CHAPTER SEVEN

PROGRAMMING THE BASIC COMPUTER

7.1. Introduction

A computer system as it was mentioned before in chapter 1, it is subdivided into two functional parts: -

1. **Hardware**, which consists of all physical components and all associated device or equipment. Parts of the hardware have been explained in chapter five concerning the basic computer. More details will be discussed in the following chapters.

2. **Software**, which refers to the programs that are written for the computer.

   The system software of a computer is divided into two main parts, **the operating system** that consists of the programs included in a system software package and **the application programs**, in which those programs are written by the user for solving particular problems.

   A program written by a user may be either dependent or independent of the computer that runs this program. For example, a program written in standard Pascal is machine independent because most computers provide a translator program that translates the standard Pascal program to the binary code of the computer available in the particular installation. In the other hand the translator program is **machine dependent** since it must translate the Pascal program to the binary code recognized by the hardware of the particular computer used.

Generally, one can be familiar with various aspects of computer software without being familiar with details of how the computer hardware operates. In the same way, it is possible to design parts of the hardware without knowledge of its software capabilities. However, the people concerned with computer architecture, they should have knowledge of both hardware and software because these two branches influence each other.

In spite of that, the computer executes only the programs that are written in binary form, there are variety types of programming languages that a programmer can write his program. In these cases there must be a translator to translate these programs to the binary form before the computer can execute them.

Programs written for computer may be in one of the following forms: -

1. **Binary Code form.**

   This is the machine language form, which is defined as a collection of all the fundamental instructions that the machine can execute, expressed together with the operands as a pattern of 1's and 0's.

2. **Octal or Hexadecimal Code form.**

   This achieved by translating of the binary code to the equivalent octal or hexadecimal representation.

3. **Symbolic Code form (Assembly Language)**

   In this form, the alphanumeric equivalent of the machine language is used. Alphanumeric mnemonics are used as an aid to the programmer, instead of the1's and 0's that the machine interpret. Each symbolic instruction can be translated into one binary coded instruction. The translator that translates the program from assembly language to machine language called ***Assembler*.**

4. **High-Level Programming Languages**

   These programming languages are closer to the programmer language and far from the machine language. These languages reflects the procedures used in solution of problems rather than be concerned with the computer hardware behavior. Examples of these languages are C, Pascal, Fortran… etc.
   Programs in these programming languages are written in a sequence of statements, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer.
   The program that translates a high-level language program to binary called a ***Compiler***.

7.2. Machine Language

A program in machine language is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

A program of adding two numbers ((13+(-2)) is used to explain the four forms listed in section 7.1 using the basic computer.

1. Table 7.1 shows a binary program for adding the two numbers. Here the program is stored in the memory from location 0 to 6. Looking in this type of program, there is a difficulty to understand what is to be achieved when executed. However, the computer hardware recognizes this type of instruction code.

Table 7.1

Binary program	
location	**Instruction Code**
0000 0000 0000	0010 0000 0000 0100
0000 0000 0001	0001 0000 0000 0101
0000 0000 0010	0011 0000 0000 0110
0000 0000 0011	0111 0000 0000 0001
0000 0000 0100	0000 0000 0001 0011
0000 0000 0101	1111 1111 1111 1110
0000 0000 0110	0000 0000 0000 0000

2. Table 7.2 shows a hexadecimal program for adding the two numbers. The hexadecimal representation of the program is more convenient to use than the binary form for the programmer; however, one must convert each hexadecimal digit to its equivalent 4-bit number when the program is entered into the computer.

Table 7.2

Hexadecimal Program	
location	**Instruction**
000	2004
001	1005
002	3006
003	7001
004	0013
005	FFFE
006	0000

3. Table 7.3 shows a program with symbolic operation codes for adding the two numbers instead of their binary or hexadecimal equivalent. Here the address parts of memory-reference instructions, as well as operands, remain in their hexadecimal value. A column of comments is included for explaining the function of each instruction. Symbolic programs are easier and preferable to handle. These symbols can be converted to their binary code equivalent to produce the binary program.

