Programming the Basic Computer

lecture 8

Programming the Basic Computer

- A computer system includes both hardware and software.
- Hardware consist of the physical components.
- Software refers to computer programs.
- Hardware and software influence each other.
- Binary code is difficult to work with: there is a need for translating symbolic programs into binary programs, e.g. (Intel x86):

```
10110000 01100001 => mov a1, 0x61
```

- A written program can be machine dependent (assembly language programs) or machine independent (e.g. C-language programs).
- A program is a list of instructions for performing a data processing task.
- There is various programming languages a user can use to write programs for a computer. However, computer can execute only programs that are represented internally in a valid binary form.
- Programs written in any programming language must be translated to the binary representation prior execution.

- Program categories:
  1. Binary code: exact representation of instructions in binary form.
  2. Octal or hexadecimal code: translation of binary code into equivalent octal or hexadecimal representation.
  3. Symbolic code: symbolic representation is used for the parts of the instruction code. Each symbolic instruction is translated into one binary coded instruction by a program called an assembler.
  4. High-level programming language: developed to reflect the procedures for solving problems rather than be concerned with the computer hardware behavior. The program for translating a high-level language program to binary is called a compiler.
  - Machine language refers to categories 1 and 2.

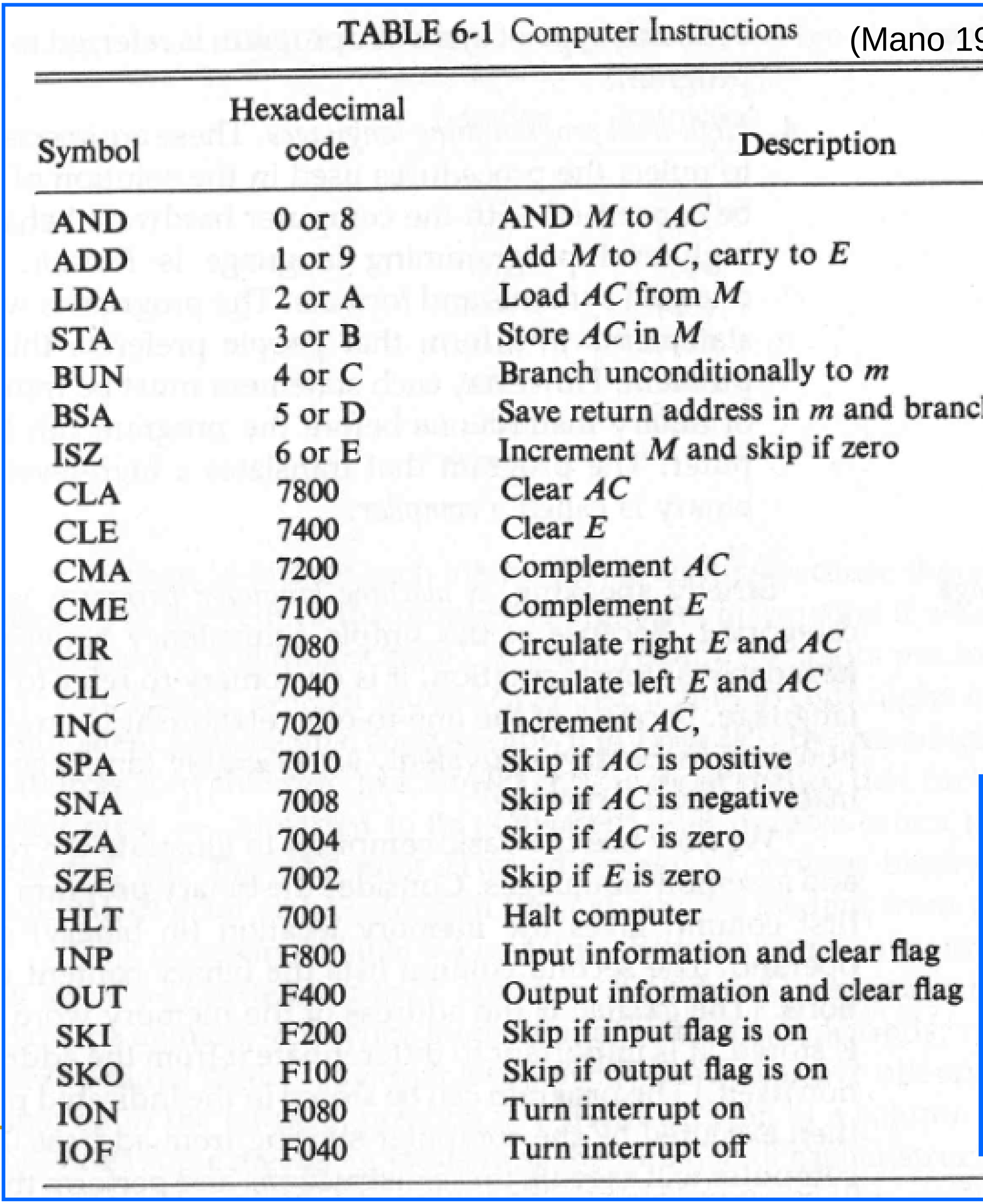


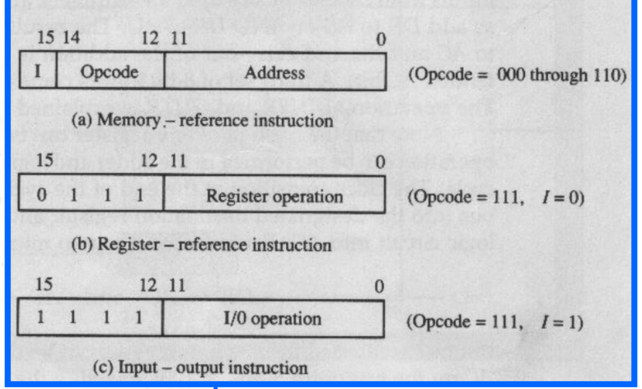
TABLE 6-1 Computer Instructions (Mano 1993)

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

M refers to a memory word found at the effective address

m denotes the effective address

Figure 5-5 Basic computer instruction formats.



(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

- Relation between binary and assembly languages:

tedious for a programmer

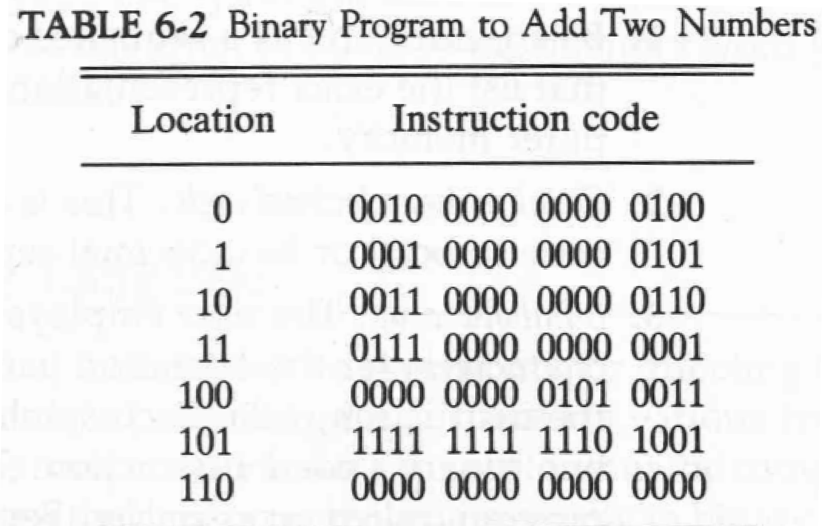


TABLE 6-2 Binary Program to Add Two Numbers

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

..a bit easier

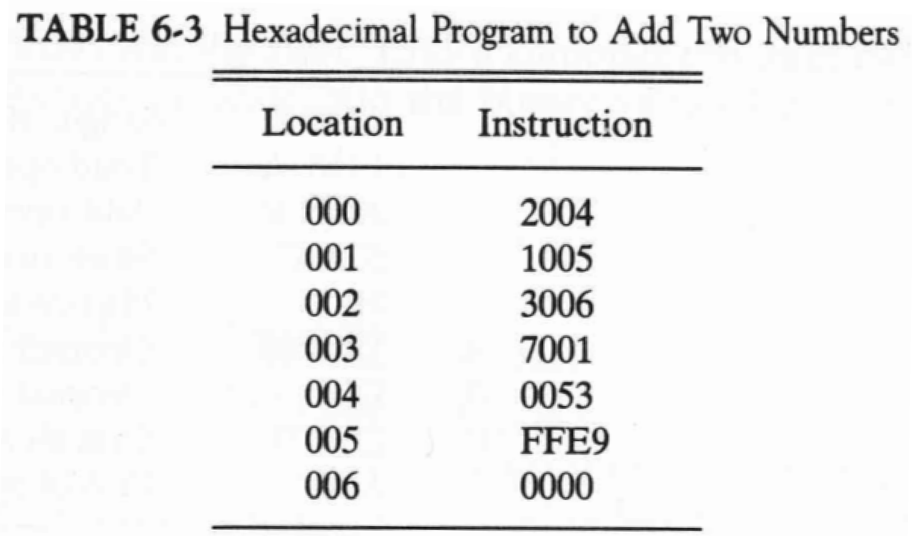


TABLE 6-3 Hexadecimal Program to Add Two Numbers

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

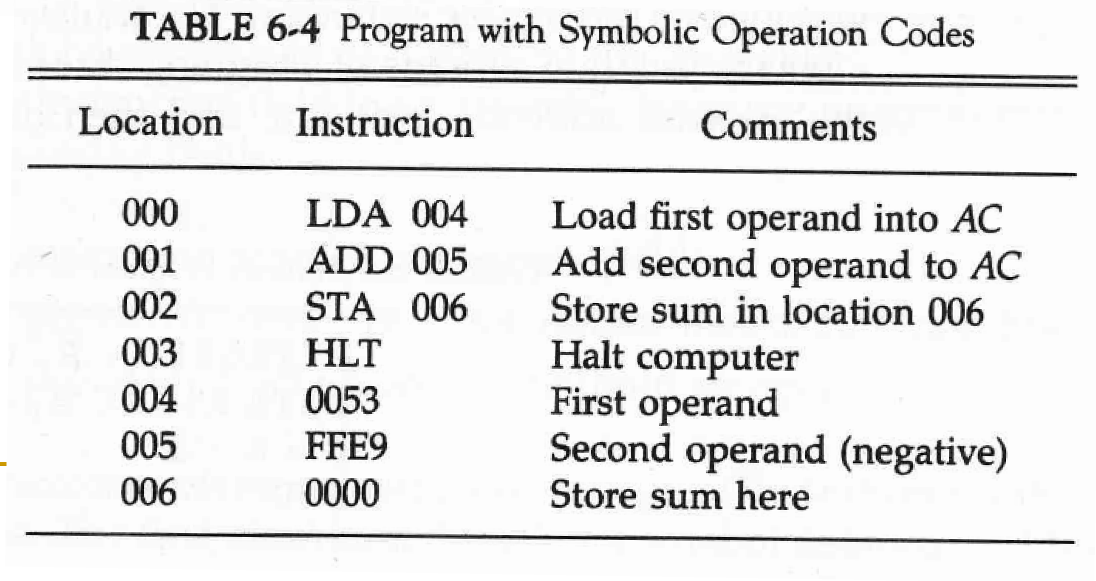


TABLE 6-4 Program with Symbolic Operation Codes

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

..much better

- Using symbolic address and decimal operands
  - numerical locations of memory operands are usually not exactly known while writing a program.
  - Decimal numbers are more familiar to humans

