**Lecture-35**

**FUNDAMENTALS OF PROGRAMMING:**

It is very difficult for the user to write his program directly in machine code. It is more common to write the program in Assembly Language and then translate the assembly language program into machine language either by hand coding or using an assembler program.

The best method to introduce software program is to take few examples first and solve.

**Problem-1**: Write a software program to obtain the sum of N natural numbers. Only 8-bit registers are to be used.

$$SUM = \sum_{i=1}^{N} i$$

First we should note down all the constraints. In this problem since we have to use only 8-bit registers, SUM cannot exceed $255_D$. Working back we can find out the constraint on N. We know SUM of N natural number is given by

$$SUM = \frac{N\,(N+1)}{2}$$

Therefore, for SUM ≤ 255 the value of N ≤ 22.

The second stage is to list down the algorithm to be used to solve the problem and draw the flow chart of the algorithm. For this problem, the flowchart is shown in fig.6.1. Fig.6.2 gives better picture in terms of FORTRAN flow chart.

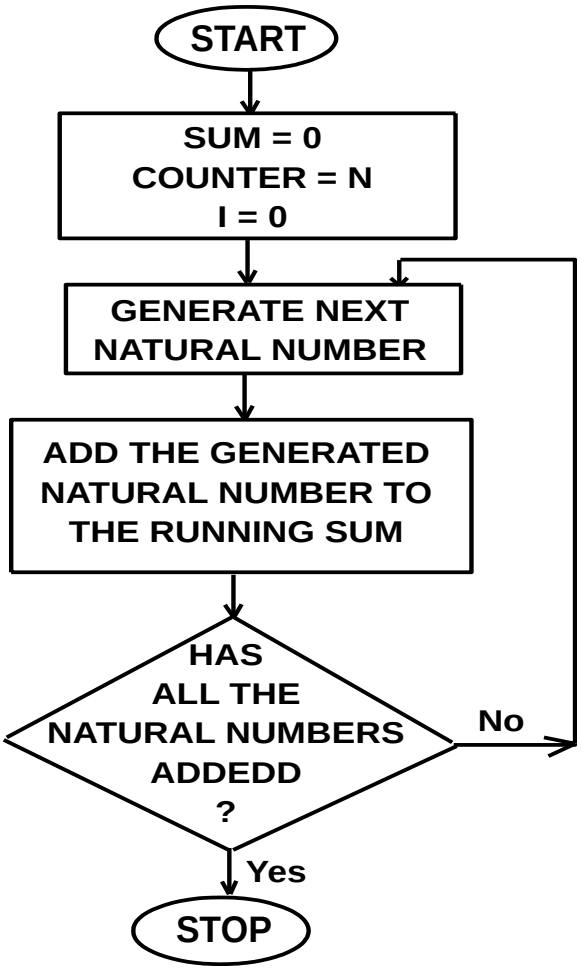



**Fig.6.1 Flow Chart of Algorithm**

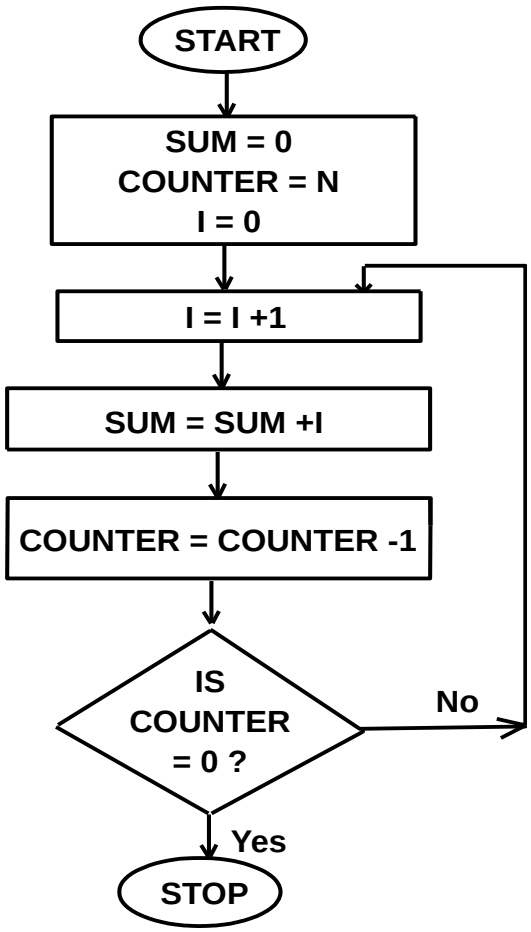



**Fig.6.2 FORTRAN Flow Chart**

Having obtained the flow chart of this problem we shall identify the registers or the memory locations. To define all the variables in the flow chart in deciding this we shall follow one general rule namely. Use internal register or register pairs only as far as possible. If they are not available then only think of a memory location as a variable.

For the problem under consideration following may be defined as the variables:

SUM → (A)

COUNTER → (C)

I → (B)

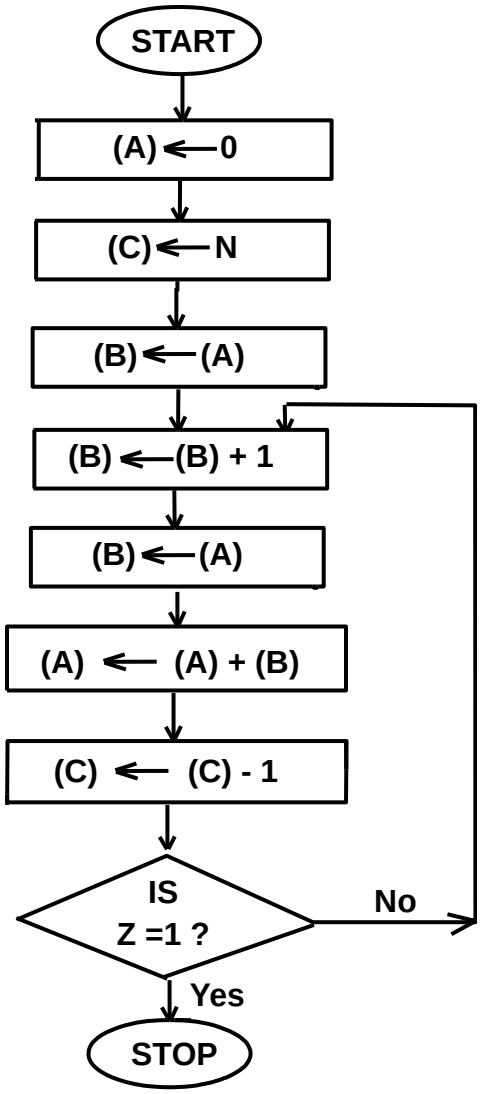



**Fig.6.3 Micro RTL Flow Chart**

Having decided the variables in the register we draw the macro RTL flow chart. While drawing this flow chart we take the help of FRTRAN flow chart drawn earlier and also the instruction set. Every block in the macro RTL flow chart must be implemented using one instruction at this stage. For the given problem this is shown in fig.6.3..

Having written the macro RTL flow chart we can directly write ALP. The preliminary ALP for fig.6.3 is shown in fig.6.4.

```
NSUM:  XRA   A      ;  Clear Accumulator
       MVI   C, N   ;  Initialize the Counter with Last Natural Number
       MOV   B, A   ;  Initialize Natural Number in (B) to zero
NEXT:  INR   B      ;  Generate Next Natural Number
       ADD   B      ;  Obtain running SUM in Accumulator
       DCR   C      ;  Has all Natural Number added?
       JNZ   NEXT;    No, Go Back to Generate Next Natural Number
       HLT          ;  Yes, stop.
```

**Fig.6.4 Assembly Language program for SUM of Natural Numbers**

Assembly language programmes are usually written in a standard form so that they can be easily translated to machine language by an assembler program (may be self on cross assembler). In general, the assembly language statements have four sectors in general known as fields:

| Label | Mnemonic | Operand | Comment |
|---|---|---|---|

*1)* Label field: The first field of ALP statement is the label field. A label is a symbol used to represent an address that is not specifically known or it is a name to be assigned to an instruction's location. The lebel field is usually ended with a colon (:).

