UNIT 15 ASSEMBLY LANGUAGE PROGRAMMING

Structure

15.0 INTRODUCTION

After discussing about the directives, program developmental tools and Input / Output in assembly language programming, let us discuss more about assembly language programs. In this unit, we will first discuss the simple assembly programs, which performs simple tasks such as data transfer, arithmetic operations, and shift operations. A key example would be to find the larger of two numbers. Thereafter, you will go through more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. This unit also discusses more complex arithmetic and string operations and modular programming. You must refer to further readings for more details on these programming concepts.

15.1 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;
- write simple assembly programs for implementing arrays;
- explain the use of stack in parameter passing; and
- use modular programming in assembly language

15.2 SIMPLE ASSEMBLY PROGRAMS

In this unit, first simple assembly programs are discussed and later more complex programs are written. In this section several simple assembly programs are explained.

15.2.1 Data Transfer

Data transfer in one of the most fundamental operation. 8086 microprocessor has two basic data transfer instructions,viz. MOV and XCHG. These instructions are explained with the help of simple example.

Program 1: Write a program using 8086 assembly language to exchange a data word stored in a memory location with the value stored in BX register and interchanges the value of AH and AL registers.

Directives **Statement**	**Discussion**

`DATA SEGMENT`
`     VAL DW4321H`
`DATA  ENDS`
	The data segment stores a variable VAL, which stores a data word.

`CODE SEGMENT`
`     ASSUME CS:CODE,DS:DATA`
`MAINP:MOV AX, DATA`
`       MOV DS, AX`
`MOVAX, 8765H`
` XCHG AH,AL`
`        MOVBX, 8765H`
` XCHG BX, VAL`
`        MOVAX, 4C00H`
` INT21H`
` CODE ENDS`
` END MAINP`
	Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register.
	Move 8765H to AX register.
	Result: AX=6587H
	Move 8765H to BX register.
	Result: BX=4321H and VAL=8765H
	Return to operating system using Interrupt 21h with function 4C.

Program 2: Write an 8086-assembly program that interchanges the values of two Memory locations.

Directives **Statement**	**Discussion**

`DATA SEGMENT`
`VAL1  DB  25h`
`VAL2  DB  65h`
`DATA ENDS`
	Define two variables VAL1 and VAL2 consisting of byte values.

`CODE SEGMENT`
`        ASSUMECS:CODE,DS:DATA`
`        MOV AX, DATA`
`        MOV DS, AX`
`        MOV AL, VAL1`
`        XCHG VAL2,AL`
`        MOV VAL1,AL`
`        MOV AX, 4C00H`
`INT 21h`
`CODE ENDS`
`END`
	Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register.
	Load the variable VAL1into AL
	Exchange AL with variable VAL2
	Now, move the AL to variable VAL1
	Return to operating system using Interrupt 21h with function 4C.

In Program 2, why have we not used XCHG VAL1, VAL2 instruction directly? To answer this question,you should look into the constraints for the MOV instructions, which are given below:

- 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, VAL1, copies the VAL1 that is 25h to the AL register:
- The instruction, XCHG   AL, VAL2exchanges the value of AL register (25h) with VAL2(65h). Thus, after the execution of this instruction AL will contain 65h and VAL2 will contain 25h. VAL1 at this time will also contain 25h only.
- Finally, the instruction MOV VALUE1, AL will put the value 65h into VAL1.

15.2.2 Simple Arithmetic Application

Let us discuss an example that uses simple arithmetic operations.

Program 3: Find the average of two-bytevalues stored in the memory locations named as FIRST and SECOND. The result of the operation can be stored in a third memory location named MEAN.

Discussion: The program should have two memory variables stored in memory locations FIRST and SECOND and a third location for storing the mean value.An add instruction cannot add two memory locations directly, so you are required to move a single value in AL first and then add the second value to it.In addition, on adding the two-byte values, there is a possibility of a carry bit. Assuming that problem is addressing two unsigned binary numbers, the problem is how to put the carry bit into the AH register such that the AX(AH:AL) reflects the added byte values. 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 addition, 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 mean value, you can divide the sum given in AX by 2. 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.

Directives **Statement**	**Discussion**

`DATA SEGMENT`
`        FIRST DB 90h`
`        SECOND DB 78h`
`        MEAN   DB ?`
`DATA    ENDS`
	Three variables

`CODE    SEGMENT`
`     ASSUME CS:CODE, DS:DATA`	

`START:   MOV   AX, DATA`	Initialize data segment
`          MOV   DS, AX`	
`       MOV      AL, FIRST`	Get FIRST number in AL
`          ADD   AL, SECOND`	Add SECOND number to AL
`       MOV      AH, 00h`	Clear AH register
`       ADC      AH, 00h`	Put carry in AH
`       MOV      BL, 02h`	Load divisor (2) in BL register
`       DIV      BL`	Divide AX by BL. Quotient in AL and remainder in AH; and copy the result to memory and return to operating system.
`          MOV  MEAN,AL`	
`          MOV  AX, 4C00H`	
`          INT 21H`	
`CODE ENDS`	
`END START`	

15.2.3 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: Write a program in 8086 assembly language to convert a two-digit ASCII code to an equivalent packed BCD number. You may assume that these two ASCII digits are stored in AL and BL registers.
Discussion: to its BCD equivalent. An ASCII digit is of 8 bit length. The lower four bits of an ASCII digit represents the equivalent BCD value of the ASCII digit. For example, ASCII digit '3' is 00110011_2, so if we replace the upper four bits by 0s, you will get 00000011_2, which is equal to BCD digit 03. The number so 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 39 is 00000011 00001001, that is, 03 09. The packed BCD will be 0011 1001, that is 39. Thus, the algorithm to convert two ASCII digits to packed BCD can be stated as:
Input: The two-digit ASCII number
Output: Two-digit packed BCD number
Process:

Convert the ASCII digits to unpacked BCD. An example is given in the table below:

Digit	ASCII	Unpacked BCD
3	00110011	00000011
9	00111001	00001001

Move most significant BCD digit to upper four bit positions in byte by using rotate instruction as shown below:

0000 0011	This is a unpacked BCD of digit 3.
00110000	Use Rotate Instructions to get this

Pack the bits of rotated BCD digit with the least significant BCD digit bits, as shown below to create a packed two-digit BCD number in a byte.

Rotated value of digit 3	0011 0000
The unpacked digit 9	0000 1001
Use OR operator	0011 1001

Display or store the results in a desired location or register.
The assembly language program for the above can be written in the following manner.