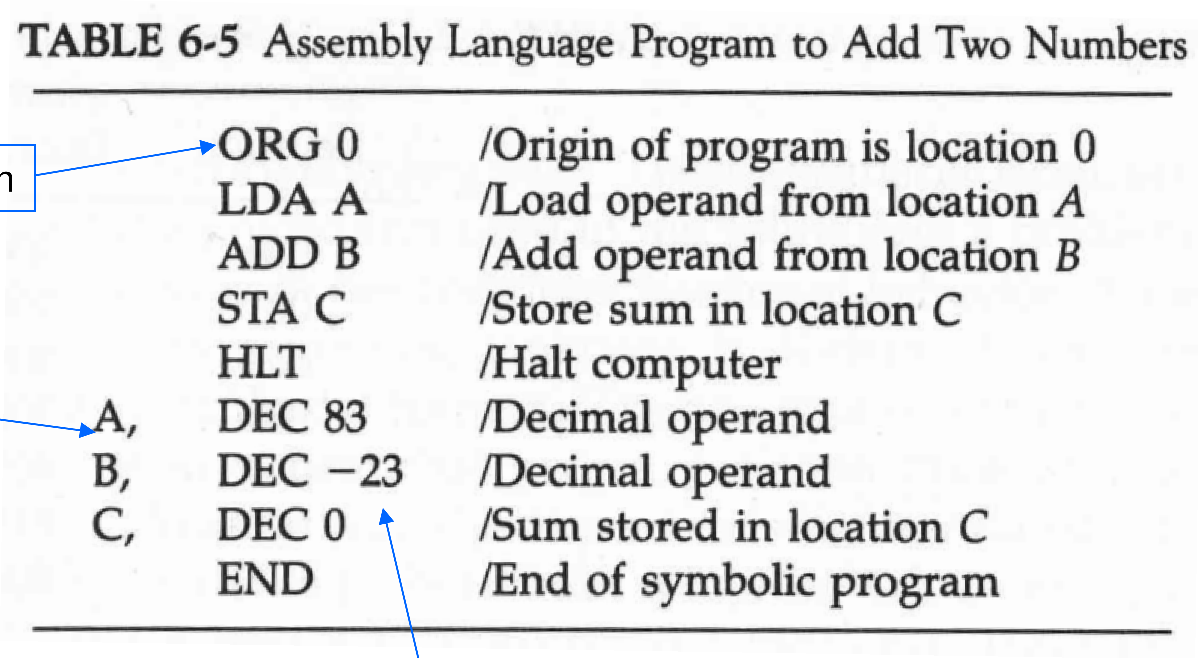


TABLE 6-5 Assembly Language Program to Add Two Numbers

	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

pseudoinstruction

label

must be translated
to binary signed-2's complement
representation

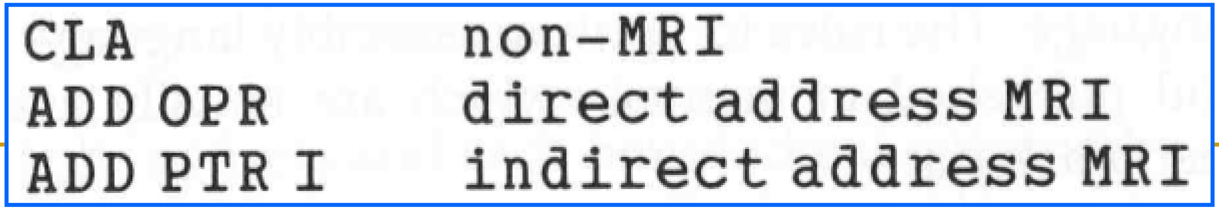
with C-language

```
int a =  83;
int b = -23;
int c;
c = a + b;
```

Assembly Language

- 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 of the assembly language of the Basic Computer
    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*. ‘/’.

- A symbolic address is restricted to three symbols – the first one is always a letter. The address is terminated by a comma.
- The instruction field may specify:
  1. A memory-reference instruction (MRI)
  2. A register-reference instruction (non-MRI)
  3. A pseudoinstruction with or without an operand
  - A memory-reference instruction occupies two or three symbols separated by spaces. The first must be a three-letter symbols defining MRI operation code from Table 6-1. The second is a symbolic address, and the third is the optional I indicating indirect address.
  - non-MRI has not an address part.



CLA	non-MRI
ADD OPR	direct address MRI
ADD PTR I	indirect address MRI

- A defined symbolic address must occur again in a label field.
- A pseudoinstruction is an instruction for the assembler and it gives information for the translation phase:

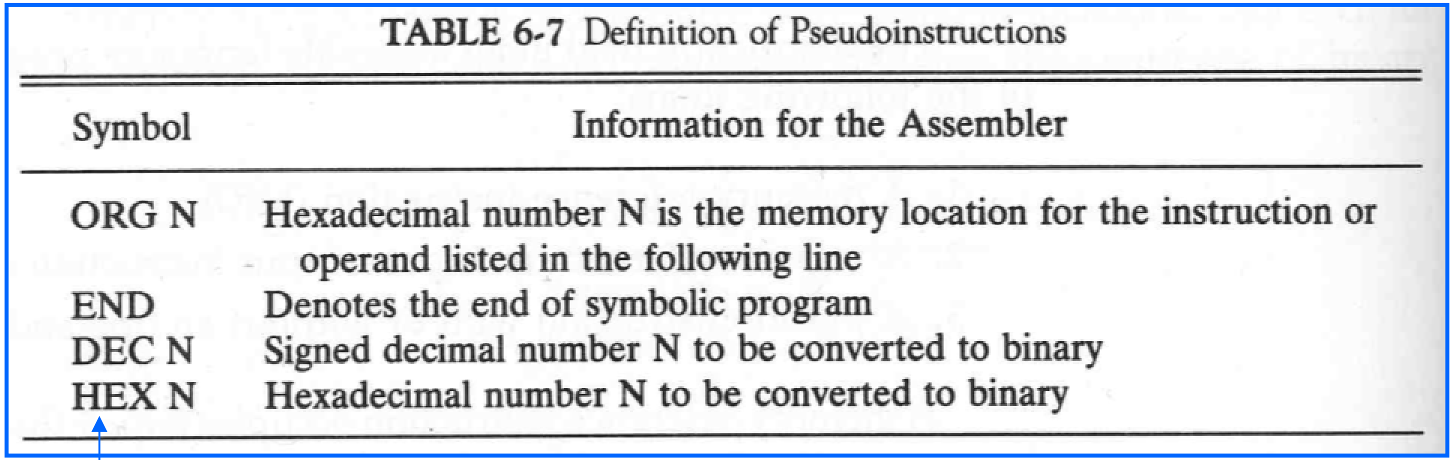


TABLE 6-7 Definition of Pseudoinstructions

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

radix

- An example assembly language program:

(Mano 1993)

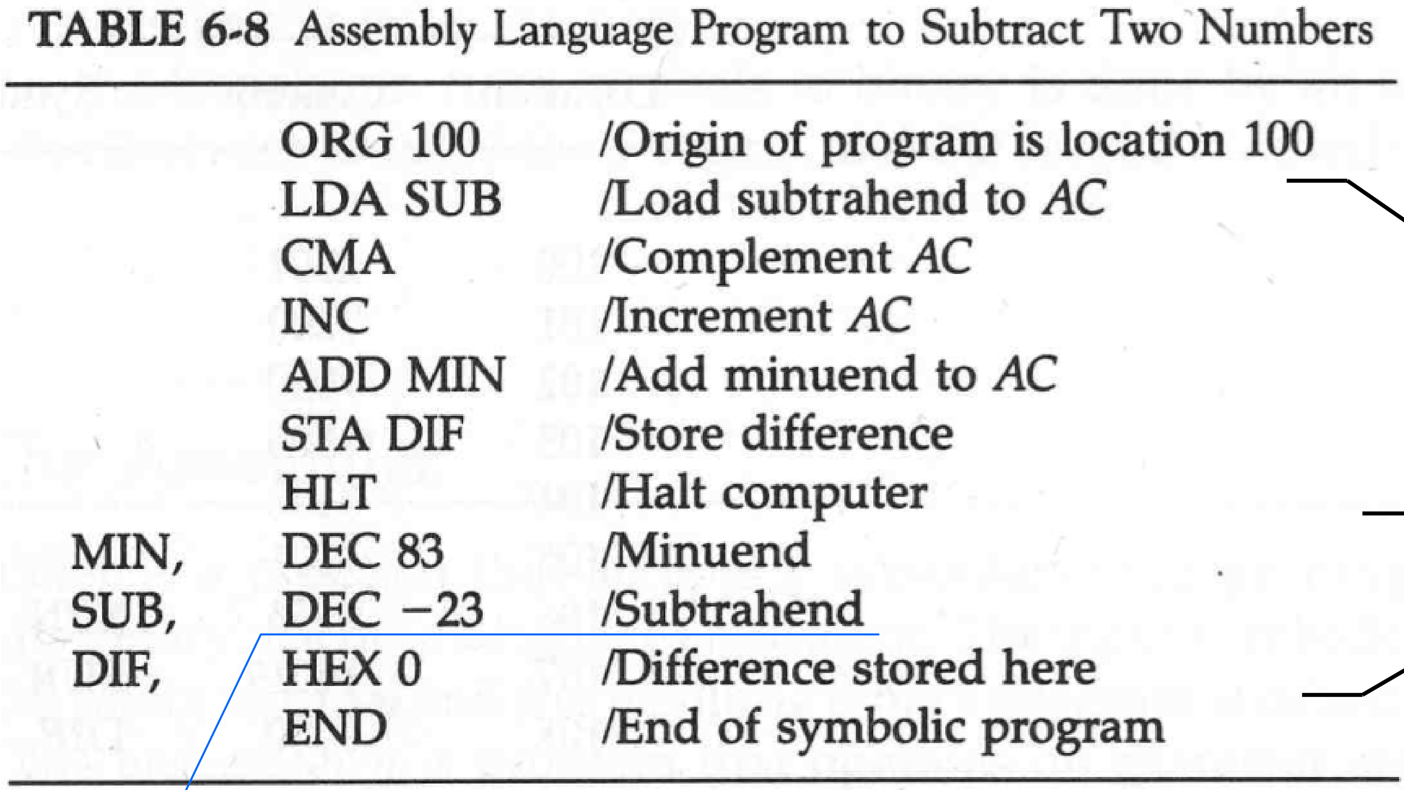
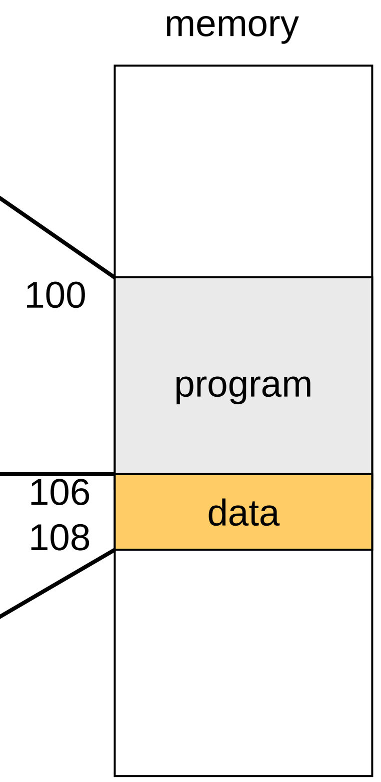


TABLE 6-8 Assembly Language Program to Subtract Two Numbers

	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 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 cross assembler (cross compiler) produces code for one processor, but runs on another
  - used *e.g.* in an embedded system software development in PC
  - the final program is uploaded into a target device
- As well as translating assembly instruction mnemonics into opcodes assemblers provide the ability to use symbolic names for memory locations (saving tedious calculations and manually updating addresses when a program is slightly modified), and macro facilities for performing textual substitution — typically used to encode common short sequences of instructions to run inline instead of in a subroutine.

