Unit 3

Assembly Level Programming

Introduction

- A computer system includes both hardware and software. The designer should be familiar with both of them.
- This unit 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
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.
- **Categories of program written in computer:-**
- **1. Binary code:** a binary representation of instructions and operands as they appear in computer memory. (electronic signals given, not easily understood by human)
- **2. Octal or hexadecimal code:** translation of binary code to octal or hexadecimal representation. (another category of program)

Binary/Hex Code

- Binary Program to Add Two Numbers:

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

Binary code

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

Hexadecimal code

- It is hard to understand the task of the program

➔ symbolic code

Hierarchy of programming languages

- **3. Symbolic code:-** The user uses symbols (Letter, numerals, or special characters) for the operation, address and other parts of the instruction code (Symbolic code).
- symbolic code → binary coded instruction
- The translation is done by a special program called an ***assembler***
- **Assembly language:** concerned with the computer hardware behavior.

Symbolic OP-Code

Location	Instruction	Comments
000	LDA 004	Load 1st operand into AC stored at address 004
001	ADD 005	Add 2nd operand to AC stored at address 005
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	1st operand
005	FFE9	2nd operand (negative)
006	0000	Store sum here

- Symbolic names of instructions instead of binary or hexadecimal code.
- The address part of memory-reference and operands ➔ remain hexadecimal.
- Symbolic programs are easier to handle.

- **4. High-level programming language:**
  C, C++,FORTRAN : used to write procedures to solve a problem or implement a task.
- More understandable by user, without considering hardware behavior
- EXAMPLE (FORTRAN)
- **INTEGER A,B,C**
- **DATA A, 83 B,-24**
- **C=A+B**
- **END**

Summary : Machine Language

- **Program:-**
- A list of instructions or statements for directing the computer to perform a required data processing task Various types of programming languages
- **Hierarchy of programming languages**
- **Machine-language**

- Binary code
- Octal or hexadecimal code

- **Assembly-language (Assembler)**

- Symbolic code

- **High-level language (Compiler)**

COMPARISON OF PROGRAMMING LANGUAGES

• Binary Program to Add Two Numbers

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

• Hexa program

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

• Program with Symbolic OP-Code

Location		Instruction	Comments
000	LDA	004	Load 1st operand into AC
001	ADD	005	Add 2nd operand to AC
002	STA	006	Store sum in location 006
003	HLT		Halt computer
004	0053		1st operand
005	FFE9		2nd operand (negative)
006	0000		Store sum here

• Assembly-Language Program

```
      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
```

• Fortran Program

```
INTEGER  A, B, C
DATA  A,83 / B,-23
C = A + B
END
```

Assembly-Language Program

- A programming language is defined by set of rules
- Almost every commercial computer has its own particular assembly language.
- All formal rules of the language must be conformed in order to translate the program correctly.
- Rules for writing program in assembly language are documented and published in manuals
- Symbols can be combined to write line of code

Assembly-Language Program

- Rules of the assembly language of the Basic Computer
- Each lines of assembly language program is arranged in three columns called fields, the fields are:
  - 1. **The label field** may be empty or it may specify a symbolic address
  - 2. **The instruction field** specifies a machine instruction of pseudo instruction.
  - 3. **The comment field** may be empty or it may include a comment, which must be preceded by a slash i.e. '/'.

  Example:
  ORG Lab
  Lab, ADD op1 / this is an add operation.
  Label Instruction comment

- **1. Symbolic address:**
- Consists of one, two, or three (maximum) alphanumeric characters.
- The first character must be a letter, the next two may be letters or numerals.
- A symbolic address in the label field is terminated by a comma so that it will be recognized as a label by the assembler.
- Examples: Which of the following is a valid symbolic address:
- r2 : Yes
- Sum5: No
- tmp : Yes.

- **2. Instruction field:**
- The instruction field contains:

  1- Memory-Reference Instruction (MRI)

  2- A register-reference or I/O instruction (non-MRI)

  3- A pseudo instruction with or without an operand.

  Ex: ORG, DEC 83

1-Memory-Reference Inst.

- Occupies two or three symbols separated by spaces.
- The first must be a three-letter symbol defining an MRI operation code. (ADD)
- The second is a symbolic address.

  Ex: ADD OPR ***direct address MRI***
- The third (optional)-Letter I to denote an indirect address instruction

  Ex: ADD OPR I ***Indirect address MRI***
- A symbolic address in the instruction field specifies the memory location of the operand.
- This location MUST be defined somewhere in the program by appearing ***again*** as a label in the first column.

2-Non-Memory-Reference Inst.

- Does not have an address part.
- It is recognized in the instruction field by one of the three-letter symbols (CLA, INC, CLE,..).

3-Pseudo instruction

- Not a machine instruction
- It is an instruction to the assembler giving information about some phase of the translation from assembly level to machine level program

Ex:

ORG N

Hexadecimal number N is the memory loc. for the instruction or operand listed in the following line (start of program)

END

Denotes the end of symbolic program (end of assembly level program)

DEC N

Signed decimal number N to be converted to binary

HEX N

Hexadecimal number N to be converted to binary

Comment Field

- A line of code may or may not have a comment. (Ex: STA A0 / storing at A0)

- A comment must be preceded by a slash for the assembler to recognize the beginning of the comment field.

