Module -1

System software

System software consists of variety of programs that supports the operation of a computer.

System Software	Application Software
Used for operating computer hardware	Used by user to perform specific task
Installed on the computer when operating system is installed.	Installed according to user's requirements
The user does not interact with system software because it works in the background	The user interacts with application software.
Run independently	Can't run without the presence of system software
Example: (compiler, assembler, debugger, driver, etc)	Example: (word processor, web browser, media player, etc.)

Simplified Instructional Computer (SIC)

Simplified Instructional Computer (SIC) is a hypothetical computer that has hardware features which are often found in real machines. There are two versions of this machine:

1. SIC standard Model
2. SIC/XE(extra equipment or expensive)

Object program for SIC can be properly executed on SIC/XE which is known as upward compatability.

SIC Machine Architecture/Components –

1. **Memory –**

SIC Memory

- Memory storage in SIC consists of 8-bit bytes, and all memory addresses in SIC are byte addresses.
- Any 3 consecutive bytes form a word (24 bits)
  - All addresses in SIC are byte address.
  - Words are addressed by the location of their lowest numbered byte.
- Total of 2^{15} (32,768) bytes of total memory
  - Least significant part of a numeric value is stored at the lowest numbered address

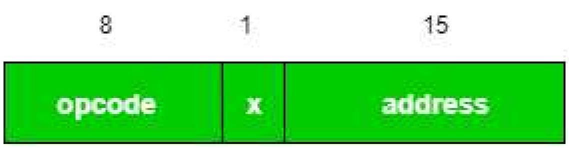
2. **Registers –**

SIC Registers

Register	Number	Function	Use
A	0	Accumulator	Arithmetic operations
X	1	Index register	Addressing
L	2	Linkage register	Storing return address for subroutine jump instruction (JSUB)
PC	8	Program counter	Contains Address of next instruction to be fetched for execution
SW	9	Status word	Flags, other information including Condition Code (CC)

3. **Data Format –**
   - Integers are represented by 24 bit.
   - Negative numbers are represented in 2's complement.
   - Characters are represented by 8 bit ASCII value.
   - No floating point representation is available.

4. **Instruction Format –**
   All instructions in SIC have 24 bit format.



   - If x=0 it means direct addressing mode.
   - If x=1 it means indexed addressing mode.
5. **Instruction Set –**
   - Load And Store Instructions: To move or store data from accumulator to memory or vice-versa. For example LDA, STA, LDX, STX etc.
   - Comparison Instructions: Used to compare data in memory by contents in accumulator. For example COMP data.
   - Arithmetic Instructions: Used to perform operations on accumulator and memory and store result in accumulator. For example ADD, SUB, MUL, DIV etc.
   - Conditional Jump: compare the contents of accumulator and memory and performs task based on conditions. For example JLT, JEQ, JGT
   - Subroutine Linkage: Instructions related to subroutines. For example JSUB, RSUB
6. **Input and Output –**
   It is performed by transferring 1 byte at a time from or to rightmost 8 bits of accumulator. Each device has 8 bit unique code.
   There are 3 I/O instructions:
   - Test Device (TD) tests whether device is ready or not. Condition code in Status Word Register is used for this purpose. If cc is < then device is ready otherwise device is busy.
   - Read data(RD) reads a byte from device and stores in register A.
   - Write data(WD) writes a byte from register A to the device.

Extensions for SIC/XE:

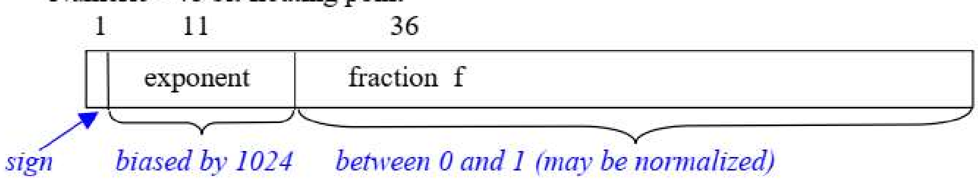
Memory: 2^{20} (1 M) bytes

Added Registers:

mnemonic	number	
B	3	base register
S	4	general working register
T	5	general working register
F	6	floating point accumulator; it uses the 24 bits that could be R7 to provide a 48 bit register

Added Data formats:

Numeric - 48 bit floating point



The actual exponent: exponent - 1024

value represented = (sgn) $f \times 2^{(\text{exponent - 1024})}$

Instruction formats: (4 in all)

1. 1-byte format (e.g., SIO, HIO, TIO, NORM) - not used in COP 3601

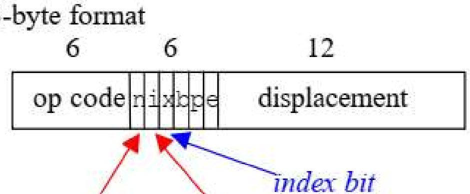
8
op code

2. 2-byte format

8	4	4
op code	R1	R2

The two addresses typically represent registers, so no memory access is needed to execute these.

3. 3-byte format



If the "n-bit" and the "i-bit" are both 0 then the instruction is interpreted as a simple SIC instruction.

4. 4-byte format

6			6				20
op code	n	i	x	b	p	e	address

n = indirect bit
i = immediate bit
x = index bit
b = base bit
p = PC relative bit
e = extended bit

These bits alone or in combination determine variations of the instruction interpretation:

e = 0 $\Rightarrow$ *3 byte format*
e = 1 $\Rightarrow$ *4 byte format*

x = 1 $\Rightarrow$ *indexed addressing*

b = 1 and p = 0 $\Rightarrow$ *base/displacement* addressing
b = 0 and p = 1 $\Rightarrow$ *PC relative* addressing

n = 1 and i = 1 $\Rightarrow$ *direct* addressing
n = 1 and i = 0 $\Rightarrow$ *indirect* addressing
n = 0 and i = 1 $\Rightarrow$ *immediate* addressing

n = 0 and i = 0 $\Rightarrow$ *simple SIC* interpretation
(so the last 15 bits is treated as an address, including the *bpe* bits).

SIC/XE Addressing Mode[1]

- Addressing modes
  - Base relative/b (n=1, i=1, b=1, p=0)
  - Program-counter relative/p (n=1, i=1, b=0, p=1)
  - Direct/ (n=1, i=1, b=0, p=0)
  - Immediate (n=0, i=1, x=0)
  - Indirect (n=1, i=0, x=0)
  - Indexing (both n & i = 0 or 1, x=1)
  - Extended (e=1)
- Note: Indexing cannot be used with immediate or indirect addressing

SIC/XE Instruction Set

- New registers: LDB, STB, etc.
- Floating-point arithmetic: ADDF, SUBF, MULF, DIVF
- Register move: RMO
- Register-register arithmetic: ADDR, SUBR, MULR, DIVR
- Supervisor call: SVC
  - Generates an interrupt for OS

SIC/XE Input/Output

- 1 byte at a time, TD, RD, and WD
- SIO, TIO, HIO:
  - start, test, halt the operation of I/O device

- Looping and indexing: copy one string to another

```
          LDT     #11          initialize register T to 11
          LDX     #0           initialize index register to 0
MOVECH    LDCH    STR1,X       load char from STR1 to reg A
          STCH    STR2,X       store char into STR2
          TIXR    T            add 1 to index, compare to 11
          JLT     MOVECH       loop if "less than" 11
          .
          .
          .
STR1      BYTE    C'TEST STRING'
STR2      RESB    11
```

SIC Programming Example[2]

- Arithmetic operations: BETA = ALPHA+INCR-1

```
          LDA     ALPHA
          ADD     INCR
          SUB     ONE
          STA     BETA
          LDA     GAMMA
          ADD     INCR
          SUB     ONE
          STA     DELTA
          —
ONE       WORD    1            one-word constant
ALPHA     RESW    1            one-word variables
BETA      RESW    1
GAMMA     RESW    1
DELTA     RESW    1
INCR      RESW    1
```

Assembler Design

Assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to the assembler is a source code written in assembly language (using mnemonics) and the output is the object code. The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

1. Basic Assembler Functions:

The basic assembler functions are:
- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.

