Chapter 6:

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**
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.
- 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.

Hierarchy of programming languages

- 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***
- High-level programming language:
  C++ : used to write procedures to solve a problem or implement a task.
- Assembly language: concerned with the computer hardware behavior.

Binary/Hex Code

- Binary Program to Add Two Numbers:

Location	Instruction Code	Location	Instruction
0	0010 0000 0000 0100	000	2004
1	0001 0000 0000 0101	001	1005
10	0011 0000 0000 0110	002	3006
11	0111 0000 0000 0001	003	7001
100	0000 0000 0101 0011	004	0053
101	1111 1111 1110 1001	005	FFE9
110	0000 0000 0000 0000	006	0000
	Binary code	Hexadecimal code	

- It is hard to understand the task of the program
  ➔ symbolic code

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

- 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.

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
```

Assembly-Language Program

- A further step is to replace:

  hexadecimal address → symbolic address,
  hexadecimal operand → decimal operand.

If the operands are placed in memory following the instructions, and if the length of the program is not known in advance, the numerical location of operands is not known until the end of program is reached.

Decimal numbers are more familiar than hex. equivalents.

Assembly Language

- Following the rules of the language → the programs will be translated correctly.
- Almost every commercial computer has its own particular assembly language.

Rules of the Language

- Each line of an assembly language program is arranged in three columns called fields:

  1- The **Label** field: May be empty or specify a symbolic address.

  2- The **Instruction** field: Specifies a machine instruction or pseudo instruction.

  3- The **Comment field:** May be empty or it may include a comment.

  – Example:

  ORG Lab

  Lab, ADD op1 / this is an add operation.

  Label Instruction comment

  Note that Lab is a symbolic address.

Cont'

- 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.

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***

Memory-Reference Inst. / cont.

- 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. Ex:

```
        LDA X1
X1,     HEX 40
```

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,..).

Pseudo instruction

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

Ex:

ORG N

Hexadecimal number N is the memory loc. 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

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:

- An 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

Label   Instruction  comment
```

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

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

Example: storing the symbolic program in Memory

- PL3, LDA SUB I
- By referring to the ASCII code table, we get:

Memory word	Symbol	Hex code
1	P L	50 4C
2	3 ,	33 2C
3	L D	4C 44
4	A	41 20
5	S U	53 55
6	B	42 20
7	I CR	49 0D

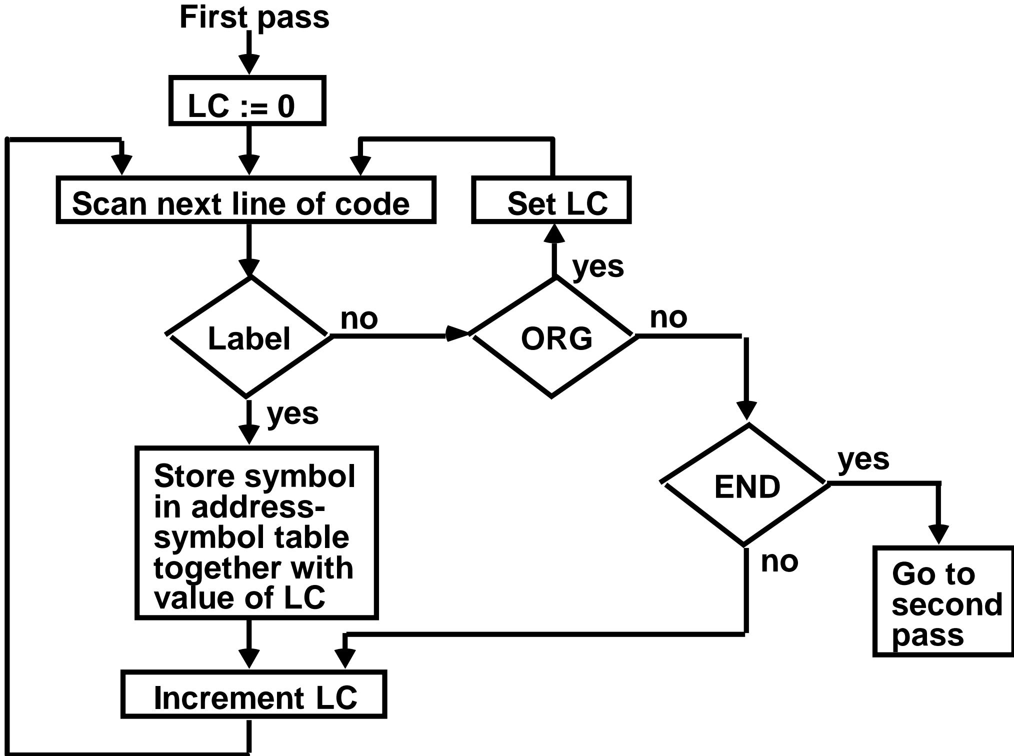
First Pass

- 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

First Pass /cont.