Example of Assembly level program

- **An assembly language program to subtract two numbers**

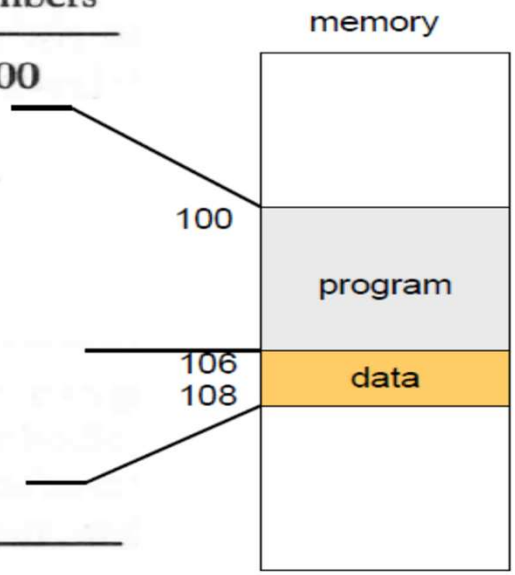
```
        ORG  100     / Origin of program is location 100
        LDA  SUB     / Load subtrahend to AC (SUB is symbol replaced for memory location )
        CMA          / Complement AC
        INC          / Increment AC
        ADD  MIN     / Add minuend to AC
        STA  DIF     / Store difference
        HLT          / Halt computer (no further instruction will be executed)
MIN,    DEC  83      / Minuend
SUB,    DEC  -23     / Subtrahend
DIF,    HEX  0       / Difference stored here
        END          / End of symbolic program

Label   Instruction  comment
```

- An example assembly language program:

(Mano 1993)

