Programming the Basic Computer

Introduction

- A computer system includes both hardware and software. The designer should be familiar with both of them.
- This chapter introduces some basic programming concepts and shows their relation to the *hardware representation* of instructions.
- A program may be : dependent or independent on the computer that runs it.

Instruction Set of the *Basic Computer*

Symbol	***Hex code***	***Description***
– **AND**	**0 or 8**	**AND M to AC**
– **ADD**	**1 or 9**	**Add M to AC, carry to E**
– **LDA**	**2 or A**	**Load AC from M**
– **STA**	**3 or B**	**Store AC in M**
– **BUN**	**4 or C**	**Branch unconditionally to m**
– **BSA**	**5 or D**	**Save return address in m and branch to m+1**
– **ISZ**	**6 or E**	**Increment M and skip if zero**
– **CLA**	**7800**	**Clear AC**
– **CLE**	**7400**	**Clear E**
– **CMA**	**7200**	**Complement AC**
– **CME**	**7100**	**Complement E**
– **CIR**	**7080**	**Circulate right E and AC**

Contd..

- CIL 7040 Circulate left E and AC
- INC 7020 Increment AC, carry to E
- SPA 7010 Skip if AC is positive
- SNA 7008 Skip if AC is negative
- SZA 7004 Skip if AC is zero
- SZE 7002 Skip if E is zero
- HLT 7001 Halt computer
- INP F800 Input information and clear flag
- OUT F400 Output information and clear flag
- SKI F200 Skip if input flag is on
- SKO F100 Skip if output flag is on
- ION F080 Turn interrupt on
- IOF F040 Turn interrupt off

Machine Language

- Program: A list of instructions that direct the computer to perform a data processing task.
- Many programming languages (C++, JAVA). However, the computer executes programs when they are represented internally in binary form.
- So any program written in any other language (like C, BASIC, Java etc.) must be translated to the binary representation of instructions before they can be executed by the computer.
- Machine language program is a binary program that runs on a computer directly with out the need of assembler or compiler.
- Machine Language:
  - Binary code: a binary representation of instructions and operands as they appear in computer memory.
  - Octal or hexadecimal code: translation of binary code to octal or hexadecimal representation.

Program Categories

- Binary code:
  - sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.
- Octal or hexadecimal code:
  - an equivalent translation of the binary code to octal or hexadecimal representation
- Symbolic code (assembly code):
  - User employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code.
  - Each symbolic instruction can be translated into one binary coded instruction.
  - This is done by assembler.

Contd..

- High-level programming language:
  - special language developed to reflect the procedures used in the solution of a problem.
  - Example: FORTRAN, C, JAVA.
  - The program is written in a sequence of statements in a form that people prefer to think in when solving a program.
  - Each statement must be translated into a sequence of binary instructions before the program can be executed in a computer.
  - This is done by compiler or interpreter.

Example

- Binary program to add two number

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

- Hexadecimal program to add two number

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

Example

- Assembly program to add two numbers

Location	Instruction	Comments
000	LDA 004	Load first operand into *AC*
001	ADD 005	Add second operand to *AC*
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

- Assembly program in which we replace each hexadecimal address by symbolic address and each hexadecimal operand by a decimal operand

	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 83	/Decimal operand
B,	DEC −23	/Decimal operand
C,	DEC 0	/Sum stored in location *C*
	END	/End of symbolic program

Assembly language

- The rules for writing assembly language program
  - Documented and published in manuals(from the computer manufacturer)
- Rules of the Assembly Language
  - Each line of an assembly language program is arranged in three columns called fields. The fields specify the following information:
    - 1) Label field : empty or symbolic address
    - 2) Instruction field : machine instruction or pseudo instruction
    - 3) Comment field : empty or comment
- Example:

Lab,	ADD op1	/ this is an add operation.
Label	**Instruction**	**comment**

**Note that Lab is a symbolic address.

Contd..