• The design of assembler can be to perform the following:
- Scanning (tokenizing)
- Parsing (validating the instructions)
- Creating the symbol table
- Resolving the forward references
- Converting into the machine language

• The design of assembler in other words:
- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Decide the proper instruction format Convert the data constants to internal machine representations
- Write the object program and the assembly listing

So for the design of the assembler we need to concentrate on the machine architecture of the SIC/XE machine. We need to identify the algorithms and the various data structures to be used. According to the above required steps for assembling the assembler also has to handle *assembler directives*, these do not generate the object code but directs the assembler to perform certain operation. These directives are:

• SIC Assembler Directive:
- START: Specify name & starting address.
- END: End of the program, specify the first execution instruction.

– BYTE, WORD, RESB, RESW
– End of record: a null char(00)
End of file: a zero length record

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

Single-pass Assembler:

In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference. This is shown with an example below:

```
10      1000        FIRST       STL     RETADR          141033
--
--
--
--
95      1033        RETADR      RESW        1
```

In the above example in line number 10 the instruction STL will store the linkage register with the contents of RETADR. But during the processing of this instruction the value of this symbol is not known as it is defined at the line number 95. Since I single-pass assembler the scanning, parsing and object code conversion happens simultaneously. The instruction is fetched; it is scanned for tokens, parsed for syntax and semantic validity. If it valid then it has to be converted to its equivalent object code. For this the object code is generated for the opcode STL and the value for the symbol RETADR need to be added, which is not available.

Due to this reason usually the design is done in two passes. So a multi-pass assembler resolves the forward references and then converts into the object code. Hence the process of the multi-pass assembler can be as follows:

Pass-1

- Assign addresses to all the statements
- Save the addresses assigned to all labels to be used in *Pass-2*
- Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
- Defines the symbols in the symbol table(generate the symbol table)

Pass-2

- Assemble the instructions (translating operation codes and looking up addresses).
- Generate data values defined by BYTE, WORD etc.
- Perform the processing of the assembler directives not done during *pass-1*.
- Write the object program and assembler listing.

Assembler Design:

The most important things which need to be concentrated is the generation of Symbol table and resolving *forward references*.

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

Example Program:

The example program considered here has a main module, two subroutines

- Purpose of example program
  - Reads records from input device (code F1)
  - Copies them to output device (code 05)
  - At the end of the file, writes EOF on the output device, then RSUB to the operating system
- Data transfer (RD, WD)
  - -A buffer is used to store record
  - -Buffering is necessary for different I/O rates
  - -The end of each record is marked with a null character (00)16
  - -The end of the file is indicated by a zero-length record
- Subroutines (JSUB, RSUB)
  - -RDREC, WRREC
  - -Save link register first before nested jump

```
Loc     Source statement                    Object code

1000    COPY     START   1000
1000    FIRST    STL     RETADR             141033
1003    CLOOP    JSUB    RDREC              482039
1006             LDA     LENGTH             001036
1009             COMP    ZERO               281030
100C             JEQ     ENDFIL             301015
100F             JSUB    WRREC              482061
1012             J       CLOOP              3C1003
1015    ENDFIL   LDA     EOF                00102A
1018             STA     BUFFER             0C1039
101B             LDA     THREE              00102D
101E             STA     LENGTH             0C1036
1021             JSUB    WRREC              482061
1024             LDL     RETADR             081033
1027             RSUB                       4C0000
102A    EOF      BYTE    C'EOF'             454F46
102D    THREE    WORD    3                  000003
1030    ZERO     WORD    0                  000000
1033    RETADR   RESW    1
1036    LENGTH   RESW    1
1039    BUFFER   RESB    4096
         .
```

```
         .
         .       SUBROUTINE TO READ RECORD INTO BUFFER
         .
2039    RDREC    LDX     ZERO               041030
203C             LDA     ZERO               001030
203F    RLOOP    TD      INPUT              E0205D
2042             JEQ     RLOOP              30203F
2045             RD      INPUT              D8205D
2048             COMP    ZERO               281030
204B             JEQ     EXIT               302057
204E             STCH    BUFFER,X           549039
2051             TIX     MAXLEN             2C205E
2054             JLT     RLOOP              38203F
2057    EXIT     STX     LENGTH             101036
205A             RSUB                       4C0000
205D    INPUT    BYTE    X'F1'              F1
205E    MAXLEN   WORD    4096               001000
```

```
195              .
200              .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205              .
210   2061   WRREC    LDX     ZERO            041030
215   2064   WLOOP    TD      OUTPUT          E02079
220   2067            JEQ     WLOOP           302064
225   206A            LDCH    BUFFER,X        509039
230   206D            WD      OUTPUT          DC2079
235   2070            TIX     LENGTH          2C1036
240   2073            JLT     WLOOP           382064
245   2076            RSUB                    4C0000
250   2079   OUTPUT   BYTE    X'05'           05
255                   END     FIRST
```

The first column shows the line number for that instruction, second column shows the addresses allocated to each instruction. The third column indicates the labels given to the statement, and is followed by the instruction consisting of opcode and operand. The last column gives the equivalent object code.

The *object code* later will be loaded into memory for execution. The simple object program we use contains three types of records:

- Header record
  - Col. 1 H
  - Col. 2~7 Program name
  - Col. 8~13 Starting address of object program (hex)
  - Col. 14~19 Length of object program in bytes (hex)
- Text record
  - Col. 1 T
  - Col. 2~7 Starting address for object code in this record (hex)
  - Col. 8~9 Length of object code in this record in bytes (hex)
  - Col. 10~69 Object code, represented in hex (2 col. per byte)
- End record
  - Col.1 E
  - Col.2~7 Address of first executable instruction in object program (hex) "^" is only for separation only

Object code for the example program:

Session –II

1. Simple SIC Assembler

The program below is shown with the object code generated. The column named LOC gives the machine addresses of each part of the assembled program (assuming the program is starting at location 1000). The translation of the source program to the object program requires us to accomplish the following functions:

1. Convert the mnemonic operation codes to their machine language equivalent.
2. Convert symbolic operands to their equivalent machine addresses.
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into their internal machine representations in the proper format.
5. Write the object program and assembly listing.

All these steps except the second can be performed by sequential processing of the source program, one line at a time. Consider the instruction

10	1000	LDA	ALPHA	00-----

This instruction contains the forward reference, i.e. the symbol ALPHA is used is not yet defined. If the program is processed (scanning and parsing and object code conversion) is done line-by-line, we will be unable to resolve the address of this symbol. Due to this problem most of the assemblers are designed to process the program in two passes.

In addition to the translation to object program, the assembler has to take care of handling assembler directive. These directives do not have object conversion but gives direction to the assembler to perform some function. Examples of directives are the statements like BYTE and WORD, which directs the assembler to reserve memory locations without generating data values. The other directives are START which indicates the beginning of the program and END indicating the end of the program.

The assembled program will be loaded into memory for execution. The simple object program contains three types of records: Header record, Text record and end record. The header record contains the starting address and length. Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded. The end record marks the end of the object program and specifies the address where the execution is to begin.

The format of each record is as given below.

Header record:

Col 1	H
Col. 2-7	Program name
Col 8-13	Starting address of object program (hexadecimal)
Col 14-19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T

Col 2-7.	Starting address for object code in this record (hexadecimal)
Col 8-9	Length off object code in this record in bytes (hexadecimal)
Col 10-69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

Col. 1	E
Col 2-7	Address of first executable instruction in object program (hexadecimal)

The assembler can be designed either as a single pass assembler or as a two pass assembler. The general description of both passes is as given below:

- Pass 1 (define symbols)
  - Assign addresses to all statements in the program
  - Save the addresses assigned to all labels for use in Pass 2
  - Perform assembler directives, including those for address assignment, such as BYTE and RESW
- Pass 2 (assemble instructions and generate object program)
  - Assemble instructions (generate opcode and look up addresses)
  - Generate data values defined by BYTE, WORD
  - Perform processing of assembler directives not done during Pass 1
  - Write the object program and the assembly listing

2. Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

OPTAB: It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length. In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR. In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB: This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).

During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.

SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2. A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

LOCCTR: Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

The Algorithm for Pass 1:

```
Begin
 read first input line
  if OPCODE = ‘START’ then begin
     save #[Operand] as starting addr
     initialize LOCCTR to starting address
     write line to intermediate file
      read next line
    end( if START)
   else
     initialize LOCCTR to 0
```

```
   While OPCODE != 'END' do
    begin
      if this is not a comment line then
          begin
            if there is a symbol in the LABEL field then
               begin
                 search SYMTAB for LABEL
                  if found then
                     set error flag (duplicate symbol)
                  else
                     (if symbol)
             search OPTAB for OPCODE
             if found then
                 add 3 (instr length) to LOCCTR
             else if OPCODE = 'WORD' then
                  add 3 to LOCCTR
             else if OPCODE = 'RESW' then
                   add 3 * #[OPERAND] to LOCCTR
             else if OPCODE = 'RESB' then
                   add #[OPERAND] to LOCCTR
             else if OPCODE = 'BYTE' then
                   begin
                             find length of constant in bytes
                             add length to LOCCTR
                    end
             else
                   set error flag (invalid operation code)
        end (if not a comment)
        write line to intermediate file
         read next input line
    end { while not END}
    write last line to intermediate file
    Save (LOCCTR – starting address) as program length
End {pass 1}
```

The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line. If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value. If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol. It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction. If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes. If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

The Algorithm for Pass 2:

```
Begin
  read 1st input  line
    if OPCODE =  'START' then
      begin
         write listing line
          read next input line
       end
    write Header record to object program
    initialize 1st Text record
while OPCODE != 'END' do
     begin
        if this is not comment line then
           begin
              search OPTAB for OPCODE
                if found then
                   begin
                      if there is a symbol in OPERAND field then
                           begin
                               search SYMTAB for OPERAND field then
                               if found then
                               begin
                                          store symbol value as operand address
                               else
                                  begin
                                          store 0 as operand address
                                          set error flag (undefined symbol)
                                  end
                           end (if symbol)
                        else store 0 as operand address
                               assemble the object code instruction
                         else if OPCODE = 'BYTE' or 'WORD" then
                              convert constant to object code
                          if object code doesn't fit into current Text record then
                              begin
```

```
                    Write text record to object code
                     initialize new Text record

                  end
                  add object code to Text record
      end {if not comment}
     write listing line
     read next input line
    end
   write listing line
   read next input line
   write last listing line
  End {Pass 2}
```

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code. If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code(for example, for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written.

Design and Implementation Issues

Some of the features in the program depend on the architecture of the machine. If the program is for SIC machine, then we have only limited instruction formats and hence limited addressing modes. We have only single operand instructions. The operand is always a memory reference. Anything to be fetched from memory requires more time. Hence the improved version of SIC/XE machine provides more instruction formats and hence more addressing modes. The moment we change the machine architecture the availability of number of instruction formats and the addressing modes changes. Therefore the design usually requires considering two things: Machine-dependent features and Machine-independent features.

3. Machine-Dependent Features:

- Instruction formats and addressing modes
- Program relocation

3.1 Instruction formats and Addressing Modes

The instruction formats depend on the memory organization and the size of the memory. In SIC machine the memory is byte addressable. Word size is 3 bytes. So the size of the memory is 2^{12} bytes. Accordingly it supports only one instruction format. It has only two registers: register A and Index register. Therefore the addressing modes supported by this architecture are direct, indirect, and indexed. Whereas the memory of a SIC/XE machine is 2^{20} bytes (1 MB). This supports four different types of instruction types, they are:

- 1 byte instruction
- 2 byte instruction
- 3 byte instruction
- 4 byte instruction

- Instructions can be:
  - Instructions involving register to register
  - Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
  - Extended instruction format
- Addressing Modes are:
  - Index Addressing(SIC): Opcode m, x
  - Indirect Addressing: Opcode @m
  - PC-relative: Opcode m
  - Base relative: Opcode m
  - Immediate addressing: Opcode #c

1. Translations for the Instruction involving Register-Register addressing mode:

During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass 2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2. Translation involving Register-Memory instructions:

In SIC/XE machine there are four instruction formats and five addressing modes. For formats and addressing modes refer chapter 1.

Among the instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other operand is in the memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

There are two ways: *Program-counter relative and Base-relative.* This addressing mode can be represented by either using format-3 type or format-4 type of instruction format. In format-3, the instruction has the opcode followed by a 12-bit displacement value in the address field. Where as in format-4 the instruction contains the mnemonic code followed by a 20-bit displacement value in the address field.

2. Program-Counter Relative: In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction. This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:

```
10      0000         FIRST        STL          RETADR
```

RETADR is at address $(0030)_{16}$
After the SIC fetches this instruction, (PC) = $(0003)_{16}$
TA = (PC) + disp ⇒ disp = TA – (PC) = 0030 – 0003 = $(02D)_{16}$

op	n	i	x	b	p	e	disp	
000101	1	1	0	0	1	0	02D	⇒ 17202D

```
40      0017         J            CLOOP
```

CLOOP is at address $(0006)_{16}$
After the SIC fetches this instruction, (PC) = $(001A)_{16}$
TA = (PC) + disp ⇒ disp = TA – (PC) = 0006 – 001A = $(\boxed{FEC})_{16}$

op	n	i	x	b	p	e	disp	
001111	1	1	0	0	1	0	FEC (12-bits)	⇒ 3F2FEC

```
70      002A         J            @RETADR
```

Indirect addressing

CLOOP is at address $(0030)_{16}$
After the SIC fetches this instruction, (PC) = $(002D)_{16}$
TA = (PC) + disp ⇒ disp = TA – (PC) = 0030 – 002D = $(0003)_{16}$

op	n	i	x	b	p	e	disp	
001111	1	0	0	0	1	0	003	⇒ 3E2003

3. Base-Relative Addressing Mode: in this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE. The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.

When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

LDB #LENGTH (*instruction*)
BASE LENGTH (*directive*)
:
NOBASE

For example:

12	0003	LDB	#LENGTH		69202D
13		BASE	LENGTH		
: :					
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
: :					
160	104E	STCH	BUFFER,	X	57C003
165	1051	TIXR	T	B850	

In the above example the use of directive BASE indicates that Base-relative addressing mode is to be used to calculate the target address. PC-relative is no longer used. The value of the LENGTH is stored in the base register. If PC-relative is used then the target address calculated is:

The LDB instruction loads the value of length in the base register which 0033. BASE directive explicitly tells the assembler that it has the value of LENGTH.

BUFFER is at location $(0036)_{16}$
$(B) = (0033)_{16}$
disp = 0036 – 0033 = $(0003)_{16}$

op	n	i	x	b	p	e	disp		
010101	1	1	1	1	0	0	003	⇒	57C003

20	000A		LDA	LENGTH	032026
: :					
175	1056	EXIT	STX	LENGTH	134000

Consider Line 175. If we use PC-relative

Disp = TA – (PC) = 0033 –1059 = EFDA

PC relative is no longer applicable, so we try to use BASE relative addressing mode.

4. Immediate Addressing Mode

In this mode no memory reference is involved. If immediate mode is used the target address is the operand itself.

```
55     0020     LDA     #3
```

Immediate operand

TA = $(0003)_{16}$

op	n i	x b p e	disp
000000	0 1	0 0 0 0	003 ⇒ 010003

```
133    103C     +LDT    #4096
```

Extended instruction format

TA = $(01000)_{16}$

op	n i	x b p e	disp(20 bits)
011101	0 1	0 0 0 1	01000 ⇒ 75101000

If the symbol is referred in the instruction as the immediate operand then it is immediate with PC-relative mode as shown in the example below:

```
12     0003     LDB     #LENGTH
```

LENGTH is at address 0033

TA = (PC) + disp ⇒ disp = 0033 – 0006 = $(002D)_{16}$

op	n i	x b p e	disp
011010	0 1	0 0 1 0	02D ⇒ 69202D

***5. Indirect and PC-relative mode*:**

In this type of instruction the symbol used in the instruction is the address of the location which contains the address of the operand. The address of this is found using PC-relative addressing mode. For example:

```
70     002A             J       @RETADR
                        :       :
95     0030   RETADR    RESW    1
```

RETADR is at address 0030

TA = (PC) + disp ⇒ disp = 0030 – 002D = $(0003)_{16}$

op	n i	x b p e	disp
001111	1 0	0 0 1 0	003 ⇒ 3E2003

The instruction jumps the control to the address location RETADR which in turn has the address of the operand. If address of RETADR is 0030, the target address is then 0003 as calculated above.