TABLE 6-8 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
```



converted into a binary number of signed 2's complement form (by the assembler)

Translation to Binary:

- Translation to binary is done by **an assembler.**
- An assembler is a computer program for translating assembly language essentially, a mnemonic representation of machine language — into object code.
- A program will be scanned and machine code will be written for symbols in program
- first ORG is encountered which tells us program starts at location 100
- Second line has two symbols LDA and SUB as I is missing it is direct address memory reference instruction, SUB is address symbol

TRANSLATION TO BINARY

Hexadecimal Code		*Symbolic Program*	
Location	*Content*		
			ORG 100
100	2107		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC -23
108	0000	DIF,	HEX 0
			END

TABLE 6-1 Computer Instructions

Symbol	Hexadecimal 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
CIL	7040	Circulate left E and AC
INC	7020	Increment AC,
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

Address Symbol Table

- The translation process can be simplified if we scan the entire symbolic program twice.
- No translation is done during the first scan. We just assign a memory location to each instruction and operand.
- Such assignment defines the address value of labels and facilitates the translation during the second scan.
- ORG & END are not assigned a numerical location because they do not represent an instruction or operand.

Address symbol	*Hex Address*
MIN	106
SUB	107
DIF	108

Example

LDA SUB

Address mode: direct → I=0

Instruction: LDA → 010

Address : SUB → 107

Instruction 0 010 107 → 2107

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

Representation of Symbolic Program in Memory

- user types the symbolic program on a terminal.
- A loader program is used to input the characters of the symbolic program into memory.
- Since user inputs symbols, program's representation in memory uses alphanumeric characters (8-bit ASCII).
- A line of code is stored in consecutive memory locations with two 8-bit characters in each location (we have 16-bit wide memory).
- End of line is recognized by the CR code.

TABLE 6-10 Hexadecimal Character Code (Mano 1993)

Character	Code	Character	Code	Character	Code	
A	41	Q	51	6	36	
B	42	R	52	7	37	
C	43	S	53	8	38	
D	44	T	54	9	39	
E	45	U	55	space	20	
F	46	V	56	(	28	
G	47	W	57	)	29	
H	48	X	58	*	2A	
I	49	Y	59	+	2B	
J	4A	Z	5A	,	2C	
K	4B	0	30	−	2D	
L	4C	1	31	.	2E	
M	4D	2	32	/	2F	
N	4E	3	33	=	3D	
O	4F	4	34	CR	0D	(carriage return)
P	50	5	35			

- E.g. a line of code: PL3, LDA SUB I is stored in seven consecutive memory locations

(Mano 1993)

TABLE 6-11 Computer Representation of the Line of Code: PL3, LDA SUB I

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

- Each symbol is terminated by the code for space (0x20) except last, which is terminated by the code of carriage return (0x0D).
- If a line of code has a comment, the assembler recognizes it from code 0x2F (slash): assembler ignores all characters in the comment field and keeps checking for a CR code.
- The input for the assembler program is the user's symbolic language program in ASCII.
- The binary program is the output generated by the assembler.

- A two-pass assembler scans the entire symbolic program twice
- **First pass:** address table is generated for all address symbols with their binary equivalent value .(symbols are stored in symbol table along with their location)
- **Second pass:** binary translation with the help of address table generated during the first pass.
- To keep track of the location of instructions, the assembler uses a memory word (variable) called location counter (LC): LC stores the value of the memory location assigned to the instruction or operand currently being processed.
- The ORG pseudo instruction initializes the LC to the value of the first location.
- If ORG is missing LC is initially set to 0.
