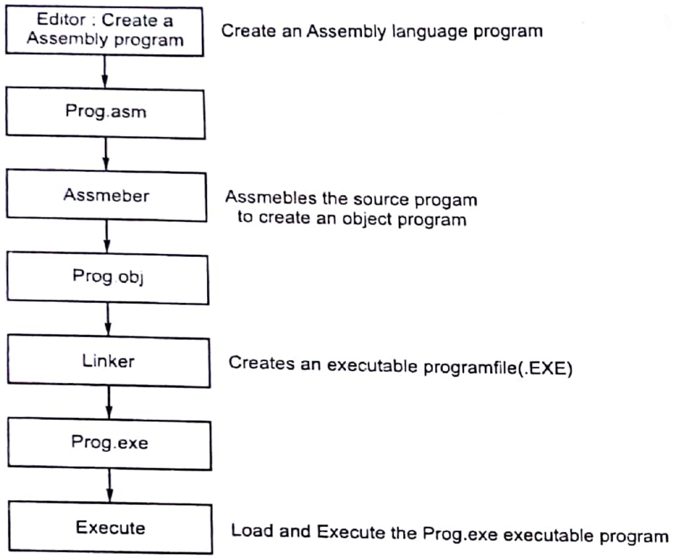


Different Steps Required to Execute an Assembly Language Program
Fig. 3.2

Step II: Link Step:

The link step involves converting the **.OBJ** file to an **.EXE** machine code. The linker's tasks also includes combining separately assembled program into one executable code. Thus, the **Linker**

(i) Combines assembled module into one executable program and

(ii) Generating an **.EXE** module and initializes with special instructions to facilitate its subsequent loading for execution.

Step III: Execute Step:

The last step is to load the program in memory for execution which is done by Loader.

3.5 PROGRAM LOOP

A **program loop** is a sequence of instructions that are executed many times, each time with a different set of data. **Program loops** are specified in FORTRAN by a **DO statement.**

A program loop is a method which runs a logic until it will not catch the desired result.

A **System Program** that translates a program written in a high-level programming language such as the **FORTRAN** to a machine language program is called a compiler. A compiler is a more complicated program than an **Assembler** and requires knowledge of systems programming to fully understand its operation.

A **compiler** may use an **Assembly Language** as an intermediate step in the translation or may translate the program directly to binary.

Line			
1		ORG 100	/Origin of program is SEX 100
2		LDA ADS	/Load first address of operands
3		STA PTR	/Store in pointer
4		LDA NBR	/Load minus 100
5		STA CTR	/Store in counter
6		CLA	/Clear accumulator
7	LOP,	ADD PTR I	/Add an operand to AC
8		ISZ PTR	/Increment pointer
9		ISZ CTR	/Increment counter
10		BUN LOP	/Repeat loop again
11		STA SUM	/Store sum
12		HLT	/Halt
13	ADS,	HEX 150	/First address of operands
14	PTR,	HEX 0	/This location reserved for a pointer
15	NBR,	DEC –100	/Constant to initialized counter
16	CTR,	HEX 0	/This location reserved for a counter
17	SUM,	HEX 0	/Sum is stored here
18		ORG 150	/Origin of operands is HEX 150
19		DEC 75	/First operaand
•			
•			
•			
118		DEC 23	/Last operand
119		END	/End of symbolic program

Symbolic Program to ADD 100 numbers
Program 3.1

The **program loop** specified by the DO statement is translated to the sequence of instructions listed in lines 7 through 10. Line 7 specified an indirect **ADD instruction** because it has the symbol I. The address of the current operand is stored in location **PTR.** When this location is addressed indirectly the computer takes the content of PTR to be the address of the operand. As a result, the operand in location 150 is added to the **Accumulator**. Location PTR is then incremented with the **ISZ** instruction in line 8, so its value changes to the value of the address of the next sequential operand. Location **CTR** is incremented in line 9, and if it is not zero, the computer does not skip the next instruction. The next instruction is **branch (BUN)** instruction to the beginning of the loop, so the computer returns to repeat the loop once again.

When location CTR reaches zero (after the loop is executed 100 times), the next instruction is skipped and the computer executes the instructions in line 11 and 12. The sum formed in the accumulator is stored in **SUM** and the computer halts. The **Halt** instruction is inserted here for clarity, actually, the program will branch to a location where it will continue to execute the rest of the program or branch to the beginning of another program. Note that ISZ line 8 is used merely to add 1 to the address pointer PTR. Since the address is a positive number, a skip will never occur.

The program of above Table introduces the idea of a **Pointer** and a **Counter** which can be used, together with the indirect address operation, to form a program loop. The pointer points to the address of the current operand and the counter counts the number of times that the program loop is executed. In this example we use two memory locations for these functions. In computers with more than one **Processor Register**, it is possible to use one processor register as a pointer, another as a counter, and a third as an **accumulator**. When **Processor Registers** are used as pointers and counters they are called **Index Registers.**

3.6 Programming Arithmetic & Logic Operations

The number of instructions available in a computer may be a few hundred in large system or a few dozen in a small one. Some computer perform a given operation with one **Machine Instruction**; other may require a large number of machine instructions to perform the same operation. As an illustration, consider the four basic arithmetic operations. Some computers have machine instructions to Add, Subtract, Multiply and Divide. Others, such as the basic computer, have only one arithmetic instruction, such a ADD. Operations not included 'in the set of machine instructions must be implemented by a program.