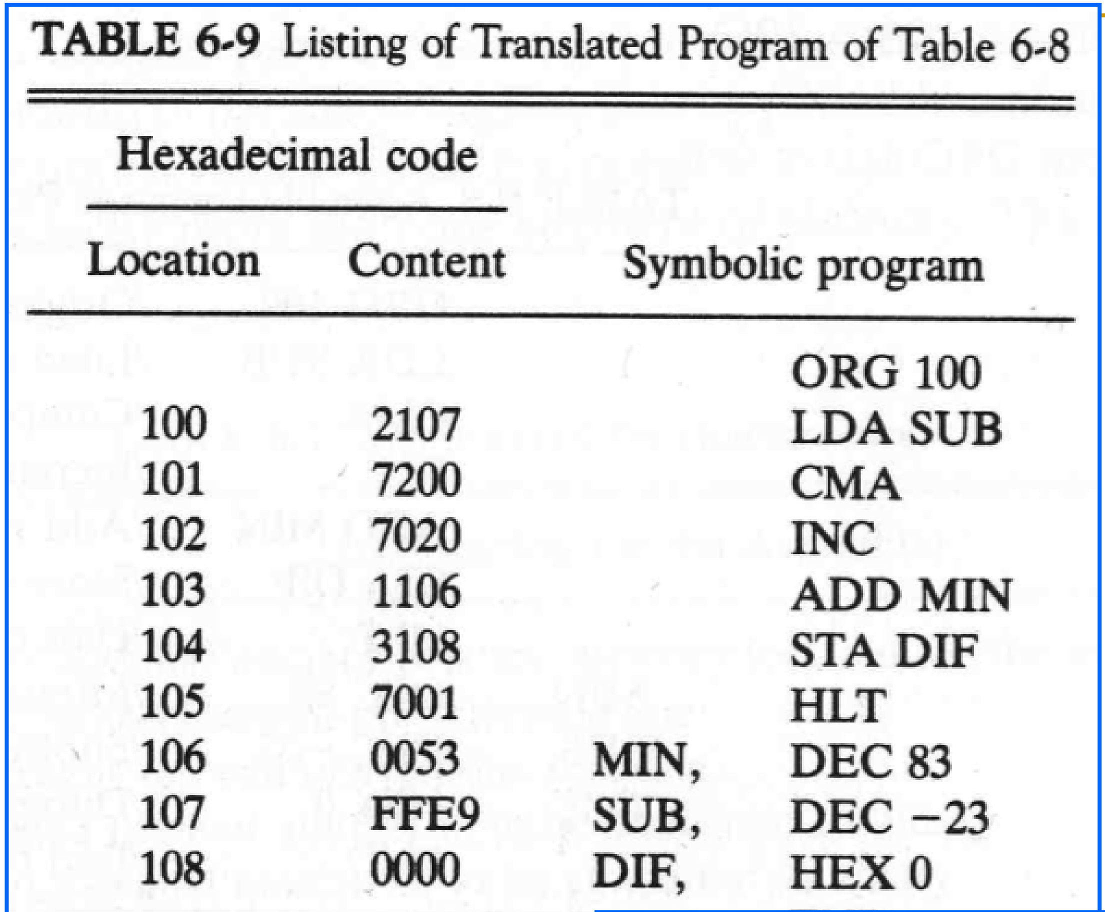
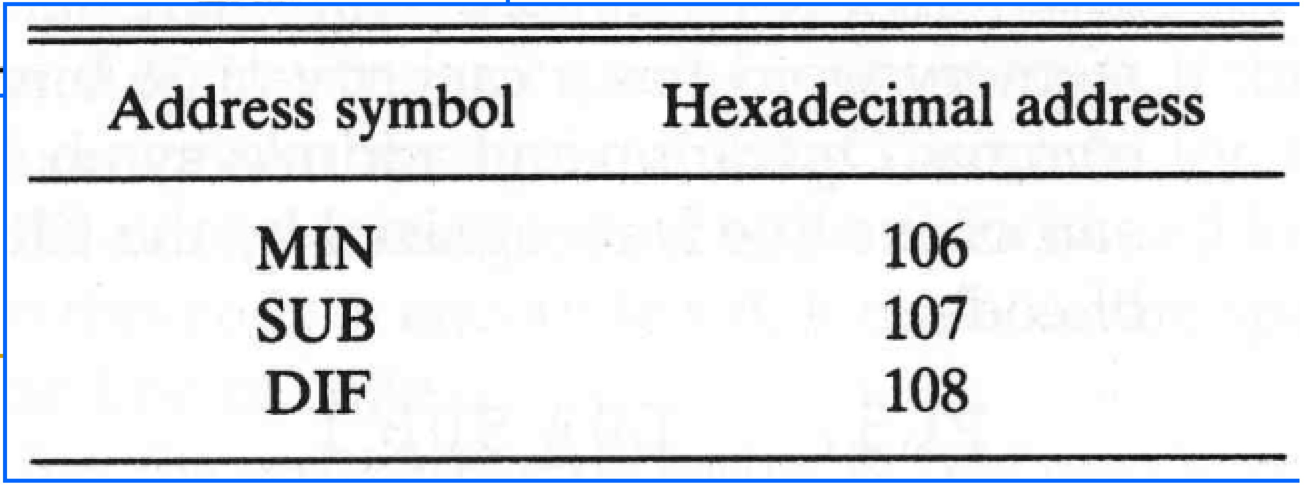


TABLE 6-9 Listing of Translated Program of Table 6-8

Hexadecimal code			
Location	Content	Symbolic program	
			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

(Mano 1993)

address symbol table



Address symbol	Hexadecimal address
MIN	106
SUB	107
DIF	108

- 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; see Table 6-10).
  - 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.

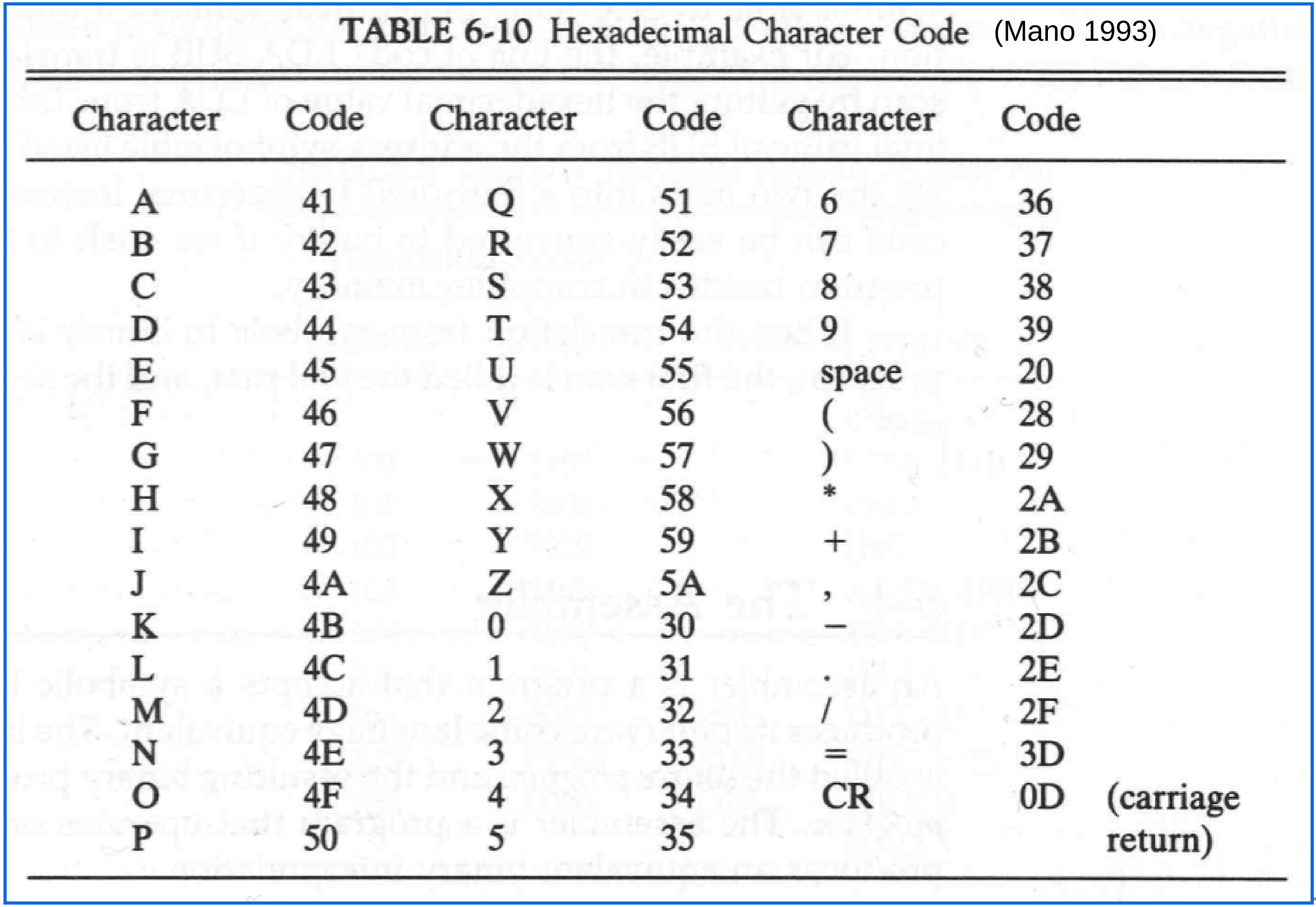


TABLE 6-10 Hexadecimal Character Code (Mano 1993)

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 (see Table 6-11):

(Mano 1993)

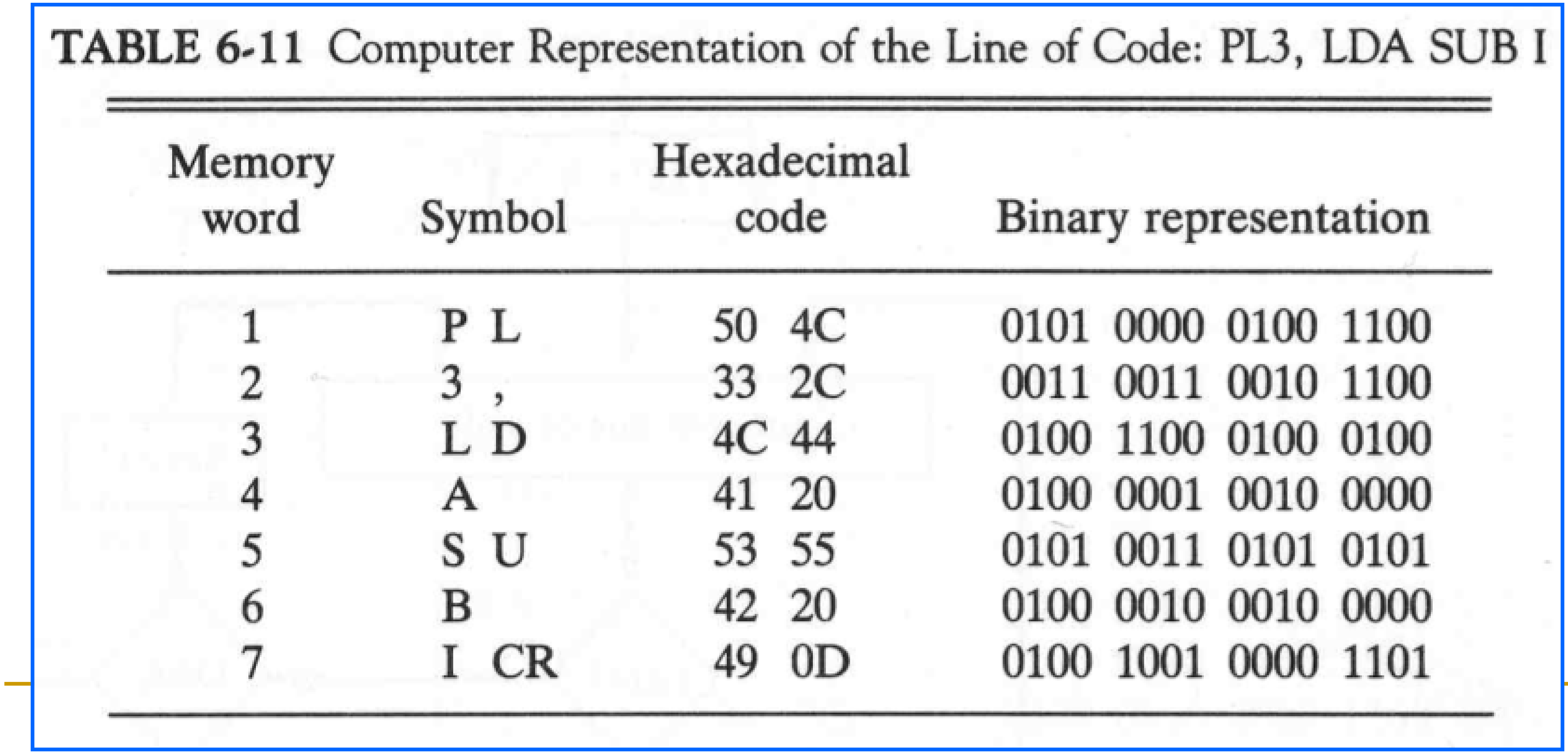
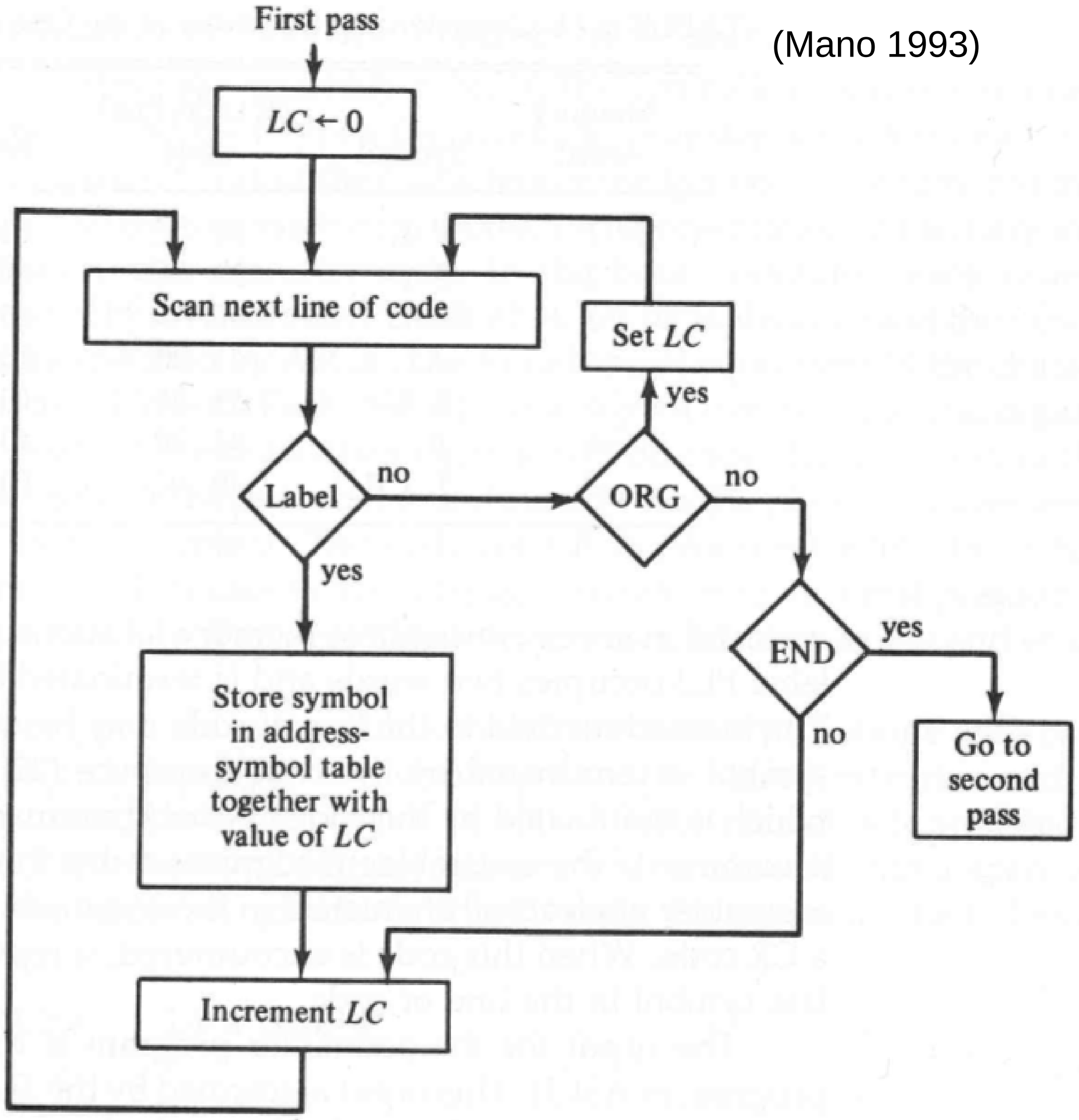