- 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/ cont.



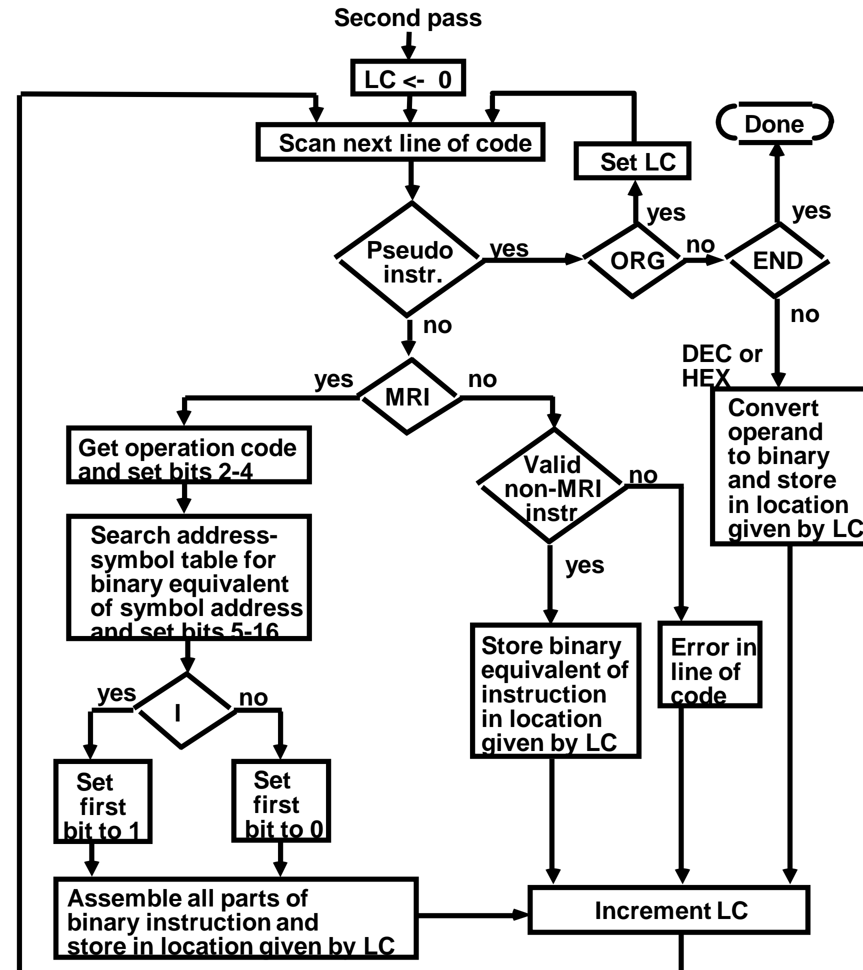
Second Pass

- 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/ cont.



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)
```

Program Loops/ cont.

```
        ORG  100        / Origin of program is HEX 100
        LDA  ADS        / Load first address of operand
        STA  PTR        / Store in pointer
        LDA  NBR        / Load -100
        STA  CTR        / Store in counter
        CLA             / Clear AC
LOP,    ADD  PTR  I     / Add an operand to AC
        ISZ  PTR        / Increment pointer
        ISZ  CTR        / Increment counter
        BUN  LOP        / Repeat loop again
        STA  SUM        / Store sum
        HLT             / Halt
ADS,    HEX  150        / First address of operands
PTR,    HEX  0          / Reserved for a pointer
NBR,    DEC  -100       / Initial value for a counter
CTR,    HEX  0          / Reserved for a counter
SUM,    HEX  0          / Sum is stored here
        ORG  150        / Origin of operands is HEX 150
        DEC  75         / First operand
        .
        .
        DEC  23         / Last operand
        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