- The LC is incremented (by 1) after processing each line of code.

(Mano 1993)

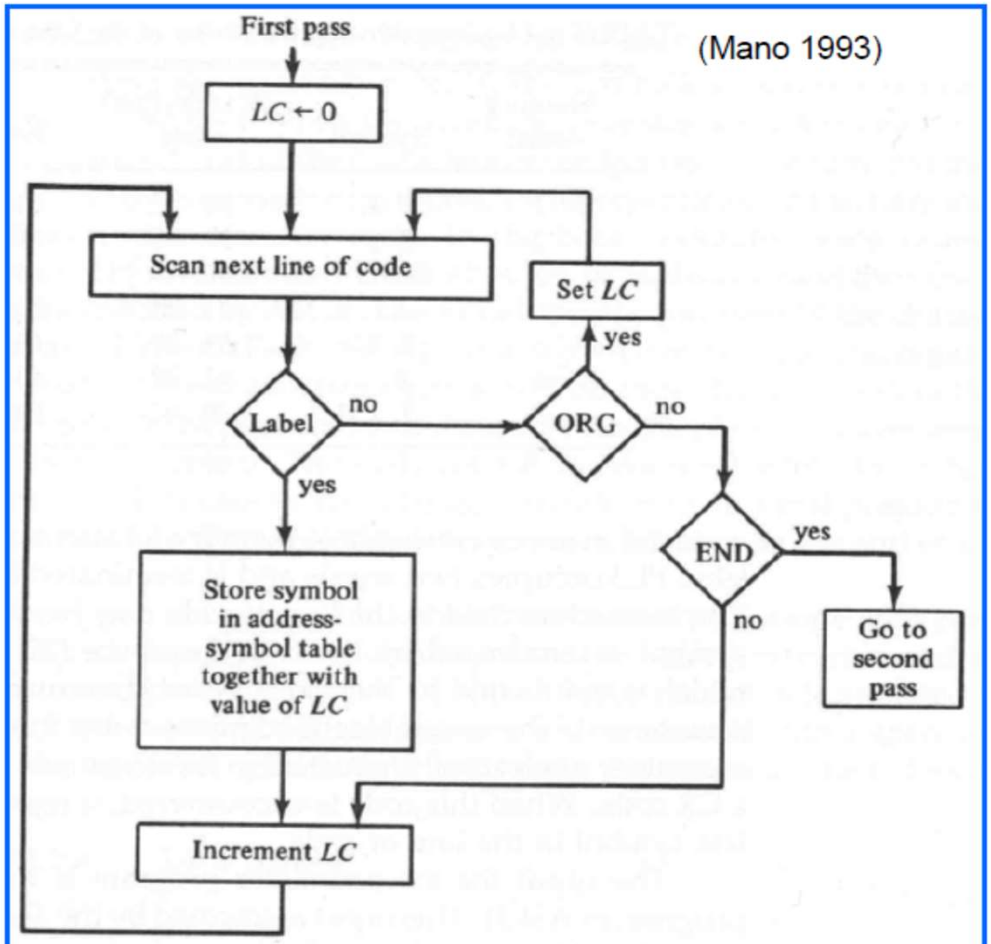


Figure 6-1 Flowchart for first pass of assembler.

- 1.Value of location counter will be set to 0
- 2.Scan the next line
- 3. Is it labelled , if not is it ORG?
- 4. If it is ORG we will set our LC to the address specified by ORG because that is where our program starts from, than scan next line
- 5. If it is label, than store it in symbol table together with value of LC, increment LC by 1 and scan next line
- 6. If it is not ORG than END , increment LC and scan next line

- Address symbol table occupies three words for each label symbol encountered and constitutes the output data that the assembler generates during the first pass.

TABLE 6-12 Address Symbol Table for Program in Table 6-8

Memory word	Symbol or (LC)*	Hexadecimal code	Binary representation
1	M I	4D 49	0100 1101 0100 1001
2	N ,	4E 2C	0100 1110 0010 1100
3	(LC)	01 06	0000 0001 0000 0110
4	S U	53 55	0101 0011 0101 0101
5	B ,	42 2C	0100 0010 0010 1100
6	(LC)	01 07	0000 0001 0000 0111
7	D I	44 49	0100 0100 0100 1001
8	F ,	46 2C	0100 0110 0010 1100
9	(LC)	01 08	0000 0001 0000 1000

* (LC) designates content of location counter.

(Mano 1993)

- **Second pass:**
- Machine instructions are translated by means **of table-lookup procedures**: search of table entries to determine whether a specific item matches one of the items stored in the table.
- The assembler uses four tables. Any symbol encountered must be available as an entry in one of the tables otherwise symbol is not valid and cannot be interpreted:
- 1**. Pseudo instruction table**
- 2. **MRI table:** 7 symbols of memory-reference instructions and their 3-bit operation codes.
- 3**. Non-MRI table:** 18 register-reference and io-instructions and their 16-bit binary codes.
- 4. **Address symbol table (generated during 1st pass)** The assembler searches the four tables to determine the binary value of the symbol that is currently processed.

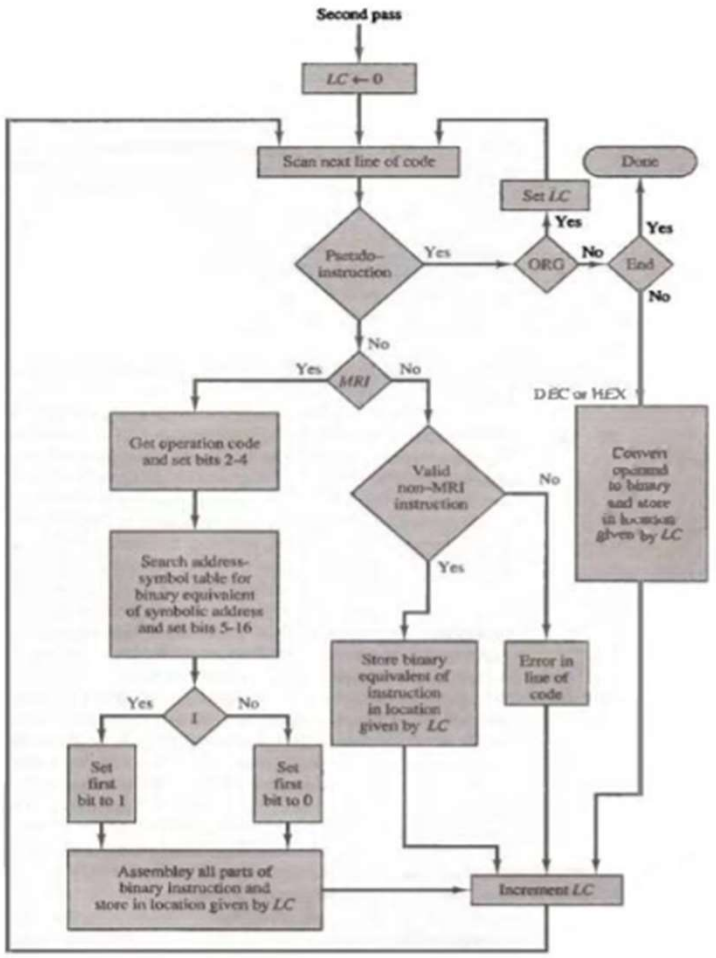


Figure 6-2 Flowchart for second pass of assembler.

- 1. set location counter to 0
- 2. scan the next code line
- 3.check whether given instruction is pseudo instruction, if yes than check if it is ORG. if it is ORG than we will set our LC and scan next instruction
- 4. if it is not ORG than it might be END, HEX or DEC, if it is END that means we are done with second pass of the assembler
- 5. but if it is not END than it might be DEC or HEX, if it is any of these two instruction than convert the given data in binary and store it at address given by location counter

- 6. after than increment LC and scan the next instruction
- 7. if next instruction is not pseudo instruction than we check whether it is MRI instruction
- 8. if it is MRI instruction means we will get operation code corresponding to instruction and set bit 2-4(for example if it is LDA than bit 2-4 is 010)
- 9. than rest bits we will search in symbol table and find the corresponding address and set bits 5-16 out of 1-16 instruction bits we have set 2-4 and 5-16, 1st bit is to check whether it is direct or indirect addressing

- 10. check I bit if it is set means 1 so it is indirect addressing than assemble all the parts of binary instruction and store it at location specified by LC
- 11. if I bit is not set means 0 so it is direct addressing than assemble all the parts of binary instruction and store it at location specified by LC
- 12. but if it is not MRI instruction it can be valid non MRI instruction (register reference or input/output), if so than store the corresponding binary code at location given by LC than increment LC and scan next instruction line
- 13. if it is not valid non MRI instruction means there is error in the given line of code we have to detect error, increment LC and scan next line

Program Loops

- Program loop is a sequence of instructions that are executed many, times (within the loop) with a different set of data.