Operations that are implemented in a computer with one machine instruction are said to be implemented by Hardware. Operations implemented by a set of instructions that constitute a program are said to be implemented by Software. Some computers provides an extensive set of Hardware instruction designed to speed up common tasks. Others contain a smaller set of hardware instructions and depend more heavily on the software implementation of many operations. Hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation results in long programs both in number of instructions and in execution time.

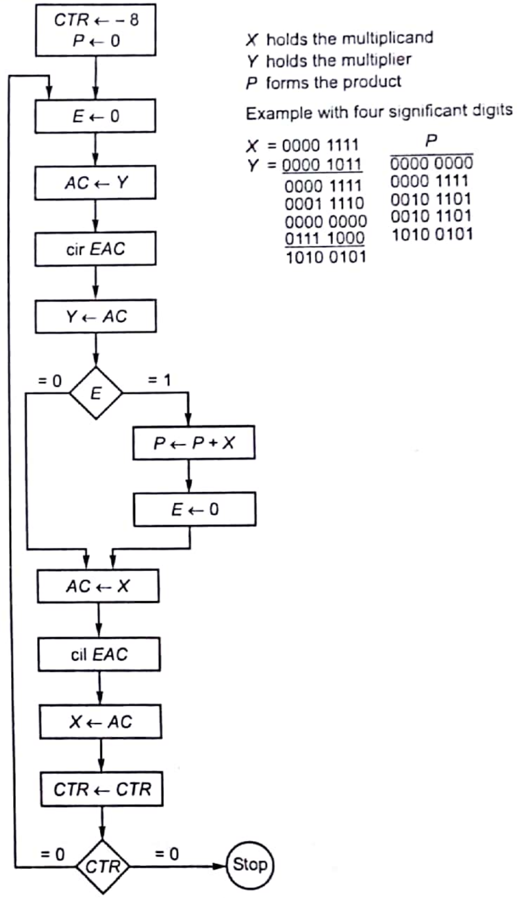
3.6.1 Multiplication Program

The program for multiplying two numbers is based on the procedure we use to multiply numbers with paper and pencil. As shown in the numerical example of below Fig. 3.3, the **multiplication** process consists of checking the bits of the **multiplier** Y and adding the **multiplicand** X as many times as there are 1's in Y, provided that the value of X is shifted left from one line to the next. Since the computer can add only two numbers at a time, we reserve a **memory location**, denoted by P, to store intermediate sums. The intermediate sums are called partial products since they hold a partial product until all numbers are added. As shown in the numerical example under P, the partial product starts with zero. The **multiplicand** X is added to the content of P for each bit of the **multiplier** Y that is 1.

The value of X is shifted left after checking each bit of the multiplier. The final value in P forms the product. The numerical example has number with four significant bits. When multiplied, the product contains eight significant bits. The computer can use numbers with eight significant bits to produce a product of up to 16 bits.

The flowchart of above Fig. 3.3 shows the step-by-step procedure for programming the

multiplication operation. The program has a loop that is traversed eight times, once for each significant bit of the multiplier. Initially, location X holds the multiplicand and location Y holds the multiplier. A counter CTR is set to -8 and location P is cleared to zero.



Flow Chart for Multiplication Program
Fig. 3.3

The multiplier bit can be checked if it is transferred to the E Register. This is done by clearing E, loading the value of Y into the **AC**, circulating right E and AC and storing the shifted number back into location Y. This bit stored in E is the low-order bit of the multiplier. We now check the value of E. If it is 1, the multiplicand X is added to the partial product P. If it is 0, the partial product does not change. We then shift the value of X once to the left by loading it into the AC and circulating left E and

AC. The loop is repeated eight times by incrementing location **CTR** and checking when it reaches zero. When the **counter** reaches zero, the program exits from the **loop** with the product stored in location P.

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load Multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

Program to Multiply Two Positive Numbers
Program 3.2

3.7 Subroutines

Frequently, the same piece of code must be written over again in many different parts of a program. Instead of repeating the code every time it is needed, there is an obvious advantage if the common instructions are written only once. A set of common instructions that can be used in a program many times is called a Subroutine. Each time that a subroutine is used int the main part of the program, a branch is executed to the beginning of the Subroutine. A subroutine consists of a self-contained sequence of instruction that carries out given task. A branch can be mad to the subroutine from any part of the main program. This poses the problem of how the subroutine knows

which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return. Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instruction to facilitate subroutine entry and return.