*2)* Mnemonic field: This field contains the mnemonic for the instruction to be performed. Sometimes mnemonics are referred to as operation codes or opcode. It may consist of pseudo mnemonics.

*3)* Operand field: Operand field consists of operand and operands - either constants or variables with reference to the instruction in the

mnemonic field. It may be any register, data, or address on which the instruction is to be performed. Depending upon the instruction, the operand filed may be absent, may contain one operand or two operands separated by a comma. A comma is required between register initials or between register initial and a data byte.

*4)* Comment field: A very important part of an ALP is the comment field. For most assemblers, the comment field is started with a semicolon (;). Comments do not become part of the machine program. They are written for the reference of the user. If you write a program without comment and set it aside for 6 months it may be very difficult for you to understand the program again when you look back to it. It may even appear that someone else has written it. Comments should be written to explain in detail what each instruction or group of instructions is doing. Comments should not just describe the mnemonic, but also the function of the instruction in the particular routine. For best result, write comments as if you are trying to explain the program to someone who initially knows nothing about the program's purpose.

In addition to the comments, a program or subroutine should start with a series of comments describing what the program is supposed to do. The staring comments should also include a list of parameters, registers and memory locations used.

For example, an ALP statement is given below:

**NEXT: INR B; Generate next natural number**

In this statement **NEXT** is in the LEBEL field. **INR** is the mnemonic. **B** is the operand and rest is the comment. It is also clear that label field & remark field are optional.

All the programs written down in assemble language should start from a specific address which is not known before hand. Therefore, the symbolic name is given to the starting address of ALP. In the program written **NSUM** is given the name to the program.

The program is said to be completely written if all the instruction mnemonics including labels are listed along with proper comments. Next the program is assembled, i.e. the source code (instruction mnemonics) is translated to object code (machine code), a form in which the program can be loaded into the program memory. It is done by hand or using an assembler.

Hand assembling a program consists of assigning memory address and machine codes for each of the labels, opcode and operands. For hand assembly, two blank columns are allocated to the left of the label field. After assembly, the first column i.e. the address column contains the address of the first byte of each instruction and the second column contains the hexadecimal representation of the 1, 2, or 3 bytes of code that comprise the instructions.

Manual assembly is carried out in two steps, each requiring a complete scan or pass through the program. The first pass or scan determines the memory location into which the first byte of each instruction is assembled and creates a table for the values of the all symbolic names used in the program. A starting address is assigned to the first byte of the first instruction and is recorded in the address column. This is the initial value for a count that is increment by the

number of bytes in each instruction. This count corresponds to the location in memory which the first byte of each instruction is to be placed. A counter refers to as the location counter keeps the count. The counter location of the first byte of each instruction is recorded in the address column. The label (for each instruction that has one) is recorded is a symbol table together with the address of the first byte of the instruction. At the completion of the first pass (or first scan) through the ALP, all the symbolic labels and operands appear in the symbol table along with their assigned values. Thus, at the end of the first pass, the address columns & symbol tables are complete.

The second pass (or second scan) of the assembly process fills in the object code column. During this pass, each instruction is examined and the instruction mnemonic is replaced by its machine code written in hexadecimal notation. If the address or constant constituting the additional byte is written symbolically, the symbol table is consulted to determine the hexadecimal code for the symbolic operand. At the end of the second pass, the assembly process is complete. The resulting object code can be loaded into the microprocessor's memory and executed.

Insertion or deletion of instruction requires program reassembly. To avid reassembly, NOP instruction may be inserted after every few instruction. Few NOPs should left between the main program and subroutines, so that the main program can be expanded without having to change the addresses contained in the subroutine calls.

If we are not using an assembler to obtain machine language program (MLP) from assembly language program (ALP) then fig.4 is

sufficient to obtain MLP directly using hand coding procedure. However, then the starting address should be known to us. Obviously this should be in the RWM area. Let us suppose that the starting address for the problem is $2000_H$. Then the hand coding for fig.6.4 is done as shown in fig.6.5.

| Lebel | Addr | Contents | Mnemonics & Operands | Remarks |
|---|---|---|---|---|
| NSUM | $2000_H$ | AF | XRA A | Clear accumulator. |
| | $2001_H$ | 0E ….. | MVI C,N | Initialize the counter with 'N' |
| | $2003_H$ | 47 | MOV B,A | Initialize next natural number in (B) to 00. |
| NEXT | $2004_H$ | 04 | INR B | Generate next natural number |
| | $2005_H$ | 80 | ADD B | Obtain running SUM |
| | $2006_H$ | 0D | DCR C | Has all natural numbers added? |
| | $2007_H$ | C2 04 20 | JNZ NEXT | No, go back to generate next natural number |
| | $200A_H$ | 76 | HLT | Yes, Stop |

**Fig.6.5 Hand Coding of Assembly Language Program**

If the hand coding has to be performed automatically using an assembler, then assembler needs directions from the user w.r.t the following:

1) From which address (ADDR) the programme assembly is to start?
2) What is the value of the constant like 'N'?

Such directions are given to the assembler by pseudo instruction. These are also written similar to ALP statements with mnemonics in the mnemonic field. But remember these mnemonics have no operation codes like mnemonic of instruction set. They are only used for the directions given to the assembler,

**Lecture-36**

## ASSEMBLER DIRECTIVES

To assemble a program automatically the assembler needs information in the form of assembler directives that controls the assembly. For example, the assembler must be told at what address to start assembling the program. These assembler directives are command placed in the program by the designer that provides information to the assembler. They do not become part of the final program as they are not the part of the instruction set of the microprocessor nor did they translate into executable code. Therefore, they are also known as pseudo-instruction on false instructions.

Each assembler has its own unique pseudo instructions or assembler directives. These instructions differ from assembler to assembler but most of the assembler contains an equivalent set of pseudo instructions written in assembly language format.

**1. ORG:** The origin (ORG) instruction tells the assembler the address of the memory location for the next instruction or data byte should be assembled. ORG is entered at the beginning of a program. When different parts of a programme (e.g. subroutines) are to be placed in different areas of memory, an ORG pseudo instruction is used before each part of the program to specify the starting location for assembly of that part of the program. The origin instruction has the following form,

**ORG expression**

where **expression** evaluates to a 16-bit address, i.e., ORG is followed by an address. If no origin pseudo instruction appears before the first instruction in the program, assembly will begin, by default, at memory location $0000_H$.

For example, ORG $0100_H$ tells the assembler to start assembling the immediately following program at $0100_H$ in memory.

**2. <u>END:</u>** When an assembler scans the program to be assembled it must know, where the program ends. It cannot depend on a HLT instruction for this because some programmes don't contain a halt instruction as the last instruction and other don't contain a halt at all. An application program used, e.g., in process monitoring on control might run continuously and, therefore, not contain a halt instruction. Thus, an end assembly, END directive must be the last instruction. The directive has a form.

**END**

The END statement explicitly indicates the end of the program to the assembler. If no END statement is given, then the assembler just keeps on running through all the memory.

The ORG and END assembler directives, in effect, frame the program to be assembled.

```
ORG    0000H
[Assembly language instructions]
END
```

When there is more than one ORG assembler directive, then the assembly of group of instruction start at the location specified by

the origin assemble directive that proceeds. But there will be only one END instruction to tell the assembler the physical end of the program. For example,

**ORG 0000H**

**[Assembly language instructions]**

{This block of instructions is assembled starting at location **$0000_H$**}

**ORG 0100 H**

**[Assembly language instructions]**

{This block of instructions is assembled starting at location **$0100_H$**}

**END**

**3. <u>EQU</u>:** Symbolic names, which appear in assembly language programs as labels, instructions mnemonics and operands are translated to binary values by the assembler. As discussed in hand-assembly the labels are assigned the current value of the assembler's location counter when encountered in the first pass of the assembly. Instruction mnemonics have predefined values that the assembler obtains from a table that is part of the assembler.