```
int a[100];
.
.
int sum = 0;
int i;
for (i=0;i<100;i++)
    sum = sum + a[i];
```

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

- A program that translates a program written in a high level programming language to a machine language program is called a compiler.
- A compiler is a more complicated program than an assembler.
- Demonstration of basic functions of a compiler: translating
- the previous c-program (loop) to an assembly language program.

TABLE 6-13 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

corresponds assignment **SUM = 0**

loop counter

program loop

indexing of **do** statement

if counter is zero then exit from the loop

DIMENSION and **INTEGER** statements

NOTE: indirect addressing provides the pointer mechanism. Registers used to store pointers and counters are called index registers (memory words are used in this example).

AC	JJR			
170	150			

- Program is consisting of 4 parts first part is (ORG 100-CLA) which is initialization portion
- Second part is (LOP to HLT) for looping
- Third part is (ADS-SUM) for variables
- Fourth part is (ORG 150-END) for data storage
- In the last portion we have stored the data starting from memory location 150 onwards, it will be stored sequentially
- Program start at location 100
- First instruction nis to load the AC with ADS, where ADS holds symbolic address 150

- Next is store AC value in PTR, so we will store 150 in PTR(this PTR variable points towards array of 100 operands)
- Load AC with NBR, where NBR is having decimal number -100
- Store AC in counter, so -100 will be copied in CTR (CTR will tell us how many times loop will be executed)
- Finally clear accumulator and our initialization is done
- Now in second part, we will add PTR value to AC, now its indirect addressing so we don't use 150 directly but we will use data stored at 150 which is 75. now our AC content is 0000 in which 75 would be added

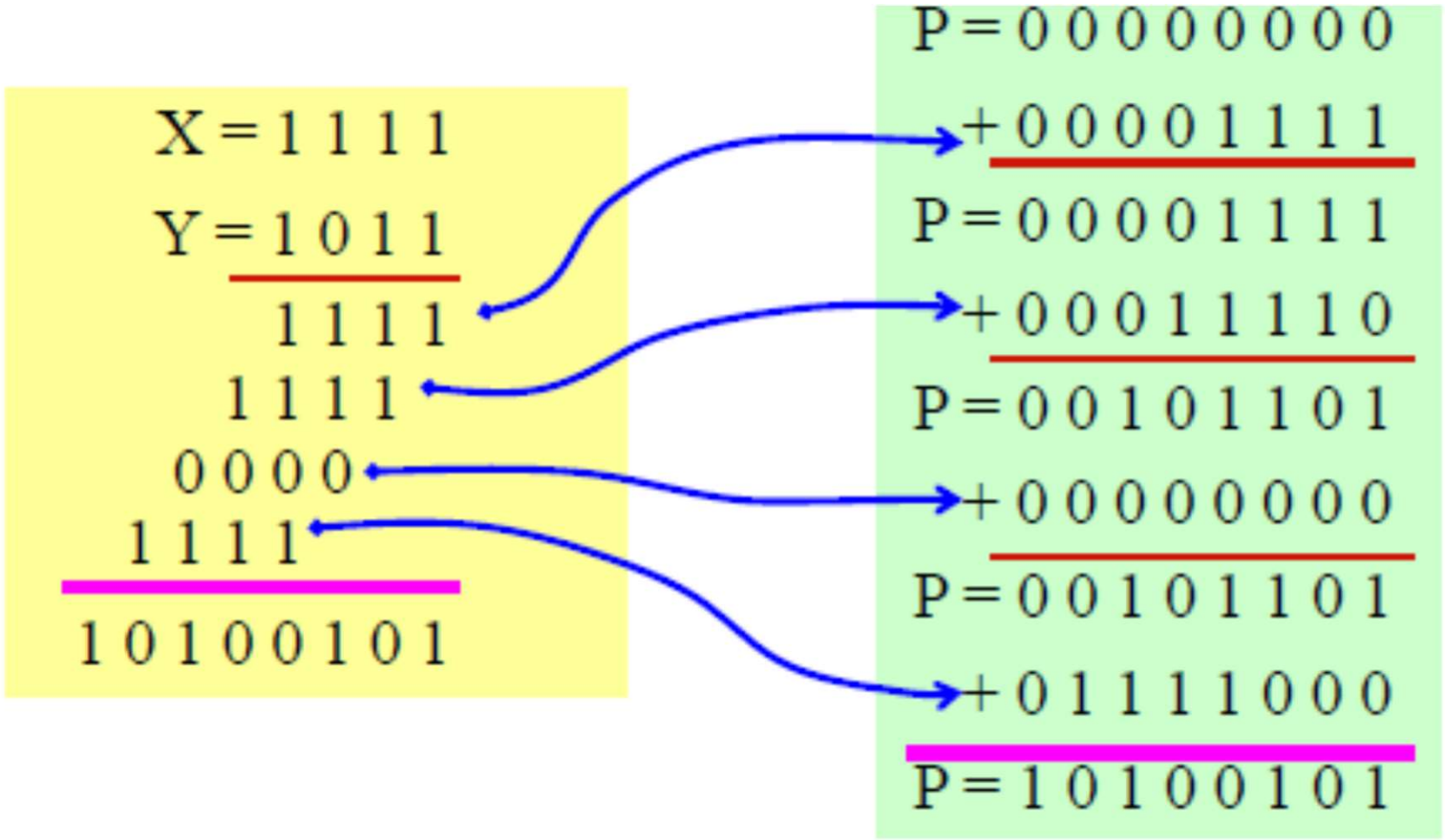
- Now we will increment PTR and CTR with ISZ instruction
- Now BUN will lead us again to LOP where we have ADD PTR I
- This process will go on till CTR will be 0, after that as it is zero next instruction which is BUN LOP will be skipped
- Now STA SUM so content of AC is stored at SUM

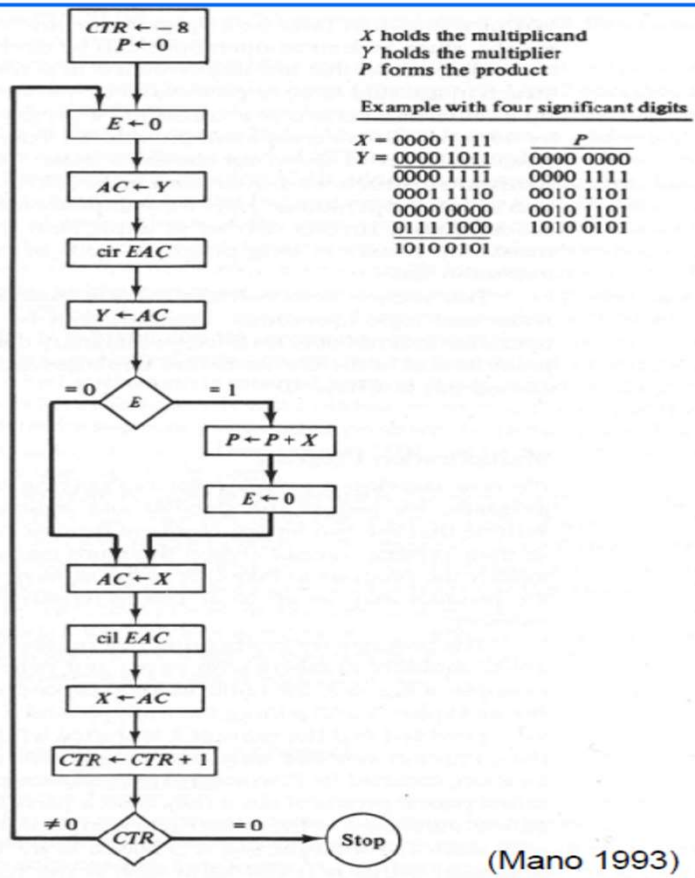
Programming Arithmetic and Logic operation

- Number of instructions available in computer can be few hundred in large system or few dozen in small one
- Four basic arithmetic operations: add, subtract, multiply, divide
- Basic computers just have ADD instruction, complex system can have all 4 instructions
- For such cases we have to write program to perform arithmetic operations
- **Hardware implementation**
  - Operations are implemented in a computer with one machine instruction, it is costly due to circuit implementation
  - Ex) **ADD**, **SUB**
- **Software implementation**
  - Operations are implemented by a set of instruction(Subroutine)
  - Ex) **MUL**, **DIV**

Multiplication Program

- **Multiplication Program**
- Positive Number Multiplication
- X = multiplicand
- Y = multiplier
- P = Partial Product Sum
- multiplication of two 8-bit unsigned numbers (integers) 16-bit product.
- Program loop is traversed eight times, once for each significant bit.
- X holds the multiplicand, Y holds the multiplier, and P holds the product.


X = 1 1 1 1
Y = 1 0 1 1
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1
1 0 1 0 0 1 0 1
P = 0 0 0 0 0 0 0 0
+ 0 0 0 0 1 1 1 1
P = 0 0 0 0 1 1 1 1
+ 0 0 0 1 1 1 1 0
P = 0 0 1 0 1 1 0 1
+ 0 0 0 0 0 0 0 0
P = 0 0 1 0 1 1 0 1
+ 0 1 1 1 1 0 0 0
P = 1 0 1 0 0 1 0 1



(Mano 1993)

Figure 6-3 Flowchart for multiplication program.

TABLE 6-14 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
```