TABLE 6-11 Computer Representation of the Line of Code: PL3, LDA SUB I

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 (see Table 6-11) 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.
- <u>The binary program is the output generated by the assembler</u>.

- 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 (see Fig. 6-1).
  - 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 pseudoinstruction 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.

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

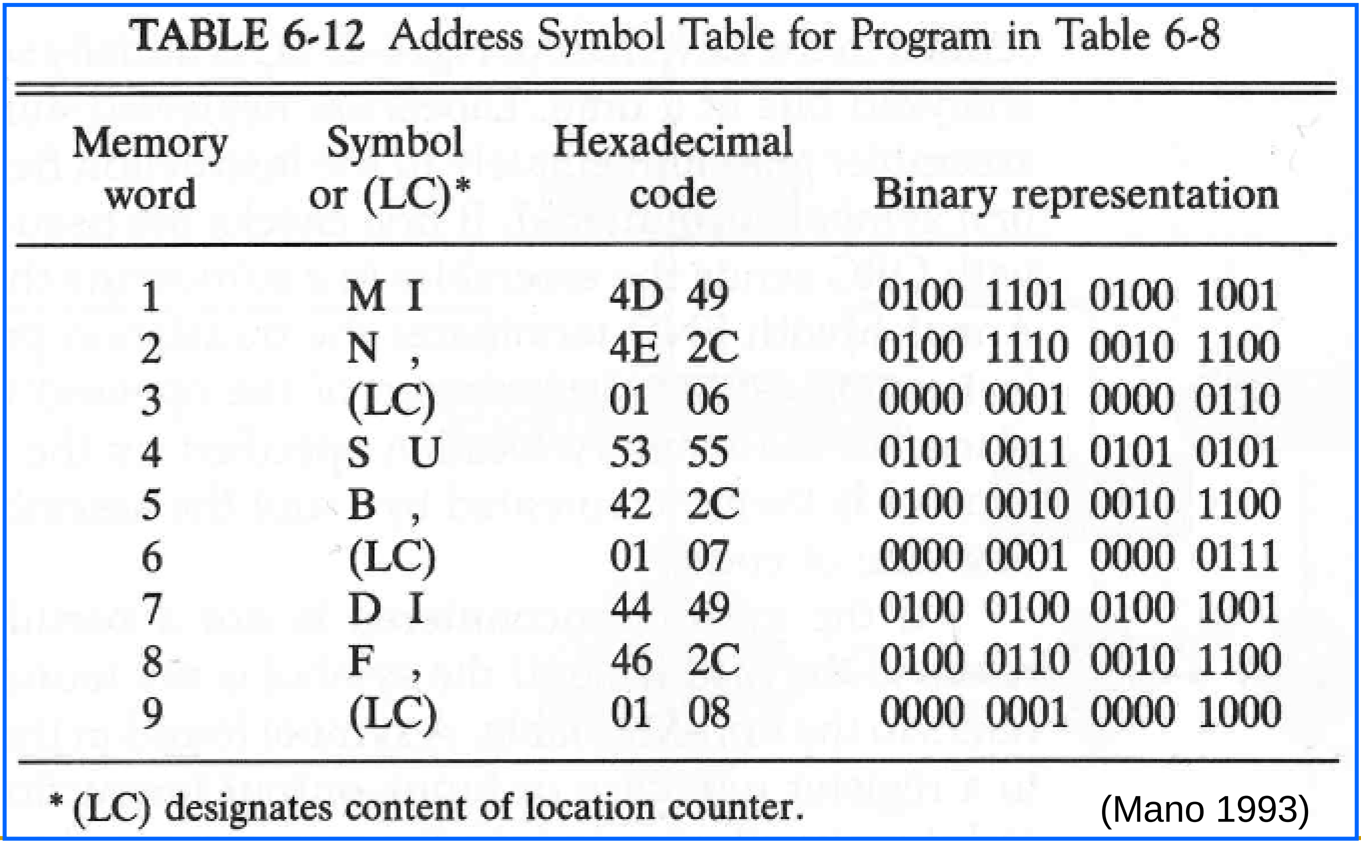


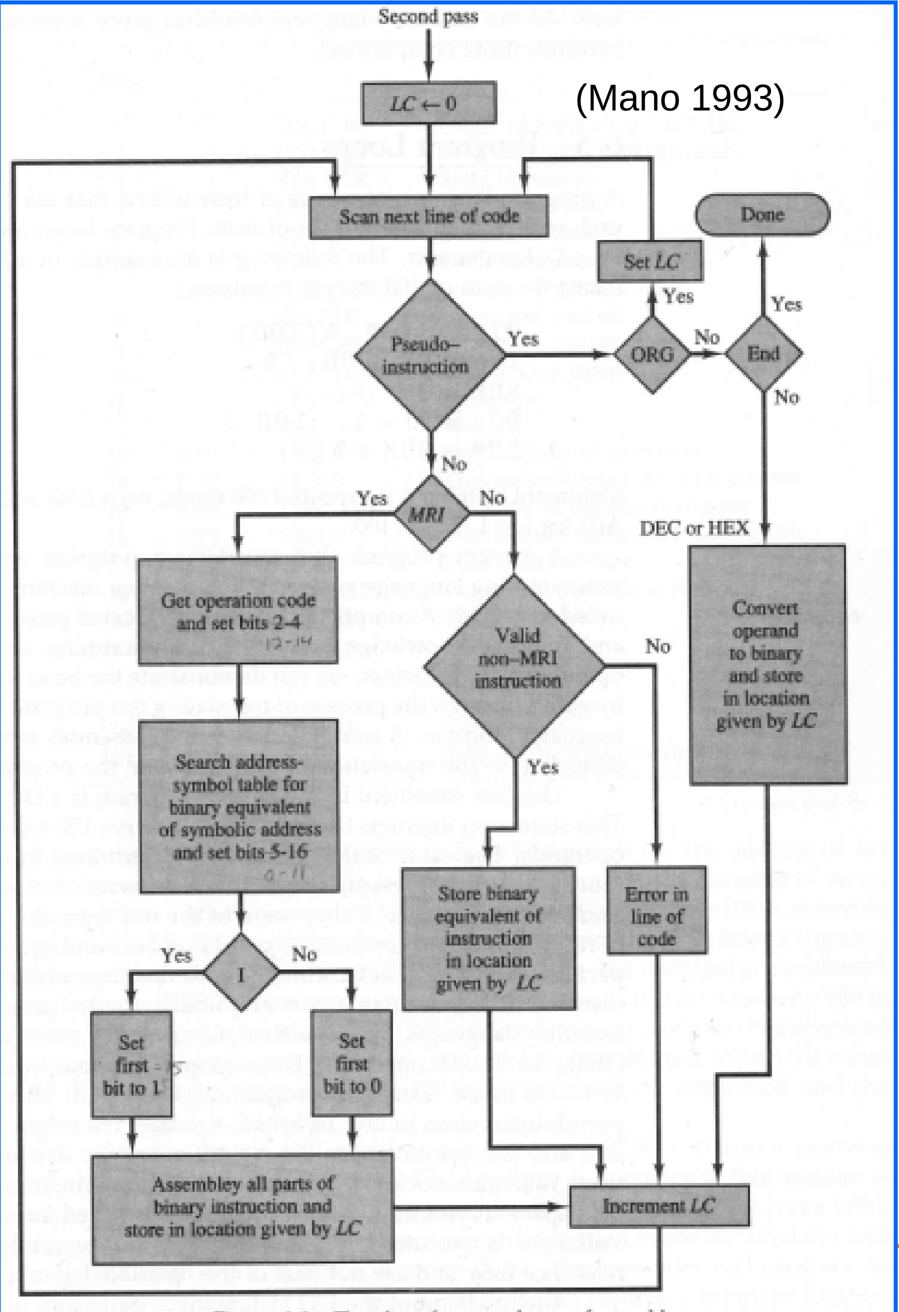
TABLE 6-12 Address Symbol Table for Program in Table 6-8

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:
    1. Pseudoinstruction 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.



(Mano 1993)

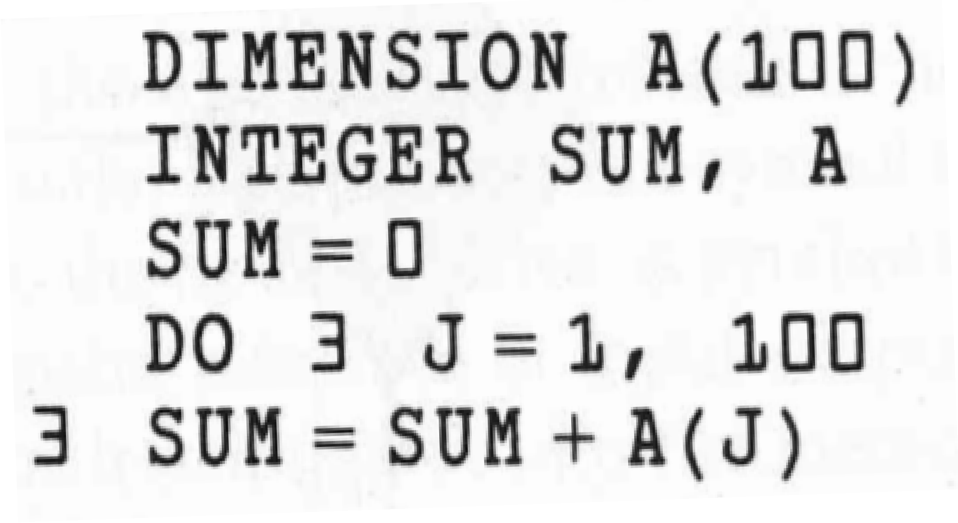
Figure 6-2 Flowchart for second pass of assembler.

- Error diagnostics:
  - invalid machine code not found in the MRI or non-MRI tables.
  - Symbolic address not found from the address table.
  - ⇒ cannot be translated because the binary value is not known: error message for the user.