A symbolic operand can be a register name, an address or a data constant. Register names have predefined values. All addresses correspond to labels in the program and their values are defined. Data constants, on the other hand, are defined by the designer using an equate assembler directive. The equate instruction EQU defines symbols used in the program. Equate assembler directives usually appear as a group at the beginning of a program and have the form.

name **EQU** expression.

'name' stands for the symbolic name. The assembler evaluates the expression and equates the symbolic name to it by placing the name in its symbol table along with the value of the expression. Therefore, whenever the name appears in the program, it is replaced by the value the expression in the equate pseudo instruction. For example,

COUNT **EQU** $0100_H$

Note the symbolic name is not followed by a colon and is not a label even though it appears in the label field. The symbolic name in one equate statement cannot be used in another nor can it be used as the label of another instruction. That is, the name in an equate directive cannot be redefined. If its value is changed, the equate assembler directive must be changed and the program reassembled.

**4. SET:** SET is similar to EQU assembler directive. This directive also assigns a value to the name associated it. However, the same symbol can be redefined by another SET statement later in the program. Thus, more than one SET instructions can have the same name the SET assembler directive has the form.

name **SET** expression.

**5. DS:** Another pseudo instruction, the define storage, reserves or allocates read/write memory locations for storage of temporary data. The first of the locations allocated can be referred to by an optional symbolic label. The define storage instruction has the form

opt. label: **DS** expression.

A number of bytes of memory equal to the value of the expression are reserved. However, no assumptions can be made about the initial

values of the data in these reserves locations, i.e, the assembler does not initialize the contents of these locations in anyway.

If has a symbolic name is used with the DS pseudo instructions, it has the value of the address of the first reserved location. Some of the assemblers use **DFS** for define storage. For example, to establish two 1-byte storage registers in RWM memory with the names TEMP1 & TEMP2, the instruction are written.

TEMP1: **DS** 1

TEMP2: **DS** 2

During the first pass, the assembler assigns the values of its location counter to TEMP1 and TEMP2 respectively and thus an address is associated with each label. Instructions in the program can read or write these locations using memory reference instructions such as STA TEMP1 or LDA TEMP2.

A memory buffer is a collection of consecutive memory locations and also used to store data temporarily. For example,

BUFFER: **DS** $50_D$

It tells the assembler to reserve 50 memory locations for storage when it assembles the program. The address of the first location is BUFFER (BUFFER to BUFFER + 50 -1). Such a buffer is usually written and read sequentially using register indirect addressing.

**6. DB:** When a table of fixed data values is required, memory must also be allocated. However, unlike the DS, each memory locations must have a defined value that is assembled into it. The pseudo instructions for this is define, DB and the general form is

opt. name: **DB** list

'list' refers either to one or more arithmetic or logic expressions that evaluate to 8-bit data quantities or to strings of character enclosed in quotes that the assembler replaces with their equivalent ASCII representations. Assembled bytes of data are stored in successive memory locations until the list is exhausted. Some of the assemblers use **DFB** for this assembler directive.

For example,

**DB** $07A_H$

It stores $7A_H$ in memory location right after the preceding instruction. Another example is

**DB** 'J', 'O', 'H', 'N'

It stores $4A_H$, $4F_H$, $48_H$ & $4E_H$ in the four successive memory locations to represent the string of ASCII characters.

**7. DW:** Define word DW instruction is similar to define byte pseudo instruction.

opt. name: **DW** list

The only difference between the DB & DW is that expression in this define word list is evaluated to 16-bit quantity and stored as 2-bytes. It is stored with the lower order byte in the lower of the two memory locations and the higher order byte in the next higher location. This is consistent with the convention for storing 16- bit quantities in 8085A systems. Some of the assemblers use **DFW** for this assembler directive.

**Macros:** Sometimes it is required that same set of instructions are to be repeated again & again. One way to simplify the problem is the

use of subroutine. This increases the execution time due to overhead. The other way is the use of macros. The assemblers which have the capability to process macro instructions are called macro assemblers. The assemblers are designed such that the programmer need to write set of instruction once and then refer it many times as desired.

A **macro instruction** is a single instruction that the macro assemble replaces with a group of instruction whenever it appears in an assembly language program. The macro instruction and the instruction that replace it are defined by the system design only once in the program. Macros are useful when a small group of instruction must be repeated several times in a program, with only minor or no changes in each repetition.

The use of macro in ALP entails three groups:

1) The macro definition
2) The macro reference
3) The macro expansion.

The macro definition defines the group of instructions equivalent to macro. Macro reference is the use of the macro instruction as an instruction in the program. A macro expansion is the replacement of the macro instruction defined by its equivalent. The first two steps are carried out by the system designer and the third by the macro assembler.

The macro definition has the following format:

| **LABEL** | **CODE (Mnemonic)** | **OPERAND** |
|---|---|---|
| **Name** | **MACRO** | **List** |
| | **[Macro body]** | |
| | **ENDM** | |

‘Name’ stands for the name of the macro that appears in the label field of the macro definition. A list of dummy parameters may be specified as List and, if so, these parameters also appear in the macro body. The macro body is the sequence of assembly language instructions that replace the macro reference into program when assembled. The macro definition produces no object code (hexadecimal number); it simply indicates to the assembler what instructions are represented by the macro name.

**Example-1:**

Consider the use of a macro involving a large amount of indirect addressing. An indirect addressing input capability is provided by the two instructions:

```
        LHLD    addr
        MOV     r, M
```

This sequence can be written as a macro named LDIND; with a macro definition of

```
LDIND   MACRO   REG, ADDR
        LHLD    ADDR
        MOV     REG, M
        ENDM
```

To have the macro body appear at any given point in the program, it requires a macro reference. This format is identical to that of an assembly language instruction.

| **Label** | **Code (Mnemonic)** | **Operand** |
|---|---|---|
| **optional label** | **name** | **parameters list** |

‘Name’ is the label by which the macro is referenced or called. The following macro instructions load register (C) indirectly through the address PRT

```
        LDIND   C, PTR
```

When a program containing macro is input to a macro assembler the assembler carries out a test substitution, the macro expansion, substituting for each macro reference the macro body specified in the macro definition. And for each dummy parameter list in the macro body, the appropriate parameter from the parameter list of the macro reference macro assembler encounter the macro instruction

```
        LDIND   C, PTR
```

It replaces it with the instructions

```
        LHLD    PTR
        MOV     C, M
```

**Example-2:**

The following macro rotation the contents of the accumulator to the left through carry, N times. This is done with a loop that is terminated when a register is counted down to zero. The numbers of rotation, N. and the register to be used as the counter are the parameters in the macro definition:

```
RALN    MACRO   N, REG
        MVI     REG, N
LOOP:   RAL
        DCR     REG
        JNZ     LOOP
        ENDM
```

If this macro appears in a program, a problem results as it will be referred twice. When the macro is expanded, the label LOOP will appear twice in the program, resulting in a multiply define symbol error when the program is assembled. This problem is avoided by use of the LOCAL directive; which is placed in the macro definition. The LOCAL directive has the form:

| LABEL | CODE | OPERAND |
|---|---|---|
| --- | LOCAL | label names |

The specified label names are defined to have meaning only within the current macro expansion. Each time the macro is referenced and expanded; the assembler assigns each local symbol a unique symbol in the form ‘??nnnn’. The assembler assigns ‘??0001’ to the first symbol, ‘??0002’ to the second symbol and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. These symbols are never duplicated by the assembler.