Table 7.3

Program with Symbolic Operation Codes		
location	**Instruction**	**Comments**
000	LDA 004	Loud first operand Into AC
001	LDD 005	Add second operand to AC
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0013	The first operand
005	FFFE	The second operand
006	0000	Store the sum here

Note. Negative numbers must be converted to binary in the signed 2's complement representation in the above three programming forms.

4. Table 7.4 shows an assembly language program for adding the above two numbers. The symbols ORG, DEC, and END are called **Pseudo Instructions** and not a machine instructions.

The purpose of ORG Pseudo instruction is to specify an origin, i.e. indicates the memory location of the first instruction of the program is below it.

The purpose of DEC Pseudo instruction is to specify a decimal operand.

The purpose of END Pseudo instruction is to indicate the end of the program.

The three lines having symbolic addresses A, B, and C, their value is specified by they being present as a label in the first column.

Table 7.4

Assembly Language Program		
location	**Instruction**	**Comments**
	ORG 0	/Origin of program is location 0
	LDA A	/Load operand from location A
	ADD B	/Add operand from location B
	STA C	/Store sum in location C
	HLT	/Halt computer
A,	DEC 13	/First decimal operand
B,	DEC -2	/Second decimal operand
C,	DEC 0	/Sum stored in location C
	END	/End of symbolic program

5. Table 7.5 shows an equivalent FORTRAN program for adding the two numbers.
   The FORTRAN program is translated into the binary values listed in the program of table 7.1, by what is called a **Compiler Program.**

Table 7.5

Fortran Program
INTEGER A, B, C DATA A, 13 B, -2 C = A + B END

7.3. Assembly Language

Every computer has its own particular assembly language. The rules for writing assembly language programs are documented and published in manuals which are usually available from the computer manufacturer.

The rules of an assembly language for the basic computer are the following: -

The line of an **assembly language** program is divided to three columns called fields as follows: -

1, **The Label Field**. This field may be empty or it may specify a symbolic address. **A symbolic address has the following characteristics: -**

- **a.** It consists up to three alphanumeric characters, the first character must be a letter, and the other two if they exist may be letters or numerals.
- **b.** The programmer chooses the symbol arbitrarily.
- **c.** A symbolic address is terminated by a comma.

2. **The Instruction Field.** This field specifies a machine instruction or a pseudo instruction. **The instruction field may specify one of the following: -**

- **a.** A memory reference instruction (**MRI**).
- **b.** A register reference Instruction (**RRI**).
- **c.** Input-output instruction (**IOI**).
- **d.** A pseudo instruction with or without operand.

A memory reference instruction (MRI) occupies two or three symbols separated by spaces: -

- **a.** The first part must be a three letter symbol defining an **MRI**.
- **b.** The second part is the symbolic address, which specifies the memory location of an operand.
- **c.** The third part is the letter **I, this may or may not present**. If it's present, it denotes indirect address, while if **I** is missing, it denotes direct addressing.

Example

AND A1	Direct address **MRI**
AND ABC I	Indirect address **MRI**
ADD A	Direct address **MRI**

A non-MRI or RRI & IOI are defined as those instructions that do not have an address part, and presented by a three letter symbol only in the instruction field.

Example

CMA	**non-MRI (RRI)**
OUT	**non-MRI (IOI)**

pseudo instruction as mentioned before is not a machine instruction but an instruction which gives information to the assembler about some phase of the translation. There are four pseudo instructions as shown in table 7.6.

Table 7.6

Symbol	Information for the Assembler
ORG N	N indicates the memory location for the instruction or operand listed in the following line.
END	Indicates the end of the program.
DEC N	N indicates the decimal signed number to be converted to binary.
HEX N	N indicates the hexadecimal signed number to be converted to binary.

3. **The Comment Field.** This field may be empty or it may include a comment. Comments must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are inserted for explanation process only and are neglected during the binary translation process.