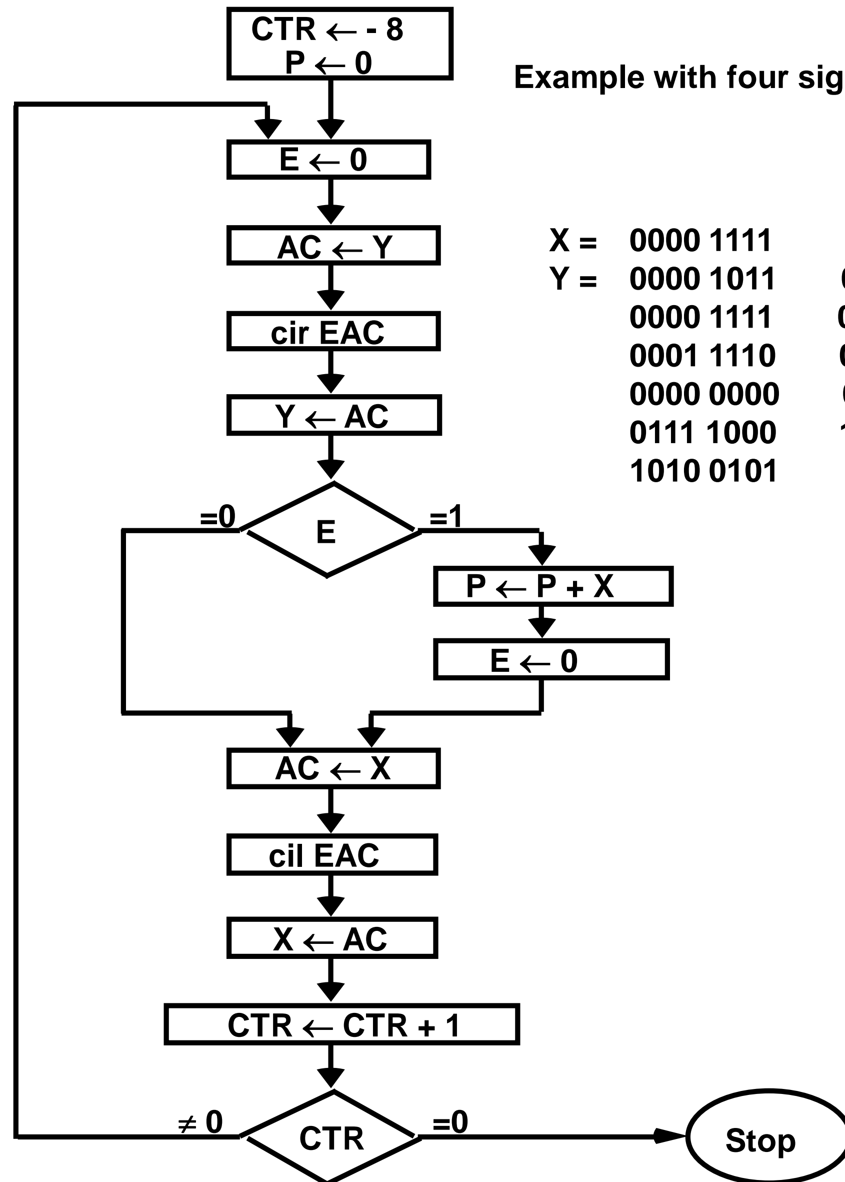
- We will develop a program to multiply two numbers.
- Assume positive numbers and neglect the sign bit for simplicity.
- Also, assume that the two numbers have no more than 8 significant bits → 16-bit product.

Multiplication / cont.

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



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

```
        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; goto ONE
        BUN  ZRO     / Bit is zero; goto 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

Double Precision Addition / cont.

- 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 / cont.

```
        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
                        / Location of operands
AL,     ______
AH,     ______
BL,     ______
BH,     ______
CL,     ______
CH,     ______
```

Logic Operations

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

Buffer

- The symbolic program is stored in a section of the memory called the *buffer.*
- A buffer is a set of consecutive memory locations that stores data entered via the input device. Ex: store input characteres in a buffer

```
        LDA ADS     / Load first address of buffer
        STA PTR     / Initialize pointer
LOP,    BSA IN2     /Go to subroutine IN2
        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
```

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

```
Loc.
  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
```