**Lecture-37**

**Problem-2:** Write the software program for the same problem as in problem-1 if the largest natural number exceeds 22.

In this case the sum shall exceed 8-bits; therefore, 16-bit operations are involved. Therefore, let us take the sum to be in (H, L) register pair (16-bits). If the sum is in (H, L) pair, the constraint on the sum is

$$SUM \leq 65535.$$

This shall give the largest value for N ≤ 361. Thus the largest number also needs 16-bit register for 16 bit arithmetic operation.

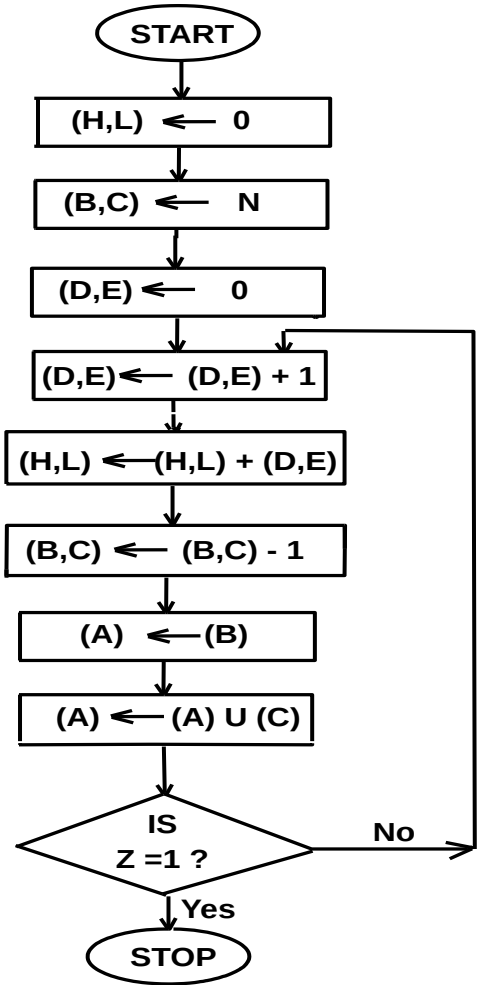



**Fig.6.6 ALP for SUM of N Natural Numbers**

Therefore, both counter and next natural numbers be placed in 16-bit registers

COUNTER ⟶ (B, C)

Next Natural Number, I ⟶ (D, E)

Therefore the macro RTL flow chart will be as shown in fig.6.6 and the ALP is given in fig.6.7

```
; This programme / sum of N natural numbers for N greater than 22.
        ORG     2000H
LNN     EQU     360 D
NSUM:   LXI     H, 0000 H   ; Initialize SUM
        LXI     B, LNN      ; Initialize COUNTER
        LXI     D, 0000H    ; Initialize current natural no.
NEXT:   INX     D           ; Generate next natural no.
        DAD     D           ; Obtain running sum in (H,L)
        DCX     B           ; Decrement counter
        MOV     A,B         ; Is counter zero?
        ORA     C
        JNZ     NEXT        ;No, go to generate natural number
        HLT                 ; Yes, SUM is in (H,L)
```

**Fig.6.7 ALP of SUM of N Natural Numbers**

**Problem-3:** Write a program me to introduce 1msec delay in the main programme. Assume 4MHz crystal is used in $\mu p$ .

The philosophy of software delay is to ask the microprocessor to do some irrelevant job for the requested delay duration. In general, the irrelevant job is to load an internal register with a pre-calculated

number ‘N’, ask the microprocessor to decrement it repeatedly till the register or counter becomes zero, and came out of the loop when the counter becomes zero. In this process the processor must have spent the required time delay duration. The two points to be noted in this connection are:

i) Number ‘N’ to be loaded is to be recalculated

ii) The register used for initial loading the number should not contain any useful data of main programme.

The necessary macro RTL flow chart for introducing the necessary 1msec delay is shown in fig.6.7.

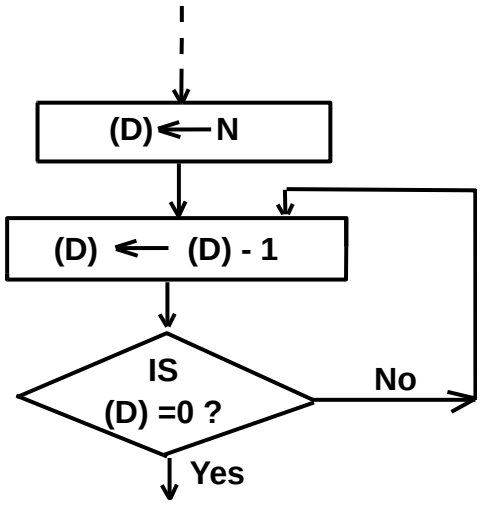



**Fig.6.7 Flow Chart to Introduce 1 ms Delay**

The corresponding ALP is shown in fig.6.8.

```
        :
        MVI     D, N        7 states
LP:     DCR     D           4 x N states
        JNZ     LP          10 x (N-1) + 7
        :
```

**Fig.6.8 Assembly Language Program to Introduce 1 ms Delay**

The fig.6.7 also shows the number of states elapsed in executing each instruction.

Total number of states elapsed = 7 + 4N + 10 (N-1) +7

= 14N + 4

If T secs is the duration of a state then time delay introduced is $\approx t_d=(K\,N + 4)\times T$. This must be equal to 1msec. For 4MHz clock, the time period T=0.5μsec.

On substitution, we get,

$$(14N + 4) \times 0.5 \times 10^{-6} = 1 \times 10^{3}$$

$$7 \times 10^{-6} \times N = 1 \times 10^{-3} - 2 \times 10^{-6}$$

$$\approx 1x\ 10^{-3}$$

Therefore, $N = 142.88 \simeq 143D = 8FH.$

Therefore, in fig.11 the number 'N' to be loaded in to the (D) register is $8F_H$ to introduce 1msec delay in the main pregame.

**Problem-4:** Modify problem-3 to introduce K msec delay

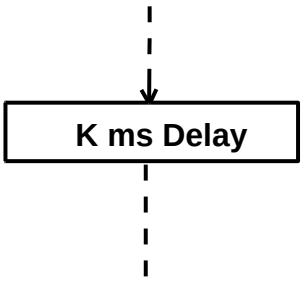



**Fig.6.9 Flow Chart to Introduce K ms Delay**

In this problem we have to introduce a variable time delay from 1 msec up to K msec. The register pair (B, C) can be used for loading, the constant K. K shall be 1 for 1msec delay and K shall be 60000D for 1min delay. The corresponding macro RTL flow chart in given in flg-14