m

Tab. 6-14 Program to Multiply Two Positive numbers

Line				
1		ORG	100	
2	LOP,	CLE		/ A = 0
3		LDA	Y	/ A = Y (000B)
4		CIR		/ Circular Right to E
5		STA	Y	/ Store shifted Y
6		SZE		/ Check if E = 0
7		BUN	ONE	/ E = 1
8		BUN	ZRO	/ E = 0
9	ONE,	LDA	X	A = X (000F)
10		ADD	P	/ X = X + P
11		STA	P	/ Store A to P
12		CLE		/ Clear E
13	ZRO,	LDA	X	/ A = X
14		CIL		/ A = 00011110 (00001111)
15		STA	X	/ Store A to X
16		ISZ	CTR	/ CTR = – 7 = –8 + 1
17		BUN	LOP	/ Repeat until CTR = 0
18		HLT		
19	CTR,	DEC	–8	
20	X,	HEX	000F	
21	Y,	HEX	000B	
22	P,	HEX	0	
23		END		

Double-precision addition

- addition of two 32-bit unsigned integers.
- Added numbers place in two consecutive memory locations, AL and AH, and BL and BH.
- Sum is stored in CL and CH

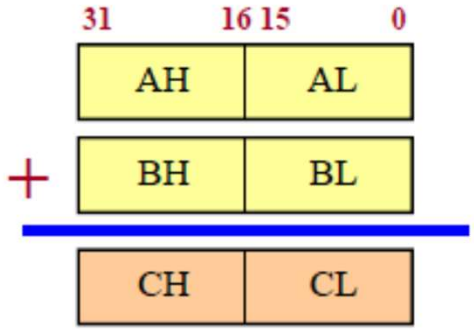


TABLE 6-15 Program to Add Two Double-Precision Numbers

```
        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,     —
```

Line				
1		LDA	AL	/ A = AL
2		ADD	BL	/ A = AL + BL
3		STA	CL	/ Store A to CL
4		CLA		/ A = 0
5		CIL		/ 0000 0000 0000 000?(?=E)
6		ADD	AH	/ A = 00(E=0) or 01(E=1)
7		ADD	BH	/ A = A + AH + BH
8		STA	CH	/ Store A to CH
9		HLT		
10	AL,	DEC	?	/ Operand
11	AH,	DEC	?	
12	BL,	DEC	?	
13	BH,	DEC	?	
14	CL,	HEX	0	
15	CH,	HEX	0	

Logic Operations

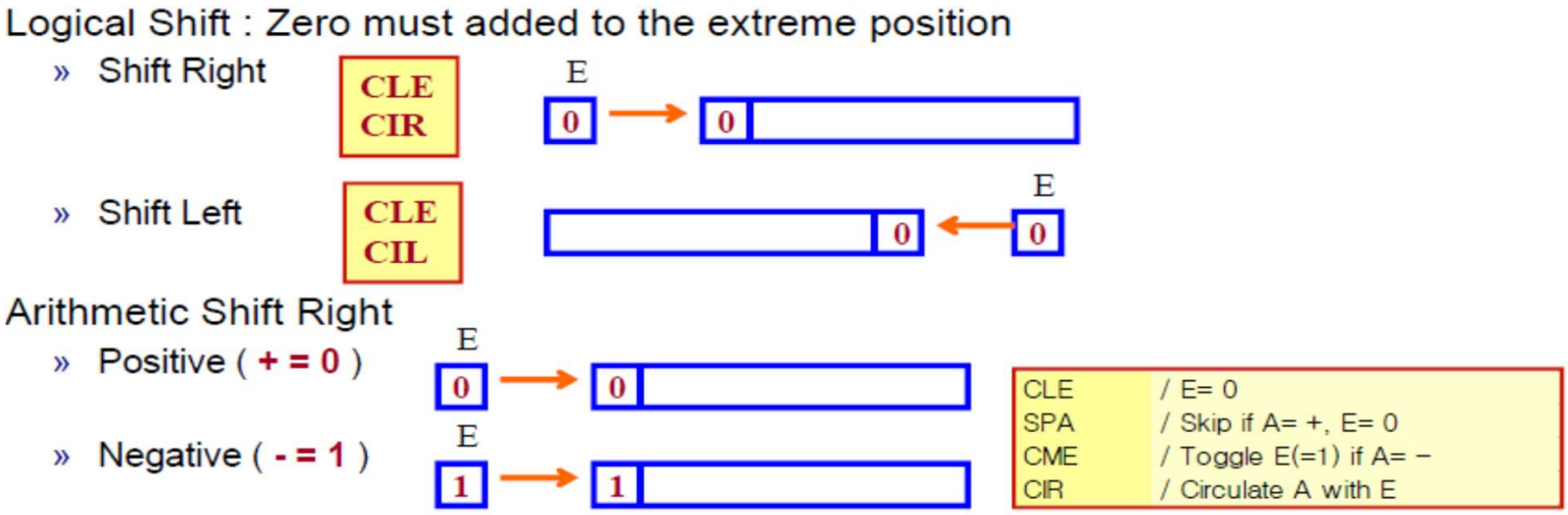
- Any logic operation can be implemented by a program using AND and complement operations.
- *E.g.* x + y = (x'y')' by DeMorgan's theorem.
- OR operation of two logic operands A and B:

```
LDA A     Load first operand A
CMA       Complement to get Ā
STA TMP   Store in a temporary location
LDA B     Load second operand B
CMA       Complement to get B̄
AND TMP   AND with Ā to get Ā ∧ B̄
CMA       Complement again to get A ∨ B
```

- Other logical operations can be implemented in a similar fashion.

```
LDA     A          / Load A
CMA                / Complement A
STA     TMP     P  / St    o
LDA     B          / Load B
CMA                / Complement
AND     TMP        / AND
CMA                / Complement
```

- The basic computer has two shift instructions: CIL, CIR.
- Logical and arithmetic shifts can be programmed.