Directives Statement	Discussion
`DATA SEGMENT`	
`HighDigit DB '3'`	The data segment stores the ASCII
`LowDigit  DB '9'`	value of the two digits, the assumed
`PackedBCD DB ?`	digits are '3' and '9'.
`DATA  ENDS`	
`CODE SEGMENT`	Initialize data segment register
`ASSUME CS:CODE, DS: DATA`	
`START: MOV AX, DATA`	Move the Higher digit (3) in BL
`MOV DS, AX`	Move the lower digit (9) to AL
`MOVBL,  HighDigit`	Mask upper 4 bits of BL
`MOVAL,  LowDigit`	Mask upper 4 bits of AL
`ANDBL,  0Fh`	Move the rotate count to CL
`ANDAL,  0Fh`	Rotate BL register using CL
`MOVCL,  04h`	OR to get the packed BCD in AL
`ROLBL,  CL`	Store the result in Packed BCD
`ORAL,   BL`	Return to Operating system
`MOV PackedBCD, AL`	
`MOV AX 4C00H`	
`INT 21H`	
`CODE ENDS`	
`END START`	

Discussion on Program 4:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore you need to use the rotate instruction4 times. You can choose any of the two rotate instructions, ROL and RCL. In this example, 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. Rest the entire program proceeds as per the algorithm.

Program 5: Write a program using 8086 assembly language that adds two binary numbers (assume the number are of byte type) stored in the consecutive memory locations. The result of the addition and carry, if any are also stored in the memory locations.

Directives Statement	Discussion
`DATA SEGMENT`	
`NUM1DB 25h`	First number contains 25h
`NUM2 DB 80h`	Second number contains 80h
`RES   DB ?`	Will store sum of the two numbers
`CARY DB  ?`	Will store carry bit, if any
`DATA  ENDS`	
`CODE  SEGMENT`	
`ASSUME CS:CODE, DS:DATA`	
`START:MOV AX, DATA`	Initialize data segment register
`MOV DS, AX          MOV`	
`AL, NUM1`	Load the first number in AL, addthe
`ADD AL, NUM2`	second number in AL and store the
`MOV RES,AL          RCL`	result into RES
`AL, 01          AND AL,`	Rotate AL with carry, to bring carry bit
`00000001B       MOV CARY, AL`	to the least significant bit, AND it with
`MOV  AH, 4CH`	00000001_2 to mask out all bits except
`INT 21H`	the least significant bit. Store the carry
`CODEENDS`	bit to CARY and return to the operating
`ENDSTART`	system.

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 AL.

15.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP destination, source. However, this instruction only sets the flags on comparing two operands both of which should be either of 8 bits or 16 bits. Compare instruction just subtracts the value of the source operand from the destination operand without storing the result, but setting the flag during the process. In general,the following three comparisons may be able to address most of the comparison operations:

Instruction: CMP destination, source

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

The following examples show how the flags are set when the numbers are compared.
Example 1:

```
MOV BL, 02h              ;Move 02h to BL
CMP BL, 10h              ; Compare BL with 10h. Sets carry flag = 1
```

As the value of BL is less than 10h, the carry flag would be set as borrow would be needed to subtract 10h from BL.

Example 2:

```
MOV AX,F0F0h      ;  Same value is moved to AX
MOV DX, F0F0h     ;  and BX
CMP AX,DX         ;On comparison, it sets Zero flag = 1
```

The zero flag is set as both the operands are same.

Example 3:

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

The destination register (BX) contains a value greater than the source (0), so both the zero and the carry flags are cleared.

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 15.2.2 you can change instruction DIV BL with a shift instruction. ☐

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. ☐

15.3 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.

15.3.1 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 given 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 assembly code given above is not efficient, but suggests that there are many ways through which a conditional jump can be implemented. You should select the most optimum way based on your program requirements.

Example 6:

```
      CMP   DX, 00           ; checks if DX is zero.
      JE    Label1           ; if yes, jump to Label1 i.e., if ZF=1
      …
Label1:other instructions    ; 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    Label1              ; carry flag = 1 then jump to Lab1
        …
Label1:   other instructions      ; control comes here if condition is satisfied
```

LOOPING

Program 6: Write a program using 8086 assembly language that computes the new prices from series of prices data stored in the memory. You may assume a constant inflation factor that is added to each old price value. Also assume that all the prices are given in the BCD form.
Discussion:
Input: A list of prices stored in the memory and a constant inflation factor
Output: The new prices
Process:

```
Repeat the following steps
    Read a price (in BCD) from the input array
    Add inflation factor
    Adjust result to correct BCD
    Put result back in the same array
Until all prices are converted to new price
```

Directives Statement	**Discussion**

`ARRAYSSEGMENT`
` PRICES DB 25h, 35h, 45h, 65h, 75h`
`ARRAYS ENDS`
	The data segment is named ARRAYS and consist of a list of 5 PRICES.

`CODESEGMENT`
`ASSUME CS:CODE, DS:ARRAYS`
`START:MOVAX, ARRAYS`
`MOVDS, AX`
`LEA   BX, PRICES`
`MOV   CX, 0005h`
 	
`DO_NEXT:MOVAL,[BX]`
`ADD   AL, 0Ah`
 	
`DAA`
`MOV[BX], AL`
 	
`INC   BX`
`DEC   CX`
 	
`JNZ   DO_NEXT`
`MOV   AH,4CH`
`INT 21H`
`CODE  ENDS`
`END START`
	Initialize data segment. Please note the use of name ARRAYS
	
	Move address of variable PRICES to BX register and move 5 to CX as PRICES has 5 values.
	Load the first value from array to AL and add constant 0Ah, which is assumed as inflation factor
	Since input is BCD, DAA adjusts the addition, and results are stored in the PRICES again.
	BX is incremented to point to next value and counter CX is decremented
	If the decrement operation does not result in zero, then jump is taken to DO_NEXT label, else all the values of PRICES has been processed, so program exits to Operating system.

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 named `ARRAYS`. [BX] is an

indirection through BX and points to the value stored at that element of array named 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. The LOOP instruction is demonstrated with the help of following program:

Program 7: Write a program using 8086 assembly language that prints the alphabets A-Z on the screen. This program is written using LOOP statement.

Directives Statement	**Discussion**
`CODE SEGMENT`	
`ASSUMECS:CODE`	
`MAINP:MOVCX,1AH`	1AH=26 (number of alphabets to be displayed)
`MOV DL, 41H`	41H is hexadecimal equivalent of ASCII ‘A’.
`NEXTC: MOV AH, 02H` `INT 21H`	Function 02H of Interrupt 21h is used to display the character stored in DL.
`INC DL`	Increment DL to next alphabet value
`LOOP NEXTC`	Loop instruction will decrement CX by 1 and check if CX=0, if not it loops to label NEXTC to print remining characters.
`MOV AX, 4C00H` `INT21H`	Once all the characters are printed, the program returns to the operating system.
`CODE ENDS`	
`END MAINP`	

Let us now discuss slightly more complex program in the next section.