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.

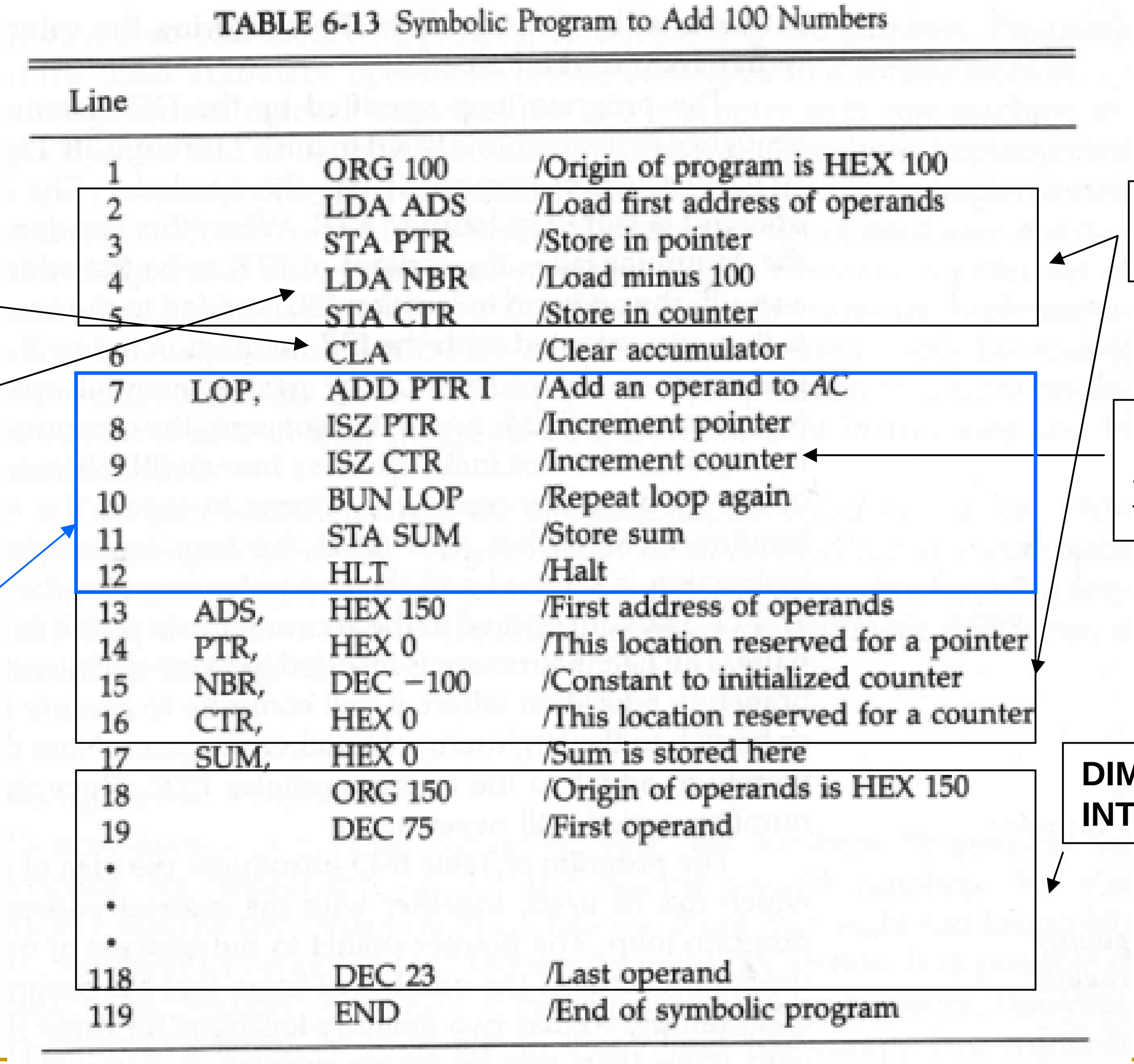


TABLE 6-13 Symbolic Program to Add 100 Numbers

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

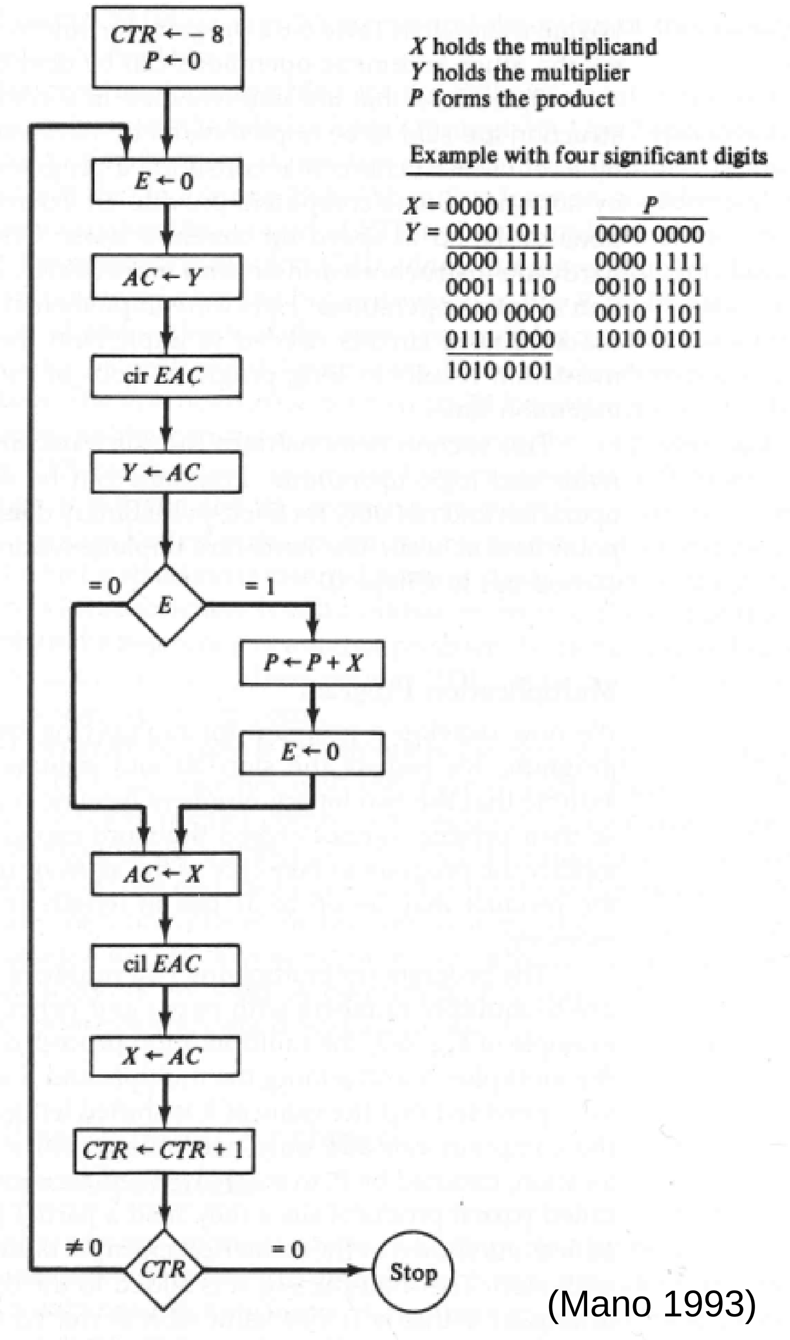
if counter is zero then exit from the loop

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).

Programming Arithmetic and Logic Operations

- Fig. 6-3 shows a flowchart of a multiplication program of the basic computer
  - 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.
  - Example shows how an arithmetic operation can be implemented by a program.



X holds the multiplicand
Y holds the multiplier
P forms the 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	

Figure 6-3 Flowchart for multiplication program.

(Mano 1993)

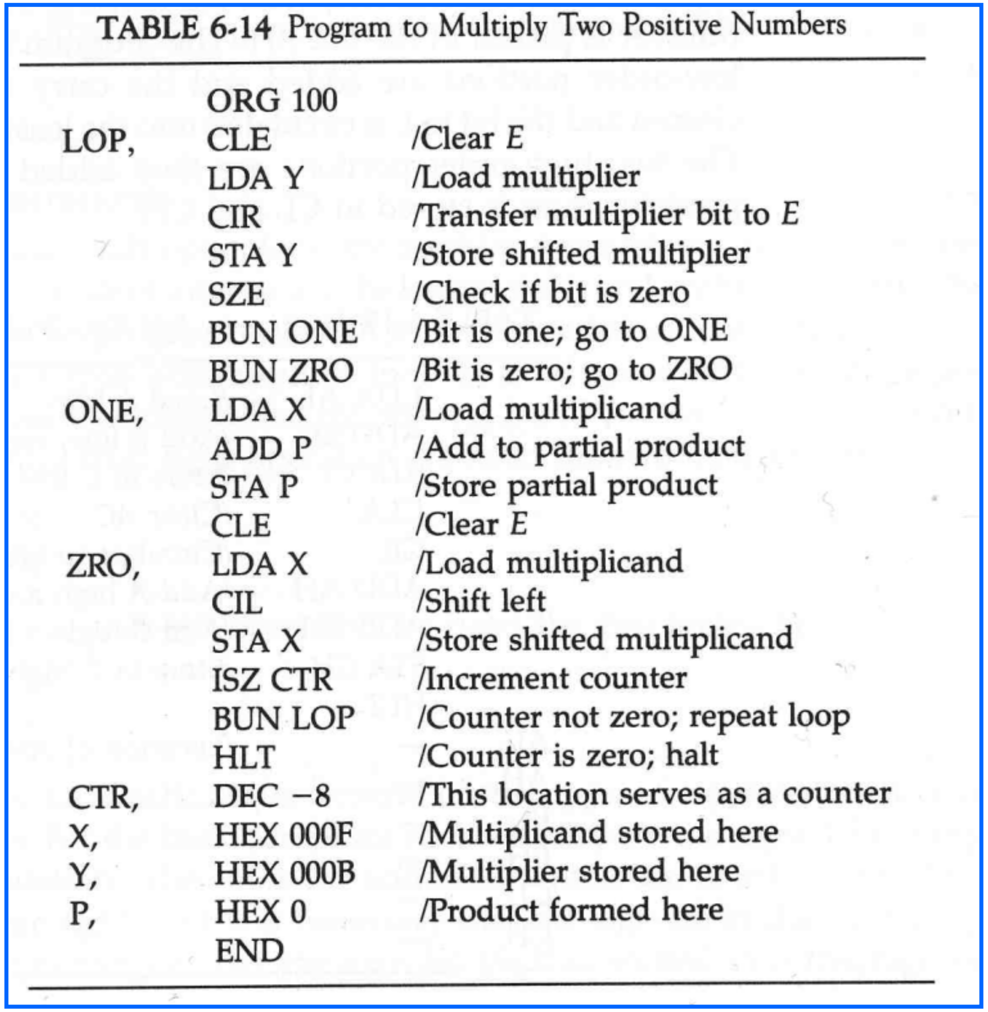


TABLE 6-14 Program to Multiply Two Positive Numbers

```
          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
```

- 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:

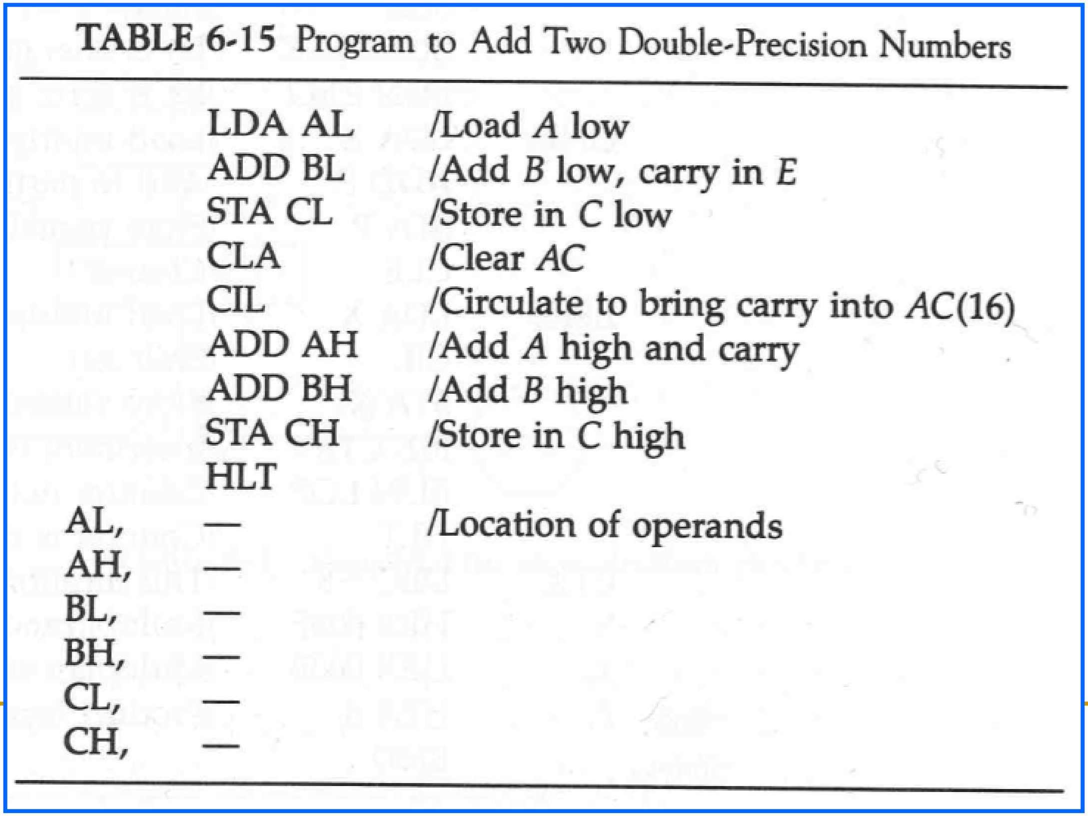
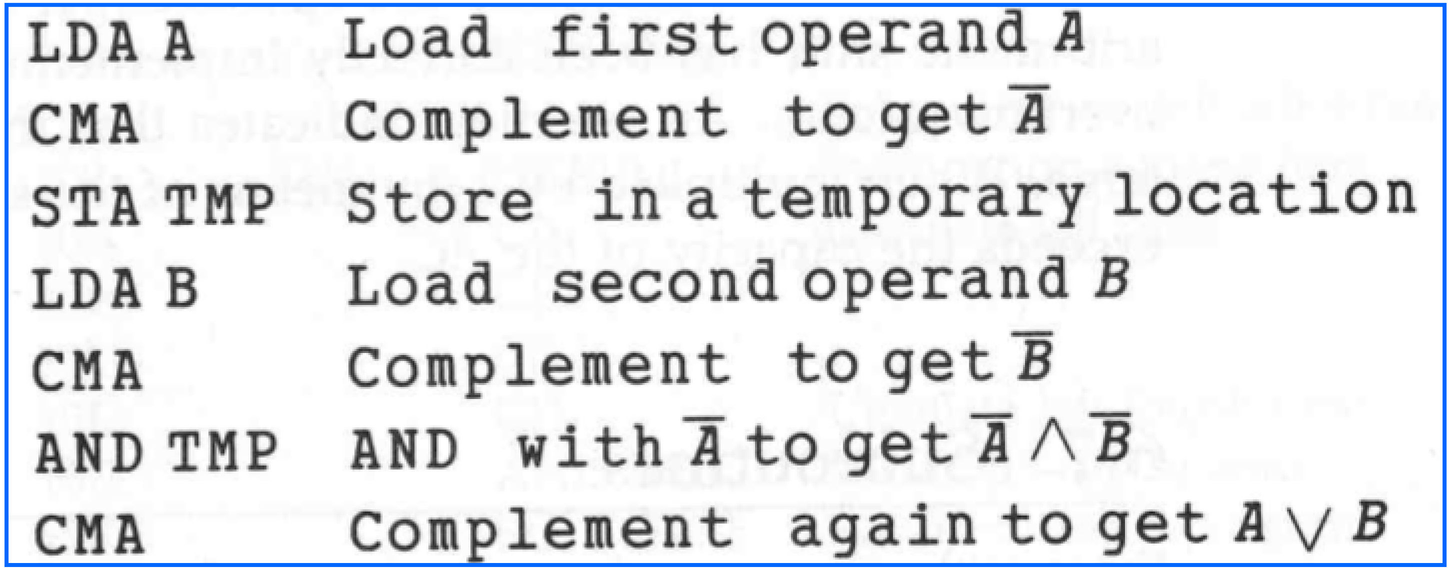


TABLE 6-15 Program to Add Two Double-Precision Numbers

```
          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,       —
```

- 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 $\bar{A}$
STA TMP	Store in a temporary location
LDA B	Load second operand B
CMA	Complement to get $\bar{B}$
AND TMP	AND with $\bar{A}$ to get $\bar{A} \wedge \bar{B}$
CMA	Complement again to get $A \vee B$

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

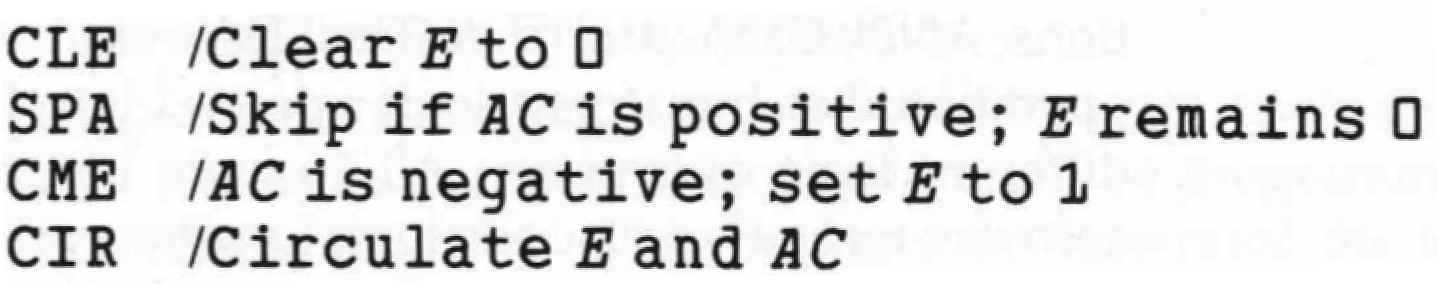
- The basic computer has two shift instructions: CIL, CIR. Logical and arithmetic shifts can be programmed.
- Logical shift-right (zeros added to the leftmost position):

```
CLE
CIR
```

- Logical shift-left (zeros added to the rightmost position):

```
CLE
CIL
```

- 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.
- E.g. a subroutine (Table 6-17) for shifting the content of AC four times to the left.

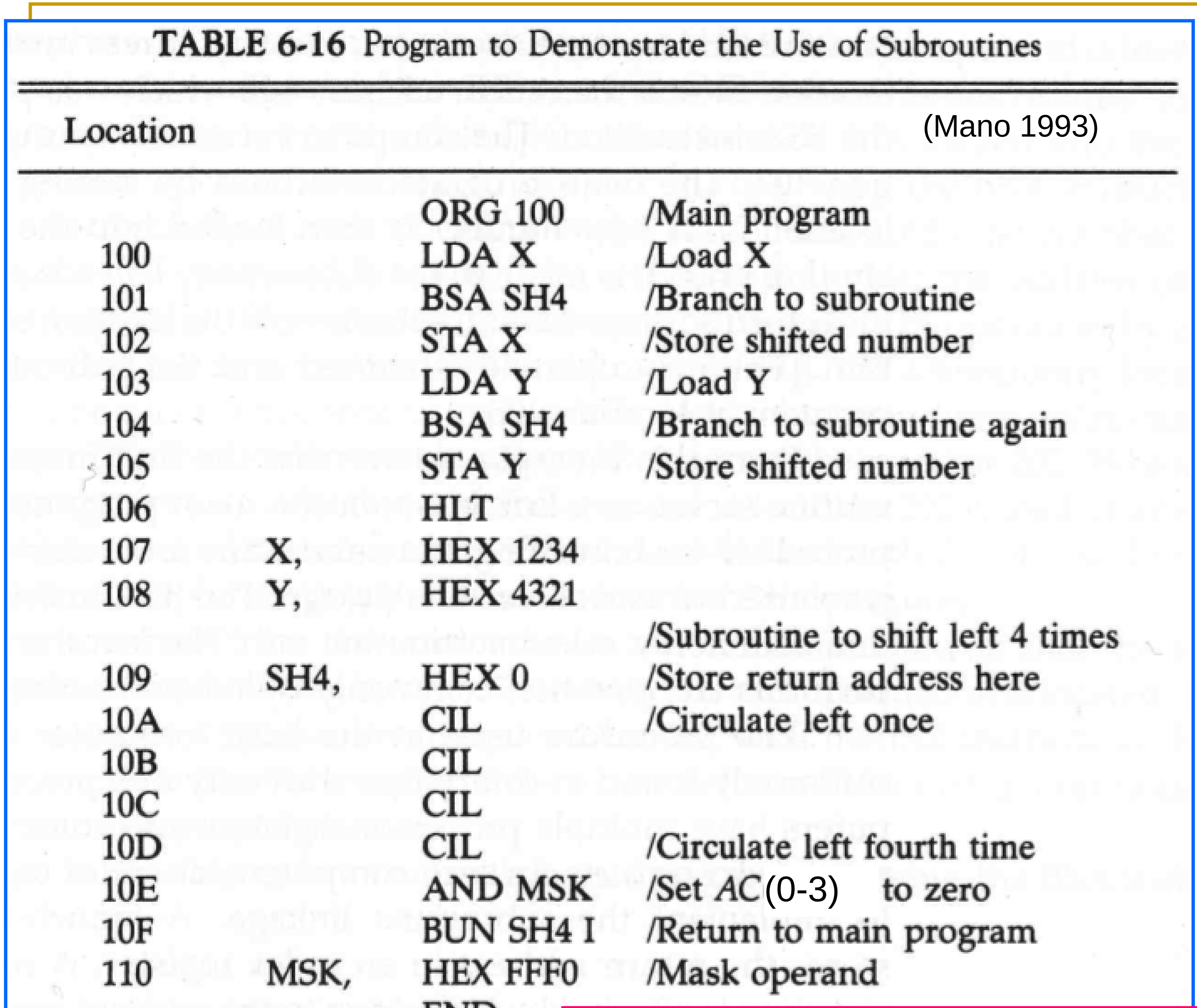
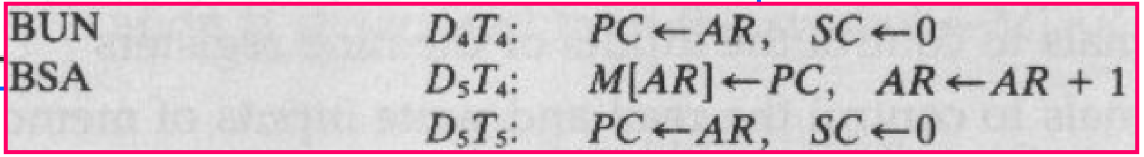


TABLE 6-16 Program to Demonstrate the Use of Subroutines

(Mano 1993)

```
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
```



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: $PC \leftarrow AR, \quad SC \leftarrow 0$

- From the example (Table 6-17) 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 <u>parameter linkage</u> (Table 6-17): OR operation.
- The subroutine must increment the return address for each operand.
- *E.g.* of subroutine to move <u>a block of data</u> is presented in Table 6-18.