- Arithmetic right-shift (sign bit remains):

```
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 left-shift (zeros added to the rightmost position) – E must be checked for an overflow, *e.g.*:

```
      CLE          /clear E
      CIL          /circulate left E and AC
      SZE          /skip if E is zero (= AC was positive)
      BUN  NEG     /branch for checking the negative case
      SPA          /skip if AC is positive
      BSA  OVF     /branch to overflow handling
      BUN  RET I   /return main program
NEG,  SNA          /skip if AC is negative
      BSA  OVF
      BUN  RET I
```

Subroutines

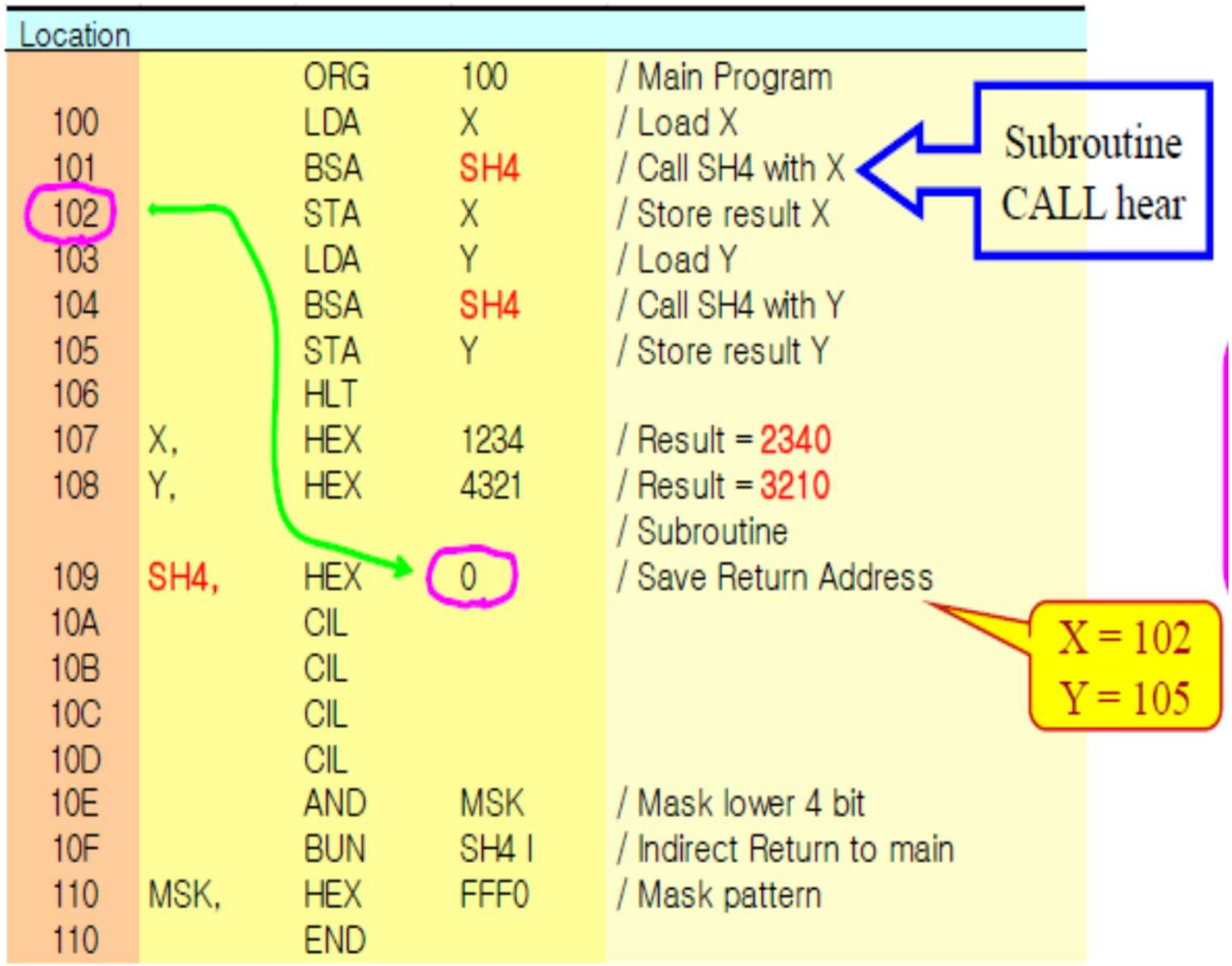
- A set of common instructions that can be used (called) in a program many times is called a subroutine.
- A branch can be made to the subroutine from any part of the main program.
- The return address must be stored (somewhere) in order to successfully return from the subroutine.
- In the basic computer the link between main program and subroutine is the BSA instruction.

TABLE 6-16 Program to Demonstrate the Use of Subroutines

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 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*(0-3) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

(Mano 1993)

BUN	D_4T_4:	$PC \leftarrow AR, \quad SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR, \quad SC \leftarrow 0$


Location
ORG 100 / Main Program
100 LDA X / Load X
101 BSA SH4 / Call SH4 with X
102 STA X / Store result X
103 LDA Y / Load Y
104 BSA SH4 / Call SH4 with Y
105 STA Y / Store result Y
106 HLT
107 X, HEX 1234 / Result = 2340
108 Y, HEX 4321 / Result = 3210
/ Subroutine
109 SH4, HEX 0 / Save Return Address
10A CIL
10B CIL
10C CIL
10D CIL
10E AND MSK / Mask lower 4 bit
10F BUN SH4 I / Indirect Return to main
110 MSK, HEX FFF0 / Mask pattern
110 END
Subroutine CALL hear
X = 102
Y = 105

- From the 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 as a **subroutine linkage.**
- The BSA instructions performs a subroutine call.
- The last instruction of the subroutine (indirect BUN) performs a **subroutine return.**
- In many computers, **index registers** are employed to implement the subroutine linkage: registers are used to store and retrieve the return address.

- Data can be transferred to a subroutine by using registers (*e.g.* AC in previous example) or through the memory.
- Data can be placed in memory locations following the call (return from subroutine must be correspondingly modified).
- Data can also be placed in a block of storage (structure):
- the first address of the block in then placed in the memory location following the subroutine call.
- *E.g.* of parameter linkage : OR operation.
- The subroutine must increment the return address for each operand.
- *E.g.* of subroutine to move a block of data is presented in Table 6-18.

TABLE 6-17 Program to Demonstrate Parameter Linkage

(Mano 1993)

Location			
		ORG 200	
200		LDA X	/Load first operand into *AC*
201		BSA OR	/Branch to subroutine OR
202		HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand
20B		CMA	/Complement second operand
20C		AND TMP	/AND complemented first 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	

BUN	D_4T_4:	$PC \leftarrow AR,\quad SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC,\quad AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR,\quad SC \leftarrow 0$

Tab. 6-17 Program to Demonstrate Parameter Linkage

Location				
		ORG	200	
200		LDA	X	/ Load first operand X
201		BSA	OR	/ Call OR with X
202		HEX	3AF6	/ Second operand
203		STA	Y	/ Subroutine return here(Y=result)
204		HLT		
205	X,	HEX	7B95	/ First operand
206	Y,	HEX	0	/ Result store here
207	OR,	HEX	0	/ Return address = 202
208		CMA		/ Complement X
209		STA	TMP	/ TMP = X
20A		LDA	OR I	/ A = 3AF6 (202 번지의 내용)
20B		CMA		/ Complement Second operand
20C		AND	TMP	/ AND
20D		CMA		/ 전체 Complement
20E		ISZ	OR	/ Return Address = 202 + 1 = 203
20F		BUN	OR I	/ Return to main
210	TMP,	HEX	0	
		END		

* OR Subroutine
First Operand : X = 7B95
Second Operand : BSA후 = 3AF6

return address must be incremented three times

TABLE 6-18 Subroutine to Move a Block of Data