- Symbolic Address(***Label field***)
  - One, two, or three, but not more than three alphanumeric characters
  - The first character must be a letter; the next two may be letters or numerals
  - A symbolic address is terminated by a comma(***recognized as a label by the assembler***)
- Instruction Field
  - 1) A memory-reference instruction(***MRI***)
    - » Example: **ADD OPR** (*direct address MRI*), **ADD PTR I**(*indirect address MRI*)
  - 2) A register-reference or input-output instruction(***non-MRI***)
    - » Example: **CLA**(*register-reference*), **INP**(*input-output*)
  - 3) A pseudo instruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation
    - » Example: ORG, End

Example

- Assembly language program to subtract two numbers

```
        ORG 100     /Origin of program is location 100
        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 83      /Minuend
SUB,    DEC −23     /Subtrahend
DIF,    HEX 0       /Difference stored here
        END         /End of symbolic program
```

The Assembler

- An Assembler is a program that accepts a symbolic language and produces its binary machine language equivalent.
- The input symbolic program :*Source program.*
- The resulting binary program: Obj*ect program.*
- Prior to assembly, the program must be stored in the memory.
- A line of code is stored in consecutive memory locations with two characters in each location. (each character 8 bits)→ memory word 16 bits

First Pass

- The assembler scans the symbolic program twice.
- First pass: generates an "Address Symbol Table" that connects all user-defined address symbols with their binary equivalent value.
- Second Pass: Binary translation
- To keep track of instruction locations: the assembler uses a memory word called a *location counter (LC).*
- LC stores the value of the memory location assigned to the instruction or operand presently being processed.
- LC is initialized to the first location using the ORG pseudo instruction. If there is no ORG → LC = 0

First Pass (Contd..)