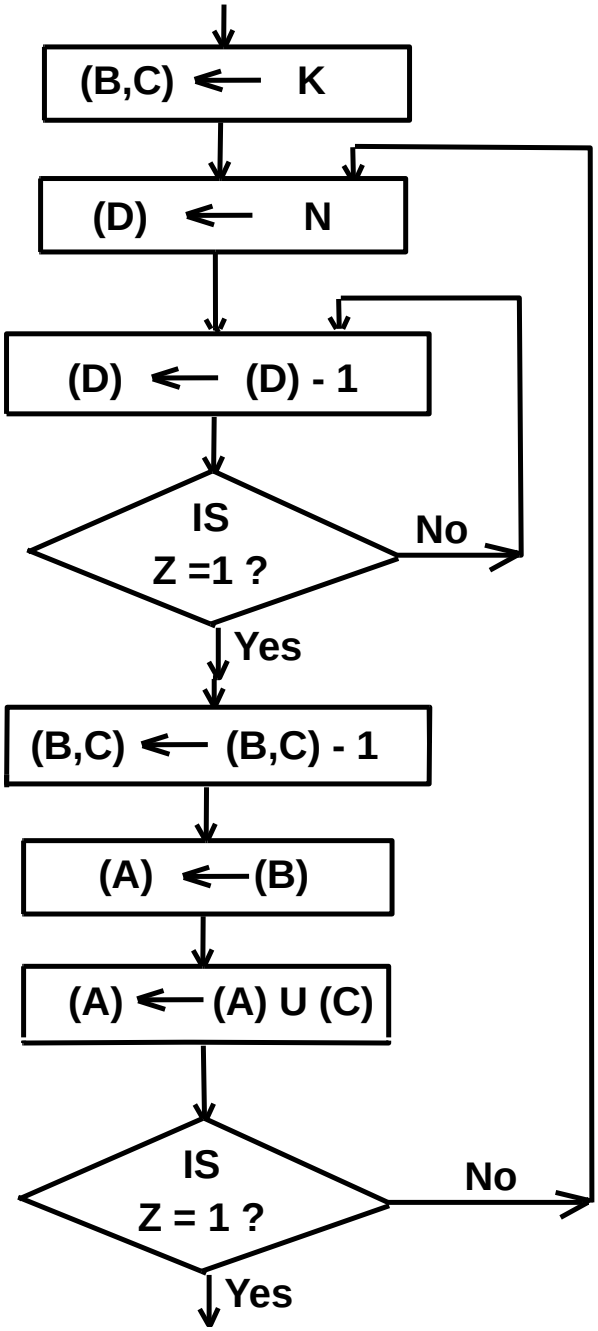



**Fig.6.10 Macro RTL Flow Chart to Introduce K ms Delay**

While drawing the macro RTL flow chart of fig-14 it is assumed that register pain (B, C), the register (D) and the accumulator (A) are available to the user. The corresponding ALP for fig- 14 is shown in fig.6.11.

```
        :                       Number of states
        :                       required for K=1
        LXI     B, K    10
LP2:    MVI     D, N    7
LP1:    DCR     D       4N
        JNZ     LP1     10(N-1)+7
        DCX     B       6
        MOV     A,B     4
        ORA     C       4
        JNZ     LP2     7
        :
```

**Fig.6.11 ALP to Introduce K ms Delay**

The constant N can be calculated again to introduce 1msec delay. The constant so calculated shall not differ from $8F_H$ calculated in problem -3 because only few extra instructions are involved in this problem. However we shall calculate the value of N?

For K = 1, the delay is given by

$$Td = [10 + 7 + 4N + 10(N-1) + 7 + 6 + 4 + 4 + 7] T$$

$$= [14N + 35] T$$

In this case, outer loop will not be traversed at all.

Assuming T=0.5 μsec, i.e., 2MHz external clock

$$1x\ 10^{-3} = [14N + 35] \times 0.5 \times 10^{6}$$

or, $$14N + 35 = 2 \times 10^{3}$$

or, $$N = 140D = 8C_H$$

8CH can be taken as 1msec constant for introducing 1mesc delay using ALP of fig.15. This constant shall not change even if some more instructions are added to fig.15.

Consider now that of we have to implement Kmsec delay, K variable at different points in the main programme as shown in fig.6.12.

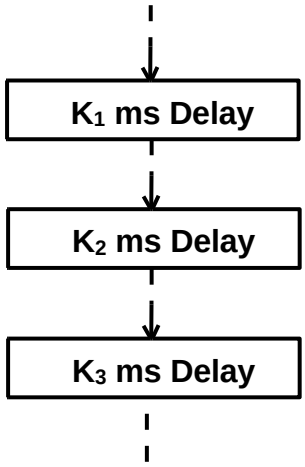



**Fig.6.12 Flow Chart to Introduce K ms Delay at Different Points**

One way to solve the problem of fig.6.12 is to repeat the flow chart of fig.6.10 every time loading the (B, C) pair in the appropriate constant K. This is clumsy and occupies tremendous amount of memory space unnecessarily. We can, therefore, write a subroutine program for Kmsec time delay starting from the symbolic address KDELAY. While writing the subroutine, it is assumed that the value of K is available in the (B, C) register pair. This means that the value of K must be loaded in to the (B, C) pair in the main programme before calling the subroutine. This is known as the input to subroutine or parameter passing from the main programme to subroutine programme. The second point has to be noted while writing the subroutine. Contents of all the registers made use of in the subroutine programme should not be destroyed. They should be saved on the top of stack using PUSH instructions. Further, the contents of these

registers should be restored by POP operations before returning to the main programme. These extra instructions increases the delay introduced by the subroutine. The subroutine incorporating the above details is shown in fig6.13.

```
KDELAY:   PUSH    B
          PUSH    D
          PUSH    PSW
LP2:      MVI     D, 8CH
LP1:      DCR     D
          JNZ     LP1
          DCX     B
          MOV     A,B
          ORA     C
          JNZ     LP2
          POP     PSW
          POP     D
          POP     B
          RET
```

**Fig.6.13 ALP for K ms Delay Subroutine**

**Lecture-38**

**Problem-5:**

An output port with an 8-bit register latch driver is interfaced using isolated I/O PORT address $30_H$. This register latch driver output drives 8 LEDs (0-0FF, 1-ON). Write a software programme in ALP to simulate an 8-bit ring counter at the PORT $30_H$. Ring counter must go from one state to the next in 10 sec. Kmsec delay subroutine programme (KDELAY) is available from the starting address $0430_H$.

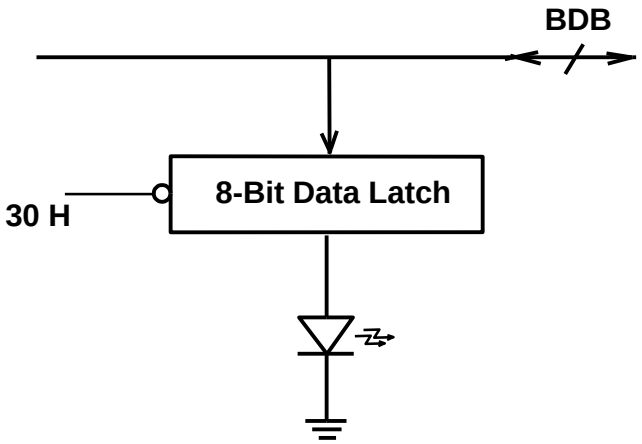



**Fig.6.14 LEDs connected to an Output Port 30H**

The 8-LEDs are connected to the 8-bit latch as shown in fig.6.14. The latch is capable to driving the LEDs. In the ring counter only one flip-flop is SET at a time. Therefore, only one LED is to be switched ON at a time as shown in fig.6.15.

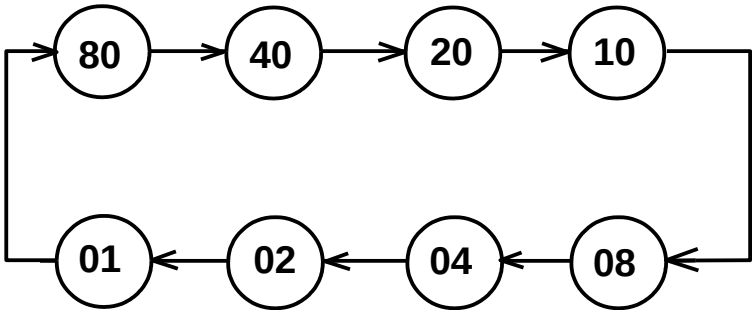



**Fig.6.15 8-bit Ring Counter**

The assembly language program to implement 8-bit ring counter is given in fig.6.16. Each bit of the port 30H is made HIGH one by one at an interval of 10 sec. 10 sec delay is implemented by calling K ms delay subroutine with (B,C) register pair initialized with K (=10000D) value.

```
        LXI     B, 10000D
        MVI     A, 80H          Set MSB of Accumulator
NEXT:   OUT     30H
        CALL    KDELAY
        RRC                     Set next bit of A
        JMP     NEXT
```

**Fig.6.16 ALP for 8-bit Ring Counter**

**Tutorial Problem for students:**

Simulate a BCD counter for up counting. The counting should be MOD-25 BCD up counting. This should go from one state to other in 10sec.

