
;         Add to first and second digit to get 4560 = 11D0h
;         Extract the last digit (7)
;         by masking out the other three digits (0007)
;         Add the first, second, and third digit to get 4567 = 11D7h
; PORTS      : None used
; REGISTERS: Uses CS, DS, AX, CX, BX, DX

THOU      EQU     3E8h         ; 1000 = 3E8h
DATA      SEGMENT
          BCD     DW   4567h
          HEX     DW   ?       ; storage reserved for result
DATA      ENDS

CODE      SEGMENT
       ASSUME CS:CODE, DS:DATA
START:    MOV     AX, DATA     ; initialise data segment
          MOV     DS, AX       ; using AX register
          MOV     AX, BCD      ; get the BCD number AX = 4567
          MOV     BX, AX       ; copy number into BX; BX = 4567
          MOV     AL, AH       ; place for upper 2 digits in AX = 4545
          MOV     BH, BL       ; place for lower 2 digits in BX = 6767
                               ; split up numbers so that we have one digit
                               ; in each register
          MOV     CL, 04       ; bit count for rotate
          ROR     AH, CL       ; digit 1 (MSB) in lower four bits of AH.
                               ; AX = 54 45
          ROR     BH, CL       ; digit 3 in lower four bits of BH.
                               ; BX = 76 67
          AND     AX, 0F0FH    ; mask upper four bits of each digit.
                               ; AX = 04 05
```

```
        AND     BX, 0F0FH       ; BX = 06 07
        MOV     CX, AX          ; copy AX into CX so that can use AX for
                                ; multiplication CX = 04 05

        ; CH contains digit 4 having place value 1000, CL contains digit 5
        ; having place value 100, BH contains digit 6 having place value 10 and
        ; BL contains digit 7 having unit place value.
        ; so obtain the number as CH × 1000 + CL × 100 + BH × 10 + BL

        MOV     AX, 0000H       ; zero AH and AL
                                ; now multiply each number by its place
                                ; value
        MOV     AL, CH          ; digit 1 to AL for multiply
        MOV     DI, THOU        ; no immediate multiplication is allowed so
                                ; move thousand to DI
        MUL     DI              ; digit 1 (4)*1000
                                ; result in DX and AX. Because BCD digit
                                ; will not be greater than 9999, the result will
                                ; be in AX only. AX = 4000
        MOV     DH, 00H         ; zero DH
        MOV     DL, BL          ; move BL to DL, so DL = 7
        ADD     DX, AX          ; add AX; so DX = 4007
        MOV     AX, 0064h       ; load value for 100 into AL
        MUL     CL              ; multiply by digit 2 from CL
        ADD     DX, AX          ; add to total in DX. DX now contains
                                ; (7 + 4000 + 500)
        MOV     AX, 000Ah       ; load value of 10 into AL
        MUL     BH              ; multiply by digit 3 in BH
        ADD     DX, AX          ; add to total in DX; DX contains
                                ;  (7 + 4000 + 500 +60)
        MOV     HEX, DX         ; put result in HEX for return
        MOV     AX, 4C00h
        INT     21h
CODE    ENDS
        END     START
```

Check Your Progress 3

1. Why should we perform string processing in assembly language in 8086 and not in high-level language?

....

....

....

2. What is the function of direction flag?

....

....

....

3. What is the function of NOP statement?

....

....

....

3.5 SUMMARY

In this unit, we have covered some basic aspects of assembly language programming. We started with some elementary arithmetic problems, code conversion problems, various types of loops and graduated on to do string processing and slightly complex arithmetic. As part of good programming practice, we also noted some points that should be kept in mind while coding. Some of them are:

- An algorithm should always precede your program. It is a good programming practice. This not only increases the readability of the program, but also makes your program less prone to logical errors.
- Use comments liberally. You will appreciate them later.
- Study the instructions, assembler directives and addressing modes carefully, before starting to code your program. You can even use a debugger to get a clear understanding of the instructions and addressing modes.
- Some instructions are very specific to the type of operand they are being used with, example signed numbers and unsigned numbers, byte operands and word operands, so be careful !!
- Certain instructions except some registers to be initialised by some values before being executed, example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialised by the segment and the offset of the string instructions, and ES:DI to be with the destination strings, INT 21h expects AH register to contain the function number of the operation to be carried out, and depending on them some of the additional registers also to be initialised. So study them carefully and do the needful. In case you miss out on something, in most of the cases, you will not get an error message, instead the 8086 will proceed to execute the instruction, with whatever junk is lying in those registers.

In spite of all these complications, assembly languages is still an indispensable part of programming, as it gives you an access to most of the hardware features of the machine, which might not be possible with high level language. Secondly, as we have also seen some kind of applications can be written and efficiently executed in assembly language. We justified this with string processing instructions; you will appreciate it more when you actually start doing the assembly language programming. You can now perform some simple exercises from the further readings.

In the next block, we take up more advanced assembly language programming, which also includes accessing interrupts of the machine.

3.6 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. False 2. False 3. True 4. True 5. True 6. False

Check Your Progress 2

1.
```
MOV   AX, A        ; bring A in AX
SUB   AX, B        ; subtract B
MOV   DX, 0000h    ; move 0 to DX as it will be used for word division
MOV   BX, 10       ; move dividend to BX
IDIV  BX           ; divide
IMUL  C            ; ( (A-B) / 10 * C) in AX
IMUL  AX           ; square AX to get (A-B/10 * C) * * 2
```

2. Assuming that each array element is a word variable.

```
        MOV     CX, COUNT     ; put the number of elements of the array in
                              ; CX register
        MOV     AX, 0000h     ; zero SI and AX
        MOV     SI, AX
; add the elements of array in AX again and again
AGAIN:  ADD AX, ARRAY[SI]     ; another way of handling array
        ADD     SI, 2         ; select the next element of the array
        LOOP    AGAIN         ; add all the elements of the array. It will
                                terminate when CX becomes zero.
        MOV     TOTAL, AX     ; store the results in TOTAL.
```

3. Yes, because the conversion efforts are less.
4. We may use two nested loop instructions in assembly also. However, as both the loop instructions use CX, therefore every time before we are entering inner loop we must push CX of outer loop in the stack and reinitialize CX to the inner loop requirements.

Check Your Progress 3

1. The object code generated on compiling high level languages for string processing commands is, in general, found to be long and contains several redundant instructions. However, we can perform string processing very efficiently in 8086 assembly language.

2. Direction flag if clear will cause REPE statement to perform in forward direction. That is, in the given example the strings will be compared from first element to last.

3. It produces a delay of a desired clock time in the execution. This instruction is useful while development of program. A collection of these instructions can be used to fill up some space in the code segment, which can be changed with new code lines without disturbing the position of existing code. This is particularly used when a label is specified.