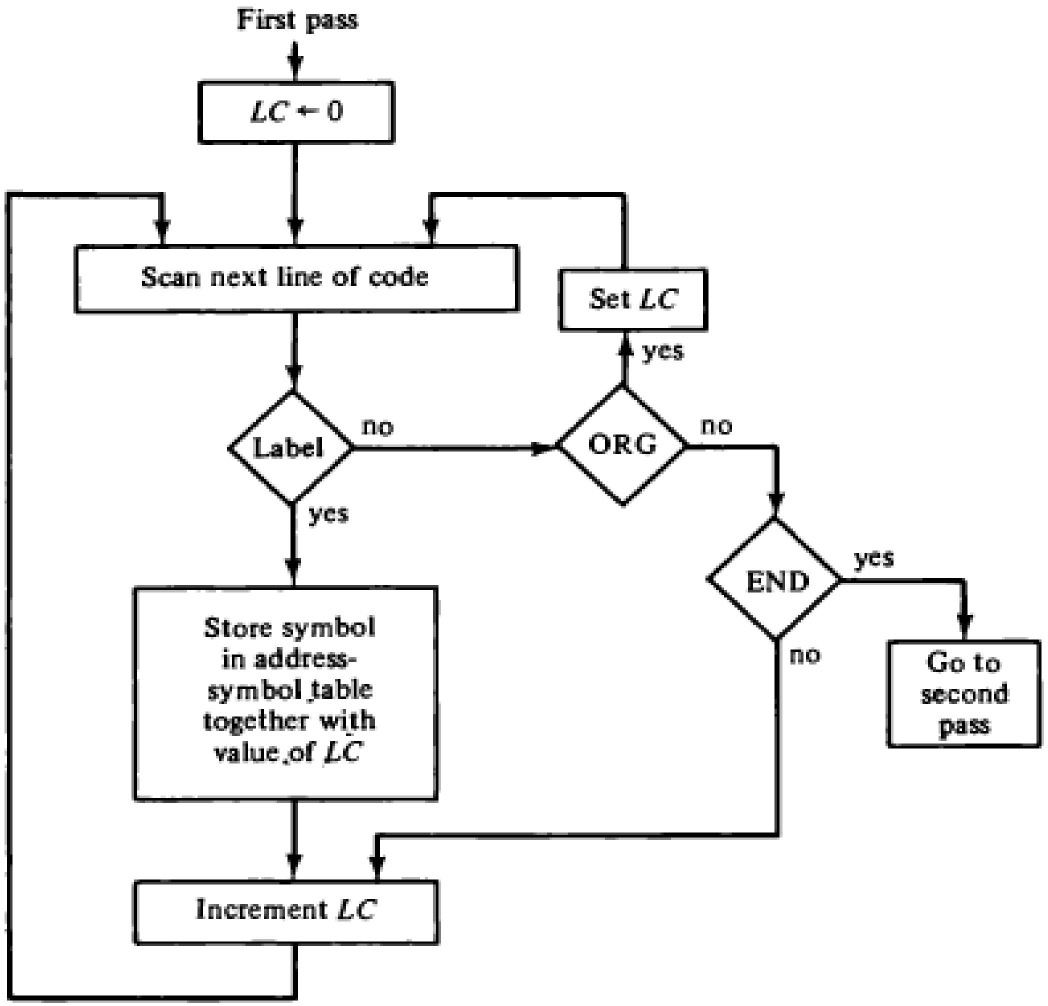


Figure Flowchart for first pass of assembler.

Second Pass

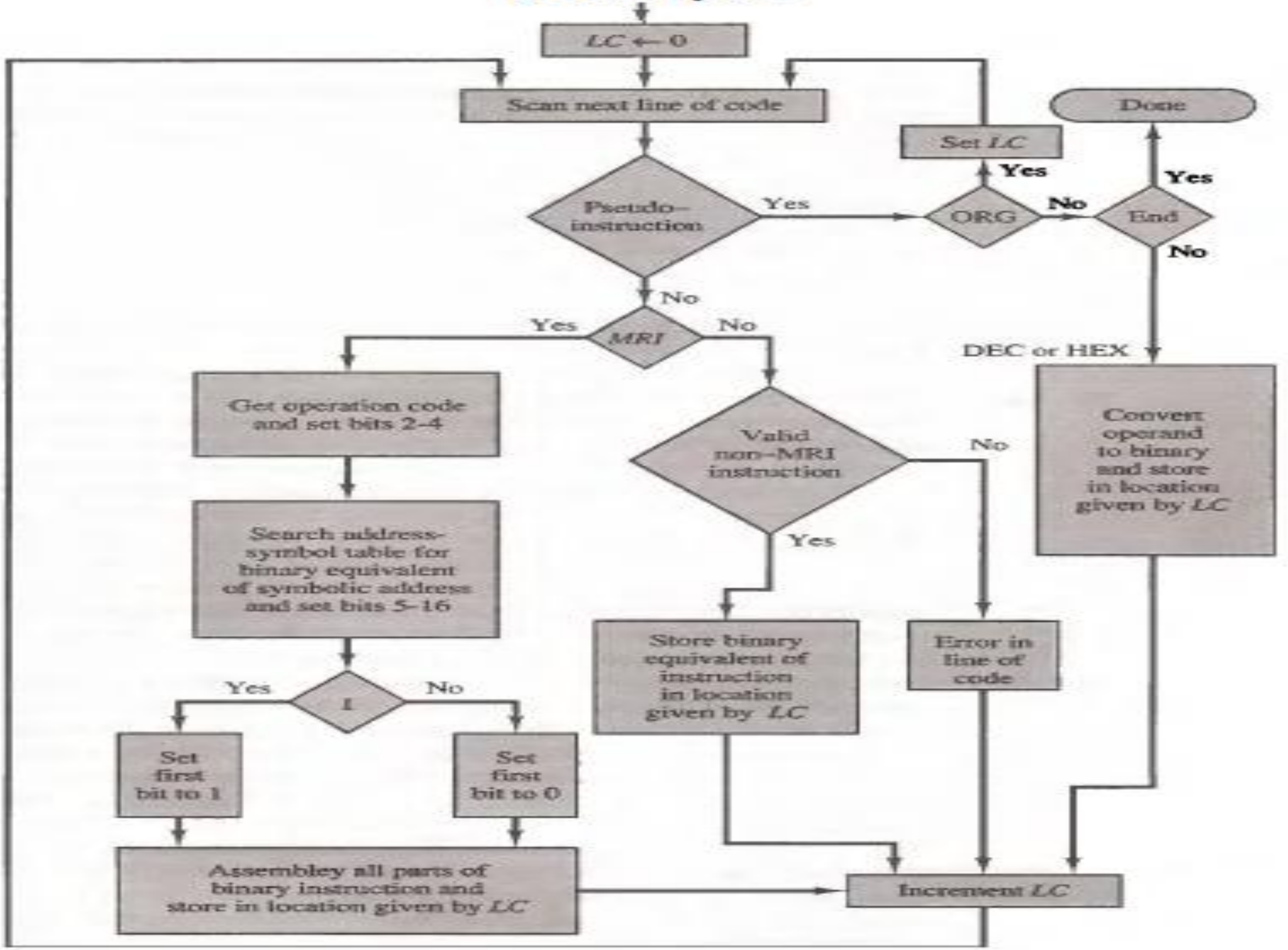
- Machine instructions are translated in this pass by means of a *table lookup* procedure.