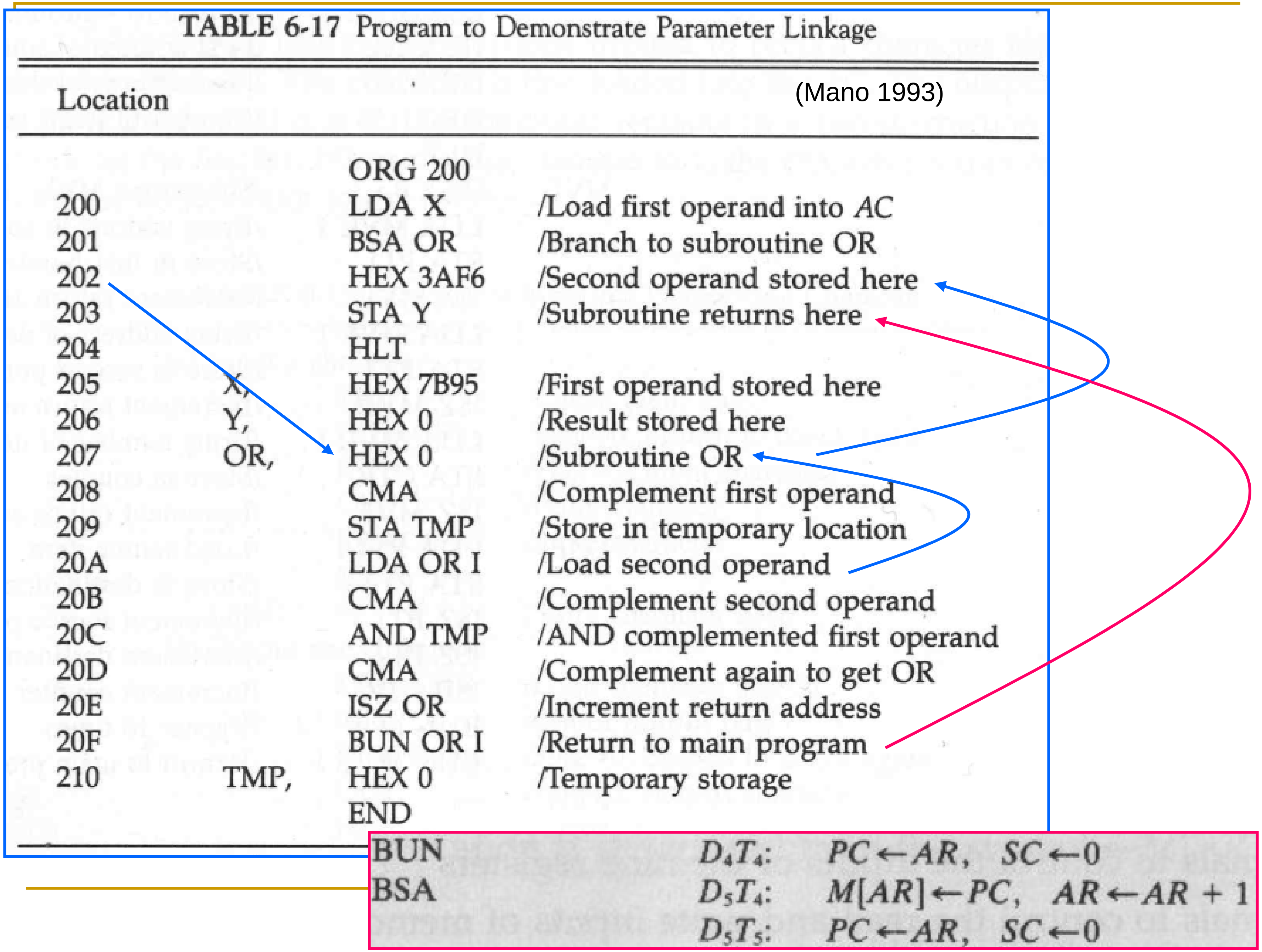


TABLE 6-17 Program to Demonstrate Parameter Linkage

(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: $PC \leftarrow AR, \quad SC \leftarrow 0$

return address
must be incremented
three times

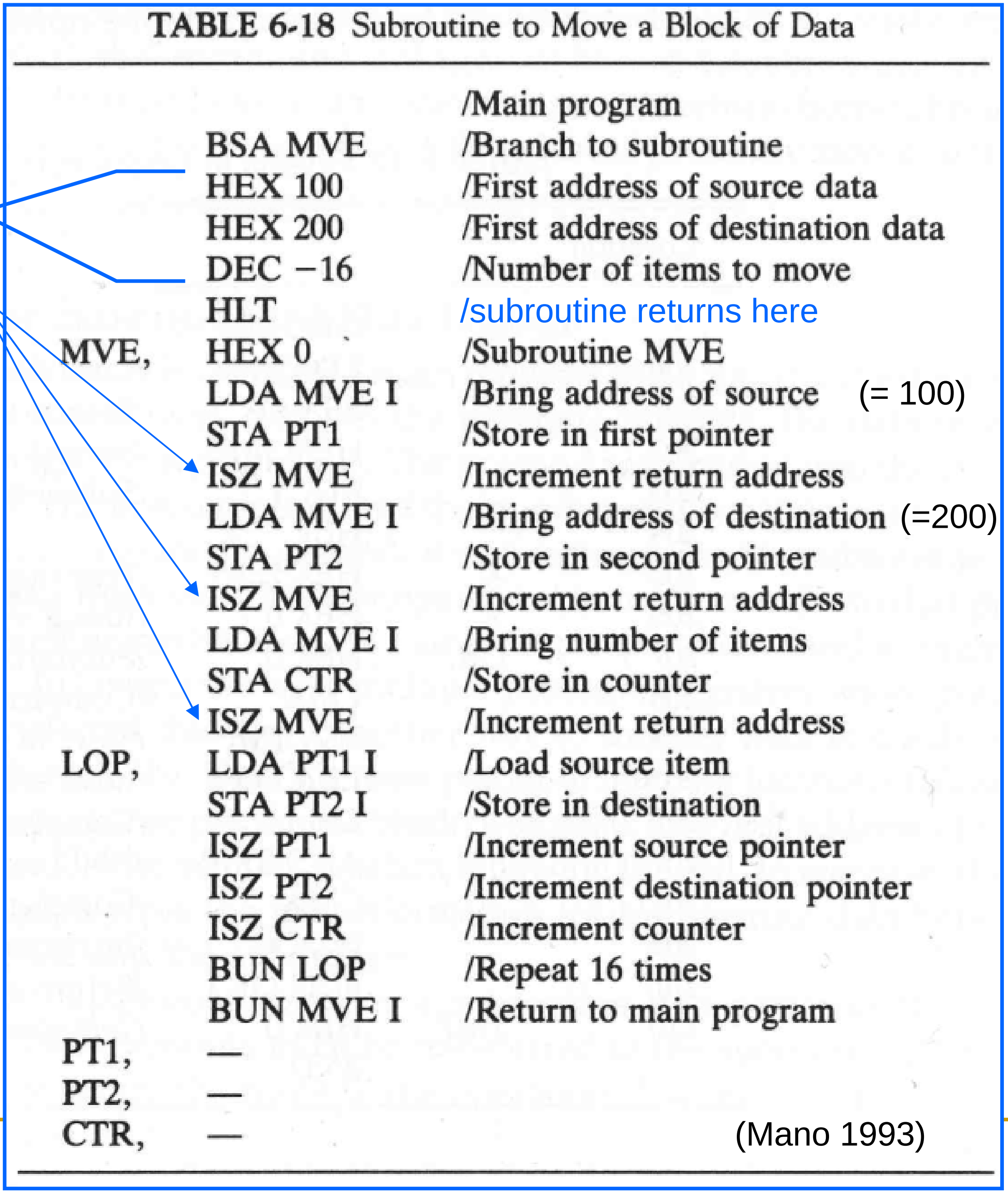


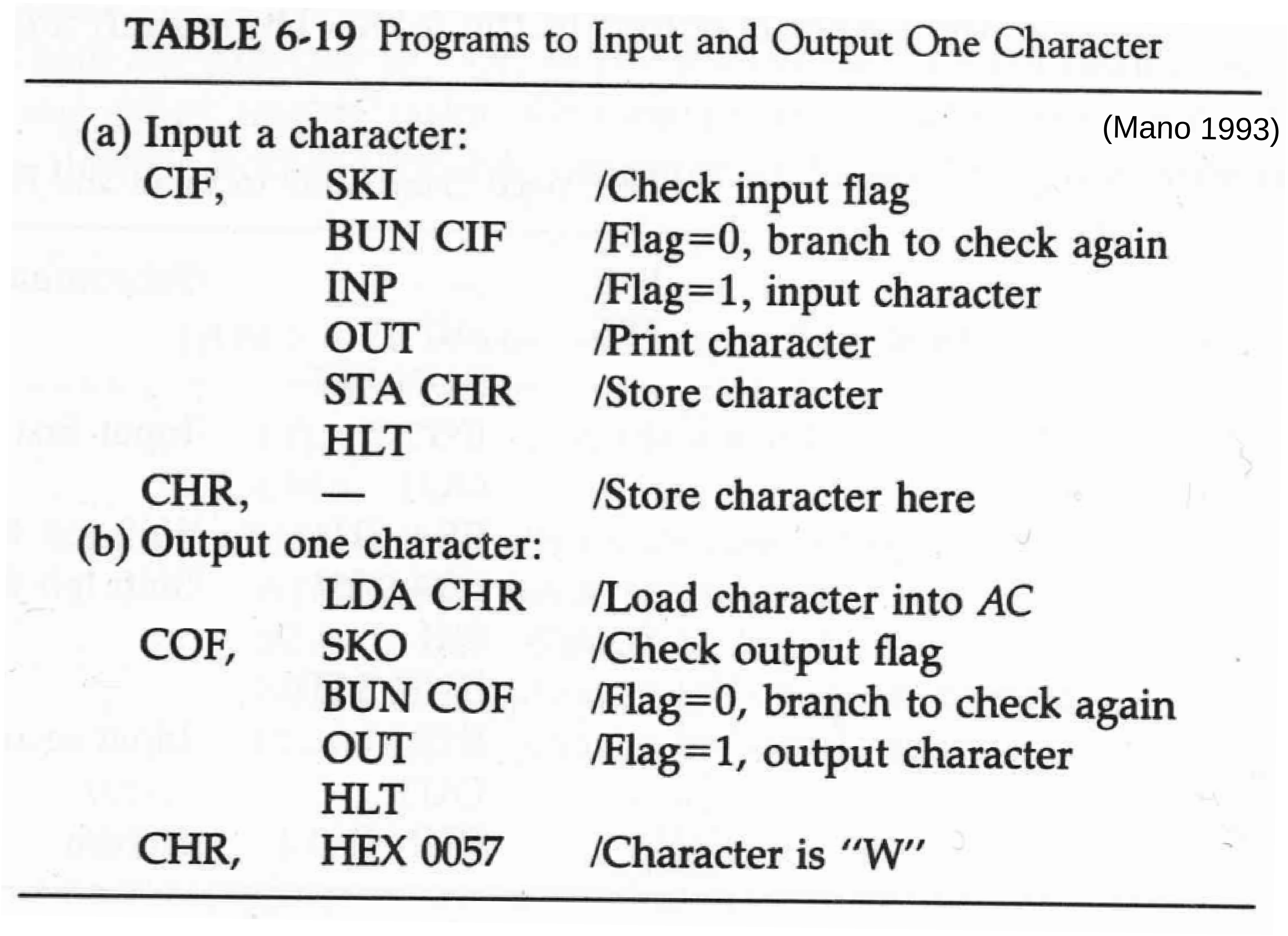
TABLE 6-18 Subroutine to Move a Block of Data

```
                  /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)

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.
- Table 6-19 lists programs for the Basic Computer to input and output one character: non-interrupt based programs.



(Mano 1993)

TABLE 6-19 Programs to Input and Output One Character

```
(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:

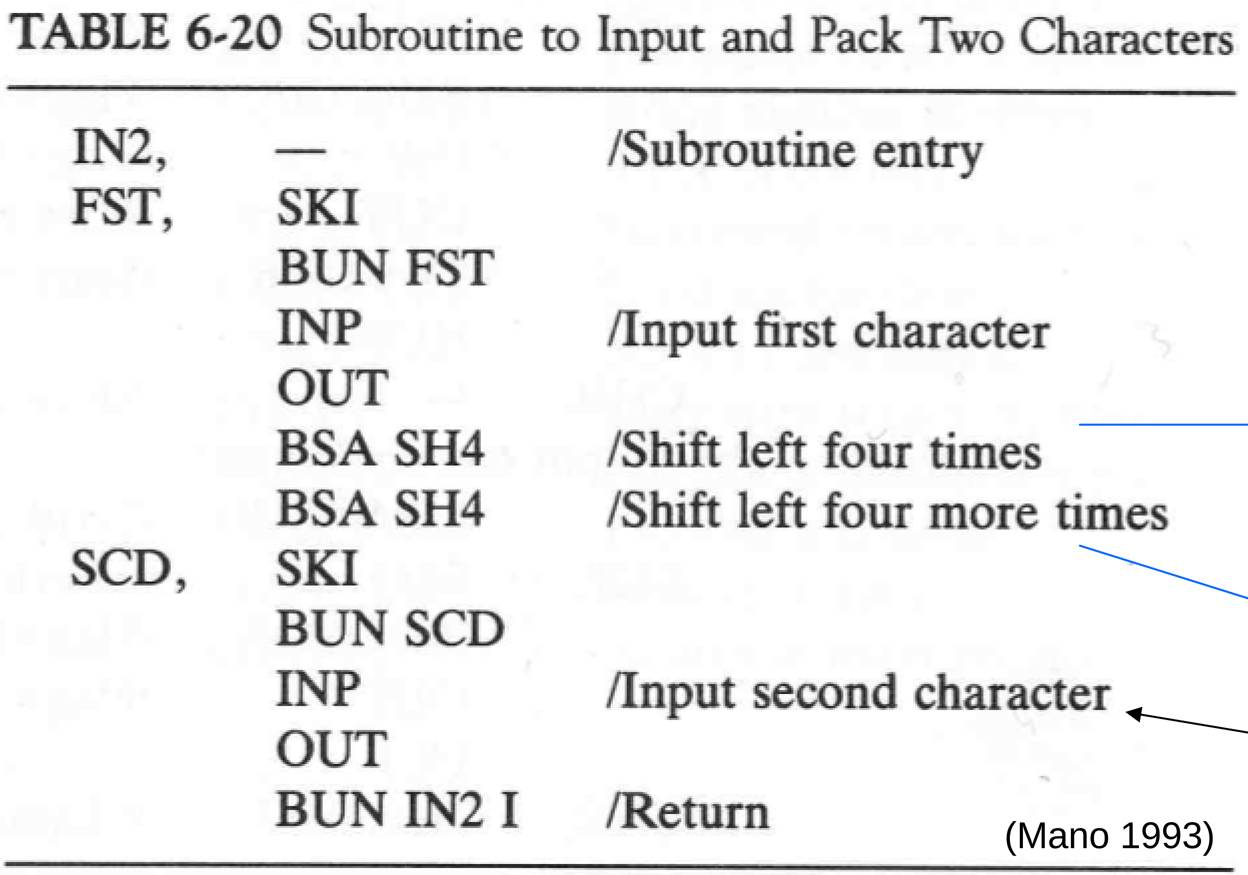


TABLE 6-20 Subroutine to Input and Pack Two Characters

```
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 (Table 6-21) 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.

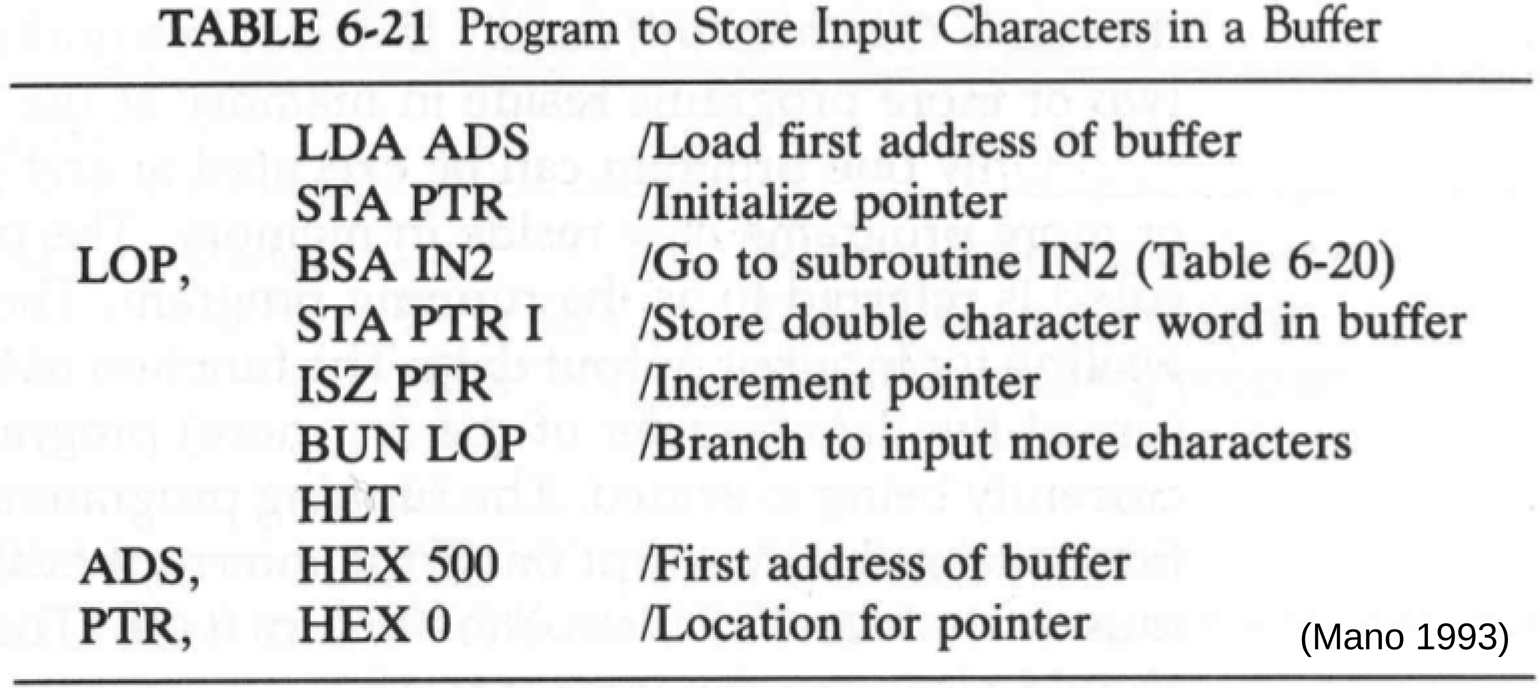


TABLE 6-21 Program to Store Input Characters in a Buffer

```
        LDA ADS     /Load first address of buffer
        STA PTR     /Initialize pointer
LOP,    BSA IN2     /Go to subroutine IN2 (Table 6-20)
        STA PTR I   /Store double character word in buffer
        ISZ PTR     /Increment pointer
        BUN LOP     /Branch to input more characters
        HLT
ADS,    HEX 500     /First address of buffer
PTR,    HEX 0       /Location for pointer
```

(Mano 1993)

- The fourth example (Table 6-22) describes a program that compares two memory words: the program can be utilized, for example, when implementing assembler program’s second-pass table lookup procedures.

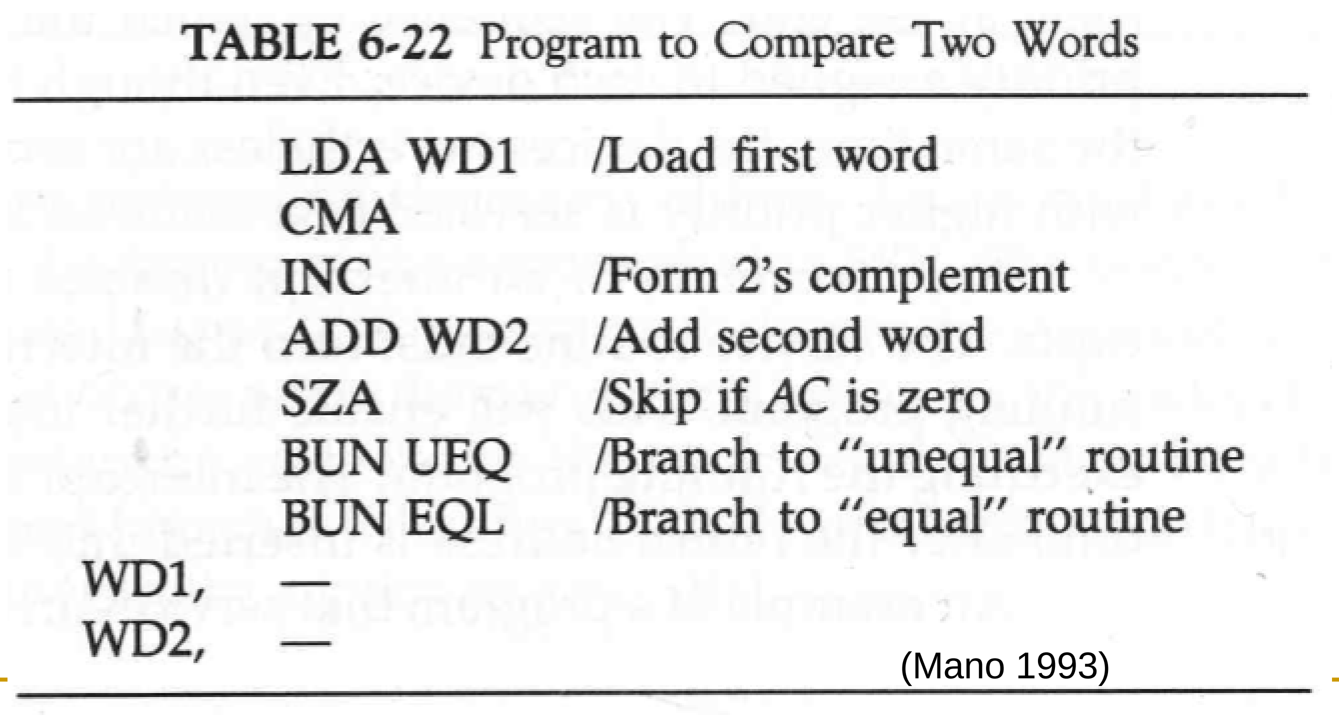


TABLE 6-22 Program to Compare Two Words

```
        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.
- E.g. in Table 6-23.

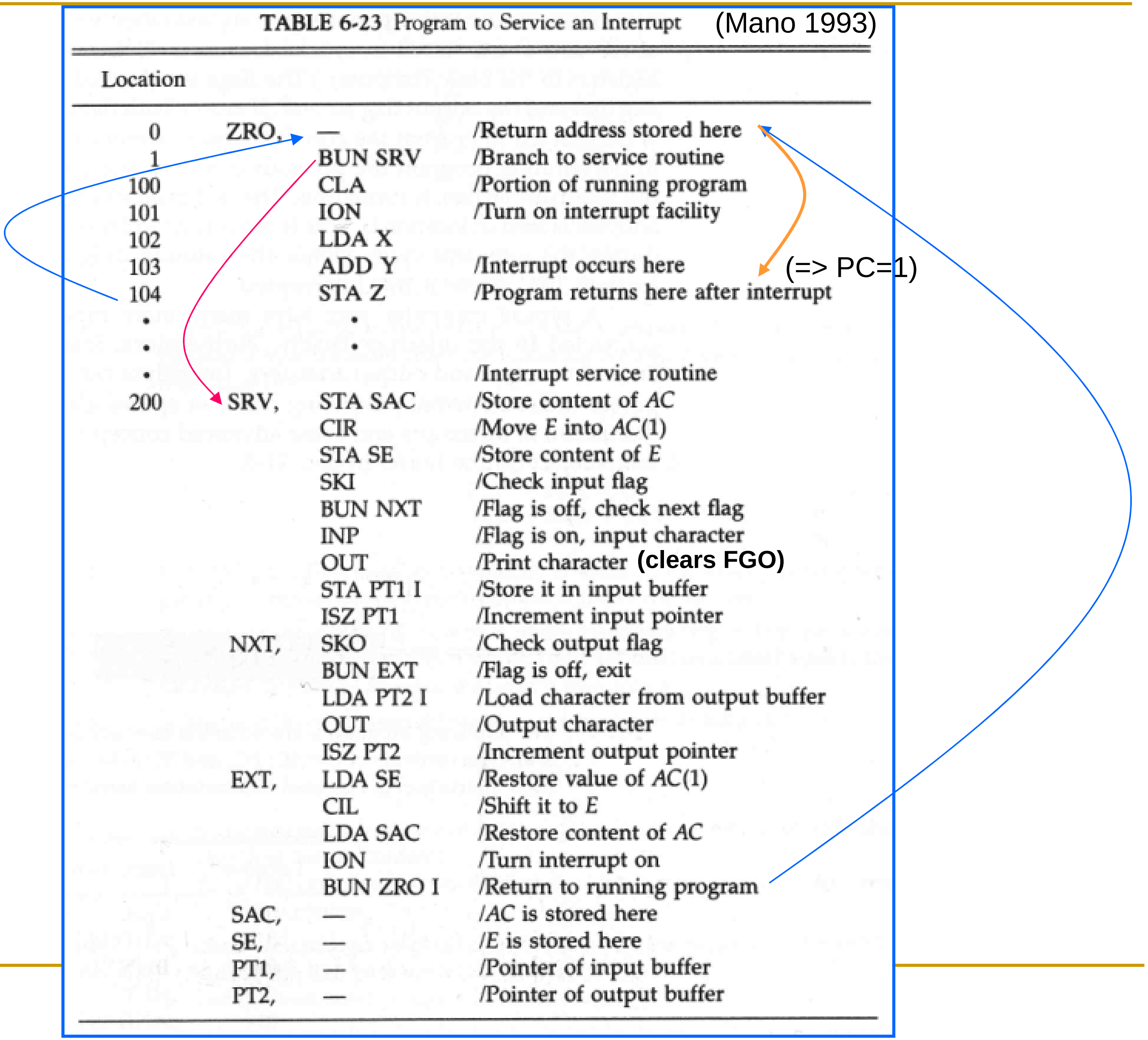


TABLE 6-23 Program to Service an Interrupt (Mano 1993)

```
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
```