3.2 Program Relocation

Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

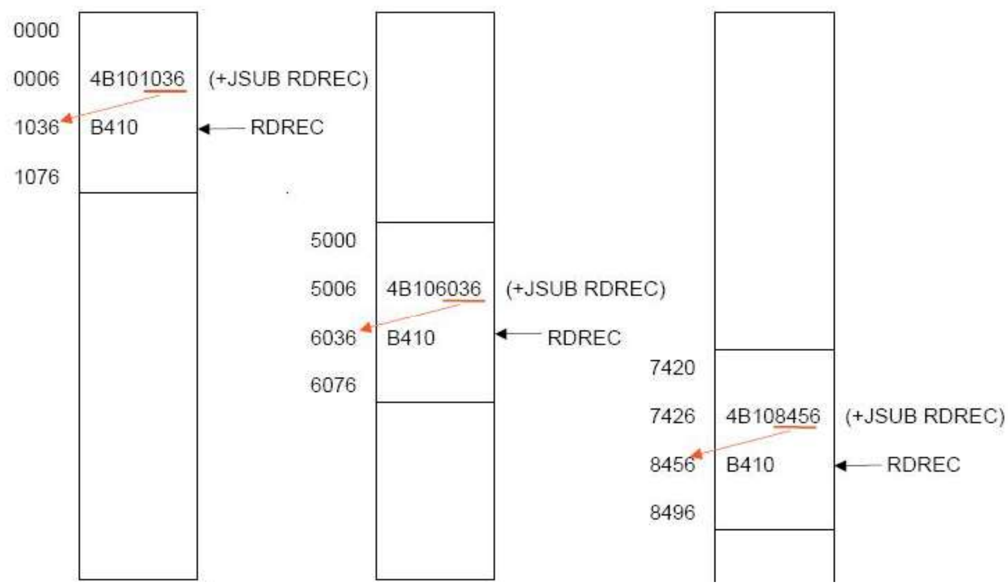
Absolute Program

In this the address is mentioned during assembling itself. This is called *Absolute Assembly.* Consider the instruction:

55	101B	LDA	THREE	00102D

This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000. Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000. Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.

Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification. An object program that has the information necessary to perform this kind of modification is called the relocatable program.



The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006. The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000. The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.

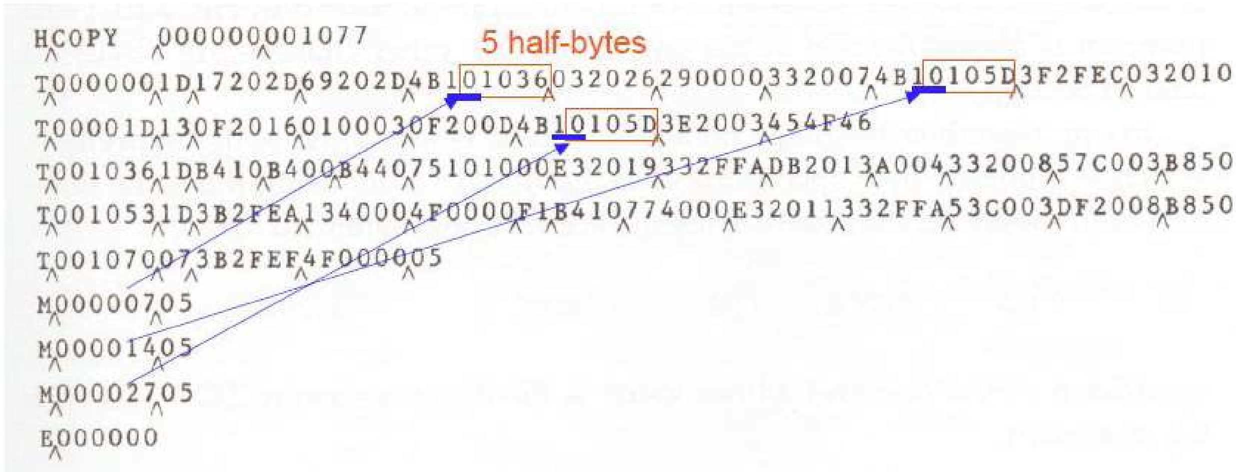
The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions. From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.

For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes (4 bits) The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.




```
H^COPY ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

In the above object code the red boxes indicate the addresses that need modifications. The object code lines at the end are the descriptions of the modification records for those instructions which need change if relocation occurs. M00000705 is the modification suggested for the statement at location 0007 and requires modification 5-half bytes. Similarly the remaining instructions indicate.

<u>Assembler Design</u> Session 4 & 5

3.2 Machine-Independent features:

These are the features which do not depend on the architecture of the machine. These are:

- Literals
- Expressions
- Program blocks
- Control sections

3.2.1 Literals:

A literal is defined with a prefix = followed by a specification of the literal value. Example:

```
45      001A   ENDFIL      LDA    =C'EOF'      032010
-
-
93                         LTORG
        002D   *                  =C'EOF'      454F46
```

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

215	1062	WLOOP	TD	=X’05’	E32011

It is important to understand the difference between a constant defined as a literal and a constant defined as an immediate operand. In case of literals the assembler generates the specified value as a constant at some other memory location In immediate mode the operand value is assembled as part of the instruction itself. Example

55	0020		LDA	#03	010003

All the literal operands used in a program are gathered together into one or more *literal pool*s. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

A literal table is created for the literals which are used in the program. The literal table contains the *literal name, operand value and length.* The literal table is usually created as a hash table on the literal name.

Implementation of Literals:

During Pass-1:

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG for literal definition. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITTAB

SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062

LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

3.2.2 Symbol-Defining Statements:

EQU Statement:

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this **EQU** (Equate). The general form of the statement is

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example

```
+LDT        #4096
```

This loads the register T with immediate value 4096, this does not clearly what exactly this value indicates. If a statement is included as:

```
MAXLEN      EQU         4096 and then
            +LDT        #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

Another common usage of EQU statement is for defining values for the general-purpose registers. The assembler can use the mnemonics for register usage like a-register A , X – index register and so on. But there are some instructions which requires numbers in place of names in the instructions. For example in the instruction RMO 0,1 instead of RMO A,X. The programmer can assign the numerical values to these registers using EQU directive.

```
A         EQU      0
X         EQU      1  and  so on
```

These statements will cause the symbols A, X, L… to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed. As another usage if in a machine that has many general purpose registers named as R1, R2,…, some may be used as base register, some may be used as accumulator. Their usage may change from one program to another. In this case we can define these requirement using EQU statements.

```
BASE      EQU      R1
INDEX     EQU      R2
COUNT     EQU      R3
```

One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined. For example, the following statement is not valid:

```
BETA      EQU      ALPHA
ALPHA     RESW     1
```

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

ORG Statement:

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin). Its general format is:

```
ORG       value
```

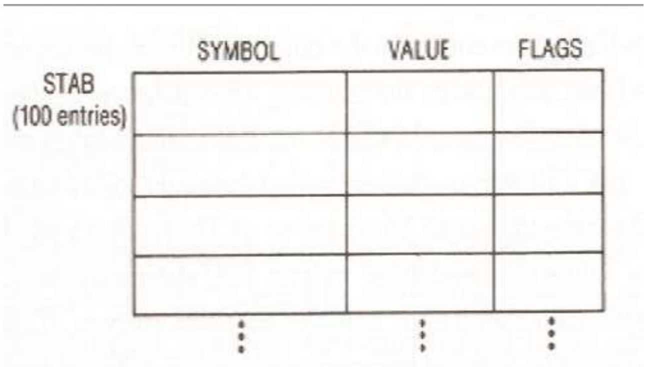
Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

SYMBOL 6 Bytes
VALUE 3 Bytes

FLAG 2 Bytes

The table looks like the one given below.