- A search of table entries is performed to determine whether a specific item matches one of the items stored in the table.

Assembler Tables

- Four tables are used:
  - Pseudoinstruction table. (ORG, DEC, HEX, END)
  - MRI table. (7 symbols for memory reference and 3-bit opcode equivalent)
  - Non-MRI table. (18 Reg. & I/O instruction and 16-bit binary code equivalent)
  - Address symbol table. (Generated during first pass)

Second pass



LC ← 0
Scan next line of code
Done
Set LC
Yes
Pseudo-instruction
Yes
ORG
No
End
Yes
No
No
Yes
MRI
No
DEC or HEX
Get operation code and set bits 2-4
Valid non-MRI instruction
No
Convert operand to binary and store in location given by LC
Search address-symbol table for binary equivalent of symbolic address and set bits 5-16
Yes
Yes
I
No
Store binary equivalent of instruction in location given by LC
Error in line of code
Set first bit to 1
Set first bit to 0
Assembley all parts of binary instruction and store in location given by LC
Increment LC

Error Diagnostics

- One important task of the assembler is to check for possible errors in the symbolic program.

Example:

- Invalid machine code symbol.
- A symbolic address that did not appear as a label.

Program Loops

- A sequence of instructions that are executed many times, each time with a different set of data
- Fortran program to add 100 numbers:

```
      DIMENSION A(100)
      INTEGER SUM, A
      SUM = 0
      DO 3 J = 1, 100
3     SUM = SUM + A(J)
```

TABLE Symbolic Program to Add 100 Numbers

Line			
1		ORG 100	/Origin of program is HEX 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 operand
•			
•			
•			
118		DEC 23	/Last operand
119		END	/End of symbolic program

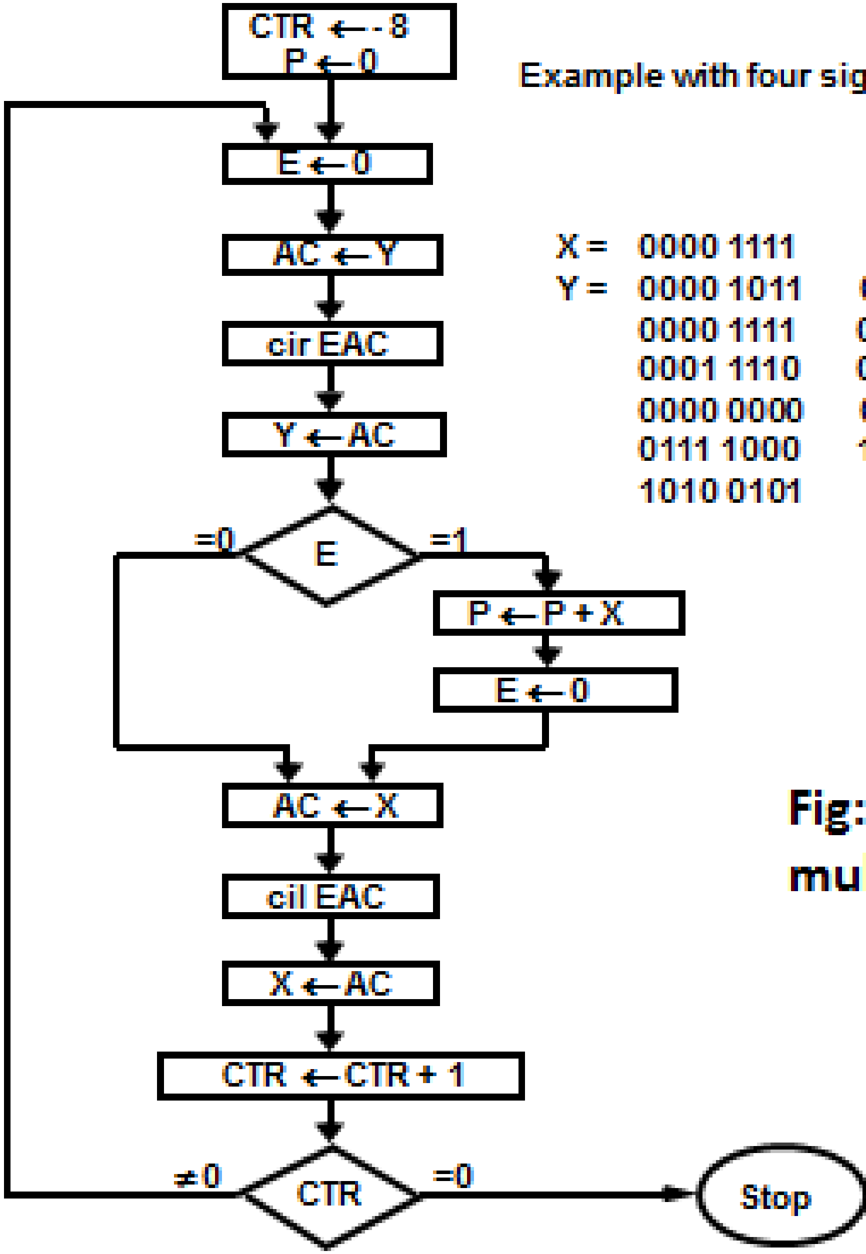
Programming Arithmetic & Logic Operations

- Software Implementation
  - Implementation of an operation with a program using machine instruction set
  - Usually used: when the operation is not included in the instruction set

- Hardware Implementation
  - Implementation of an operation in a computer with one machine instruction

Multiplication

- We will develop a program to multiply two numbers.
- Assume positive numbers and neglect the sign bit for simplicity.
- Also, assume that the two numbers have no more than 8 significant bits → 16-bit product.



Example with four significant digits

		P
X =	0000 1111	
Y =	0000 1011	0000 0000
	0000 1111	0000 1111
	0001 1110	0010 1101
	0000 0000	0010 1101
	0111 1000	1010 0101
	1010 0101	

X holds the multiplicand
Y holds the multiplier
P holds the product

Fig: Flowchart for multiplication program

TABLE Program to Multiply Two Positive Numbers

```
        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
```

Double-Precision Addition

- When two 16-bit unsigned numbers are multiplied, the result is a 32-bit product that must be stored in two memory words.
- A number stored in two memory words is said to have double precision.
- When a partial product is computed, it is necessary to add a double-precision number to the shifted multiplicand, which is also double-precision.
- This provides better accuracy
- One of the double precision numbers is stored in two consecutive memory locations, AL & AH. The other number is placed in BL & BH.
- The two low-order portions are added and the carry is transferred to E. The AC is cleared and E is circulated into the LSB of AC.
- The two high-order portions are added and the sum is stored in CL & CH.

Double-Precision Addition

TABLE Program to Add Two Double-Precision Numbers

```
        LDA AL       /Load A low
        ADD BL       /Add B low, carry in E
        STA CL       /Store in C low
        CLA          /Clear AC
        CIL          /Circulate to bring carry into AC(16)
        ADD AH       /Add A high and carry
        ADD BH       /Add B high
        STA CH       /Store in C high
        HLT
AL,     —            /Location of operands
AH,     —
BL,     —
BH,     —
CL,     —
CH,     —
```

Logic Operations

- All 16 logic operations (table 4-6) can be implemented using the AND & complement operations.
- Example: OR : x + y = (x'.y')' *De-morgan's*

```
LDA   A          / Load 1st operand
CMA              / Complement to get A'
STA   TMP        / Store in a temporary location
LDA   B          / Load 2nd operand B
CMA              / Complement to get B'
AND   TMP        / AND with A' to get A' AND B'
CMA              / Complement again to get A OR B
```

Shift Operations

- The circular shift operations are machine instructions in the basic computer.
- Logical and Arithmetic shifts can be programmed with a small number of instructions.

Logical Shift Operations

- Logical shift right

  CLE

  CIR
- Logical shift left

  CLE

  CIL

Arithmetic Shift Operations

- Arithmetic shift right: it is necessary that the sign bit in the leftmost position remain unchanged. But the sign bit itself is shifted into the high-order bit position of the number.

```
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 and AC
```

Arithmetic Shift Operations /cont.

- Arithmetic shift left: it is necessary that the added bit in the LSB be 0.

  CLE
  CIL

- The sign bit must not change during this shift.
- With a circulate instruction, the sign bit moves into E.

Arithmetic Shift Operations /cont.

- The sign bit has to be compared with E after the operation to detect overflow.
- If the two values are equal → No overflow.
- If the two values are not equal → Overflow.

Subroutines

- The same piece of code might be written again in many different parts of a program.
- Write the common code only once.
- ***Subroutines :***A set of common instructions that can be used in a program many times
- Each time a subroutine is used in the main program, a branch is made to the beginning of the subroutine. The branch can be made from any part of the main program.

Subroutines /cont.

- After executing the subroutine, a branch is made back to the main program.
- It is necessary to store the return address somewhere in the computer for the subroutine to know where to return.
- In the basic computer, the link between the main program and a subroutine is the BSA instruction.

Subroutines example- (CIL 4 times)

Loc.			
		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 shift 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	

Subroutines /cont.

- 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 the *call.*
- The BUN instruction performs the return.

Subroutine Parameters and Data Linkage

- When a subroutine is called, the main program must transfer the data it wishes the subroutine to work with.
- It is necessary for the subroutine to have access to data from the calling program and to return results to that program.
- The accumulator can be used for a *single* input parameter and a *single* output parameter.

Subroutine Parameters and Data Linkage /cont.

- In computers with multiple processor registers, more parameters can be transferred this way.

- Another way to transfer data to a subroutine is through the memory.

- Data are often placed in memory locations following the call.

Parameter Linkage

```
Loc.            ORG  200
200             LDA  X          / Load 1st operand into AC
201             BSA  OR         / Branch to subroutine OR
202             HEX  3AF6       / 2nd operand stored here
203             STA  Y          / Subroutine returns here
204             HLT
205     X,      HEX  7B95       / 1st operand stored here
206     Y,      HEX  0          / Result stored here
207     OR,     HEX  0          / Subroutine OR
208             CMA             / Complement 1st operand
209             STA  TMP        / Store in temporary location
20A             LDA  OR  I      / Load 2nd operand
20B             CMA             / Complement 2nd operand
20C             AND  TMP        / AND complemented 1st operand
20D             CMA             / Complement again to get OR
20E             ISZ  OR         / Increment return address
20F             BUN  OR  I      / Return to main program
210     TMP,    HEX  0          / Temporary storage
                END
```

Subroutine Parameters and Data Linkage /cont.

- It is possible to have more than one operand following the BSA instruction.

- The subroutine must increment the return address stored in its first location for each operand that it extracts from the calling program.

Data Transfer

- If there is a large amount of data to be transferred, the data can be placed in a block of storage and the address of the first item in the block is then used as the linking parameter.

```
SUBROUTINE  MVE (SOURCE, DEST, N)
DIMENSION  SOURCE(N), DEST(N)
DO  20  I = 1, N
DEST(I) = SOURCE(I)
RETURN
END
```

Data transfer

```
                        / Main program
        BSA  MVE        / Branch to subroutine
        HEX  100        / 1st address of source data
        HEX  200        / 1st address of destination data
        DEC  -16        / Number of items to move
        HLT
MVE,    HEX  0          / Subroutine MVE
        LDA  MVE  I     / Bring address of source
        STA  PT1        / Store in 1st pointer
        ISZ  MVE        / Increment return address
        LDA  MVE  I     / Bring address of destination
        STA  PT2        / Store in 2nd pointer
        ISZ  MVE        / Increment return address
        LDA  MVE  I     / Bring number of items
        STA  CTR        / Store in counter
        ISZ  MVE        / Increment return address
LOP,    LDA  PT1  I     / Load source item
        STA  PT2  I     / Store in destination
        ISZ  PT1        / Increment source pointer
        ISZ  PT2        / Increment destination pointer
        ISZ  CTR        / Increment counter
        BUN  LOP        / Repeat 16 times
        BUN  MVE  I     / Return to main program
PT1,    --
PT2,    --
CTR,    --
```

Input-Output Programming

- Users of the computer write programs with symbols that are defined by the programming language used.

- The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in a computer memory.

Input-Output Programming /cont.

- A binary coded character enters the computer when an INP instruction is executed.

- A binary coded character is transferred to the output device when an OUT instruction is executed.

Character Input

Program to Input one Character(Byte)

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

Character Output

Program to Output a 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"
```

Character Manipulation

- The binary coded characters that represent symbols can be manipulated by computer instructions to achieve various data-processing tasks.
- One such task may be to pack two characters in one word.
- This is convenient because each character occupies 8 bits and a memory word contains 16 bits.

Subroutine to Input 2 Characters and pack into a word

```
IN2,    --              / Subroutine entry
FST,    SKI
        BUN  FST
        INP             / Input 1st character
        OUT
        BSA  SH4        / Logical Shift left 4 bits
        BSA  SH4        / 4 more bits
SCD,    SKI
        BUN  SCD
        INP             / Input 2nd character
        OUT
        BUN  IN2  I     / Return
```

Table lookup

- A two pass assembler performs the table lookup in the second pass.

- This is an operation that searches a table to find out if it contains a given symbol.

- The search may be done by comparing the given symbol with each of the symbols stored in the table.

Table lookup /cont.

- The search terminates when a match occurs or if none of the symbols match.
- The comparison is done by forming the 2's complement of a word and arithmetically adding it to the second word.
- If the result is zero, the two words are equal and a match occurs. Else, the words are not the same.

Table Lookup / cont.

Comparing two words:

```
        LDA WD1     / Load first word
        CMA
        INC         / Form 2's complement
        ADD WD2     / Add second word
        SZA         / Skip if AC is zero
        BUN UEQ     / Branch to "unequal" routine
        BUN EQL     / Branch to "equal" routine
WD1,    ---
WD2,    ---
```

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 wait loop that checks the flags wastes a large amount of time.
- Use interrupt facility to notify the computer when a flag is set → eliminates waiting time.

Program Interrupt /cont.

- Data transfer starts upon request from the external device.
- Only one program can be executed at any given time.
- Running program: is the program currently being executed
- The interrupt facility allows the running program to proceed until the input or output device sets its ready flag

Program Interrupt /cont.

- When a flag is set to 1: the computer completes the execution of the instruction in progress and then acknowledges the interrupt.
- The return address is stored in location 0.
- The instruction in location 1 is 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.

Service Routine

- Must have instructions to perform:
  - Save contents of processor registers.
  - Check which flag is set.
  - Service the device whose flag is set.
  - Restore contents of processor registers
  - Turn the interrupt facility on.
  - Return to the running program.

Service Routine /cont.

- The contents of processor registers before and after the interrupt must be the same.

- Since the registers may be used by the service routine, it is necessary to save their contents at the beginning of the routine and restore them at the end.

Service Routine /cont.

- The sequence by which flags are checked dictates the priority assigned to each device.
- The device with higher priority is serviced first.
- Even though two or more flags may be set at the same time, the devices are serviced on at a time.

Service Routine /cont.

- The occurrence of an interrupt disables the facility from further interrupts.

- The service routine must turn the interrupt on before the return to the running program.

- The interrupt facility should not be turned on until after the return address is inserted into the program counter.

Interrupt Service Program

```
Location
   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 occurs here
 104             STA  Z             / Program returns here after interrupt

                                    / Interrupt service routine
 200     SRV,    STA  SAC           / Store content of AC
                 CIR                / Move E into AC(1)
                 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(1)
                 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
```