Homework

Translate the following assembly language program to subtract two numbers to its equivalent binary from.

```
      ORG 300    /Origin of program is location 300
      LDA SUB    /Load subtrahend to AC
      CMA        /Complement AC
      INC        /Increment AC
      ADD MIN    /Add minuend to AC
      STA DIF    /Store difference
      HLT        /Halt computer
MIN,  DEC 23     /Minuend
SUB,  DEC -2     /Subtrahend
DIF,  HEX 0      /Difference stored here
      END        /End of symbolic program
```

7.3. The Assembler and the Compiler Programs

The **Assembler Program** is a program that translates a symbolic language program called the source program into binary program called the object program.
An assembler is a program that operates on character strings and produces an equivalent binary interpretation.

A **Compiler Program** is a system program that translates a program written in a high level programming language such as Pascal to a machine language program.
A compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

7.4. Programming Arithmetic and Logic Operations

Depending on the computer, some of them perform a given operation with one machine instruction; others may require a large number of machine instructions to perform the same operation.

For example the four basic arithmetic operations (addition, subtraction, multiplication, and division), there are some computers which have machine instruction to add, subtract, multiply, and divide while other computers such as the basic computer have only one arithmetic instruction, such as ADD and the remaining three operations must be implemented by a programs.

Operations that are executed with one machine instruction are said to be implemented by hardware, while those operations that are executed by a program are said to be implemented by software.

Hardware implementation is more costly than software implementation because of the additional circuits needed to implement the operation, while software implementation has a drawback of long execution time results from more instructions needed to implement the operation.

In this section, we will demonstrate the software implementation for some of arithmetic and logic operation using the basic computer.

1. Multiplication Operation

To simplify the process, assume unsigned positive binary numbers and each number have no more than 8-bits so the result of multiplication does not exceeds the word capacity of 16-bits.

The program for multiplying two numbers based on the procedure used to multiply numbers with paper and pencil. **The following example with four significant explains this multiplication process.**

Multiplicand	**X =**	00000111
Multiplier	**Y =**	00001010
		00000000
		00001110
		00000000
		00111000
		01000110

As shown in the above example, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X shifted left from one line to the next.

A memory location denoted by P is reserved to store intermediate sums, since the computer can add only two numbers at a time. The intermediate sums are called partial products since they hold a partial product until all numbers are added.

The following example shows how the result of multiplication is achieved in the memory location P for four significant digit of multiplier and multiplicand.

Initial value of P	**0000 0000**	
P after of 1st pass	**0000 0000**	
P after of 2nd pass	**0000 1110**	
P after of 3rd pass	**0000 1110**	
P after of 4th pass	**0100 0110**	***Final Result***

Note. The computer can use numbers with eight significant bits to produce a product of up to 16 bits.

The flowchart shown in figure 7.1 explains the step-by-step the multiplication operation.