The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the ttable can be reserved by the statement:

```
        STAB        RESB        1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

```
        SYMBOL      EQU         STAB
        VALUE       EQU         STAB+6
        FLAGS       EQU         STAB+9
```

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
        LDA         VALUE, X
```

The same thing can also be done using ORG statement in the following way:

```
STAB        RESB        1100
            ORG         STAB
SYMBOL      RESB        6
VALUE       RESW        1
FLAG        RESB        2
            ORG         STAB+1100
```

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. For example for the sequence of statements below:

```
            ORG         ALPHA
BYTE1       RESB        1
BYTE2       RESB        1
BYTE3       RESB        1
            ORG
ALPHA       RESB        1
```

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

3.2.3 Expressions:

Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /. Division is usually defined to produce an integer result. Individual terms may be constants, user-defined symbols, or special terms. The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement

```
BUFFEND     EQU         *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either absolute expression or relative expressions depending on the type of value they produce.

Absolute Expressions: The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. Example:

```
MAXLEN      EQU         BUFEND-BUFFER
```

In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute immaterial o the relocation of the program. The expression can have only absolute terms. Example:

MAXLEN EQU 1000

Relative Expressions: All the relative terms except one can be paired as described in "absolute". The remaining unpaired relative term must have a positive sign. Example:

STAB EQU OPTAB + (BUFEND – BUFFER)

Handling the type of expressions: to find the type of expression, we must keep track the type of symbols used. This can be achieved by defining the type in the symbol table against each of the symbol as shown in the table below:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

3.2.4 Program Blocks:

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

Assembler Directive USE:

USE [blockname]

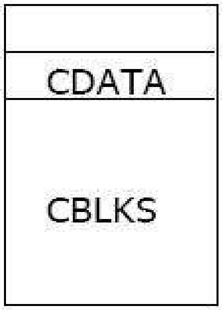
At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory



Example Code

(default) block   Block number

```
0000   0   COPY     START   0
0000   0   FIRST    STL     RETADR          172063
0003   0   CLOOP    JSUB    RDREC           4B2021
0006   0            LDA     LENGTH          032060
0009   0            COMP    #0              290000
000C   0            JEQ     ENDFIL          332006
000F   0            JSUB    WRREC           4B203B
0012   0            J       CLOOP           3F2FEE
0015   0   ENDFIL   LDA     =C'EOF'         032055
0018   0            STA     BUFFER          0F2056
001B   0            LDA     #3              010003
001E   0            STA     LENGTH          0F2048
0021   0            JSUB    WRREC           4B2029
0024   0            J       @RETADR         3E203F
0000   1            USE     CDATA
0000   1   RETADR   RESW    1
0003   1   LENGTH   RESW    1
0000   2            USE     CBLKS
0000   2   BUFFER   RESB    4096
1000   2   BUFEND   EQU     *
1000       MAXLEN   EQU     BUFEND-BUFFER
```

CDATA block

CBLKS block

```
                                            (default) block
0027   0   RDREC    USE
0027   0            CLEAR    X          B410
0029   0            CLEAR    A          B400
002B   0            CLEAR    S          B440
002D   0           +LDT      #MAXLEN    75101000
0031   0   RLOOP    TD       INPUT      E32038
0034   0            JEQ      RLOOP      332FFA
0037   0            RD       INPUT      DB2032
003A   0            COMPR    A,S        A004
003C   0            JEQ      EXIT       332008
003F   0            STCH     BUFFER,X   57A02F
0042   0            TIXR     T          B850
0044   0            JLT      RLOOP      3B2FEA
0047   0   EXIT     STX      LENGTH     13201F
004A   0            RSUB                4F0000
0006   1            USE      CDATA      CDATA block
0006   1   INPUT    BYTE     X'F1'      F1

                                            (default) block
004D   0            USE
004D   0   WRREC    CLEAR    X          B410
004F   0            LDT      LENGTH     772017
0052   0   WLOOP    TD       =X'05'     E3201B
0055   0            JEQ      WLOOP      332FFA
0058   0            LDCH     BUFFER,X   53A016
005B   0            WD       =X'05'     DF2012
005E   0            TIXR     T          B850
0060   0            JLT      WLOOP      3B2FEF
0063   0            RSUB                4F0000
0007   1            USE      CDATA      CDATA block
                    LTORG
0007   1   *        =C'EOF'             454F46
000A   1   *        =X'05'              05
                    END      FIRST
```

Arranging code into program blocks:

Pass 1

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

3.2.5 Control Sections:

A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references.*

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive

- assembler directive: **CSECT**

The syntax

secname CSECT

- separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

Implicitly defined as an external symbol

first control section

```
COPY      START    0                      COPY FILE FROM INPUT TO OUTPUT
          EXTDEF   BUFFER,BUFEND,LENGTH
          EXTREF   RDREC,WRREC
FIRST     STL      RETADR                 SAVE RETURN ADDRESS
CLOOP     +JSUB    RDREC                  READ INPUT RECORD
          LDA      LENGTH                 TEST FOR EOF (LENGTH=0)
          COMP    #0
          JEQ      ENDFIL                 EXIT IF EOF FOUND
          +JSUB    WRREC                  WRITE OUTPUT RECORD
          J        CLOOP                  LOOP
ENDFIL    LDA     =C'EOF'                 INSERT END OF FILE MARKER
          STA      BUFFER
          LDA     #3                      SET LENGTH = 3
          STA      LENGTH
          +JSUB    WRREC                  WRITE EOF
          J       @RETADR                 RETURN TO CALLER
RETADR    RESW     1
LENGTH    RESW     1                      LENGTH OF RECORD
          LTORG
BUFFER    RESB     4096                   4096-BYTE BUFFER AREA
BUFEND    EQU      *
MAXLEN    EQU      BUFFEND-BUFFER
```

Implicitly defined as an external symbol

second control section

```
RDREC     CSECT
.
.         SUBROUTINE TO READ RECORD INTO BUFFER
.
          EXTREF   BUFFER,LENGTH,BUFFEND
          CLEAR    X                      CLEAR LOOP COUNTER
          CLEAR    A                      CLEAR A TO ZERO
          CLEAR    S                      CLEAR S TO ZERO
          LDT      MAXLEN
RLOOP     TD       INPUT                  TEST INPUT DEVICE
          JEQ      RLOOP                  LOOP UNTIL READY
          RD       INPUT                  READ CHARACTER INTO REGISTER A
          COMPR    A,S                    TEST FOR END OF RECORD (X'00')
          JEQ      EXIT                   EXIT LOOP IF EOR
         +STCH     BUFFER,X               STORE CHARACTER IN BUFFER
          TIXR     T                      LOOP UNLESS MAX LENGTH HAS
          JLT      RLOOP                     BEEN REACHED
EXIT     +STX      LENGTH                 SAVE RECORD LENGTH
          RSUB                            RETURN TO CALLER
INPUT     BYTE     X'F1'                  CODE FOR INPUT DEVICE
MAXLEN    WORD     BUFFEND-BUFFER
```

Implicitly defined as an external symbol

third control section

```
WRREC       CSECT
.
.           SUBROUTINE TO WRITE RECORD FROM BUFFER
.
             EXTREF     LENGTH,BUFFER
             CLEAR      X                           CLEAR LOOP COUNTER
            +LDT        LENGTH
WLOOP        TD        =X'05'                       TEST OUTPUT DEVICE
             JEQ        WLOOP                       LOOP UNTIL READY
            +LDCH       BUFFER,X                    GET CHARACTER FROM BUFFER
             WD        =X'05'                       WRITE CHARACTER
             TIXR       T                           LOOP UNTIL ALL CHARACTERS HAVE
             JLT        WLOOP                           BEEN WRITTEN
             RSUB                                   RETURN TO CALLER
             END        FIRST
```

Handling External Reference

Case 1

15 0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference.
  - The assembler has no idea where RDREC is
  - inserts an address of zero
  - can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

Case 2

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler inserts a value of zero
- passes information to the loader
- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

Case 3

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

107 1000 MAXLEN EQU BUFEND-BUFFER

Object Code for the example program:

0000	COPY	START	0		
		EXTDEF	BUFFER,BUFFEND,LENGTH		
		EXTREF	RDREC,WRREC		
0000	FIRST	STL	RETADR	172027	
0003	CLOOP	+JSUB	RDREC	4B100000	Case 1
0007		LDA	LENGTH	032023	
000A		COMP	#0	290000	
000D		JEQ	ENDFIL	332007	
0010		+JSUB	WRREC	4B100000	
0014		J	CLOOP	3F2FEC	
0017	ENDFIL	LDA	=C'EOF'	032016	
001A		STA	BUFFER	0F2016	
001D		LDA	#3	010003	
0020		STA	LENGTH	0F200A	
0023		+JSUB	WRREC	4B100000	
0027		J	@RETADR	3E2000	
002A	RETADR	RESW	1		
002D	LENGTH	RESW	1		
		LTORG			
0030	*	=C'EOF'		454F46	
0033	BUFFER	RESB	4096		
1033	BUFEND	EQU	*		
1000	MAXLEN	EQU	BUFEND-BUFFER		

0000	RDREC	CSECT			
	.	SUBROUTINE TO READ RECORD INTO BUFFER			
	.				
		EXTREF	BUFFER,LENGTH,BUFEND		
0000		CLEAR	X	B410	
0002		CLEAR	A	B400	
0004		CLEAR	S	B440	
0006		LDT	MAXLEN	77201F	
0009	RLOOP	TD	INPUT	E3201B	
000C		JEQ	RLOOP	332FFA	
000F		RD	INPUT	DB2015	
0012		COMPR	A,S	A004	
0014		JEQ	EXIT	332009	
0017		+STCH	BUFFER,X	$\boxed{57900000}$	
001B		TIXR	T	B850	
001D		JLT	RLOOP	3B2FE9	
0020	EXIT	+STX	LENGTH	$\boxed{13100000}$	
0024		RSUB		4F0000	
0027	INPUT	BYTE	X'F1'	F1	
0028	MAXLEN	WORD	BUFFEND-BUFFER	$\boxed{000000}$	Case 2