In the basic computer, the link between the main program and a subroutine is the **BSA** instruction (branch and save return address). To explain how this instruction is used, let us write a subroutine that shifts the content of the **Accumulator** four times to the left. Shifting a word four times is a useful operation for processing binary-coded decimal numbers or alphanumeric characters. Such an operation could have been included as a **Machine Instruction** in the computer. Since it is not included, a subroutine is formed to accomplish this task. The program of above Program 3.2 starts by loaded the value of X into the AC.

Location			
		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shitf left 4 times
109	SH4,	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/Circulate left fourth time
10E		AND MSK	/Set AC(13–16) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

Program to Demonstrate the use of Subroutines
Program 3.3

The next instruction encountered is BSA SH4. The BSA instruction is in location 101. Subroutine

SH4 must return to location 102 after it finishes its task. When the **BSA** instruction is executed, the control unit stores the return address 102 into the location defined by the symbolic address SH4 (which is 109). It also transfers the value of SH4 + 1 into the program counter. After this instruction is executed, **memory location** 109 contains the binary equivalent of hexadecimal 102 and the program counter contains the binary equivalent of hexadecimal 10A. This action has saved the return address and the subroutine is now executed starting from location 10A (since this is the content of PC in the next **fetch cycle**).

The computation in the subroutine circulates the content of AC four times to the left. In order to accomplish a logical shift operation, the four low-order bits must be set to zero. This is done by masking FFF0 with the content of AC. A mask operation is a logic **AND** operation that clears the bits of the AC where the mask operand is zero and leaves the bi's of the **AC** unchanged where the mask operand bits are 1's.

The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. The address to which the computer branches are not SH4 but the value found in location SH4 because this is an indirect address instruction. What is found in location SH4 is the return address 102 which was previously stored there by the BSA instruction. The computer returns to execute the instruction in location 102. The main program continues by storing the shifted number into location X. A new number is then loaded into9 the AC from location Y, and another branch is made to the subroutine. This time location SH4 will contain the return address 105 since this is now the location of the next instruction after BSA. The new operand is shifted and the subroutine returns to the main program at location 105.

From this example we see that the first memory location of each **subroutine** serves as a link between the main program and the subroutine. The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine linkage. The BSA instruction performs an operation commonly caked subroutine call. The last instruction of the subroutine of the subroutine performs an operation commonly called subroutine return.

The procedure used in the basic computer for subroutine linkage is commonly found in computers with only one **Processor Register**. Many computers have multiple processor registers and some of them are assigned the name index register. In such computers, an Index Register is usually employed to implement the **subroutine linkage**. A branch-to-subroutine instruction stores the return address in an index register. A return-from-subroutine instruction is effected by branching to the address presently store in the **Index Register.**

3.8 INPUT-OUTPUT PROGRAMMING

Users of the computer write programs with symbols that are defined by the programming language employed. The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in computer memory. A **binary-coded** character enters the computer when an **INP** (input) instruction is executed. A binary -coded character is transferred to the output devise when an **OUT** (output) instruction is executed. The output device detects the binary code and types the corresponding character.

Following Program (a) lists the instruction needed to input a character and store it in **memory**. *The SKI* instruction checks the input flag to see if a character is available for transfer. The next

instruction is skipped if the input **flag** bit is 1. The INP instruction transfers the binary-coded character into AC (0-7). The character is then printed by mean of the OUT instruction. A terminal unit that that communicates directly with a computer does not print the character when a key is depressed. To type it, it is necessary tp send an OUT instruction for the printer. In this way, the user is ensured that the correct transfer has occurred. If the SKI instruction finds the **flag** bit at 0, the instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. Because the input device is much slower than the computer, the two instructions in the loop will be executed many times before a character is transferred into the accumulator.

Following Program (b) lists the instructions needed to print a character initially stored in **memory**. The character is first loaded into the AC. The output flag is then checked. If it is 0, the computer remains in a two-instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the **Accumulator** to the printer.

(a) Input a character :

```
CIF,    SKI          /Check input flag
        BUN CIF      /Flag = 0, branch to check again
        INP          /Flag = 1, input character
        OUT          /Print character
        STA CHR      /Store character
        HLT
CHR,    —            /Store character here
```

(b) Output one character :

```
        LDA CHR      /Load character into AC
COF,    SKO          /Check output flag
        BUN COF      /Flag = 0, branch to check again
        OUT          /Flag = 1, output character
        HLT
CHR,    HEX 0057     /Character is "W"
```

Programs to Input and Output One Character
Program 3.4

3.8.1 Program Interrupt

The running time of input and output programs is made up primarily of the time spent by the computer in waiting for the external device to set its **flag**. The waiting loop that checks the flag keeps the computer occupied with a task that wastes a large amount of time. This waiting time can be eliminated if the interrupt facility is used to notify the computer when a flag is set. The advantage of using the **interrupt** is that the information transfer is initiated upon request from the external device. In the meantime, the computer can be busy performing other useful tasks. Obviously, if no other program resides in memory, there is nothing for the computer to do, so it might as well check for the flags. The interrupt facility is useful in a **multiprogramming** environment when two or more programs reside in **memory** at the same time.

Only one program can be executed at any given time even though two or more programs may