15.3.2 Find the Largest and the Smallest Array Values

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

Program 8: Write a program using 8086 assembly language to find the largest and smallest numbers in an array.
Discussion: Initialize the **SMALL** and the **LARGE** variables as the first number in the array. They are then compared with the other array values one by one. If thevalue happens to be smaller than the assumed smallest number or larger than theassumed largest value, the **SMALL** and the **LARGE** variables are changed by this new value respectively. Let us use register DI to point the current array value andLOOP instruction for looping.

Directives Statement	**Discussion**
`DATA SEGMENT` `ARRAY DW -1, 2000, -4000,32767, 500,0` `LARGE DW ?` `SMALL DW ?`	Data segment includes a total of 6 signed values. You need to find the largest and the smallest among these values. The largest and smallest values will be stored in variables LARGE and SMALL

`DATA  ENDS`	respectively.
`CODESEGMENT`	Initialize the data segment register using AX
`ASSUME CS:CODE, DS:DATA`	
`START:MOVAX, DATA`	
`       MOV DS,AX`	Move offset of ARRAY of data segment to DI and move the array element pointed by DI to AX register. DX and BX registers are used to store the largest and smallest respectively. The first value of array is moved in both these registers. Since the size of array is 6, so move 6 to CX register.
`MOV   DI, OFFSET ARRAY`	
`MOV   AX, [DI]`	
`MOV   DX, AX`	
`MOV   BX, AX`	
`MOV   CX, 6`	
`A1:   MOV   AX, [DI]`	The element pointed to by DI is moved to AX, which is comparted with BX. In case, AX is greater than or equal to BX, which means AX is not the smallest. The program jumps to label A2.
`      CMP   AX, BX`	
`      JGE   A2`	
`MOV   BX, AX`	This instruction will be executed, if the condition as above is false, which means AX is smallest, in this case the value of AX will be moved to BX, which contains new smallest value now. The program will then jump to label A3.
`       JMP  A3`	
`A2:   CMP   AX, DX`	This statement will be executed, if junp is taken to label A2. The value in AX will be compared to largest in DX, if it is less or equal to the program will jump to label A3, otherwise the AX is largest value, so it will be moved to DX register.
`      JLE   A3`	
`MOV   DX, AX`	
`A3:ADD      DI, 2`	Next, DI is incremented by 2, so that it points to the next data word in the memory. The LOOP instruction decrements CX and checks if it is zero, if not then the steps from label A1 are repeated. Once the CX becomes zero, the DX is moved to LARGE and BX is moved to SMALL, as they contain the largest and smallest values respectively. Finally, program returns to the operating system.
`LOOP  A1`	
`MOV   LARGE, DX`	
`MOV   SMALL, BX`	
`     MOV   AX, 4C00h`	
`INT   21h`	
`CODE  ENDS`	
`END START`	

Point to Note: 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.

15.3.3 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, you may enter the numbers as:

Enter first number	1234
Enter second number	3210

As each digit is input, you 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 stored as:

```
31    32    33    34    hexadecimal values stored in memory
 1     2     3     4    equivalent 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.

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.

Let us discuss the process of conversion of ASCII digits to equivalent binary number, which can be used for computation.

15.3.4 Code Conversion

The conversion of data from one form to another is required in programs, which involve input of numbers. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to its equivalent binary form. An example showing ASCII to BCD conversion has already been explained as part of this unit.

Program 9: Write a program in 8086 assembly language to convert an ASCII input to equivalent hexadecimal value that it represents. The valid ASCII digits for this conversion are the digits 0 to 9 and alphabets A to F. The program assumes that the ASCII digit is read from a location in memory called ASCII. The hexadecimal or binary result isleft 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 ASCIIis not the proper hex digit.

```
Algorithm
Input: An ASCII digit
Output: The hexadecimal equivalent of number, if it is valid
Process:
    IfASCII digitis in the range 30h to 39h it represents hex digit 0 to 9.
    If ASCII digit is in the range 41h to 46h it represents hex digit A to F.
    For any other value of ASCII, it does not represent a hex digit.
```

Directives Statement	**Discussion**

`DATASEGMENT`
`      ASCIIDB 39h`
`DATA ENDS`
	ASCII variable contains an ASCII digit.

`CODESEGMENT`
`ASSUME CS:CODE, DS:DATA`	

`START:      MOV   AX, DATA`
`             MOV   DS, AX`
	Initialize data segment using AX

`             MOV   AL, ASCII`
`             CMP   AL, 30h`
`JB    ERROR`
	Get the ASCII digits in AL register and compare it with 30h. If it is less than 30h, it is not a valid digit. So go to label ERROR

`             CMP   AL, 3Ah`
`             JAE   ATOF`
`SUB   AL, 30h`
`             JMP   CONVERTED`
	These instructions will be executed only if the ASCII digit is 30h or more. In case the ASCII digit is equal to or above 3Ah, you jump to ATOF, otherwise the ASCII is in range 30h to 39h. So convert it and move to label CONVERTED.

`ATOF:       CMP   AL, 41h`
`             JB    ERROR`
`CMP   AL, 46h`
`             JA    ERROR`
`SUB   AL, 37h          JMP`
`      CONVERTED`
	You will be here if ASCII is greater than or equal to 3Ah. Check if it is below 41h, if yes go to label ERROR. Next, check if the ASCII digit is above 46h, if yes go to label ERROR. Otherwise convert the ASCII to hex digit equivalent by subtracting 37h. Next, jump to CONVERTED.

`ERROR:      MOV   AL, 0FFh`	Error is detected when AL has FF, which is moved to it.

`CONVERTED:  MOV   AX, 4C00h`
`             INT   21h`
`CODEENDS`
`ENDSTART`
	Otherwise, AL contains the converted hex digit so the program returns to operating system.

Discussions:

The above program demonstrates a conversion of a single ASCII character to equivalent hexadecimal digit represented by that 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 BCD addition for such numbers? Why?

   ………………………………………………………………………………………

4. How can you implement nested loops, such as given below?

```
            for (i = 1 to 10, step 1)
                  { for (j = 1 to 5, step 1)
                        add 1 to AX}
```

   in assembly language?

   ………………………………………………………………………………………

   ………………………………………………………………………………………

15.4 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 (HLL) 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. The following section discuss a program of string processing:

15.4.1 String Processing

Let us write a program for comparing two strings.

Program 10: Write a Program using 8086 assembly language to match two strings of same length stored in two separate memory locations.

Directives **Statement**	**Discussion**

`DATASEGMENT`
`PASSWORDDB'FAILSAFE'`
`DESTSTR  DB'FEELSAFE'`
`MESSAGEDB'String are equal$'`
`DATA ENDS`
	
	The source string
	The destination string
	The message to be displayed if strings are the same

`CODESEGMENT`
`ASSUMECS:CODE,DS:DATA,ES:DATA`
`MOV AX, DATA`
`MOV DS, AX`
`MOV ES, AX`
	The string matching requires two segments for data, viz. data segment for source string and extra data segment for destination string. Thus, DS and ES are initialized using AX.

`LEASI, PASSWORD`
`LEA DI, DESTSTR`
`MOV CX, 08`
`CLD`
	The offset of PASSWORD and DESTSTR are stored in SI and DI respectively. As both the strings are of 8 bytes, CX=8. The direction flag, which determines the direction of string processing is cleared.

`REPECMPSB`
`JNE  NTEQ`
`MOV  AH, 09`
`MOV  DX, OFFSET MESSAGE`
`INT  21h`
`NTEQ: MOV  AH 4CH`
`INT 21H`
`CODE  ENDS`
`END`
	Repeat the compare string operation byte by byte and move to label NTEQ, if they are not equal at any character. The branch will not be taken, if the strings are equal. In that case the MESSAGE is printed on the screen. Finally, the program returns to the operating system.

Discussion:

In the program given above, 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 the next word respectively. 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, when string instructions are used.

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

15.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

Use of array in assembly

An array is referenced using a base array value and an index. To facilitate addressing in arrays, 8086 has provided two index registers for mathematical computations, viz. BX and BP. In addition, two index registers are also provided for processing, viz. SI and DI. You can also use any general-purpose register for indexing.

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

Carry in		0	0	0	1	0
		20	11	01	10	FF
		FF	40	30	20	10
	1	1F	51	31	31	1F
Carry out						

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 11: Write a program in 8086 assembly language to add two five-byte numbers using arrays.

Algorithm:
Input: two arrays of 5 bytes each.
Output: an array of sum of size 6 bytes
Process:

```
    Repeat the following steps till all the elements of array (5) are added
        Load the byte of first array in AL
        Add the corresponding byte of second array in AL with carry
        Store the result in a memory array
        Increment to next bytes
    Rotate carry into LSB of accumulator
Mask all but LSB of accumulator
Store carry result in memory
```

Directives **Statement**	**Discussion**

`DATASEGMENT`
`NUM1DB0FFh,10h,01h,11h,20h`
`NUM2DB10h,20h,30h,40h,0FFh`
`SUMDB   6DUP(0)`
`DATAENDS`
	Two arrays of size 5 each, SUM will store the addition and overall carry out bit

`CODESEGMENT`
`   ASSUME CS:CODE, DS:DATA`
`START:  MOVAX, DATA`
`        MOVDS, AX`
`        MOVSI, 00`
`        MOVCX, 05h`
`CLC`
	Initialize segment register
	
	SI register is being used as index register, in the array, therefore, is initialized to 0. CX is initialized to the size of arrays and CLC clears the carry bit

`AGAIN:  MOV AL, NUM1[SI]`
`        ADC AL, NUM2[SI]`
`        MOV SUM[SI], AL`
`        INC SI`
`        LOOPAGAIN`
	First an element of array NUM1 is moved to AL and then the corresponding element of NUM2 and previous carry is added in AL. The result is stored in the memory and SI is incremented to the next element of the arrays. These operations are repeated for all the elements of arrays.

`        RCL AL, 01h`
`        AND AL, 01h`
`        MOVSUM[SI], AL`
`        MOV AX, 4C00h`
`        INT 21h`
`CODE ENDS`
`ENDSTART`
	Next, the final carry is rotated to LSB of AL and the higher bits are masked out. This final carry is also stored in the 6th element of SUM. Finally, program returns to operating system.

A good example of code conversion involving arithmetic is discussed next.
Program 12: Write a program using 8086 assembly programming language to convert a 4-digit BCD number into its binary equivalent. The BCD number can be stored as a word in memory location called BCD. The result is to be stored in location HEX.
The procedure to perform this number is explained with the help of following example.
Assume a 4-digit BCD number, say 4567, which is stored in a word in memory. To convert this number, you should extract each BCD digit separately and perform the following operation:
The binary number = 4 × (1000 or 3E8h) + 5 × (100 or 64h) + 6 × (10 or Ah) + 7

Directives Statement	**Operation**

`THOUEQU  3E8h`
`DATASEGMENT`
`BCDDW4567h`
`HEXDW?`
`DATAENDS`
	Constant THOU is equal to 1000, i.e. 3E8h

`CODESEGMENT`
`   ASSUME CS:CODE, DS:DATA`
`START:  MOVAX, DATA`
`        MOVDS, AX`
`        MOVAX, BCD`
`        MOVBX, AX`
`        MOVAL, AH`
`        MOVBH, BL`
`        MOVCL, 04`
`        RORAH, CL`
`        RORBH, CL`
`        ANDAX, 0F0FH`
`        ANDBX, 0F0FH`
`        MOV CX, AX`
	
	Initialize data segment Register.
	
	AX = 4567
	BX = AX = 4567
	AL = AH = 45
	BH = BL = 67
	CL = 4, as 4-bit rotation will be used
	AH = 54 due to 4-bit rotation
	BH= 76 due to 4-bit rotation
	AX=5445 AND 0F0Fh = 0405
	BX= 7667 AND 0F0Fh = 0607
	AX is moved to CX so that AX can be used for

	other operations. CX = AX = 0405
MOV AX, 0000H	AX=0
MOVAL, CH MOVDI,	AL=CH=04
THOU MULDI	DI=1000
MOVDH, 00H	AX= 04×1000 = 0FA0h
MOVDL, BL	DH=0
ADD DX, AX	DL=BL=07, thus DX= 0007
MOVAX, 0064h	DX=0FA0h+0007h=0FA7h
MULCL	AX=0064h
ADDDX, AX MOVAX,	CL=05; AX=5×100 = 01F4h
000Ah	DX= 0FA7h+01F4h= 119Bh
MULBH	AX=000A
ADDDX, AX	BH=6; AX=6×10=003Ch
MOVHEX, DX	DX=119Bh+003Ch=11D7h
MOVAX, 4C00h	Move this value to location HEX
INT21h	Return to operating system
CODEENDS	
ENDSTART	

☞ 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 REPE statement?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………

15.5 MODULAR PROGRAMMING

Modular programming refers to the practice of writing a program as a series of independently assembled source files. Each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent whole.

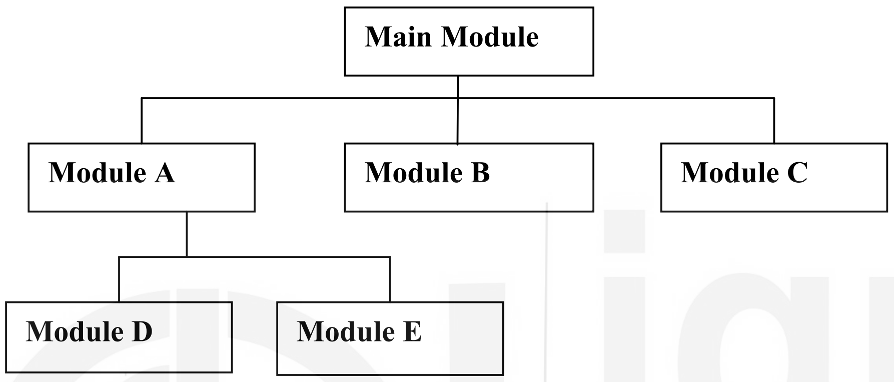
There are several reasons a programmer might choose to modularize a program.

1. Modular programming permits breaking a large program into a number of smaller modules each of more manageable size.
2. Modular programming makes it possible to link source code written in two separate languages. A hybrid program written partly in assembly language and

partly in higher level language necessarily involves at least one module for each language.

3. Modular programming allows for the creation, maintenance and reuse of a library of commonly used modules.
4. Modules are easy to comprehend.
5. Different modules can be assigned to different programs.
6. Debugging and testing can be done in a more orderly fashion.
7. Document action can be easily understood.
8. Modifications may be localized to a module.

A modular program can be represented using hierarchical diagram:



You can divide a program into subroutines or procedures. You need to CALL the procedures whenever needed. A subroutine call instruction transfers the control to subroutine instructions and the return statement brings the control back to the calling program.

15.5.1 The Stack

A procedure call is supported by a stack. Stack is a Last In First Out (LIFO) data structure. A stack in assembly language can be used for storing the return addressof procedures, for parameter passing and for storing the value returned by the procedure.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the base of stack segment and SP register stores the position of the top of the stack. A value is pushed in to top of the stack or taken out (poped) from the top of the stack. The stack of 8086 is a word stack. In order to use stack, first the stack segment register is initialized, as given below:

`STACK_SEG SEGMENT STACK`
`    DW 100 DUP (0)`
`    TOS LABEL WORD`
`STACK_SEG ENDS`
	Declaration of the stack segment.
	Assign 100-word locations to stack
	TOS is a word label to the top of stack.
	End of stack segment

`CODE SEGMENT`
`ASSUME CS:CODE,SS:STACK_SEG`
`     MOV  AX, STACK_SEG`
`     MOV  SS,AX`
	Just like a data segment, the SS register is initialized to the base of stack segment.

`     LEA  SP,TOP`
`     …`
`CODE ENDS`
`     END`
	The SP register is loaded with the maximum offset of the stack, represented

The directive STACK_SEG SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns an actual size of the stack to 100 words. All locations of this stack are initialized to zero. The label TOS defines the initial top ofthis empty stack. Please note that the stack in 8086 is a WORD stack. The stack grows from a higher offset to a lower offset.The top position of stack uses an indirect addressing mechanism through a special register called the stack pointer (SP). SP initially is made to points to a label TOS. SP is automatically decremented,whenan itemis put on the stack (called PUSH operation) and incremented as an item is retrieved from the stack (called POP operation). SP points to the address of the last element pushed on to the stack. The following table explains the PUSH and POP instructions of 8086 microprocessor

Name	Mnemonics	Description
Push a word value SRC onto the stack	PUSH SRC	SP←SP – 2 ; (decrement SP to the next word) Put the word SRC into word pointed to by present value of SP and SP+1 ;
Pop a word value from the stack in DST	POP DST	Retrieve the word stored on stack top to DST ; SP← SP + 2 ;

15.5.2 FAR and NEAR Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, i.e., they require extra code to join them together in such a way that they can communicate with each other. This extra code is sometimes referred to as linkage overhead.

A procedure call involves:

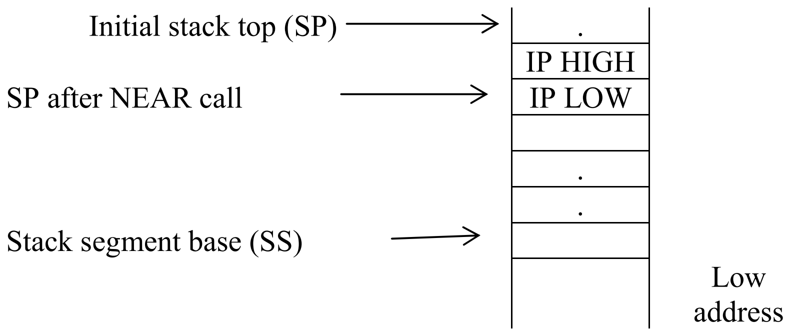
1. Unlike branch instructions, a procedure call must save the address of the next instruction to be executed of the calling program, so that after completion of execution of the procedure, the procedure can return the control to the calling program.
2. The registers used by the procedures need to be saved before their contents are changed by the procedure. These saved register values are used to restore the contents of the registers when the execution returns to the calling program.
3. A procedure must have means of communicating or sharing data with the procedures that call it, that is parameter passing.

Calls, Returns and Procedures definitions in 8086

The 8086 microprocessor supports CALL and RET instructions for procedure call.

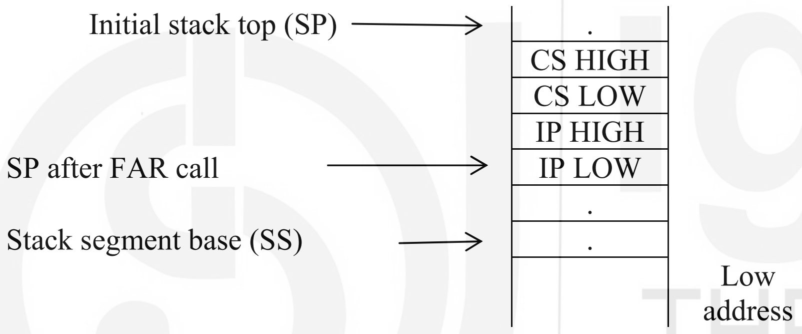
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the first instruction of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure calls namely FAR and NEAR calls.

The NEAR procedure call is also known as Intra-segment call as the called procedure is in the same segment from which call has been made. Thus, only IP is stored as the return address on the top of the stack. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base register. So, stack becomes full on an offset 0000h. Also, for push operation you decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organized memory.

FAR procedure call, also known as intersegment call, is a call made to separate code segment. Thus, the control will be transferred outside the current segment. Therefore, both CS and IP need to be stored as the return address. These values on the stack after the calls look like:



When the 8086 executes the FAR call, it first stores the contents of the code segment register followed by the contents of IP on to the stack. A RET from the NEAR procedure. Pops the two bytes into IP. The RET from the FAR procedure pops four bytes from the stack.

Procedure is defined within the source code by placing a directive of the form:

<Procedure name> PROC <Attribute>

A procedure is terminated using:

<Procedure name> ENDP

The <procedure name> is the identifier used for calling the procedure and the <attribute> is either NEAR or FAR. A procedure can be defined in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the segment containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the <attribute> code NEAR should be used as the procedure and code are in the same segment. For the latter two cases the <attribute> must be FAR.

15.5.3 Parameter Passing

Parameter passing is a very important concept in assembly language. It makes the assembly procedures more general. Parameter can be passed to and from to the main procedures. The parameters can be passed in the following ways to a procedure:

1. Parameters passing through registers
2. Parameters passing through dedicated memory location accessed by name
3. Parameters passing through pointers passed in registers
4. Parameters passing using stack.

However, in this Unit we will discuss parameter passing using a stack with the help of an example.

Passing Parameters Through Stack

The best technique for parameter passing is through stack. It is also a standard technique for passing parameters when the assembly language is interfaced with any high-level language. Parameters are pushed on the stack and are referenced using BP register in the called procedure. One important issue for parameter passing through stack is to keep track of the stack overflow and underflow to keep a check on errors.
Program 13: Write a program using 8086 assembly language, which has a procedure to convert a two-digit packed BCD number to an equivalent binary number. Use stack for parameter passing.

Discussion: The logic of conversion of packed BCD number to binary can be done in two simple steps. First convert the packed BCD digits to unpacked BCD digits and then multiply each digit with place value.

Directives Statement	Discussion

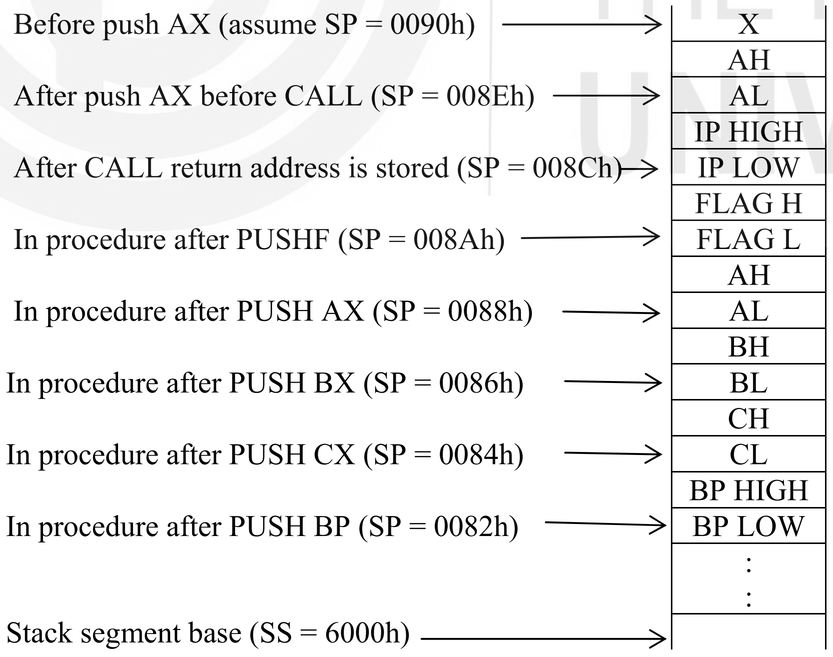
`DATA_SEGSEGMENT`
`BCDDB25h`
`BINDB?`
`DATA_SEGENDS`
	Storage for BCD value
	Storage for binary value

`STACK_SEGSEGMENT STACK`
`DW100DUP(0)`
`TOP_STACKLABELWORD`
`STACK_SEGENDS`
	Stack of 100 words
	Label for stack top

`CODE_SEGSEGMENT`
`ASSUME CS:CODE_SEG,`
`DS:DATA_SEG, SS:STACK_SEG`
`START:MOVAX, DATA`
`MOVDS, AX`
`MOVAX, STACK-SEG`
`MOV SS, AX`
`MOV SP, OFFSET TOP_STACK`
`MOVAL, BCD`
`PUSH AX`
 	
`CALL BCD_BINARY`
`POPAX`
`MOV  BIN,AL`
`MOV  AH, 4CH`
`INT 21H`
 	
`; PROCEDURE  : BCD_BINARY`
`BCD_BINARY   PROC   NEAR`
`; Store the registers`
`PUSHF`
	Initialize data segment
	
	Initialize stack segment
	
	Initialize stack pointer
	Move BCD value into ALand push it onto word stack as parameter and call the procedure. The procedure returns binary value in AX register, which is moved to AL and the program returns to operating system.
	The procedure to convert BCD value received in AX register to binary value. But, first all the registers used by the procedure

PUSHAX	and flags register is pushed in the
PUSHBX	stack. Next, the value of stack
PUSHCX	top is moved to BP register.
PUSHBP	The stack location [BP+12]
MOVBP, SP	contains the BCD value, which is
MOVAX,[BP+ 12]	moved to AX =0025h.
MOVBL, AL	BL=AL=25h
ANDBL, 0Fh	BL= 25h AND 0Fh = 05h
ANDAL, F0H	AL= 25h AND F0h = 20h
MOVCL, 04	CL=04
RORAL, CL	AL= 02h
MOVBH, 0Ah	BH=0Ah (or 10)
MULBH	AX= 02×10 = 0014h
ADDAL, BL	AL=14h+05h=19h
MOV [BP + 12], AX	Move this binary value to stack
; Restore flags and registers	
POPBP	Restore all the registers to their
POPCX	original content, restore is in
POPBX	reverse order of storage and
POPAX	
POPF	
RET	return to the calling program
BCD_BINARY ENDP	End of procedure
CODE_SEG ENDS	End of code segment
END START	End of the file

Discussion:
The parameter is pushed on the stack before the procedure call. The procedure call causes the current instruction pointer to be pushed on to the stack. In the procedure flags, AX, BX, CX and BP registers are also pushed in that order. Thus, the stack would be as follows:


Before push AX (assume SP = 0090h)
After push AX before CALL (SP = 008Eh)
After CALL return address is stored (SP = 008Ch)
In procedure after PUSHF (SP = 008Ah)
In procedure after PUSH AX (SP = 0088h)
In procedure after PUSH BX (SP = 0086h)
In procedure after PUSH CX (SP = 0084h)
In procedure after PUSH BP (SP = 0082h)
Stack segment base (SS = 6000h)
X
AH
AL
IP HIGH
IP LOW
FLAG H
FLAG L
AH
AL
BH
BL
CH
CL
BP HIGH
BP LOW
:
:

The instruction MOV BP, SP transfers the contents of the SP to the BP register. Now BP is used to access any location in the stack, by adding appropriate offset to it. For example, MOV AX, [BP + 12] instruction transfers the word beginning at the 12th byte from the top of the stack to AX register.It does not change the contents of the BP register or the top of the stack. Since the BP contains SP value, which is 0082h, therefore, BP+12 would be 0082h + 000Ch = 008Eh. This address contains the value of AX, which was pushed prior to call to the procedure. Please recall this pushed value was the BCD value, which is to be converted (this is the parameter value). Thus, this instruction copies the BCD parameter value at offset 008Eh into the AX register in the procedure. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recursive procedures. It is a good practice to make a stack diagram as above while using procedure call through stacks. This helps in reducing errors in programming.

15.5.4 External Procedures

These procedures are written and assembled in separate assembly modules, and later are linked together with the main program to form a bigger module. Since the addresses of the variables are defined in another module, you need segment combination and global identifier directives to write such programs. Let us discuss them briefly.

Concept of Segment Combination

In 8086 assembler provides a means for combining the segments declared in different modules. Some typical combine types are:

1. PUBLIC: This combine directive combines all the segments having the same name (in different modules) as a single combined segment.
2. COMMON: If the segments in different object modules have the same name and the COMMON combine type then they have the same beginning address. During execution these segments overlay each other.
3. STACK: If the segments in different object modules have the same name and the combine type is STACK, then they become one segment having the length, as the sum of the lengths of individual segments.

For more details, you may refer to the further readings.

Use of Identifiers

a) **Access to External Identifiers:** An external identifier is one that is referred in one module but defined in another. You can declare an identifier to be external by including it on as EXTRN in the modules in which it is to be referred. This tells the assembler to leave the address of the variable unresolved. The linker looks for the address of this variable in the module where it is defined to be PUBLIC.

b) **Public Identifiers:** A public identifier is one that is defined within one module of a program but potentially accessible by all of the other modules in that program. You can declare an identifier to be public by including it on a PUBLIC directive in the module in which it is defined.

The following example explains the use of external procedures in 8086 microprocessor.

Program 14: Write a program using 8086 assembly procedure that divides a 32-bit number by a 16-bit number. The procedure should be defined in one module, and other modules should be able to call this procedure.

The procedure is named a SMART_DIV procedure and first the calling program to this external procedure is given below:

Directives **Statement**	**Discussion**

`DATA_SEGSEGMENT WORD PUBLIC`
`DIVIDENDDW2345h,89ABh`
`DIVISORDW5678h`
`MESSAGEDB'INVALID','$'`
`DATA_SEGENDS`
 	
`MORE_DATASEGMENTWORD`
`QUOTIENT  DW2DUP(0)`
`      REMAINDERDW 0`
`MORE_DATA ENDS`
 	
`STACK_SEGSEGMENTSTACK`
`      DW100 DUP(0)`
`TOP_STACKLABEL        WORD`
`STACK_SEG    ENDS`
 	
`PUBLICDIVISOR`
	Public data segment
	32-bit dividend
	16-bit divisor
	Message in case division is invalid
	
	This segment is valid only for this module as it is not shared.
	
	Stack segment of 100 words
	Label to stack top
	
	DIVISOR is a public variable

`PROCEDURESSEGMENTPUBLIC`
`EXTRN SMART_DIV: FAR`
 	
`PROCEDURESENDS`
	PPOCEDURES segment is a PUBLC division, it contains an external FAR procedure

`CODE_SEGSEGMENTWORDPUBLIC`
` ASSUME CS:CODE_SEG`
`DS:DATA_SEG, SS:STACK SEG`
`START:  MOVAX, DATA_SEG`
`        MOVDS, AX`
`        MOVAX, STACK_SEG`
`        MOVSS, AX`
`        MOVSP, OFFSET TOP_STACK`
`        MOVAX, DIVIDEND`
`        MOV DX, DIVIDEND + 2`
`        MOVCX, DIVISOR`
 	
`        CALL SMART_DIV`
`        JNC   SAVE_ALL`
 	
`        JMP   STOP`
 	
`ASSUME DS:MORE_DATA`
`SAVE_ALL:PUSH DS`
`        MOV   BX, MORE_DATA`
`        MOV   DS, BX`
`        MOV   QUOTIENT, AX`
`        MOV   QUOTIENT + 2, DX`
`        MOV   REMAINDER, CX`
`ASSUMEDS:DATA_SEG`
`        POP   DS`
`        JMP   ENDING`
`STOP:   MOV   DL, OFFSET MESSAGE`
`        MOV   AH, 09H`
`        INT   21H`
 	
`ENDING: MOV   AH, 4Ch`
`        INT   21H`
`CODE_SEGENDS`
`ENDSTART`
	It declares the code segment as PUBLIC so that it can be merged with other PUBLIC segments.
	Initially, DATA_SEG is used as the data segment
	There is only one stack segment for this main program
	
	AX is loaded with lower word (2345h), DX with higher word (89ABh) of the DIVIDEND; and DIVISOR is loaded in CX.
	Procedure is Called
	In case Carry flag is NOT set all the values are saved
	Unconditional jump to label STOP is executed.
	New data segment is assumed and initialized. The old DS is pushed to stack.
	
	The values of QUOTIENT and REMAINDER are saved.
	
	After saving the data segment is restored and unconditional jump is taken to end of program
	This code will be executed in case of Carry Flag is set, it will show the message INVALID to show that division was invalid.
	
	Finally, program will terminate

Discussion on the calling program:

The linker appends all the segments having the same name and PUBLIC directive with segment name into one segment. Their contents are pulled together into consecutive memory locations. The statement to be noted is PUBLIC DIVISOR. It tells the assembler and the linker that this variable can be legally accessed by other assembly modules. The statement EXTRN SMART_DIV:FAR tells the assembler that this module will access a label or a procedure of type FAR in some assembly module. Please also note that the EXTRN definition is enclosed within the PROCEDURES SEGMENT PUBLIC and PROCEDURES ENDS, to tell the assembler and linker that the procedure SMART_DIV is located within the segment PROCEDURES and all such PROCEDURES segments need to be combined in one. Please also note that in case the procedure SMART_DIV encounters an error, such as division by zero, it sets carry flag, which is checked in the calling program to put the results in the memory or display an error message.

Let us now define the PROCEDURE module:

Directives Statement	**Discussion**
Input: Dividend is 2 words input. The low word is input in AX and high word is input in DX register The divisor is input in CX register. **Output:** The Quotient is returned in DX:AX pair and remainder is returned in CX register. In case, divisor is zero, then Carry Flag is set to indicate that division is incorrect.	

`DATA_SEGSEGMENTPUBLIC`
`EXTRN DIVISOR:WORD`
`DATA_SEG ENDS`
 	
`PUBLIC      SMART_DIV`
	This declaration informs assembler that DIVISOR is a word variable and isexternal to this procedure. It also indicates that DIVISOR can be found in public segment DATA_SEG
	
	The SMART_DIV defined in this module is PUBLIC, i.e. it is available to other modules also.

`PROCEDURESSEGMENTPUBLIC`
`SMART_DIVPROCFAR`
`ASSUME CS:PROCEDURES, DS:DATA_SEG`
 	
`CMPDIVISOR, 0`
`JEERROR_EXIT`
 	
`MOVBX, AX`
`MOVAX, DX`
`MOVDX, 0000h`
`DIV CX`
 	
`MOVBP, AX`
`MOVAX, BX`
	It declares the PROCEDURES segment as PUBLIC so that it can be merged with other PUBLIC segments with the same name.
	
	The divisor is compared to 0, to check division by 0. You can also check it using CX. In case, it is same then jump to label ERROR_EXIT.
	
	AX containing lower dividend word is moved to BX and higher divided word is moved from DX to AX. The DX is emptied. The DX: AX now contains (0000h:89ABh). This is divided by divisor in CX. Leaving remainder in DX and quotient in AX. The quotient of higher word division is moved to BP and AX is loaded with lower word of dividend.
	
	The DX:AX pair is divided by CX.

`DIVCX`	Please note DX already contained the reminder of earlier division. The quotient of this division is stored in AX and remainder in DX

`MOVCX, DX`
`MOVDX, BP`
	The final remainder is transferred from DX to CX and the higher word division quotient saved in BO is moved to DX. Thus, DX:AX contains the quotient and CX remainder of this SMART_DIV procedure.

`CLC` `JMPEXIT`	The carry flag is cleared and control jumps to label EXIT
`ERROR_EXIT: STC`	This code is executed in case divisor was zero, the STC will set the Carry flag.

`EXIT:  RET`
`SMART_DIV ENDP`
`PROCEDURESENDS`
	The procedure returns.

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the calling program, therefore the statement EXTRN DIVISOR:WORD is necessary for informing assembler that this data name is found in some other segment. The data type is defined to be of word type. Please not that the DIVISOR is enclosed in the same segment name as that of calling program that is DATA_SEG and the procedure SMART_DIV is in a PUBLIC PROCEDURES segment.

☞**Check Your Progress 4**

1. State True or False

		T	F
(a)	A NEAR procedure can be called only in the segment it is defined.		
(b)	A FAR call uses one word in the stack for storing the return address.		
(c)	While making a call to a procedure, the nature of procedure that is NEAR or FAR must be specified.		
(d)	Parameter passing through stack is used whenever assembly language programs are interfaced with any high level language programs.		
(e)	A segment if declared PUBLIC informs the linker to append all the segments with same name into one.		

2. Show the stack if the following statements are encountered in sequence.

   a) Call to a NEAR procedure FIRST at 20A2h:0050h
   b) Call to a FAR procedure SECOND at location 3000h:5055h
   c) RETURN from Procedure FIRST.

15.6 SUMMARY

This Unit presents some programs written in 806 assembly language. The programs cover elementary arithmetic problems, code conversion problems, use of arrays and jump statements in assembly, the use of near and far procedure, highlighting the use of stack in procedure calls. Some of the important points presented in this unit are:

- An algorithm should 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.
- 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 requires you to initialize certain registers prior to their use in program, for example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialized 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 etc. Study such requirements 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 you have also seen some assembly programs can be very efficient in comparison of HLL programs, for example, the assembly programs of string processing are very efficient. You should write assembly programs from the further readings.

15.7 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, 0Ah       ; move dividend to BX
IDIV    BX            ; divide DX:AX by BX. The quotient will be in AX
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 and is stored in data segment.

```
        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 repeatedly
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. You 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 you 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, you can perform string processing very efficiently in 8086 assembly language.

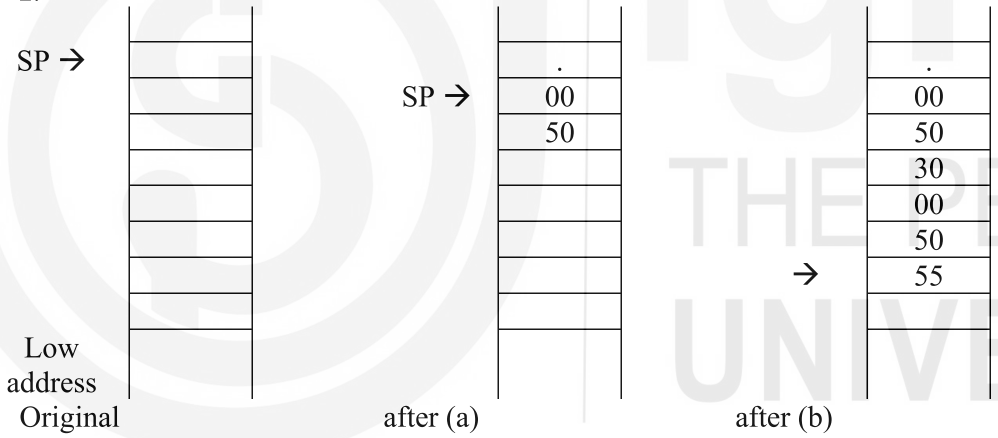
2. Direction flag if clear will cause REPE statement to perform in forward direction, i.e. comparison would be from first element to last.

3. It repeats the instruction after this instruction.

Check Your Progress 4

1. (a) True (b) False (c) False (d) True (e) True.

2.



(c) The return for FIRST can occur only after return of SECOND. Therefore, the stack will be back in Original state.