```
0000    WRREC   CSECT

        .       SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
                 EXTREF    LENGTH,BUFFER
0000             CLEAR     X                          B410
0002            +LDT       LENGTH                     77100000
0006    WLOOP    TD       =X'05'                      E32012
0009             JEQ       WLOOP                      332FFA
000C            +LDCH      BUFFER,X                   53900000
0010             WD       =X'05'                      DF2008
0013             TIXR      T                          B850
0015             JLT       WLOOP                      3B2FEE
0018             RSUB                                 4F0000
                 END       FIRST
001B    *       =X'05'                                05
```

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

Define record (EXTDEF)

- Col. 1 D
- Col. 2-7 Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

Refer record (EXTREF)

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

Modification record

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified (hexadecimal)
- Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
- Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.

A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification my be defined either in this control section or in another section.

The object program is shown below. There is a separate object program for each of the control sections. In the *Define Record* and *refer record* the symbols named in EXTDEF and EXTREF are included.

In the case of *Define,* the record also indicates the relative address of each external symbol within the control section.

For EXTREF symbols, no address information is available. These symbols are simply named in the *Refer record.*

COPY

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000
```

RDREC

```
H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER     BUFEND - BUFFER
E
```

WRREC

```
H^WRREC ^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

Handling Expressions in Multiple Control Sections:

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).

When it comes in a program having multiple control sections then we have an extended restriction that:

- Both terms in each pair of an expression must be within the same control section
  - If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located.
    - **Legal:** BUFEND-BUFFER (both are in the same control section)

  - If the terms are located in different control sections, their difference has a value that is unpredictable.
    - **Illegal:** RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.

- **How to enforce this restriction**
  - When an expression involves external references, the assembler cannot determine whether or not the expression is legal.
  - The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
  - The loader checks the expression for errors and finishes the evaluation.

3.5 ASSEMBLER DESIGN

Here we are discussing

- The structure and logic of one-pass assembler. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program.
- Notion of a multi-pass assembler, an extension of two-pass assembler that allows an assembler to handle forward references during symbol definition.

3.5.1 One-Pass Assembler

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

- One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
- The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
  - The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110		.			

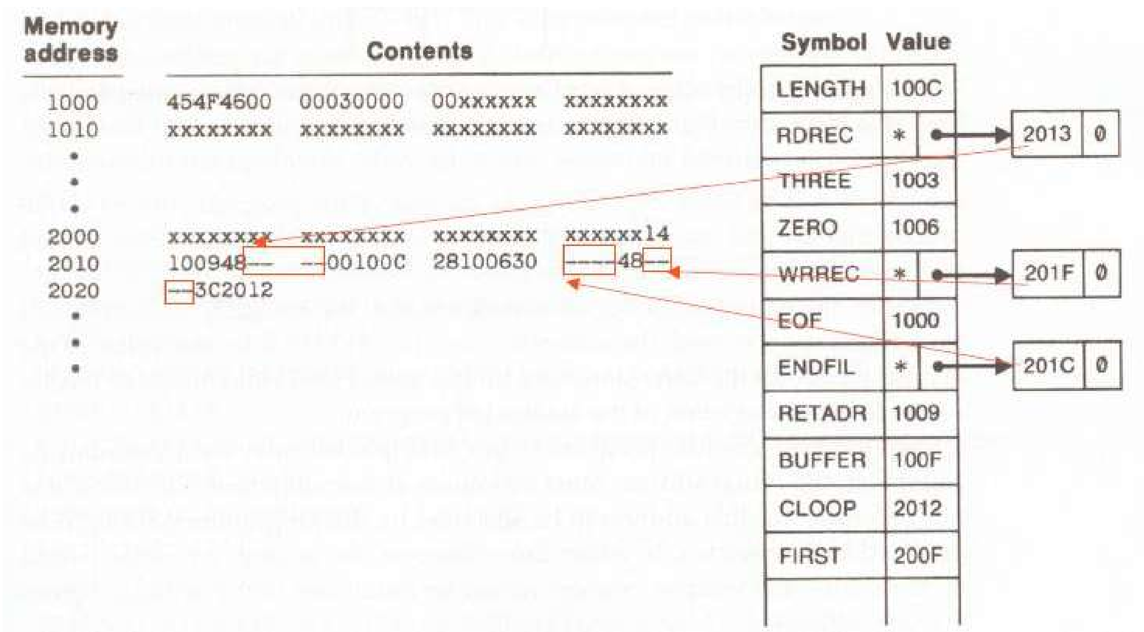
Forward Reference in One-Pass Assemblers: In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler
  - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

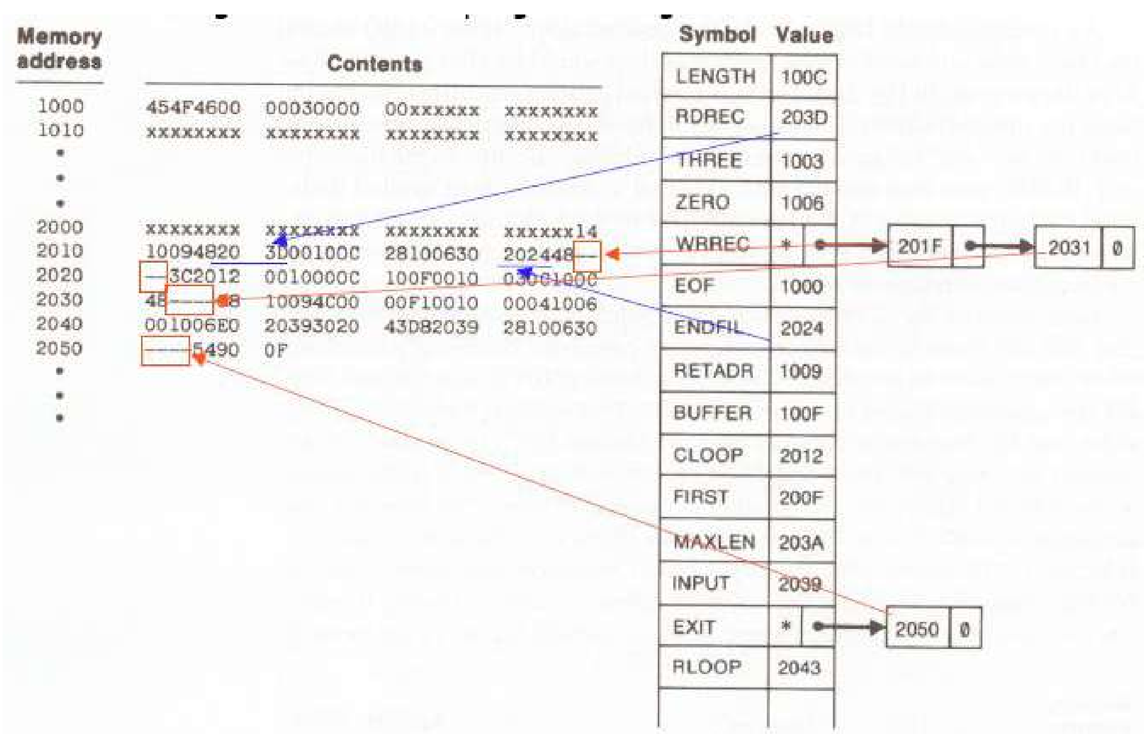
After Scanning line 40 of the program:

40 2021 J` CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:



If One-Pass needs to generate object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

Object Code Generated by One-Pass Assembler:

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

Multi_Pass Assembler:

- For a two pass assembler, forward references in symbol definition are not allowed:

```
ALPHA     EQU   BETA
BETA      EQU   DELTA
DELTA     RESW  1
```

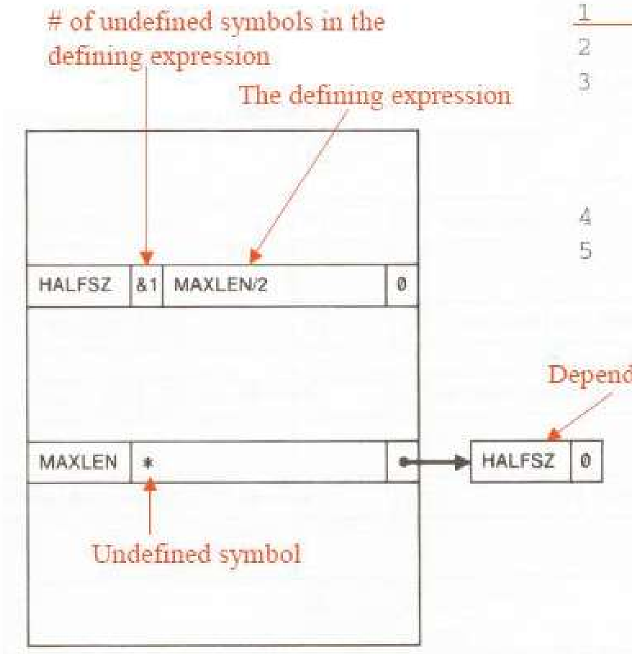
  - Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

Implementation Issues for Modified Two-Pass Assembler:

Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

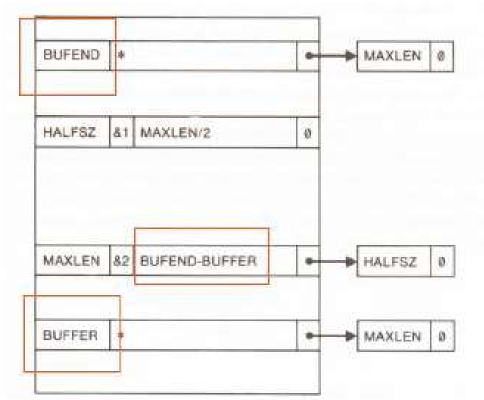
- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

Multi-Pass Assembler Example Program

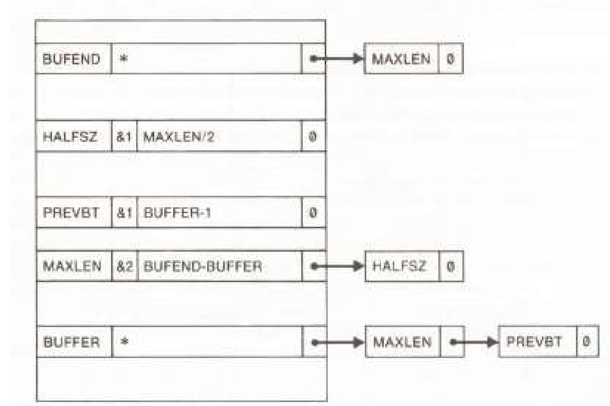