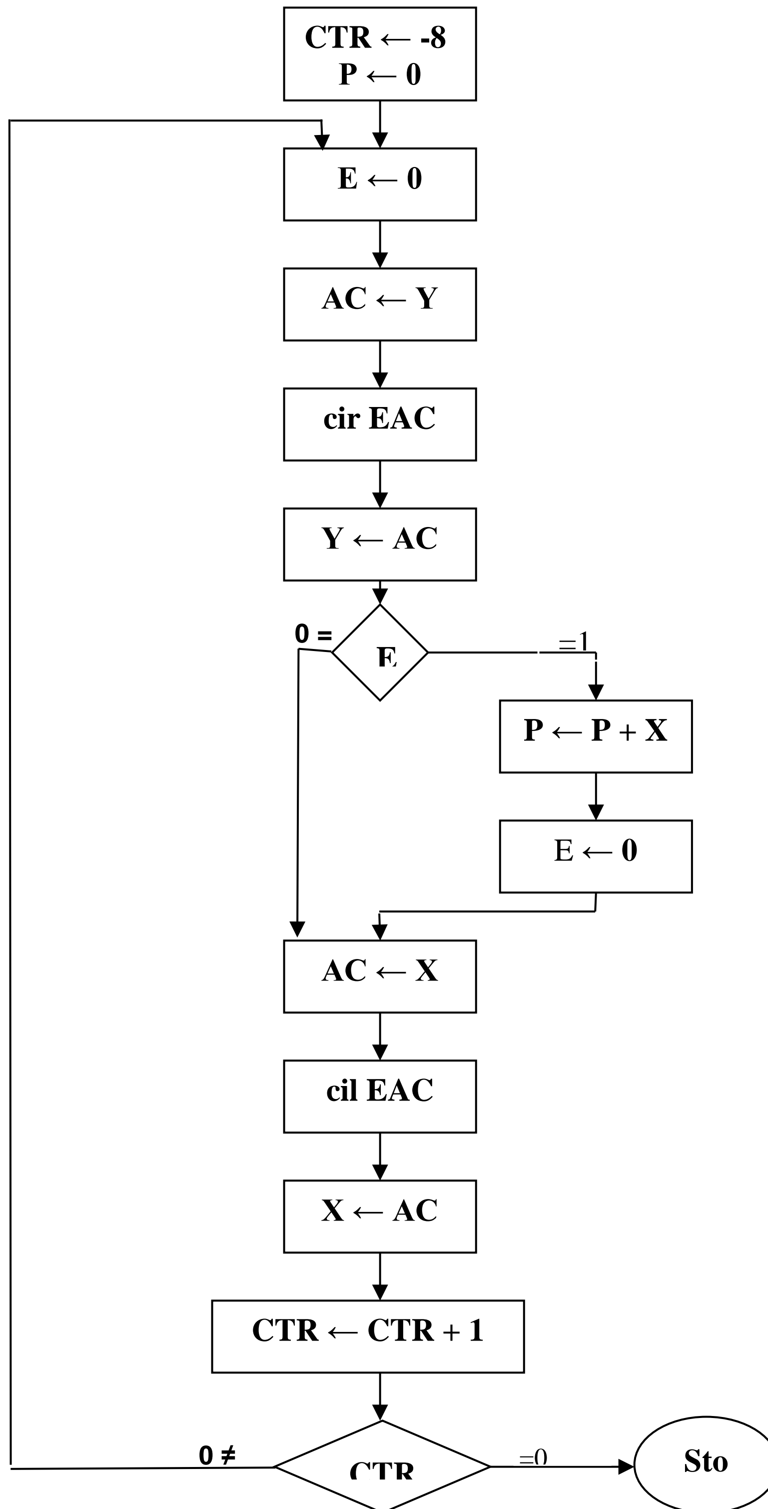

Figure 7.1

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.

The following program lists the instructions for multiplying two unsigned numbers X = 07 & Y = 0A.

```
        ORG 200
LOP,    CLE        /Clear E
        LDA Y      /Load multiplier
        CIR        /Shift multiplier bit to E
        STA Y      /Store shifted multiplier
        SZE        /Check if the shifted bit in E is zero
        BUN ONE    /If the bit is one; go to ONE
        BUN ZRO    /If the 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    /For counter not zero; repeat loop
        HLT        /If counter is zero; Halt
CTR,    DEC -8     /This location serves as a counter
X,      HEX 0007   /Location of multiplicand
Y,      HEX 000A   /Location of multiplier
P,      HEX 0      /Location of the product
        END
```

Note

The above program to be a general program for multiplying any two unsigned numbers; the initialization of the multiplicand and multiplier into locations X & Y respectively must be included. The initialization process also must include the initialization of the counter CTR to -8 and P to zero.

After generalizing the program according to the note above, **the resultant program will demonstrate the software implementation of the multiplication operation.**

2. Double-Precision Addition

A double precision number is a number that stored in two memory words of the memory.