**Problem-6:**

Write a subroutine programme to multiply two unsigned numbers. The multiplicand is inputted as a 16-bit number through (D,E) pair. The multiplier is inputted to the subroutine through the accumulator register. The product should be outputted in the (H,L) register pair on return. In the process of multiplication no register should be destroyed except (H,L) register pair.

The algorithm used for unsigned multiplication can be best on explained by taking a simple example. Consider multiplication of 2 4-bit unsigned numbers [$(7x10)_D = 70_D$]

Multiplication = 0111B ⟹ Make it an 8-bit number 0000 0111B

Multiplication = 1010B ⟹ $m_3m_2m_1m_0$

Partial Sum = 0000 0000B

Check MSB.

| | | |
|---|---|---|
| $m_3$ =1, | Add Multiplicand to partial sum | 0000 0000 |
| | | 0000 0111 |
| | | 0000 0111 |
| | Shift left the partial result | 0000 1110 |
| $m_2$ =0, | No addition | 0000 1110 |
| | Shift left the partial result | 0001 1100 |
| $m_1$=1, | Add multiplicand | 0000 0111 |
| | | 0010 0011 |
| | Shift left the partial result | 0100 0110 |
| $m_0$ =0, | No addition | 0100 0110 |
| | No shifting | 0100 0110B |
| | | = 46H = 70D |

From this example, we see clearly the algorithm. Check the multiplier bit starting from MSB. If multiplier bit is 1, add the multiplicand to current partial product and then shift the partial product by one bit to the left. If the current multiplier bit is zero, do not add the multiplicand and only shift left the partial product by one bit. Repeat this number of times for n-bit multiplier. Few more refinements will be done when we draw the flow chart. The subroutine macro RTL flowchart is shown in fig.6.17.

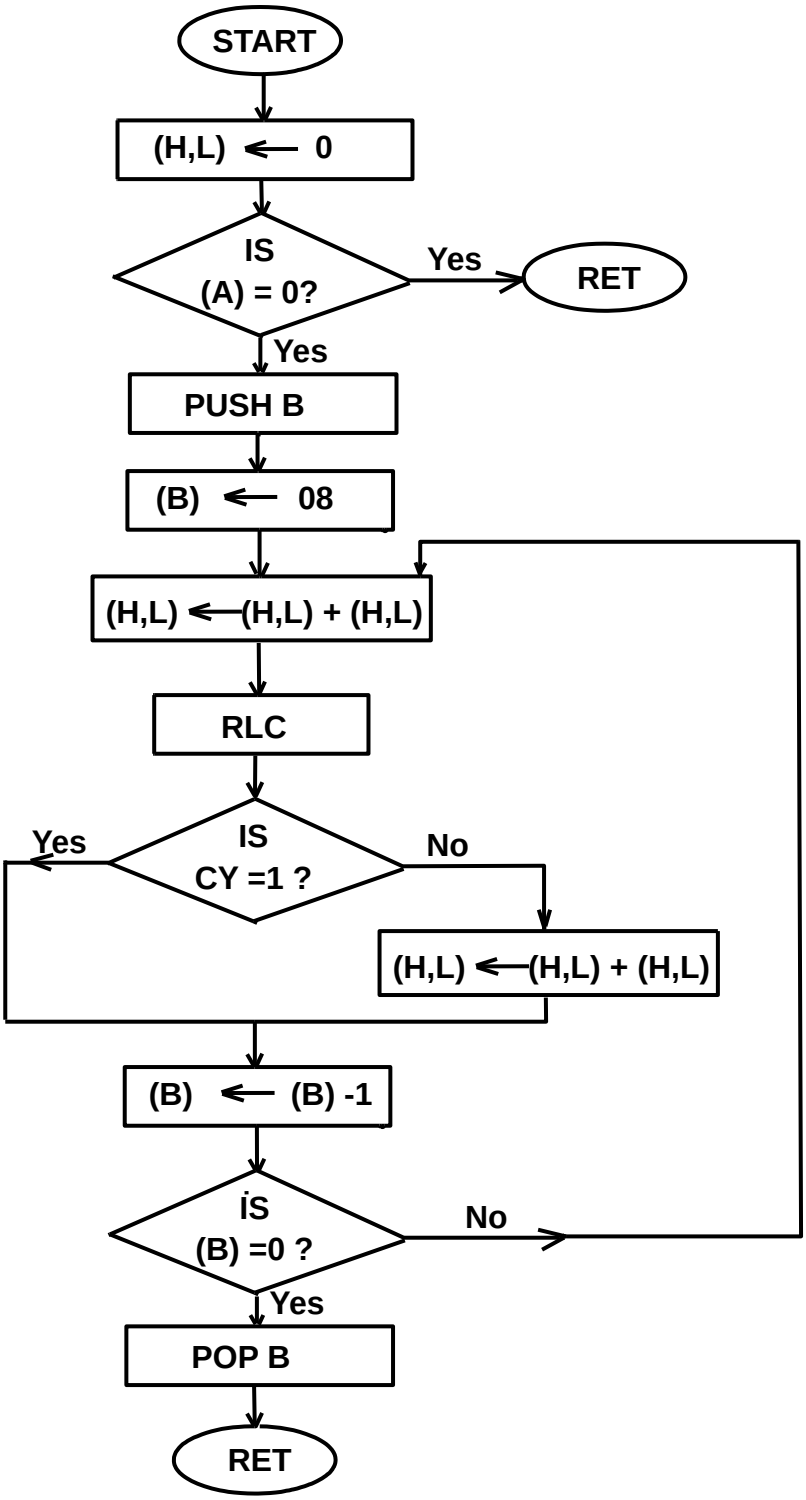



**Fig.6.17 Flow Chart of Multiplication Subroutine**

Fig.6.18 gives the ALP for subroutine programme.

```
UNSMUL:   LXI    H, 0000H      ; Initialize partial result
          ANA    A             ; Check multiplier for 0
          RZ
          PUSH   B
          MVI    B, 08H        ; Set counter for 8-bits
NEXT:     DAD    H             ;Shift result left by 1-bit
          RLC                  ;Check multiplier bit
          JNZ    TEST
          DAD    D             ;Add multiplicand
TEST:     DCR    B             ;Decrement counter
          JNZ    NEXT          ;Go for next bit if not zero
          POP    B
          RET
```

**Fig.6.18 Assembly Language Program of Multiplication Subroutine**

SUBROUTINE NAME: UNSMUL

| | |
|---|---|
| INPUT: | In this we should give the parameters passed from the main programme to the subroutine programme. In this case multiplicand is in (D, E) pair & multiplier is ACC. |
| OUTPUT: | Product in (H, L) |
| CALLS: | Nothing |
| DESTROYS: | (H, L) register pair. |
| DESCRIPTION: | This subroutine multiplies a 16-bit multiplicand by an 8-bit multiple to give 16- bit product. |

**Fig.6.19 Description of Subroutine**

The fig.6.19 gives the summary of the subtraction in the proper format. Proper formatting of subroutine is necessary because once the subroutine is satisfactorily tested; it can be used as library for further use. It can be used by anybody having access to this library provided the relevant information are given as per fig.6.19.

**Problem-7:** Write a subroutine to obtain

$$\text{SUM} = \sum_{i=0}^{N} a_i \; x^i$$

It is assumed that SUM can be accounted in 16-bits. The coefficients, $a_i$, $i = 0 - N$ are the positive integers and stored is a look up table from the starting address COEFF.

The variable $x$ is an unsigned 8-bit integer inputted from PORT whose symbolic address is PRTX through isolated I/O. The subroutine is entered with all the coefficients entered is the look up table as explained and the starting address COEFF, namely $X_3X_2X_1X_0H$ available in memory locations CLP to (CLP+1) and the number 'N' is available in B register.

| | | |
|---|---|---|
| COEFF | $a_n$ | $X_3X_2X_1X_0$ |
| | $a_{n-1}$ | |
| | : | |
| | $a_1$ | |
| | $a_0$ | $X_3X_2X_1X_0$ + N |

| | |
|---|---|
| CLP | $X_1X_0$ |
| | $X_3X_2$ |

On return from the subroutine the SUM should be available is (H, L) pair. It is the only register destroyed by the subroutine. We can make use of the subroutine written earlier for unsigned, multiplication of two numbers.