```
                        /Main program
        BSA MVE         /Branch to subroutine
        HEX 100         /First address of source data
        HEX 200         /First address of destination data
        DEC -16         /Number of items to move
        HLT             /subroutine returns here
MVE,    HEX 0           /Subroutine MVE
        LDA MVE I       /Bring address of source      (= 100)
        STA PT1         /Store in first pointer
        ISZ MVE         /Increment return address
        LDA MVE I       /Bring address of destination (=200)
        STA PT2         /Store in second 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,    —
```

(Mano 1993)

Tab. 6-18 Subroutine to Move a Block of Data

		ORG	100	
100		BSA	MVE	/ Subroutine Call
101		HEX	200	/ Source Address
102		HEX	300	/ Destination Address
103		DEC	−16	/ Number of data to move
104		HLT		
105	MVE,	HEX	0	/ Return address= 101
106		LDA	MVE I	/ A= 200
107		STA	PT1	/ PT1= 200
108		ISZ	MVE	/ Return address= 102
109		LDA	MVE I	/ A= 300
10A		STA	PT2	/ PT2= 300
10B		ISZ	MVE	/ Return address= 103
10C		LDA	MVE I	/ A= −16
10D		STA	CTR	/ CTR= −16
10E		ISZ	MVE	/ Return address= 104
10F	LOP,	LDA	PT1 I	/ A= Address 200의 내용
110		STA	PT2 I	/ Address 300에 저장
111		ISZ	PT1	/ PT1= 201
112		ISZ	PT2	/ PT2= 301
113		ISZ	CTR	/ CTR= −15 if 0 skip
114		BUN	LOP	/ Loop until CTR= 0
115		BUN	MVE I	/ 104 로 Return = HLT
116	PT1,	HEX	?	/ Source
117	PT2,	HEX	?	/ Destination
118	CTR,	DEC	?	/ Counter

3
Op

Input-Output Programming

- Input-output programs are needed for writing symbols to computer's memory and printing symbols from the memory.
- Input-output program are employed for writing programs for the computer, for example. Programs for the Basic Computer to input and output one character: non-interrupt based programs

TABLE 6-19 Programs to Input and Output One Character

(Mano 1993)

```
(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"
```

- The second example (Table 6-20) receives two 8-bit characters and places the result to 16-bit accumulator:

TABLE 6-20 Subroutine to Input and Pack Two Characters

```
IN2,    —          /Subroutine entry
FST,    SKI
        BUN FST
        INP        /Input first character
        OUT
        BSA SH4    /Shift left four times
        BSA SH4    /Shift left four more times
SCD,    SKI
        BUN SCD
        INP        /Input second character
        OUT
        BUN IN2 I  /Return
```

(Mano 1993)

shifts AC 8-bits to the left using the SH4 subroutine (see earlier example).

fills bits 0-7 of AC (bits 8-15 remain intact)

- The third example lists a program for storing characters from the input device (e.g. keyboard) to computer's memory: program can be used as a loader program when a symbolic program is inputted to computer's memory prior the usage of an assembler.

TABLE 6-22 Program to Compare 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,    —
```

(Mano 1993)

- The interrupt facility is useful in a multiprogram environment when two or more programs reside in memory at the same time: computer can perform useful computations while waiting a request (interrupt) from an external device.
- The program that is currently being executed is referred to as the running program.
- The function of the interrupt facility is to take care of the data transfer of a program while another program is being executed (which must include ION if interrupt(s) is used).

- The interrupt service routine must include instructions to perform following tasks:
- 1. Save contents of processor registers: the service routine must not disturb the running (interrupted) program.
- 2. Check which interrupt flag is set: this identifies the interrupt that occurred.
- 3. Service the device whose interrupt flag was set: the sequence by which the flags are checked dictates the priority assigned to each device.
- 4. Restore the contents of processor registers.
- 5. Turn the interrupt facility on to enable further interrupts.
- 6. Return to the running program.

TABLE 6-23 Program to Service an Interrupt (Mano 1993)

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 (=> PC=1)
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 **(clears FGO)**
		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