To add two double precision numbers; first, every number is placed in two consecutive memory locations, **AL & AH** for the first number and **BL & BH** for the other; in which **AL & BL** holds the **16** least significant bits ($b_{15} – b_0$) and **BL & BH** holds the **16** most significant bits ($b_{31} – b_{16}$) of the two numbers.

The program for adding two numbers is listed below. First the two low-order portions are added and the carry transferred into **E**, the content of **AC** stored into the memory location **CL** (the **16** least significant bits of the sum). **AC** is cleared and the bit in **E** is circulated into the least significant bit of the accumulator **AC (0).** The two high-order portions are then added to the carry and the content of AC stored into the memory location **CH** (the **16** most significant bits of the sum).

```
     LDA AL   /load A low
     ADD BL   /Load B low, carry in E
     STA CL   /Store in C low
     CLA      /Clear AC
     CIL      /Circulate to bring carry into AC (0)
     ADD AH   /Add A high and carry
     ADD BH   /Add B high
     STA CH   /Store in C high
     HLT      /Halt the computer
AL,  xxxx     /Location of the low significant part of the 1st number
AH,  xxxx     /Location of the high significant part of the 1st number
BL,  yyyy     /Location of the low significant part of the 2nd number
BH,  yyyy     /Location of the high significant part of the 2nd number
CL,  -        /Location of the low significant part of the result
CH,  -        /Location of the high significant part of the result
```

3. Logic Operations

In chapter six it is shown that the basic computer has three machine instructions that perform logic operations: AND, CMA, and CLA. Also LDA instruction can be considered as logic operation that transfers a logic operand into the AC. Any logic function can be implemented using the AND and complement operation CMA.

Example

Write a program to forms the OR logic operation of two logic 16-bit operands A & B given that $A \vee B = \overline{(\overline{A} \cdot \overline{B})}$.

The Program

	LDA A	/load the first operand A into AC
	CMA	/Complement to get A complement $(\overline{A})$
	STA TEM	/Store in temporary location TEM
	LDA B	/load the second operand B into AC
	CMA	/Complement to get B complement $(\overline{B})$
	AND TEM	/AND with the content of TEM $(\overline{A} \wedge \overline{B})$
	CMA	/Complement the result to obtain **A** $\overline{(\overline{A}\overline{B})} = A \vee B$
A,	**xxxx**	/Location of the first operand
B,	**xxxx**	/Location of the second operand

In a similar manner, the other logic operations can be implemented.

4. Shift Operations

The basic computer as it was shown before contains **only** the circular-shift instructions (**CIR & CIL**) which executes the circular-shift operations.

The logical and arithmetic shift operations can be programmed with a number of the computer instructions as follows:

a. Logical Shift operations

These shift operations requires zeroes to be loaded to the extreme positions. This is accomplished by clearing **E** and circulating the **AC & E.**

Examples

(1) The **logical shift-right operation** can be implemented by the following two instructions:

```
CLE
CIR
```

(2) The **logical shift-left operation** can be implemented by the following two instructions:

```
CLE
CIL
```

b. Arithmetic Shift operations

These shift operations depends on how the negative numbers are represented. As mentioned before the basic computer uses the 2's complement representation for the negative numbers.

(1) Arithmetic right-shift

The rules for arithmetic right-shift states that the sign bit in the leftmost position remain unchanged and the sign bit itself is shifted into the high-order bit position of the number.

The program for the arithmetic right-shift requires setting **E** to the same value as the sign bit and circulate right as follows:

```
CLE    /Clear E to 0
SPA    /Skip if AC is positive; E remains 0
CME    /AC is negative; set E to 1
CIR    /Circulate E & AC
```

(2) Arithmetic left-shift

The rules for arithmetic left-shift are that the added bit in the least significant position must be 0. This is accomplished by clearing E before circulate-left operation. Also the sign bit must not change during this shift. After the circulate instruction is executed, the sign bit moves into E, then it is necessary to compare the sign bit with the value of E after the operation. If the two bits are equal, the arithmetic shift has been correctly implemented. If they are not equal, an overflow occurs (i.e. the original number was too large).

The simplified program for the arithmetic left-shift is as follows:

```
CLE   /Clear E to 0
CIL   /Circulate E & AC
```

7.5. Subroutines

A Subroutine can be defined as a set of instructions that can be used in a program many times.

A Subroutine has general characteristics such as:

1. It consists of a self –contained sequence of instructions that carries out a given task.
2. A branch can be made to the subroutine from any part of the main program.
3. All computers provide special instructions commonly called **subroutine call** to facilitate subroutine entry and return to the main program.
4. The procedure for branching to a subroutine and returning to the main program is referred to as a **subroutine linkage**.
5. The last instruction of the subroutine performs an operation commonly called **subroutine return**.

For basic computer the instruction **BSA** (Branch and Save return Address) is the link instruction between the main program and the subroutine.

Table 7.7 shows a program that demonstrates the use of subroutines. This program contains a subroutine that shifts the content of the accumulator four times to the left which is useful operation for

Location	Label	Instruction	Comments
		ORG 200	/Main program
200		LDA X	/Load X
201		BSA SH4	/Branch To subroutine
202		STA X	/Store shifted number
203		LDA Y	/Load Y
204		BSA SH4	/Branch to subroutine a second time
205		STA Y	/Store shifted number
206		HLT	/Halt the computer
207	X,	HEX 1234	
208	Y,	HEX 4321	
			/Subroutine to shift left 4 times
209	SH4,	HEX 0	/Store return address here
20A		CIL	/Circulate left once
20B		CIL	/Circulate left once
20C		CIL	/Circulate left once
20D		CIL	/Circulate left once
20E		AND MSK	/Set AC(3-0) to zero
20F		BUN SH4 I	/Return to main program
210	MSK,	HEX FFF0	/Mask operand
		END	

processing binary-coded decimal numbers or alphanumerical characters.

Table 7.7

How the program is executed

1. The first number (**HEX 1234**) contained at location **X** (**HEX 207**) is loaded to the accumulator **AC** by the instruction **LDA X.**
2. When the next instruction **BSA SH4** is executed, the control unit stores the return address (**HEX 202**) into the location defined by the symbolic address **SH4** (location **HEX 209**), also transfers the value of **SH4+1** (i.e. **HEX 20A**) into the program counter **PC.**
3. After execution of the instruction **BSA SH4**, the subroutine is executed starting from location **20A** (since this is the content of **PC** in the next fetch cycle).
   a. The four shift instructions (**CIL**) circulate the content of **AC** four times to the left.

- **b.** In order to accomplish a logical shift instruction, the four least significant bits must be zero. This is achieved by masking **FFF0** with the content of **AC** using the **AND** operation (the instruction at location **HEX 20E**).
- **c.** The last instruction **BUN SH4 I** (indirect branch unconditional instruction) returns the computer to the main program (i.e. to location **HEX 202).**

4. The main program continues by storing the shifted number into location **X**.

5. A new number (**HEX 4321**) from location **Y** (**HEX 208**) is then loaded into the **AC** and another branch is made to the subroutine. The same procedure will be repeated, but in this case the location **SH4** will contain the return address **HEX 205** since this is now the location of the next instruction after **BSA.** After the subroutine is executed the subroutine returns to the main program at location **205.**

6. When the execution of the program is completed the contents of locations **207 & 208** are **HEX 2340 & HEX 3210** respectively.

7.6. Input-Output Programming

The character is stored in the computer as an 8-bit code. The character enters the computer by an **INP** instruction and transferred to the output device using an **OUT** instruction.

The following set of instructions, input a character and stores it in memory.

```
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
```

The program starts with the instruction **SKI** which checks the input flag to see if a character is available to transfer.
If the input flag is **1**, the next instruction **BUN** is skipped. The INP instruction transfers the binary-coded character into **AC (0-7).** By the **OUT** instruction, the character is printed by the output device.
If the input flag is **0**, the next instruction **BUN** is executed and a branch to return and check the input flag again.

The two instructions **SKI** & **BUN** will be executed many times before a character is transferred into the accumulator, because the input device is much slower than the computer.

The following set of instructions, print a character initially stored in memory.

```
      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
```

The program starts with the instruction **LDA** to load the binary-coded character into the AC from the memory location **CHR**. The instruction **SKO** checks the output flag, if its 1, the character is transferred from AC to the printer. If the output flag is 0, the computer remains in a two instruction loop (**SKO** & **BUN**) checking the flag bit.

Character Manipulation

To pack two characters in one memory word for the purpose of character manipulation, the subroutine named **IN2** that inputs two characters and packs them into the **AC** is shown below.

```
IN2,  -          /Subroutine entry
FST,  SKI        /check input flag
      BUN FST    /flag = 0, branch to check again
      INP        /flag = 1, input character
      OUT        /output character
      BSA SH4    /Shift left four times
      BSA SH4    /Shift left four times
SCD,  SKI        /check input flag
      BUN SCD    /flag = 0, branch to check again
      INP        /flag = 1, input character
      OUT        /output character
      BUN IN2 I  /Return
```

It is clear that this subroutine calls twice the subroutine **SH4** to shift the accumulator left eight times.

Program Interrupt

The interrupt facility is useful when two or more programs reside in the memory at the same time.
Even though two or more programs may reside in the computer memory, only one program can be executed at a given time and this program is referred to as the **running program**. The other programs are usually waiting for input or output data.
The function of the interrupt facility is to take care of the data transfer of one (or more) program while another program is currently being executed. The running program must include an **ION** instruction to turn the interrupt on. If the interrupt facility is not used, the program must include an **IOF** instruction to turn it off.
The interrupt facility allows the running program to proceed until the input or output device sets its ready flag.
Whenever a flag is set to **1**, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the return address is stored in location **0**. The instruction in location **1** is then performed; this initiates a service routine for the input or output transfer.
The service routine can be stored anywhere in memory provided a branch to the start of the routine is stored in location **1**.
The service routine must have instructions to perform the following tasks: -

1. Save the contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

The following program listed in table 7.8 explains an interrupt services.

1. Location **0** is reserved for the return address.
2. Location **1** has a branch instruction to the beginning of the service routine **SRV**.
3. The **ION** instruction found in the main program turns the interrupt on.
4. It is supposed an interrupt occurs while the computer is executing the instruction in location **103**.

5. The interrupt cycle stores the address of the following instruction (HEX 104) in location 0 and branches to location 1.
6. The branch instruction in location 1 branches to the service routine SRV.
7. The service routine performs the six tasks listed above.

Table 7.8

Location	Label	Instruction	Comment
0	ZRO,	-	/return address stored here
1		BUN SRV	/branch to service routine
100		CLA	/portion of running program
101		ION	/turn on interrupt facility
102		LDA X	
103		ADD Y	/interrupt occur here
104		STA Z	/program returns here after interrupt
.		.	
.		.	
.			/interrupt service here
200	SRV,	STA SAC	/store content of AC
		CIR	/move E into AC (15)
		STA SE	/store content of E
		SKI	/check input flag
		BUN NXT	/flag is off, check next flag
		INP	/flag is on, input character
		OUT	/print character
		STA PT1 I	/store it in input buffer
		ISZ PT1	/increment input pointer
	NXT,	SKO	/check output flag
		BUN EXT	/flag is off, exit
		LDA PT2 I	/load character from output buffer
		OUT	/output character
		ISZ PT2	/increment output pointer
	EXT,	LDA SE	/restore value of AC (15)
		CIL	/shift it to E
		LDA SAC	/restore content of AC
		ION	/turn interrupt on
		BUN ZRO I	/return to running program
	SAC,	-	/AC is stored here
	SE,	-	/E is stored here
	PT1,	-	/pointer of input buffer
	PT2,	-	/pointer of output buffer

A typical computer may have many more input and output devices connected to the interrupt facility.