SUBROUTINE : USMUL

INPUT : Multiplier (A)

: Multiplicand (D, E)

OUTPUT : Product (H,L)

CALLS : Nothing

DESTROY : (H, L) pair

Description of the subroutine to be written is given in following format,

SUBROUTINE : SUM (POLSM)

INPUT : (1) Coefficients are arranged in the form of look up table from the starting address “COEFF”.

(2) The number N is in (B) register.

(3) The address COEFF is parsed through two memory location CLP and (CLP + 1).

OUTPUT : SUM $= \sum_{i=0}^{N} a_i \ x^i$ is (H, L) pair.

CALLS : USMUL

DESTRCYS : (H, L)

Algorithm:

$$\text{SUM} = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots \ldots \ldots \ldots + a_1 x^1 + a_0 x^0$$

$$= \Big( \ldots \ldots \ldots \ldots . \Big( \big( (a_n x + a_{n-1})x + a_{n-2} \big) x + a_{n-3} \Big) x + \cdots \ldots \ldots a_1 \Big) x + a_0$$

```
POLSM:  PUSH  PSW       ; Save processor status word
        PUSH  D         ; Save (D,E) register pair
        PUSH  B         ; Save (B,C) register pair
        LXI   D, 0000H  ; Initialize running SUM
        IN    PRTX      ; Input X value from input port
NEXT:   LHLD  CLP       ; Load (H,L) with CLP
        MOV   C,M       ; Bring current COEFF into C
        INX   H         ; (H,L) points to next COEFF
        SHLD  CLP       ; Save the next addr in CLP
        MOV   L,C       ; Bring current COEFF in (L)
        MVI   H,00H     ; Extend the COEFF to 16-bits
        DAD   D         ; Add the current COEFF
        XCHG            ; Current multiplicand is in (D,E)
        CALL  USMUL     ; Current product is in (H,L) pair
        XCHG            ; Update the running SUM
        DCR   B         ; All done?
        JNZ   NEXT      ; No
        LHLD  CLP       ; (H,L) pair noe points to a0
        MOV   L,M       ; Bring a0 to (L)
        MVI   H,00H     ; Extend it to 16-bit
        DAD   D         ; (H,L) now contains total SUM
        POP   B
        POP   D
        POP   PSW
        RET
```

**Fig.6.20 Assembly Language Program of Problem-7**

**Problem-8:** It is desired to divide a 16-bit unsigned number in locations 2000H and 2001H (Higher byte in 2001H) by an 8-bit unsigned number in location 2002H using the division algorithm

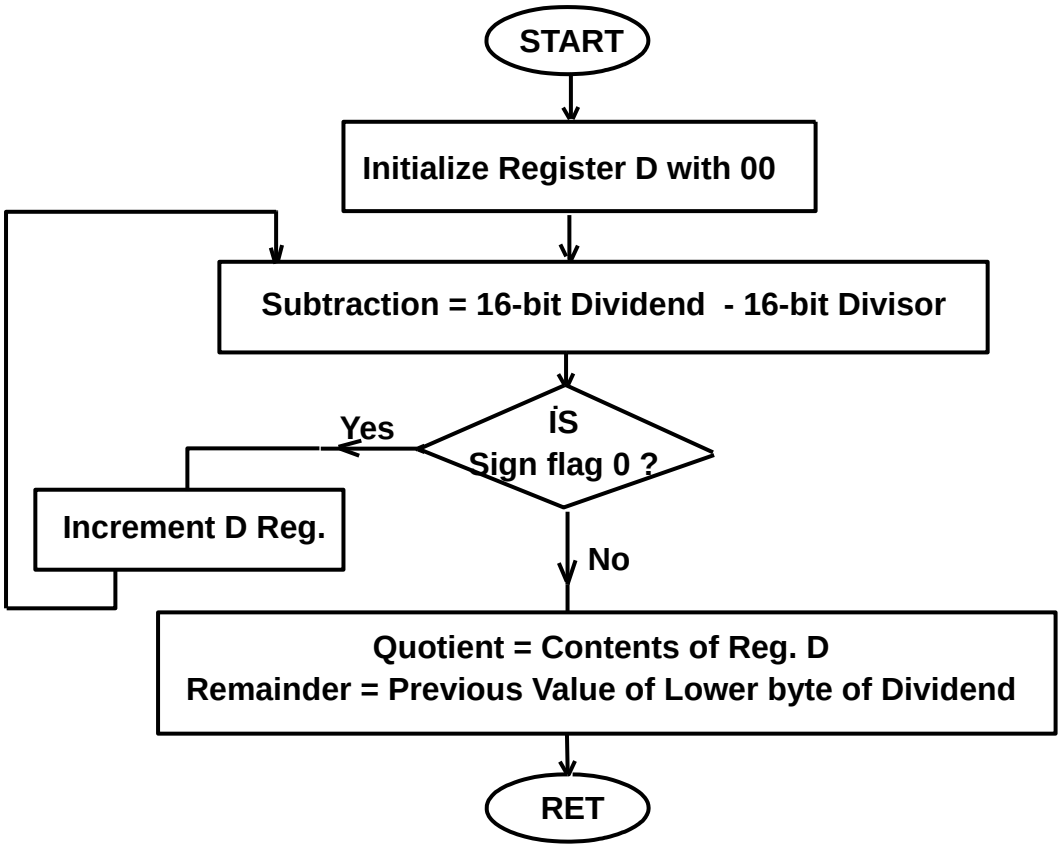



**Fig.6.21 Flow Chart for Division Program**

The division program is based on continuous subtraction of divisor from dividend till dividend becomes less than divisor. Every time subtraction s made, an index is increment which gives quotient at the end. The value left in the lower byte of 16-bit register used to store dividend gives the remainder. The ALP of the problem is shown in fig.6.22.

```
        MVI     D, 00H      ; Initialize (D) = 00
NEXT:   LXI     H, 2000H    ; (H,L) points to 2000H
        MOV     A, M        ; Lower byte of dividend in (A)
        LXI     H, 2002H    ; (H,L) points to 2002H
        ANA     A           ; To clear the CY
        SBB     M           ; Dividend – divisor (Lower byte)
        LXI     H, 2000H    ; Save the result of subtraction
        MOV     M, A        ; in 2000H
        LXI     H, 2001H
        MOV     A, M
        LXI     H, 2003H
        SBB     M           ; Dividend – divisor(Higher byte)
        LXI     H, 2001H
        MOV     M, A        ; Save it in 20001H
        JP      LP
        LXI     H, 2000H
        MOV     A, M        ; Dividend in (A)
        LXI     H, 2002H
        ADD     M           ; Divisor in (M)
        MOV     E, A
        HLT
LP:     INR     D           ; Increment (D)
        JMP     NEXT
```

**Fig.6.20 Assembly Language Program of Problem-8**

**Lecture-39**

**Passing Parameters to Subroutines**

Many subroutines accept data inputs from the calling program and provide as output results that are a function of the input data to the calling program. Subroutines are normally written such that they can be used with different programmes. One need not to write the same subroutine everytime. Thus makes the software development fast. This is possible only if subroutine is parameterized. The data, also called the subroutine parameters or arguments, must be transferred or passed to the subroutine by that partion of the program that calls the subroutine. In addition, results generated by the subroutine must be passed back to the calling program. There are a number of ways of passing parameters (data) between the calling program and the subroutine. The method is normally selected depending on number of parameters to be passed on. Data can be passed via

1. Internal registers
2. Reserved memory locations
3. Pointers to parameter lists in memory
4. The stack.

**1. Parameter Passing via Internal Registers**

When the number of parameters (data) to be passed is fewer than the internal general purpose registers available, it is convenient to transfer the data via internal registers accessible to the user. In this case, when transferring parameters to a subroutine, instructions that load the data into the specific internal

registers preceded the actual CALL instruction. In other words, before calling the subroutine, the parameters are loaded into the specific registers using instructions. These instructions and the subroutine CALL itself are together referred to as the subroutine linkage or calling sequence.