```
1    HALFSZ    EQU     MAXLEN/2
2    MAXLEN    EQU     BUFEND-BUFFER
3    PREVBT    EQU     BUFFER-1
                       .
                       .
                       .
4    BUFFER    RESB    4096
5    BUFEND    EQU     *
```

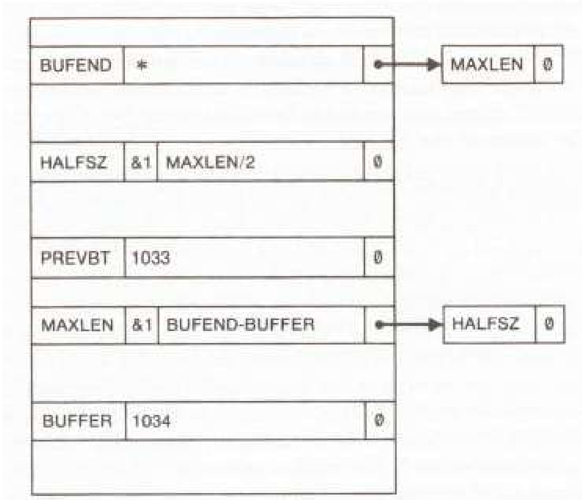
Multi-Pass Assembler (Figure 2.21 of Beck): Example for forward reference in Symbol Defining Statements:



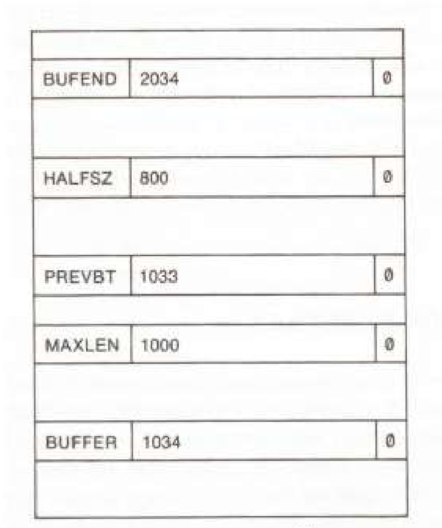
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

Chapter 3

Loaders and Linkers

This Chapter gives you…

- Basic Loader Functions
- Machine-Dependent Loader Features
- Machine-Independent Loader Features
- Loader Design Options
- Implementation Examples

3.0 Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

3.1 Basic Loader Functions

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 3.1. In figure 3.1 translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 3.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure 3.3 shows the role of both loader and linker.

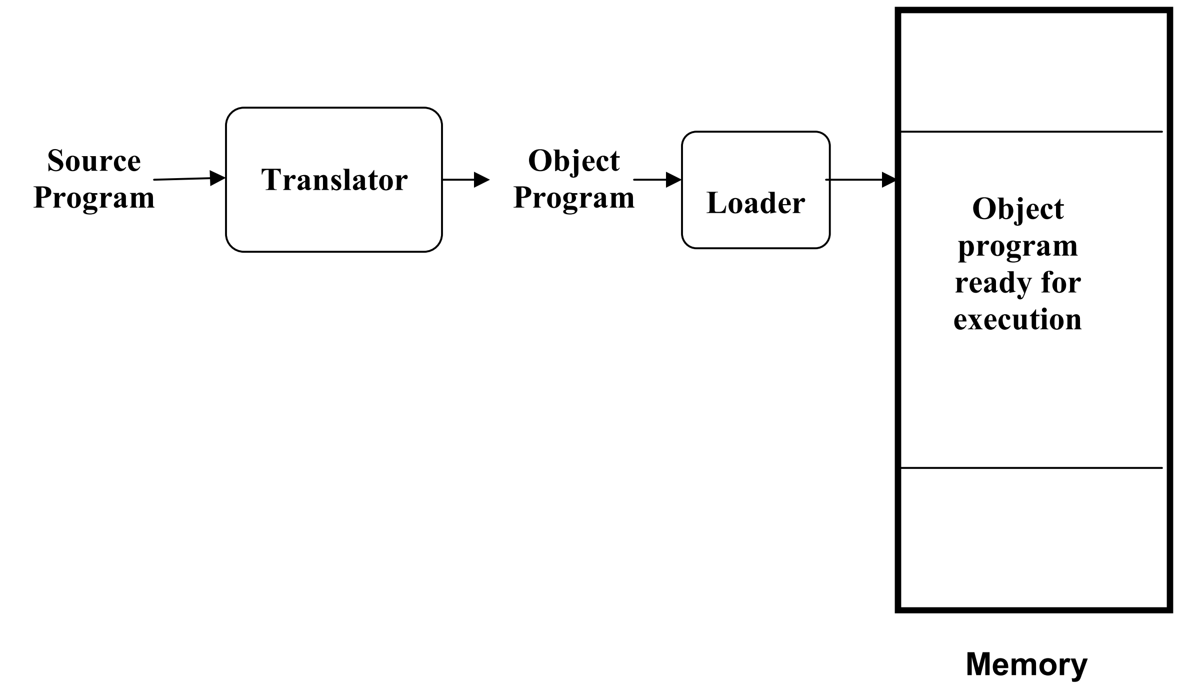


Figure 3.1 : The Role of Loader

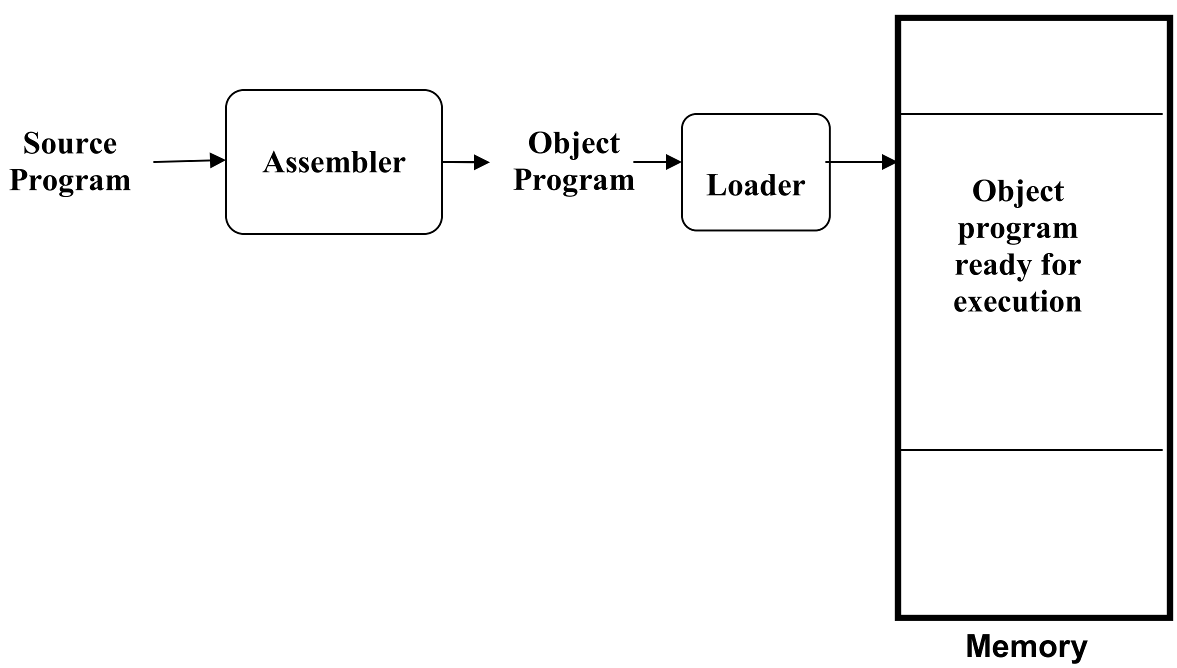


Figure 3.2: The Role of Loader with Assembler

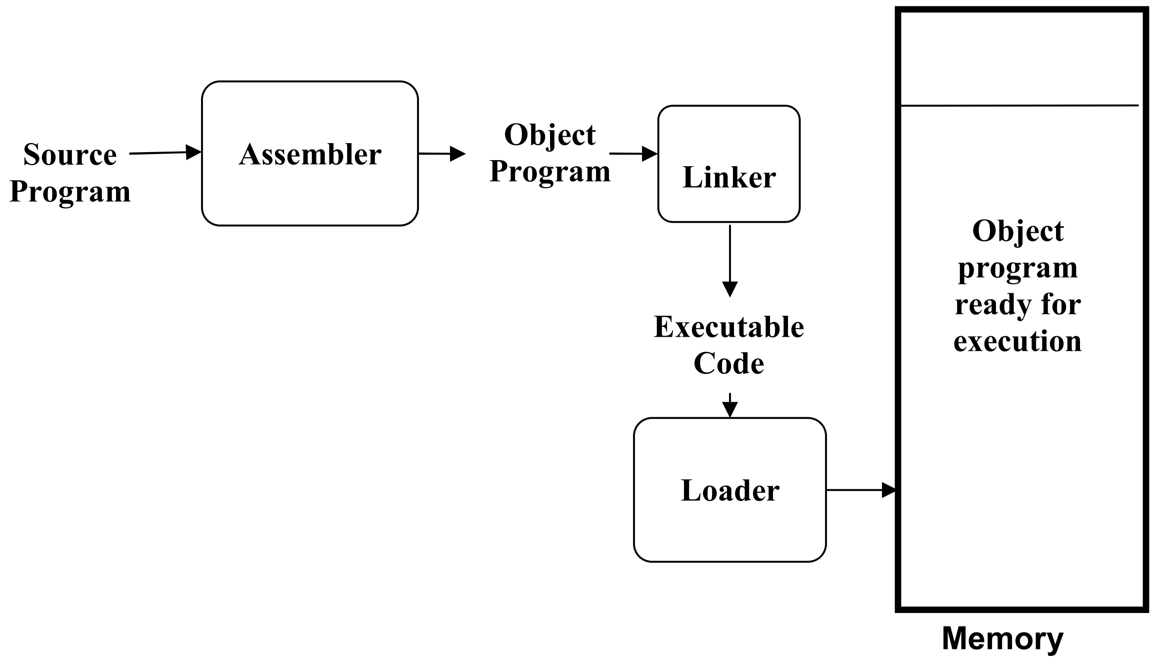


Figure 3.3 : The Role of both Loader and Linker

3.3 Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

3.3.1 Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure 3.3.1. The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

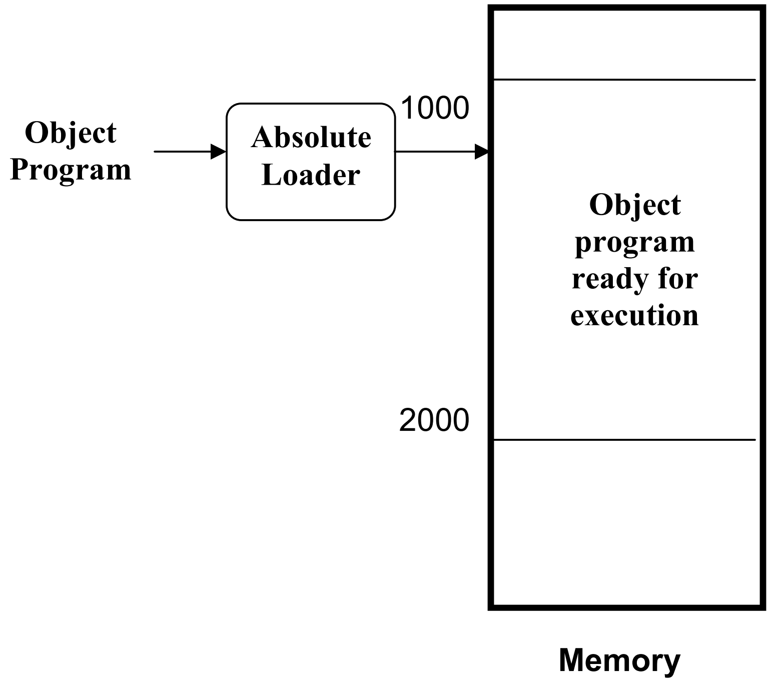


Figure 3.3.1: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

Begin
read Header record
verify program name and length
read first Text record
while record type is <> 'E' **do**
    begin
    {if object code is in character form, convert into internal representation}
    move object code to specified location in memory
    read next object program record
    end
jump to address specified in End record
end

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

Memory address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

← COPY

(b) Program loaded in memory

3.3.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin
X=0x80 (the address of the next memory location to be loaded
Loop
    A←GETC (and convert it from the ASCII character
    code to the value of the hexadecimal digit)
    save the value in the high-order 4 bits of S
    A←GETC
    combine the value to form one byte A← (A+S)
    store the value (in A) to the address in register X
    X←X+1
End

It uses a subroutine GETC, which is

```
GETC    A←read one character
        if A=0x04 then jump to 0x80
        if A<48 then GETC
        A ← A-48 (0x30)
        if A<10 then return
        A ← A-7
        return
```

3.4 Machine-Dependent Loader Features

Absolute loader is simple and efficient, but the scheme has potential disadvantages One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.

This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions.