MACHINE ARCHITECTURE

The Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer.

Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

Difference between System Software and Application Software

System Software	Application Software
System Software intended to support the operation and use of computer	Application Software is primarily concerned with the solution of some problem using computer as a tool
Related to Machine Architecture	Not related to machine architecture
Machine Dependent Example: Compilers, Assemblers, OS etc	Machine Independent Example: Payroll System, Games etc

The Simplified Instructional Computer (SIC):

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. There are two versions of SIC, they are, standard model (SIC), and, extension version (SIC/XE) (extra equipment or extra expensive).

SIC Machine Architecture:

We discuss here the SIC machine architecture with respect to its Memory and Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output

- **Memory:**

There are a total of 32,768(2^{15})bytes in the computer memory. It uses Little Endian format to store the numbers, 3 consecutive bytes form a word, and each location in memory contains 8-bit bytes.

- **Registers:**

There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

- **Data Formats:**

Integers are stored as 24-bit binary numbers. 2's complement representation is used for negative values; characters are stored using their 8-bit ASCII codes. No floating-point hardware on the standard version of SIC.

- **Instruction Formats:**

All machine instructions on the standard version of SIC have the 24-bit format as shown above

8	1	15
Opcode	x	Address

- **Addressing Modes:**

Mode	Indication	Target address calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x)

There are two addressing modes available, which are as shown in the above table. Parentheses are used to indicate the contents of a register or a memory location.

- **Instruction Set :**

1. SIC provides, load and store instructions (LDA, LDX, STA, STX, etc.). Integer arithmetic operations: (ADD, SUB, MUL, DIV, etc.).
2. All arithmetic operations involve register A and a word in memory, with the result being left in the register. Two instructions are provided for subroutine linkage.
3. COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to indicate the result. There are conditional jump instructions: (JLT, JEQ, JGT), these instructions test the setting of CC and jump accordingly.
4. JSUB jumps to the subroutine placing the return address in register L, RSUB returns by jumping to the address contained in register L.

- **Input and Output:**

Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

Data movement and Storage Definition

LDA, STA, LDL, STL, LDX, STX (A- Accumulator, L – Linkage Register, X – Index Register), all uses3-byte word. LDCH, STCH associated with characters uses 1-byte. There are no memory-memory move instructions.

Storage definitions are

- WORD - ONE-WORD CONSTANT
- RESW - ONE-WORD VARIABLE
- BYTE - ONE-BYTE CONSTANT
- RESB - ONE-BYTE VARIABLE

Example Programs (SIC):

Example 1: Simple data and character movement operation

To store the value 5 in a variable ALPHA and character Z in a variable C1

```
          LDA FIVE
          STA ALPHA
          LDCH    CHARZ
          STCH    C1
ALPHA     RESW    1
FIVE      WORD    5
CHARZ     BYTE    C'Z'
C1        RESB    1
```

Example 2: Arithmetic operations

BETA=ALPHA+INCR+1

```
        LDA ALPHA

        ADD INCR

        SUB ONE

        STA  BETA

        ........

        ........

        ........

ONE     WORD   1

ALPHA   RESW   1

BEETA   RESW   1

INCR    RESW   1
```

Example 3: Looping and Indexing operation

To perform STR2=STR1 where STR1 is a string of 11 characters.

```
         LDX    ZERO         ;  X = 0

MOVECH   LDCH   STR1, X      ;  LOAD A FROM STR1

         STCH   STR2, X      ;  STORE A TO STR2

         TIX    ELEVEN       ;  ADD 1 TO X, TEST

         JLT    MOVECH
```

```
STR1      BYTE    C 'HELLO WORLD'
STR2      RESB    11
ZERO      WORD    0
ELEVEN    WORD    11
```

Example 4: **Input and Output operation**

To read a character from the input device and to write a character to the output device.

```
INLOOP    TD      INDEV         : TEST INPUT DEVICE
          JEQ     INLOOP        : LOOP UNTIL DEVICE IS READY
          RD      INDEV         : READ ONE BYTE INTO A
          STCH    DATA          : STORE A TO DATA



OUTLOOP   TD      OUTDEV        : TEST OUTPUT DEVICE
          JEQ     OUTLP         : LOOP UNTIL DEVICE IS READY
          LDCH    DATA          : LOAD DATA INTO A
          WD      OUTDEV        : WRITE A TO OUTPUT DEVICE



INDEV     BYTE    X 'F5'        : INPUT DEVICE NUMBER
OUTDEV    BYTE    X '08'        : OUTPUT DEVICE NUMBER
```

SIC/XE Machine Architecture:

- **Memory**

  Maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} bytes).

- **Registers**

  Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC.

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

- **Data Formats**

  There is a 48-bit floating-point data type, $F*2^{(e-1024)}$

1	11	36
s	exponent	fraction

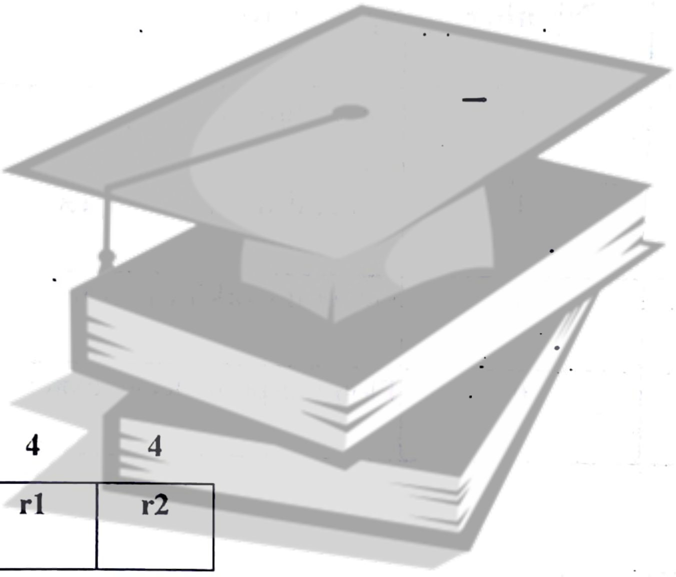
- **Instruction Formats:**

The new set of instruction formats fro SIC/XE machine architecture are as follows.

- Format 1 (1 byte): contains only operation code (straight from table).
- Format 2 (2 bytes): first eight bits for operation code, next four for register 1 and

following four for register 2. The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).

- Format 3 (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).
- Format 4 (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.



Format 1 (1 byte)

8
op

Format 2 (2 bytes)

8	4	4
op	r1	r2

Formats 1 and 2 are instructions do not reference memory at all

Format 3 (3 bytes)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

Format 4 (4 bytes)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

- Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

1. **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.

2. **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)

3. **Immediate**(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)

4. **Indirect**(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

5. **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings e -> e = 0 means format 3, e = 1 means format 4.

- **Instruction Set:**

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating- point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction: RMO Register-to-register arithmetic operations, ADDR, SUBR, MULR, DIVR and, Supervisor call instruction : SVC.

- **Input and Output:**

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

Example Programs (SIC/XE)

Example 1: Simple data and character movement operation

To store the value 5 in a variable ALPHA and character Z in a variable C1

```
          LDA     #5
          STA     ALPHA

          LDA     #90
          STCH    C1
          .
          .
ALPHA     RESW    1
C1        RESB    1
```

Example 2: Arithmetic operations

BETA=ALPHA+INCR+1

```
LDS     INCR

LDA     ALPHA

ADDR S,A

SUB     1

STA     BETA
```

```
ALPHA       RESW        1
INCR        RESW        1
BETA        RESW        1
```

Example 3: Looping and Indexing operation

To perform STR2=STR1 where STR1 is a string of 11 characters.

```
            LDT     #11

            LDX     #0

MOVECH      LDCH    STR1, X     : LOAD A FROM STR1

            STCH    STR2, X     : STORE A TO STR2

            TIXR    T           :   ADD 1 TO X, TEST (T)

            JLT     MOVECH

            ..........

STR1        BYTE    C 'HELLO WORLD'

STR2        RESB    11
```

Difference between SIC and SIC/XE

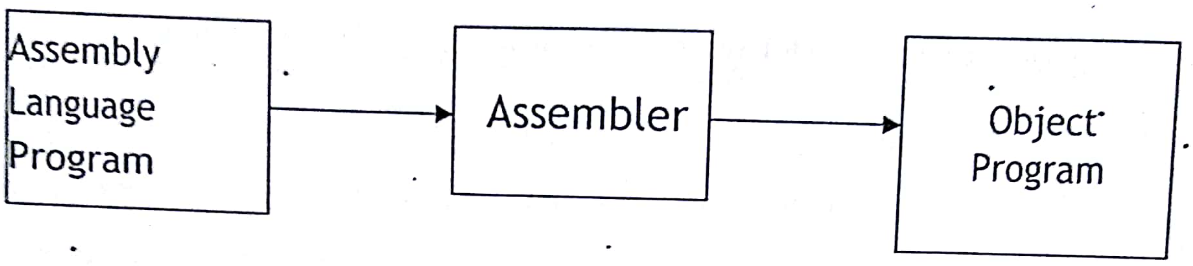
	SIC	SIC/XE
Memory	2^{15} bytes	2^{20} bytes
Registers	5 (A,X,L,PC & SW)	9(A,X,L,B,S,T,F,PC & SW)
Data Formats	No Floating Point Hardware	Supports Floating Point Hardware
Instruction Format	One	Four
Addressing Mode	Two	Five and its combination

***

ASSEMBLERS

The basic assembler functions are:

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.



SIC Assembler Directive:

START: Specify name and starting address for the program

END: Indicate End of the program and (optionally) specify the first execution instruction in the program.

BYTE: Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

WORD: Generate one-word integer constant.

RESB: Reserve the indicated number of bytes for a data area.

RESW: Reserve the indicated number of words for a data area.

A simple SIC Assembler

The design of assembler in other words (functions):

1. Convert mnemonic operation codes to their machine language equivalents.
   - Example: Translate LDA to 00.
2. Convert symbolic operands to their equivalent machine addresses.
   Example: Translate GAMMA to 400F
3. Build the machine instructions in the proper format.
4. Convert the data constants to internal machine representations.
   Example: ONE WORD 1 to 000001
5. Write the object program and the assembly listing

Two Pass Assembler

Pass-1

- Assign addresses to all the statements in the program
- 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.

DELTA=GAMMA + INCR - 1

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	ARTH	START	4000	
4000		LDA	GAMMA	00400F
4003		ADD	INCR	184012
4006		SUB	ONE	1C4015
4009		STA	DELTA	0C400C
400C	DELTA	RESW	1	
400F	GAMMA	RESW	1	
4012	INCR	RESW	1	
4015	ONE	WORD	1	000001
4018		END		

OPTAB	
MNEMONIC	OPCODE
LDA	00
ADD	18
SUB	1C
STA	0C

SYMTAB	
LABEL	ADDRESS
DELTA	400C
GAMMA	400F
INCR	4012
ONE	4015

Figure 2.1: Assembly Language Program with object code

Object Code for Instruction

LDA GAMMA

Opcode	X	Address
0000 0000	0	100 0000 0000 1111
0 0		4 0 0 F

OBJECT PROGRAM

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.

Syntax

- 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)

```
H^ARTH ^004000^000018

T^004000^0C^00400F^184012^1C4015^0C400C

T^004015^03^000001

E^004000
```

Fig 2.2 Object program corresponding to Fig 2.1

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, effi~~ciency 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:

LOCCTR is an 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 address
    initialize LOCCTR to starting address
    write line to intermediate file
    read next input 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

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.

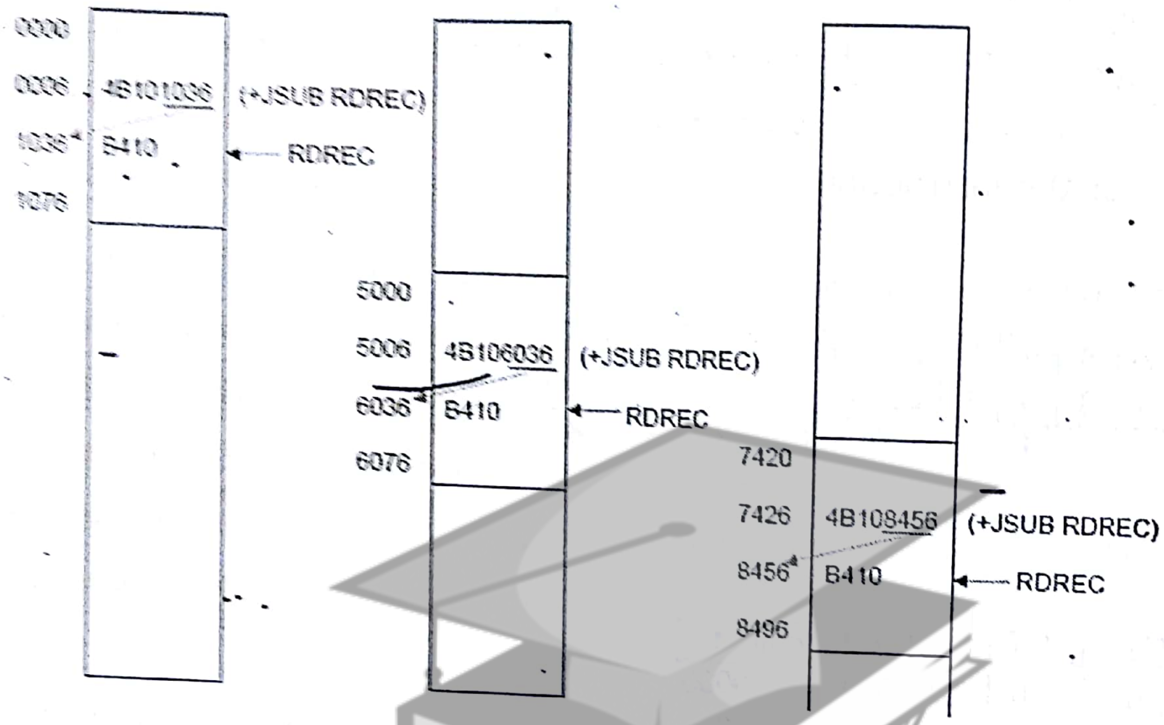


Fig: Examples of Program Relocation

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.

The Modification Record for

```
    +JSUB      RDREC
```

instruction is

```
    M00000705
```

000007 is the starting location of the address field to be modified by the loader for proper execution of the program.

05 is the length of the address field to be modified , in half bytes.

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.

ASSEMBLERS

Machine-Independent features:

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

- Literals
- Symbol-Defining Statements
- Expressions
- Program blocks
- Control sections

Literals:

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

Example:

```
001A        ENDFIL      LDA   =C'EOF'      032010
                        ...
                        ...
                         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'.

```
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

```
0020                    LDA  #03        010003
```

All the literal operands used in a program are gathered together into one or more *literal pools*. 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.

	START	0
INLOOP	TD	=X'F1'
	JEQ	INLOOP
	RD	=X'F1'
	STCH	DATA
	LTORG	
OUTLP	TD	=X'05'
	JEQ	OUTLP
	LDCH	DATA
	WD	=X'05'
DATA	RESB	1
	END	

LOCCTR				OBJECT CODE
		START	0	
0000	INLOOP	TD	=X'F1'	
0003		JEQ	INLOOP	
0006		RD	=X'F1'	
0009		STCH	DATA	
000C		=X'F1'		F1
000D	OUTLP	TD	=X'05'	
0010		JEQ	OUTLP	
0013		LDCH	DATA	
0016		WD	=X'05'	
0019	DATA	RESB	1	
		END		
001A		=X'05		05
001B				

LITTAB

Literal Name	Value	Length	Address
X'F1'	F1	1	000C
X'05'	05	1	001A

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).

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:

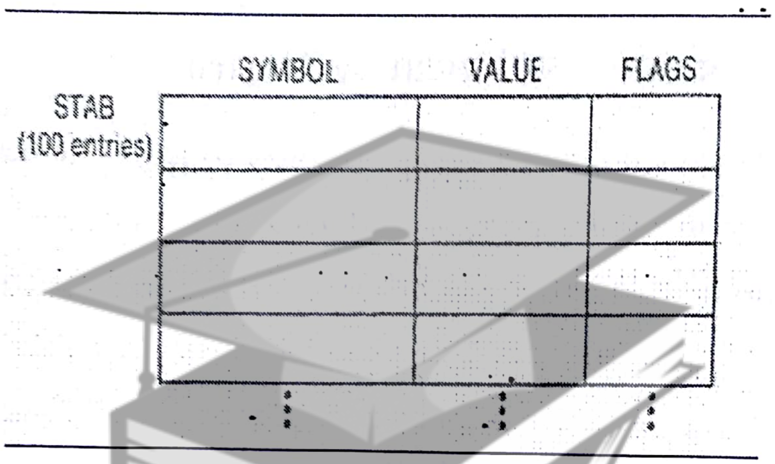
```
ORG         value
```

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).

Restrictions-EQU

In the case of EQU all the symbols used on the right hand side of the statement must have been defined previously in the program.

ALPHA RESW 1 BETA EQU ALPHA	✓	BETA EQU ALPHA ALPHA RESW 1	✗

Restriction –ORG

All symbols used to specify the new LOCCTR value must have been previously defined.

ALPHA RESB 1 ORG ALPHA BYTE1 RESB 1 BYTE2 RESB 1 BYTE3 RESB 1 ORG	✓	ORG ALPHA BYTE1 RESB 1 BYTE2 RESB 1 BYTE3 RESB 1 ORG ALPHA RESB 1	✗

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 of 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)
```

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

LOCCTR				OBJECT CODE
	READ	START	0	
0000		LDX	#0	
0003		LDT	#11	
0006	MOVECH	JSUB	RDDATA	4B200B
0009		LDCH	DATA	532019
000C		STCH	STR,X	
000F		TIXR	T	
0011		JLT	MOVECH	
		USE	CDATA	
0000	DATA	RESB	1	
0001	STR	RESB	11	
		USE	CBLKS	
0000	BUFFER	RESB	4096	
1000	BUFEND	EQU	*	
1000	MAXLEN	EQU	BUFEND-BUFFER	
		USE		
0014	RDDATA	CLEAR	A	
0016	INLOOP	TD	INDEV	E32018
0019		JEQ	INLOOP	
001C		RD	INDEV	
001F		STCH	DATA	
0022		RSUB		
		USE	CDATA	
000C	INDEV	BYTE	X'F1'	
		END		

BLOCK TABLE

BLOCK NAME	BLOCK NUMBER	ADDRESS	LENGTH
DEFAULT	0	0000	00025
CDATA	1	0025 (0000+0025)	000D
CBLKS	2	0032 (0025+000D)	1000

Program Length = 1032 (0032+1000)

JSUB RDDATA

Opcode	N	I	X	B	P	E	DISPLACEMENT
0100 10	1	1	0	0	1	0	0000 0000 1011

4 B 2 0 0 B

TA of RDDATA = (0000+0014) = 0014 Disp=TA – (PC) = 0014 – 0009 = 00B

LDCH DATA

Opcode	N	I	X	B	P	E	DISPLACEMENT
0101 00	1	1	0	0	1	0	0000 0001 1001

5 3 2 0 1 9

TA of DATA=(0025+0) = 025

Disp = TA – (PC)

= 025 – 00C = 019

TD INDEV

Opcode	N	I	X	B	P	E	DISPLACEMENT
1110 00	1	1	0	0	1	0	0000 0001 1000

E 3 2 0 1 8

TA of INDEV = 000C + 0025 = 031

Disp= TA –(PC) = 031 – 019 = 018

Advantages of Program Blocks

1. We can avoid using Format 4
2. Base register is no longer necessary.
3. The problem of placement of literals is easily solved.
4. Program readability is improved.

CONTROL SECTIONS and PROGRAM LINKING

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 of a program.

The syntax

secname CSECT

– separate location counter for each control section

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	READ	START	0	
		EXTDEF	DATA	
		EXTREF	RDDATA	
0000		LDX	#0	
0003		LDT	#11	
0006	MOVECH	+JSUB	RDDATA	4B100000
000A		LDCH	DATA	
000D		STCH	STR,X	
000F		TIXR	T	
0011		JLT	MOVECH	
0014	DATA	RESB	1	
0015	STR	RESB	11	
0020				
	~~RDDA~~TA	CSECT		
		EXTREF	DATA	
0000		CLEAR	A	B400
0002	INLOOP	TD	INDEV	E3200D
0005		JEQ	INLOOP	332FFA
0008		RD	INDEV	DB2007
000B		+STCH	DATA	57100000
000F		RSUB		4F0000
0012	INDEV	BYTE	X'F1'	F1
0013		END		

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 names symbols that are defined in this section but may be used by other control sections.

EXTREF (external Reference):

It names symbols that are used in this CONTROL section and are defined elsewhere.

For Program Linking we require Define, Refer and Modification Record.

1. Define Record: Lists symbols that are defined in this control section.

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

2. Refer Record

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

3. Modification Record

Col. 1	M
Col. 2-7	Starting address of the field to be modified (hexadecimal), relative to the beginning of control section.
Col. 8-9	Length of the field to be modified, in half-bytes (hexadecimal)
Col. 10	Modification flag(+ or -)
Col.11-16	External symbol whose value is to be added to or subtracted from the indicated field

Handling External Reference

```
MOVECH      +JSUB      RDDATA              4B100000
```

The operand RDDATA is an external reference.

- The assembler has no idea where RDDATA is
- inserts an address of zero
- can only use extended formatto 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.

Similarly for the instruction **+STCH DATA** the object code is 5 7100000

```
HRDDATA000000000013
RDATA
T00000013B400E3200D332FFADB2007571000004F0000F1
M00000C05+DATA
E
```

Figure: Object Program for control section-**RDDATA**

ASSEMBLER DESIGN OPTIONS

ONE PASS ASSEMBLERS

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)

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

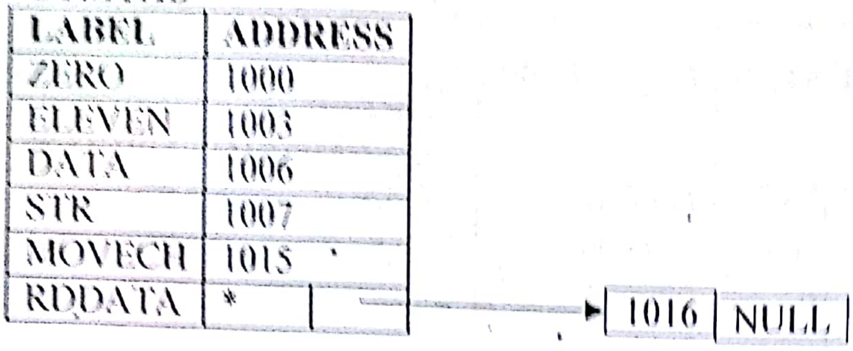
- o The efficiency of the assembly process is an important consideration.

Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

LOCCTR				OBJECT CODE
	READ	START	1000	
1000	ZERO	WORD	0	000000
1003	ELEVEN	WORD	11	00000B
1006	DATA	RESB	1	
1007	STR	RESB	11	
1012		LDX	ZERO	041000
1015	MOVECH	JSUB	RDDATA	**480000**
1018		LDCH	DATA	501006
101B		STCH	STR,X	549007
101E		TIX	ELEVEN	2C100C
1021		JLT	MOVECH	381015
1024	INDEV	BYTE	X'F1'	F1
1025	RDDATA	LDA	ZERO	001000
1028	INLOOP	TD	INDEV	
102B		JEQ	INLOOP	
102E		RD	INDEV	
1031		STCH	DATA	
1034		RSUB		
1037		END		

OPTAB	
MNEMONIC	**OPCODE**
LDX	04
LDCH	50
STCH	54
TIX	2C
JLT	38
LDA	00
JSUB	48
TD	E0
JEQ	30
RD	D8
RSUB	4C

SYMTAB



LABEL	ADDRESS
ZERO	1000
ELEVEN	1003
DATA	1006
STR	1007
MOVECH	1015
RDDATA	*

* indicate undefined symbol

Figure: The status after scanning the input statement **MOVECH JSUB RDDATA**

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.

```
HREAD 001000

T001000060000000000B

T00101216041000480000050100654900 72C100C381015F1001000

T001016021025
```

Figure: Object Program after scanning line **RDDATA LDA ZERO.**

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:

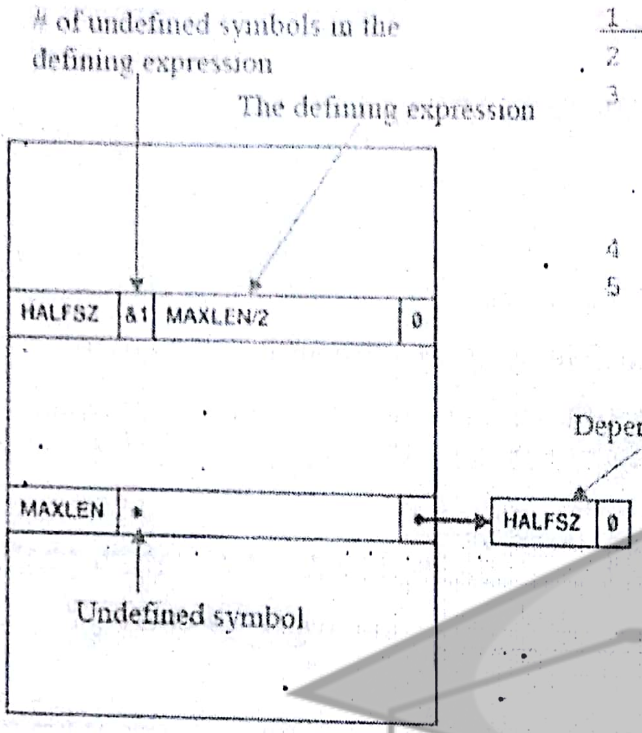
- The symbol name
- The defining expression
- The number of undefined symbols in the defining expression

The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.

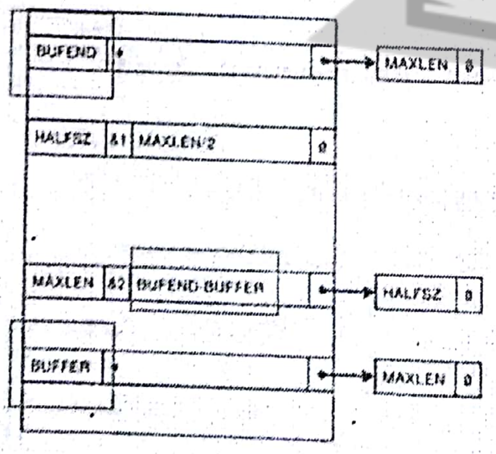
When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

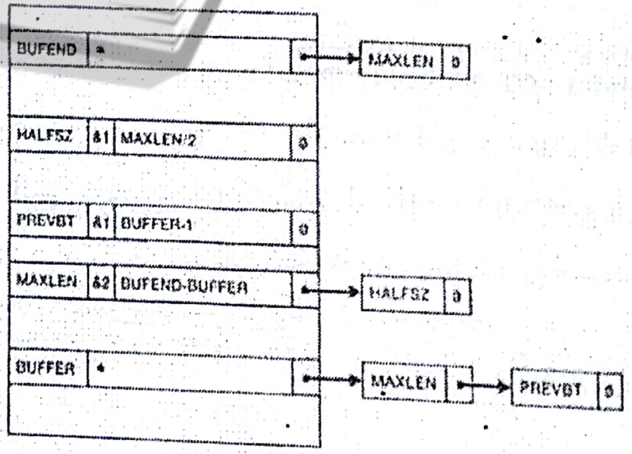
Multi-Pass Assembler Example Program

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



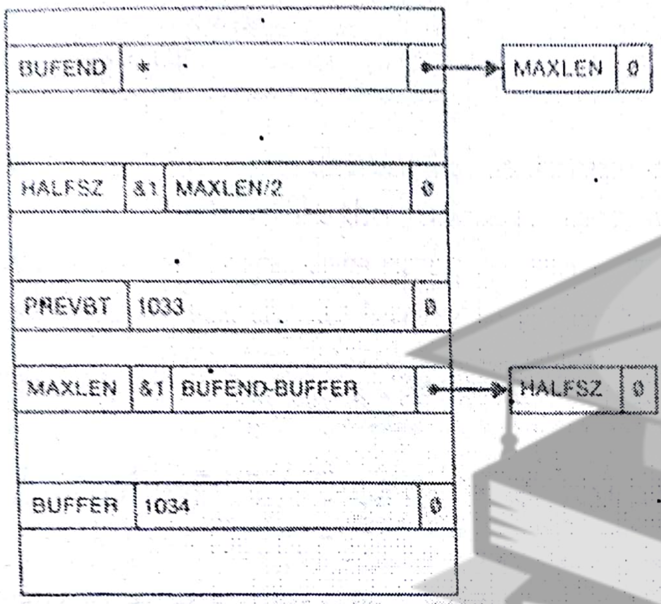
Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:



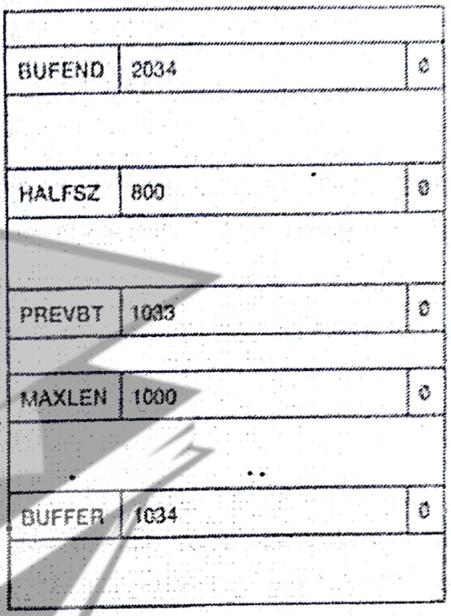


2 MAXLEN EQU BUFEND-BUFFER   3 PREVBT EQU BUFFER-1

Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034.



4 BUFFER RESB 4096



5 BUFEND EQU *

In Multi-Pass Assembler the portion of the program that involve forward references in symbol definitions are saved during Pass1. Additional Passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

Source: diginotes.in Go paperless! Save Earth!