The subroutine obtains its parameters from predetermined registers when called. The results generated are placed in predetermined registers before the return instruction is executed. For example, the 8085A microprocessor does not have any multiplication instruction, but a subroutine can be written for this purpose. In previous lecture in example-6, a subroutine UNSMUL was written to multiply a 16-bit multiplicand with an 8-bit multiplier. The multiplicand and multiplier were passed on to the subroutine via (D,E) pair and ACC respectively before calling the subroutine. The subroutine returns the result of multiplication to the main program through (H,L) register pair. Subsequent instructions in the main program can use the result as required.

## 2. Use of Reserved Memory Locations for Parameter Passing

Parameters and results are also passed between the main program and a subroutine or between subroutines by reserved memory locations. A reserved memory location could be any memory location in RWM and is set aside or reserved to hold the value of a specific variable or parameter. These locations are established by the define storage, DS or DFS, assembler directive. Instructions in the calling sequence to put the parameter in the reserved memory location and in the subroutine to return

the result, refer to the parameter by its symbolic name, the label on the assembler directive that reserves its storage. For example, in the same multiplication subroutine, instead of passing the parameters through internal general purpose registers, one may put the multiplicand and multiplier in reserved memory locations MUTLIPLICAND and MULTIPLIER as shown in fig.6.22.

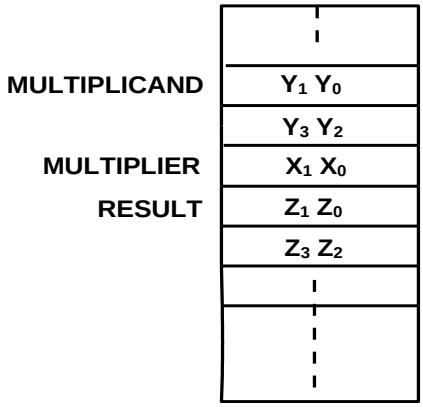



**Fig.6.22 Parameter Passing using Reserved Memory Locations**

The result of multiplication can also be returned by the putting the result in memory location RESULT before return instruction as shown in fig.6.22.

### 3. Use of Pointers to Parameter List in Memory

When a subroutine requires a large number of arguments, they can e placed in RWM, and pointers to the data can be provided in internal registers or in reserved memory locations – before the subroutine is called. For example, in example-8 of previous lecture, to calculate

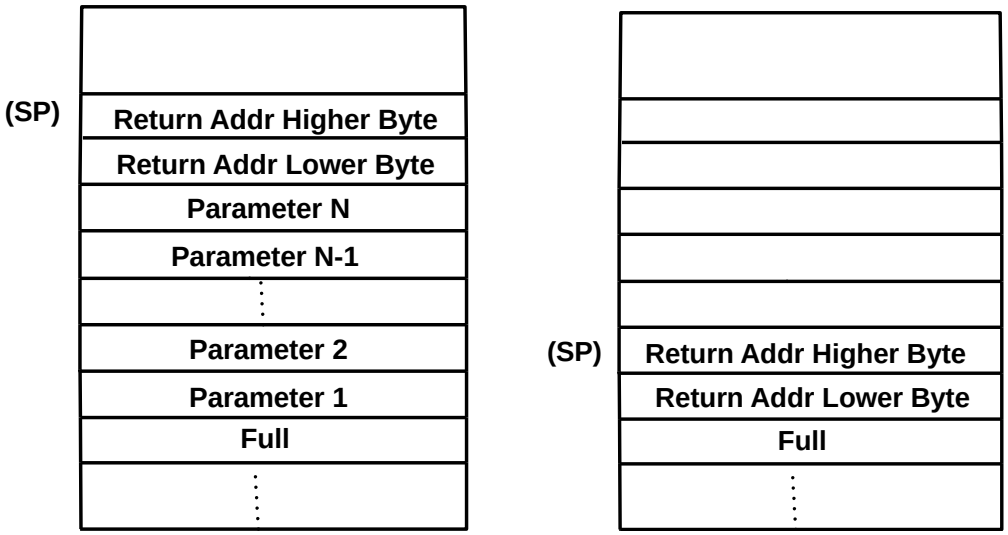
all the coefficients $a_0 \ldots a_N$ were stored in look up table or sequential memory location with the starting address COEFF

(=$X_3X_2X_1X_0$H) and this address was passed on to the subroutine through reserved memory locations CLP to (CLP+1) and the number ‘N’ through B register.

Similarly, if a subroutine is to written to compute the average of N data, the data are to be stored in sequential memory locations with the value of ‘N’ in the first location. The starting address may be passed on though (H,L) register pair used as memory pointer. In the same manner, (D,E) register pair may be used to indicate the location where the average id to be stored.

## 4. Parameter Passing Through Stack

The stack can be used to pass parameter. The parameters required by a subroutine are placed on the top of stack by the calling sequence, using the PUSH instructions before calling the subroutine. These parameters, together with the return address, which is automatically pushed onto the stack by the CALL instruction, comprise a stack frame as shown in fig. 6.23(a). After a subroutine is called, the stack pointer points to its return address, which is followed by the required parameters. The subroutine obtains the parameters from the stack, leaving the return address on the top of stack as shown in fig. 6.23(b). The number of parameters passed on the stack when calling a particular subroutine can be fixed, or the last parameter placed on the stack can be a count of parameters.




(a) (b)

**Fig.6.23 Stack Frame (a) Before Subroutine Call (b) After Subroutine Call**

A simple approach to implement this method of passing parameters to a subroutine is to first POP the return address from the top of the stack and save it in a register pair or in any reserved memory location. The subroutine than POPs the parameters from the stack as needed. When all the parameters are POPed from the stack and processed, the subroutine PUSHes the return address back onto the top of the stack and executes a return instruction. For example, the following instructions use a reserve memory location to save the return address:

```
POP     H       ; pop return address into H and L
SHLD    RADDR   ; save return address in reserve memory location
:               ; pop parameters and remove from the stack
:
LHLD    RADDR   ; obtain return address from reserve memory  location
PUSH    H       ; place the return address on top of stack
RET
```

If all the parameters pushed onto the stack are popped off, before executing the PUSH H instruction, the stack looks like as shown in fig.6.23(b). If on the other hand, they are not all popped off, the stack is left with one or more unused parameters below the return address, which causes a permanent shift of the of the top of the stack every time the subroutine is called. This condition, referred to as stack creep, is cumulative. It may result in stack to move from RWM to ROM or to undesired locations of RWM. In either case, the program will fail.

The XTHL instruction provides a means of obtaining parameters from the stack one at a time while leaving the return address on the top of the stack:

```
POP     H       ; pop return address from top of stack into H and L
XTHL            ; place parameters in H and L register and place the
                ; return address back on top of stack
:
:
XTHL            ; place parameters from H and L register on top stack
                ; and the return address in H and L registers
PUSH    H       ; save the return address on top of stack
RET             ; return from the subroutine.
```

After control has been transferred to the subroutine, this sequence pops the return address from the stack and places it in H and L, then exchanges the return address in H and L with the parameters on top of stack. As a result, the top 2 bytes of the

parameters come in H and L registers, and the return address is on top of the stack. This instruction sequence can be repeatedly executed until all the parameters placed on the stack have been removed.

Results of a subroutine can also be returned by the same technique. Before the subroutine returns, it first exchanges the results with return address available on top of stack and then pushes the return address on top of stack. Execution of return instruction pops the return address from the top of stack and put it in PC. The calling program then pops all the result from the stack.