MODULE 1
System Software

–System Software consists of a variety of programs that support the operation of a computer. System software is used for operating computer hardware.
–The programs implemented in either software and (or) firmware that makes the computer hardware usable.
–The software makes it possible for the users to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally
.–Example: BIOS (Basic Input Output System

Application software deals with the specific tasks for the user. These are created to solve various types of computing problems. Application software includes packaged software like word processing, programming languages such as BASIC, COBOL, C, Visual Basic, and other commercial and custom software.

Basis for Comparison	System Software	Application Software
Basic	System Software manages system resources and provides a platform for application software to run.	Application Software, when run, perform specific tasks, they are designed for.
Language	System Software is written in a low-level language, i.e. assembly language.	Application Software is written in a high-level language like Java, C++, .net, VB, etc.
Run	System Software starts running when the system is turned on, and runs till the system is shut down.	Application Software runs as and when the user requests.
Requirement	A system is unable to run without system software.	Application software is even not required to run the system; it is user specific.
Purpose	System Software is general-purpose.	Application Software is specific-purpose.
Examples	Operating system.	Microsoft Office, Photoshop, Animation Software, etc.

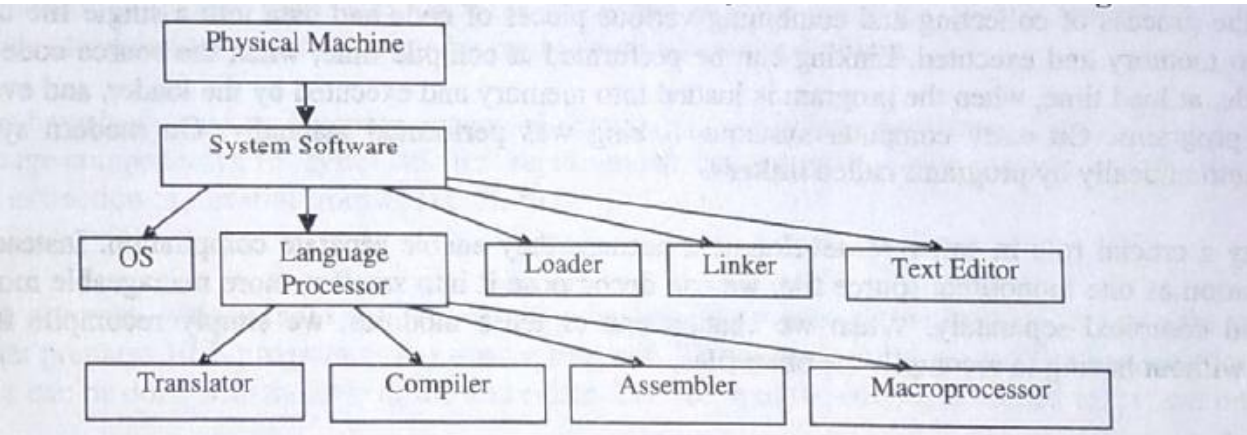


Figure 1.1: Components of System Software

Compiler:
is a program whose task is to accept as input a source program written in a certain high-level language and to produce as output an object code or assembly code.

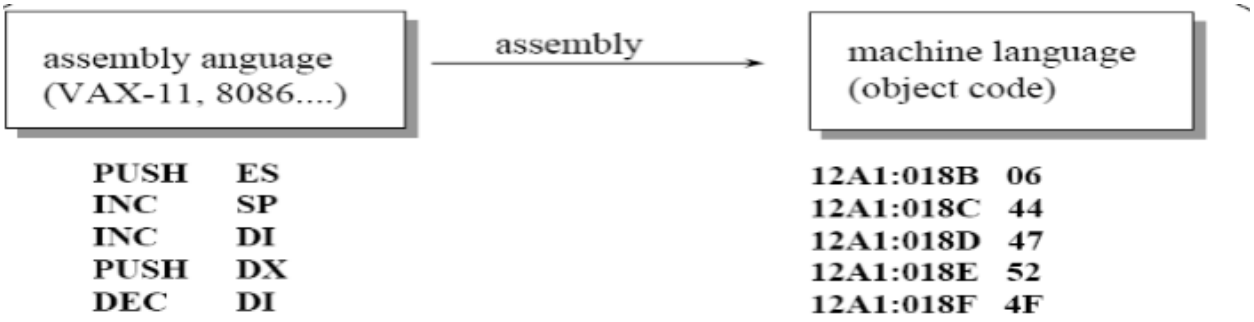

```
#include <stdio.h>

main()
{
   printf("Compiler Design Theory\n");
}
```

```
_main   proc    near
        push    ds
        mov     ax,offset,GROUP:s@
        push    ax
        call    near ptr_printf
        pop     cx
```

•**Interpreter:**
is a program that ultimately performs the same function as a compiler, but in a different manner. It works by scanning through the source program instruction by instruction. As each instruction is encountered, the interpreter translates it into machine code and executes it directly.

•**Assembler:**
is a program that automatically translates the source program written in assembly language and to produce as output an object code written in binary machine code.



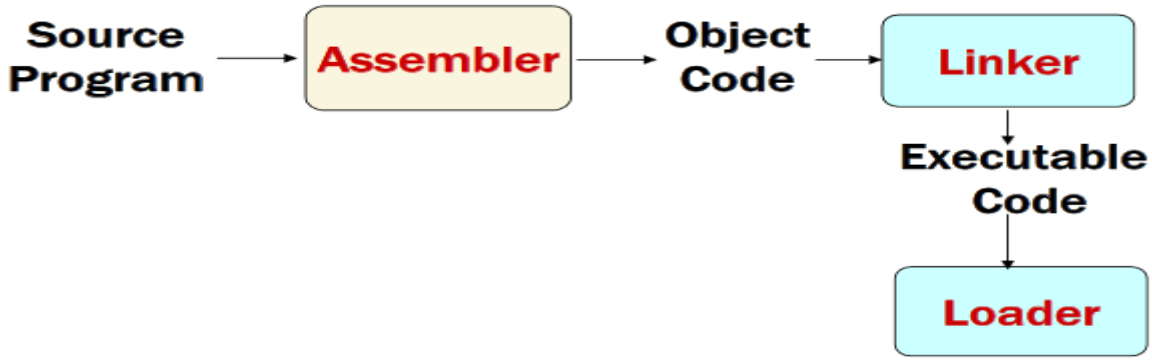
•Linker:

The Assembler generates the object code of a source program and hands it over to the linker. The linker takes this object code and generates the **executable code** for the program, and hand it over to the Loader.

Linking loader: Performs all linking and relocation operations.
Linkage editor: Produces a linked version of the program, which is normally written to a file or library for later execution.
Dynamic Linker :Dynamic linker performs dynamic loading or load on call.

•Loader:

(is a routine that) loads an object program into memory of the processor and prepares it for execution. It calculates the size of a program (instructions and data) and create memory space for it. It initializes various registers to initiate execution.



- **Macro Processor**

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. Macro processors are often embedded in other programs, such as assemblers and compilers.

#define pi 3.14

Two new assembler directives are used in macro definition

- MACRO: identify the beginning of a macro definition
- MEND:identify the end of a macro definition

- **Debuggers**

A debugger is a computer program used by programmers to test and debug a target program. Debuggers may use instruction-set simulators, rather than running a program directly on the processor to achieve a higher level of control over its execution. This allows debuggers to stop or halt the program according to specific conditions.

- **Device Driver**

A device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on.

The Simplified Instructional Computer (SIC)

- SIC is a hypothetical computer that includes the hardware features most often found on real machines.
- Designed to be similar to real computers
- Is a virtual machine

Two versions of SIC

- standard model (SIC)
- extension version (SIC/XE)
    - Upward compatible- Programs for SIC can run on SIC/XE

Characteristics of SIC -SIC Architecture

1. Memory

- 2^{15} (32,768) bytes in the computer memory
- 3 consecutive bytes form a word
- Memory consists of 8-bit bytes
- Any 3 consecutive bytes form a word (24 bits)

2. Registers

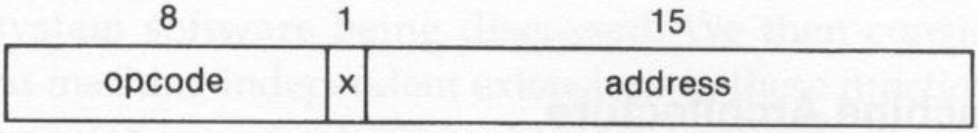
- 5 registers
- Each a full word (24-bits) registers

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

3. Data Formats

- Integers are stored as 24-bit binary number
- 2's complement representation for negative values
- Characters are stored using 8-bit ASCII codes
- No floating-point hardware on the standard SIC

4. Instruction Formats



x: indicate indexed-addressing mode

5. Addressing Modes

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

(X): the contents of register X

6. Instruction Set

a) Load and store registers

LDA, LDX, STA, STX, etc.

b) Integer arithmetic operations

ADD, SUB, MUL, DIV

All arithmetic operations involve register A and a word in memory, with the result being left in A.

c) COMP

COMP – compares A with a word in memory
Sets the CC in the SW

d) Jump instructions

JLT – Jump Less Than
JGT – Jump Greater Than
JEQ– Jump Equal to

These instructions test the setting of CC (as set by COMP) and jump accordingly

7. Subroutine linkage

- **JSUB** - jumps to the subroutine, placing the return address in register L
- **RSUB -** returns by jumping to the address contained in register L

8. Input/Output

- **Test Device instruction (TD)** The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data.
  - CC of < means device is ready
  - CC of = means device is not ready

- Read Data (RD) – read data, when the device is ready
- Write Data (WD) write data

Input and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A

i) WORD – ONE-WORD CONSTANT
ii) RESW – ONE-WORD VARIABLE
iii) BYTE – ONE-BYTE CONSTANT
iv) RESB – ONE-BYTE VARIABLE

1. SIC Programming Examples -- Data movement

```
LDA     FIVE        Load 5 into register A
STA     ALPHA       Store A into ALPHA
LDCH    CHARZ       Load character 'Z' into A
STCH    C1          Store into C1
```

```
.
.
ALPHA    RESW      1          one word variable
FIVE     WORD      5          one word constant
CHARZ    BYTE      C'Z'       one byte constant
C1       RESB      1          one byte variable
```

2. SIC Programming Example-- Arithmetic operation

```
         LDA     ALPHA       LOAD ALPHA INTO REGISTER A
         ADD     INCR        ADD THE VALUE OF INCR
         SUB     ONE         SUBTRACT 1
         STA     BETA        STORE IN BETA
         LDA     GAMMA       LOAD GAMMA INTO REGISTER A
         ADD     INCR        ADD THE VALUE OF INCR
         SUB     ONE         SUBTRACT 1
         STA     DELTA       STORE IN DELTA
           .
           .
           .
ONE      WORD    1           ONE-WORD CONSTANT
.                            ONE-WORD VARIABLES
ALPHA    RESW    1
BETA     RESW    1
GAMMA    RESW    1
DELTA    RESW    1
INCR     RESW    1
```

(a)

BETA=ALPHA+INCR-ONE

DELTA=GAMMA+INCR-ONE

All arithmetic operations are performed using register A, with the result being left in register A.

3. SIC Programming Example -- Looping and indexing

```
          LDX    ZERO       initialize index register to 0
MOVECH    LDCH   STR1,X     load char from STR1 to reg A
          STCH   STR2,X
          TIX    ELEVEN     add 1 to index, compare to 11
          JLT    MOVECH     loop if "less than"
          .
          .
          .
STR1      BYTE   C'TEST STRING'
STR2      RESB   11
ZERO      WORD   0
ELEVEN    WORD   11
```

4. SIC Programming Example -Input and output

```
INLOOP  TD      INDEV    test input device
        JEQ     INLOOP   loop until device is ready (<)
        RD      INDEV    read one byte into register A
        STCH    DATA
        .
        .
OUTLP   TD      OUTDEV   test output device
        JEQ     OUTLP    loop until device is ready (<)
        LDCH    DATA
        WD      OUTDEV   write one byte to output device
        .
        .
INDEV   BYTE    X'F1'    input device number
OUTDEV  BYTE    X'05'    output device number
DATA    RESB    1
```

SIC/XE Machine Architecture

1. Memory

- 2^{20} (1MB) in the computer memory
- 3 consecutive bytes form a word
- Byte Addressable
- Any 3 consecutive bytes form a word (24 bits)

2. Registers

- 5 registers of SIC + 4 additional
- Each a full word
- Each a full word (24-bits) registers

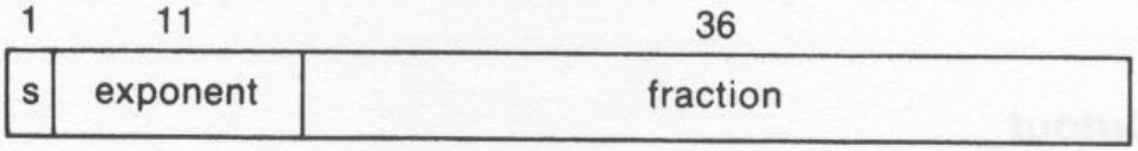
Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register—no special use
T	5	General working register—no special use
F	6	Floating-point accumulator (48 bits)

3. Data Formats

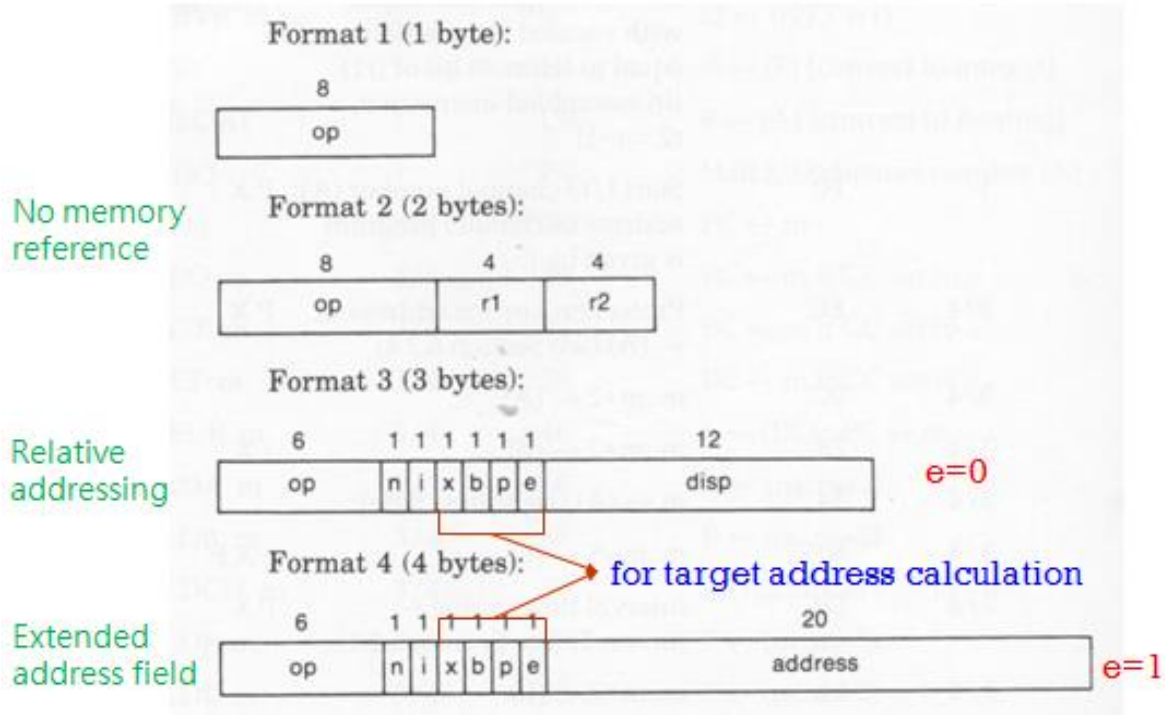
- Integers are stored as 24-bit binary number
- 2's complement representation for negative values
- Characters are stored using 8-bit ASCII codes
- Floating point – 48 bit floating point

Floating point

- the exponent is e and the fraction is f
- The number is $f * 2^{(e+1024)}$
- frac: 0~1
- exp: 0~2047



4. Instruction Formats



- Format 2 – 16 bit (2 bytes)

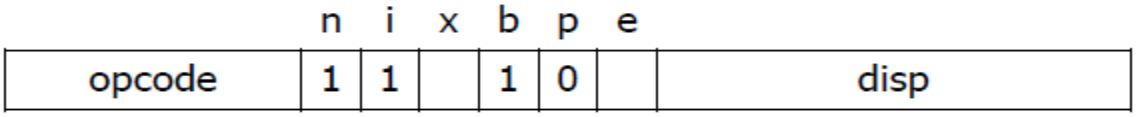
  ADDR T,A

, COMPR A,S (Compare the contents of register A and S).

Opcode		A	S
1010	0000	0000	0100

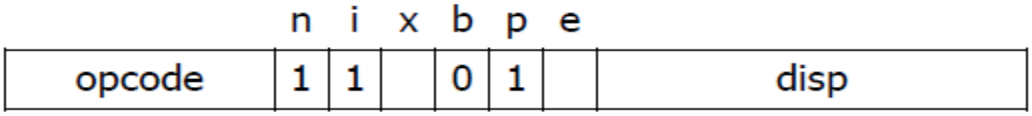
Mode	Indication	Target address calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (X)

- Base Relative Addressing Mode



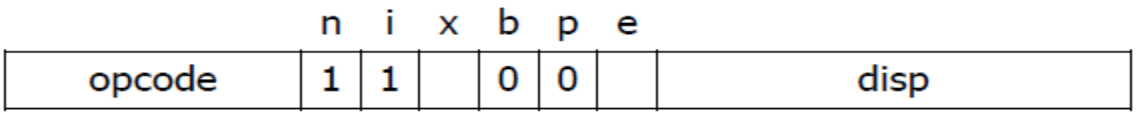
n=1, i=1, b=1, p=0, TA=(B)+disp (0≤disp ≤4095)

- Program-Counter Relative Addressing Mode

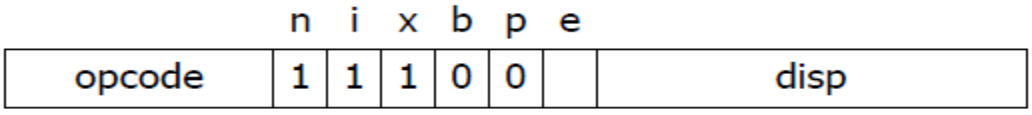


n=1, i=1, b=0, p=1, TA=(PC)+disp (-2048≤disp ≤2047)

- Direct Addressing Mode

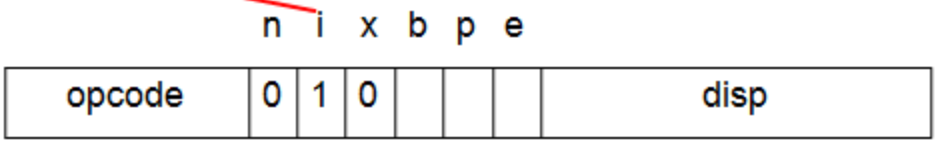


n=1, i=1, b=0, p=0, TA=disp (0≤disp ≤4095)



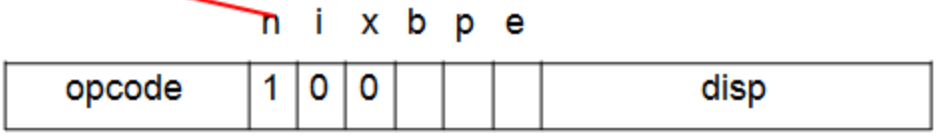
n=1, i=1, b=0, p=0, TA=(X)+disp
(with index addressing mode)

- Immediate Addressing Mode



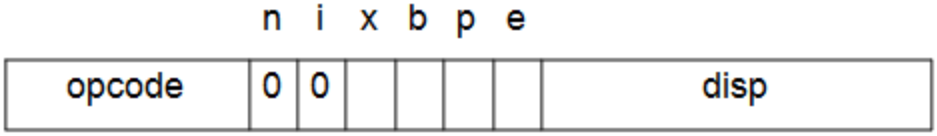
n=0, i=1, x=0, operand=disp

- Indirect Addressing Mode

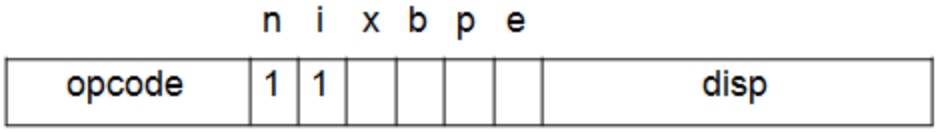


n=1, i=0, x=0, TA=(disp)

- Simple Addressing Mode



i=0, n=0, TA=bpe+disp (SIC standard)
opcode+n+i = SIC standard opcode (8-bit)



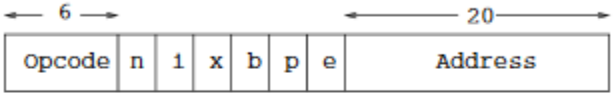
i=1, n=1, TA=disp (SIC/XE standard)

- Bits x,b,p,e: how to calculate the target address
  - relative, direct, and indexed addressing modes
- Bits i and n: how to use the target address (TA)
  - i=1, n=0: immediate addressing
    - TA is used as the operand value, no memory reference
  - i=0, n=1: indirect addressing
    - The word at the TA is fetched

        - Value in this word is taken as the address of the operand value
    - i=0, n=0 (in SIC), or
    - i=1, n=1 (in SIC/XE): simple addressing
        - TA is taken as the address of the operand value
- Any of these addressing modes can also be combined with indexed addressing.

– Direct	b=0, p=0	TA=disp
– Index	x=1	$TA_{new}=TA_{old}+(X)$
– Index+Base relative	x=1, b=1, p=0	TA=(B)+disp+(X)
– Index+PC relative	x=1, b=0, p=1	TA=(PC)+disp+(X)
– Index+Direct	x=1, b=0, p=0	
– Format 4	e=1	

4-Byte format:



Notes:

- Bits b and p are always 0. (Thus, 4-byte format does not allow relative addressing.)
- Bit e is always 1 (to distinguish between 3-byte and 4-byte instructions).

5. Addressing Modes

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

- Base relative (n=1, i=1, b=1, p=0)
- Program-counter relative (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 for format 4, e=0 for format 3)

6. **Instruction Set**

Instruction Set

- 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
- 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

- RMO S,B
  - Register S content is moved to Register B
- ADDR S,B
  - Add value of S with B and store in Register B
- IO channels
  - Perform IO while CPU is executing other instructions
  - Three instructions:

- SIO: start the operation of IO channel
- TIO: test the operation of IO channel
- HIO: halt the operation of IO channel

SIC/XE Programming Example

SIC version

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

SIC/XE version

```
        LDA     #5
        STA     ALPHA
        LDCH    #90
        STCH    C1
        .
        .
        .
ALPHA   RESW    1
C1      RESB    1
```

ASCI Value of 'Z' 90

Indirect addressing indicated by '@':

```
JEQ     @RADDR
```

Immediate operands are indicated by '#':

```
LDA     #50
```

Addition and Subtraction in SIC\XE

```
        LDS     INCR
        LDA     ALPHA   BETA=ALPHA+INCR-1
        ADDR    S,A
        SUB     #1
        STA     BETA
        LDA     GAMMA   DELTA=GAMMA+INCR-1
        ADDR    S,A
        SUB     #1
        STA     DELTA
        ...
        ...
ALPHA   RESW    1       one-word variables
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

- 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/XE Subroutine Example

```
          JSUB    READ          CALL READ SUBROUTINE
                  .
                  .
                  .
.                               SUBROUTINE TO READ 100-BYTE RECORD
READ      LDX     #0            INITIALIZE INDEX REGISTER TO 0
          LDT     #100          INITIALIZE REGISTER T TO 100
RLOOP     TD      INDEV         TEST INPUT DEVICE
          JEQ     RLOOP         LOOP IF DEVICE IS BUSY
          RD      INDEV         READ ONE BYTE INTO REGISTER A
          STCH    RECORD,X      STORE DATA BYTE INTO RECORD
          TIXR    T             ADD 1 TO INDEX AND COMPARE TO 100
          JLT     RLOOP         LOOP IF INDEX IS LESS THAN 100
          RSUB                  EXIT FROM SUBROUTINE
                  .
                  .
                  .
INDEV     BYTE    X'F1'         INPUT DEVICE NUMBER
RECORD    RESB    100           100-BYTE BUFFER FOR INPUT RECORD
```

SIC/XE Input and Output Operation

```
INLOOP    TD      INDEV         TEST INPUT DEVICE
          JEQ     INLOOP        LOOP UNTIL DEVICE IS READY
          RD      INDEV         READ ONE BYTE INTO REGISTER A
          STCH    DATA          STORE BYTE THAT WAS READ
                  .
                  .
                  .
OUTLP     TD      OUTDEV        TEST OUTPUT DEVICE
          JEQ     OUTLP         LOOP UNTIL DEVICE IS READY
          LDCH    DATA          LOAD DATA BYTE INTO REGISTER A
          WD      OUTDEV        WRITE ONE BYTE TO OUTPUT DEVICE
                  .
                  .
                  .
INDEV     BYTE    X'F1'         INPUT DEVICE NUMBER
OUTDEV    BYTE    X'05'         OUTPUT DEVICE NUMBER
DATA      RESB    1             ONE-BYTE VARIABLE
```

- **Direct addressing** A
  - E.g. LDA ZERO
- **Immediate addressing** #A
  - E.g. LDA #0
- **Indexed addressing** A, X
  - E.g. STCH BUFFER, X
- **Indirect addressing** @A
  - E.g J @RETADR

SIC Program to swap the values of ALPHA and BETA

Code	Description
LDA ALPHA	Load the value of ALPHA in Accumulator
STA GAMMA	Store the value of Accumulator to GAMMA
LDA BETA	Load the value of BETA to Accumulator
STA ALPHA	Store the value of Accumulator to ALPHA
LDA GAMMA	Load the value of GAMMA to Accumulator
STA BETA	Store the value of Accumulator to BETA
ALPHA RESW 1	Reserve 1 word for ALPHA
BETA RESW 1	Reserve 1 word for BETA
GAMMA RESW 1	Reserve 1 word for GAMMA

Write a sequence of instructions for SIC to ALPHA equal to the product of BETA and GAMMA.

Assembly Code:

```
LDA BETA
MUL GAMMA
STA ALPHA
:
:
ALPHA RESW 1
```

```
BETA RESW 1
GAMMA RESW 1
```

Write a sequence of instructions for SIC to set ALPHA equal to the integer portion of BETA ÷ GAMMA

Assembly Code:

```
LDA BETA
DIV GAMMA
STA ALPHA
:
:
ALPHA RESW 1
BETA RESW 1
GAMMA RESW 1
```

Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the integer portion of the quotient and DELTA to the remainder. Use register-to-register instructions to make the calculation as efficient as possible.

Assembly Code:

```
LDA  BETA
LDS  GAMMA
DIVR  S, A
STA  ALPHA
MULR  S, A
LDS  BETA
SUBR  A, S
STS  DELTA
:
:
ALPHA          RESW      1
BETA           RESW      1
```

```
GAMMA       RESW    1
DELTA       RESW    1
```

Write a sequence of instructions for SIC/XE to set ALPHA equal to 4*BETA – 9.

```
        LDA BETA
        LDS #4

        MULR  S,A

        SUB #9

        STA  ALPHA

ALPHA   RESW    1
```

Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks.

Assembly Code:

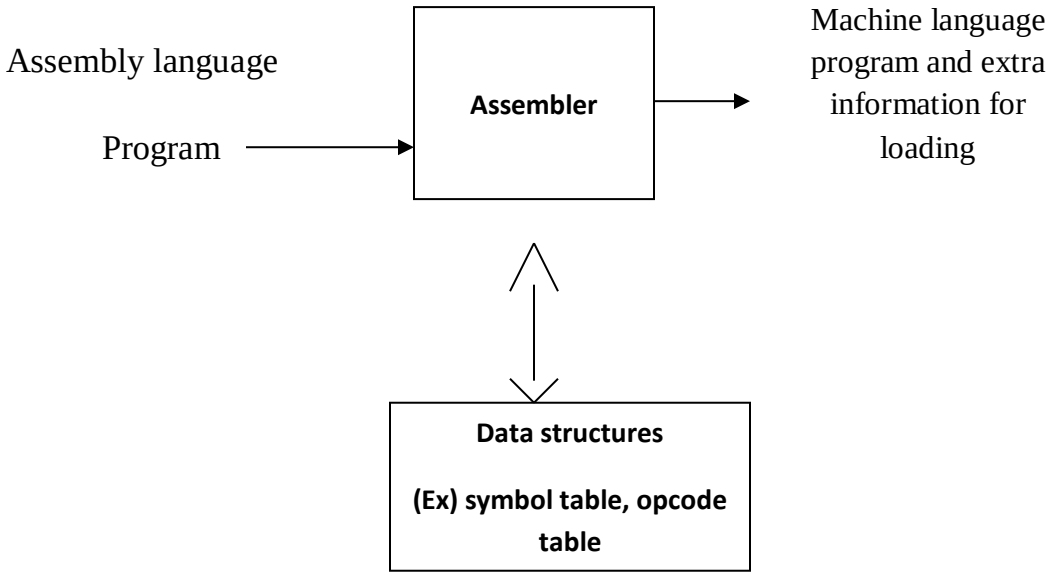
```
            LDX   ZERO
LOOP        LDCH  BLANK
            STCH  STR1,X
            TIX   TWENTY
            JLT   LOOP
:
:
            STR1        RESW    20
            BLANK       BYTE    C ' '
            ZERO        WORD    0
            TWENTY      WORD    20
```

Assembler directives

- Assembler directives are instructions that direct the assembler to do something.
- Directives do many things; some tell the assembler to set aside space for variables, others tell the assembler to include additional source files, and others establish the start address for your program.
- SIC Assembler Directive:
  – START: Specify name & starting address.
  – END: End of the program, specify the first execution instruction.
  – BYTE, WORD, RESB, RESW
- These statements are not translated into machine instructions.
- Instead, they provide instructions to the assembler itself.

MODULE 2

Assembly language is converted into executable machine code by a utility program referred to as an assembler. An assembler is system software that accepts an assembly language program as its input and produces its machine language equivalent along with information for the loader as its output. It is a translator that converts the assembly language program into machine language program. The structure of the assembler is given as



Assembly language program

The sequence of instructions to the assembler is called as assembly language program that uses set of mnemonics.The format of the instruction varies from system to system based on the machine architecture. In SIC, the format of the assembly language instruction is given as

Label	Opcode or Mnemonics	Operands

(Ex)

FIRST	SLT	RETADR
CLOOP	JSUB	RDREC
	LDA	LENGTH
	RSUB	

Basic Assembler Directives

- Assembler directives are pseudo instructions
- They provide instructions to the assembler itself
- They are not translated into machine operation codes

The Assembler directives are:

- START:Specify name & starting address for the program.
- END: Indicate the end of the source program and specify the first execution instruction in the program.
- BYTE: Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
- EQU: Symbol EQU value
- When the assembler encounters the EQU statement, it enters "symbol" into SYMTAB with the value of "symbol"
- WORD: Generate one-word integer constant.
- RESB: Reserves the indicated number of bytes for a data area.
- RESW: Reserves the indicated number of words for a data area.
- End of record: a null char (00).
- End of file: a zero length record.

Assembler language program for basic SIC version

The example program considered has a main module and 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

Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	

Figure 1: Assembler language program

- Data transfer (RD, WD)

  -A buffer is used to store record

  -Buffering is necessary for different I/O rates

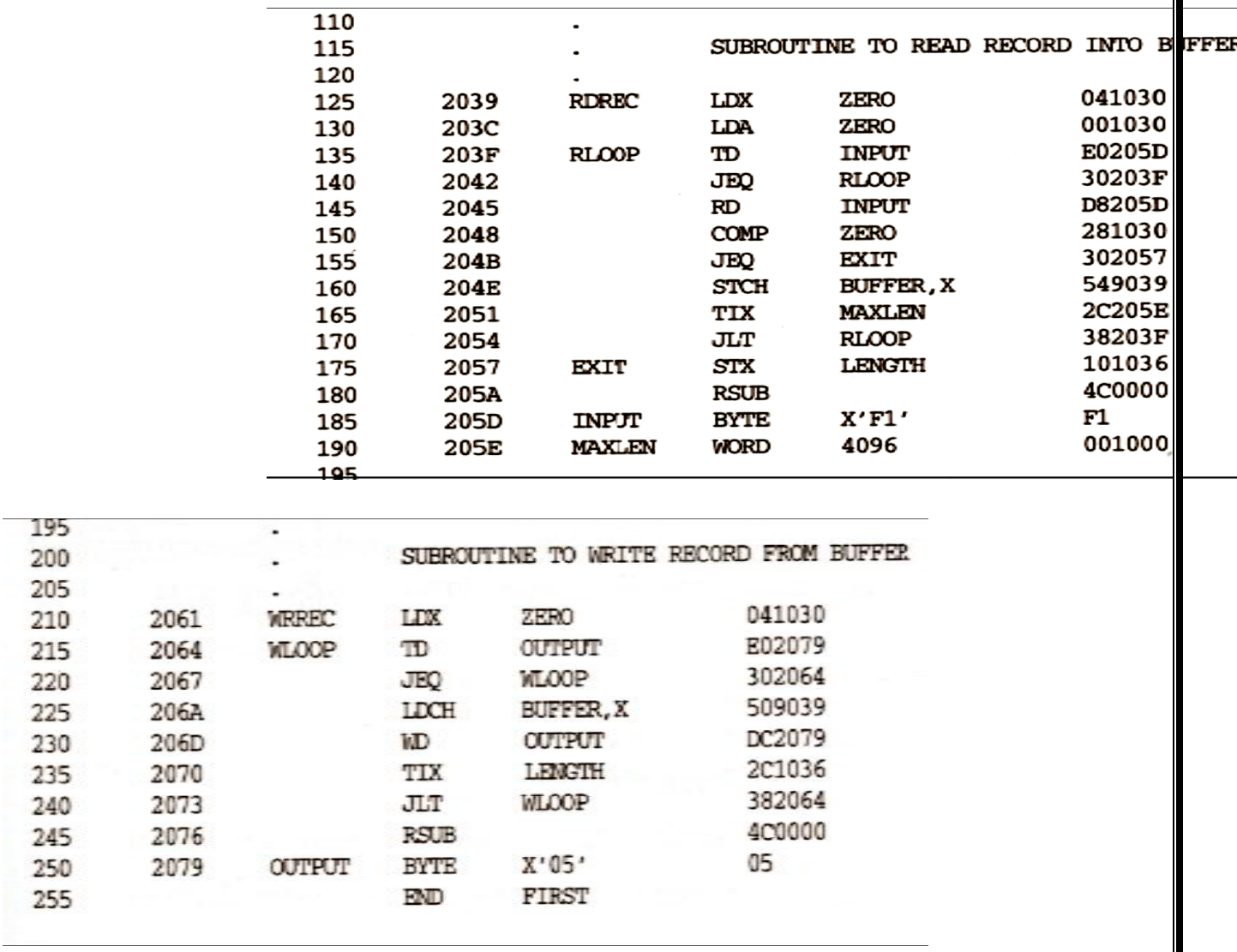
  -The end of each record is marked with a null character (00) hexadecimal.

  -The end of the file is indicated by a zero-length record

The main routine calls subroutines:

- **RDREC** – To read a record into a buffer.
- **WRREC** – To write the record from the buffer to the output device.

The end of each record is marked with a null character (hexadecimal 00).



```
110                 .
115                 .       SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
125     2039    RDREC   LDX     ZERO            041030
130     203C            LDA     ZERO            001030
135     203F    RLOOP   TD      INPUT           E0205D
140     2042            JEQ     RLOOP           30203F
145     2045            RD      INPUT           D8205D
150     2048            COMP    ZERO            281030
155     204B            JEQ     EXIT            302057
160     204E            STCH    BUFFER,X        549039
165     2051            TIX     MAXLEN          2C205E
170     2054            JLT     RLOOP           38203F
175     2057    EXIT    STX     LENGTH          101036
180     205A            RSUB                    4C0000
185     205D    INPUT   BYTE    X'F1'           F1
190     205E    MAXLEN  WORD    4096            001000
195
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
```

Figure 2: Assembler language program for basic SIC version with Object code

1. The column "Loc" in the example program(Figure 1: Assembler language program) gives the machine address in hexadecimal for each and every line of the assembled program.
2. The program starts at address 1000 (it's only assumption – for simple SIC machine the starting address will always be assumed and the value will not be 0).
3. The translation of source program to object code requires the following functions:
   a. Convert mnemonic operation codes to their machine language equivalents. Eg: Translate STL to 14 (line 10).

b. Convert symbolic operands to their equivalent machine addresses. Eg:Translate RETADR to 1033 (line 10).
c. Build the machine instructions in the proper format.
d. Convert the data constants specified in the source program into their internal machine representations. Eg: Translate EOF to 454F46(line 80).
e. Write the object program and the assembly listing.

4. All the statements in the program except statement 2 can be established by sequential processing of source program one line at a time.
   Consider the statement

   10 1000 FIRST STL RETADR 141033

5. This instruction contains a **forward reference** (i.e.) - a reference to a label (RETADR) that is defined later in the program.
6. It is unable to process this line because the address that will be assigned to RETADR is not known.
7. Hence most assemblers make two passes over the source program.
8. The first pass does little more than scan the source program for label definitions and assign address, such as those in the Loc column,
9. The second pass performs most of the actual translation.
10. The assembler must also process statements called **assembler directives or pseudo instructions** which are not translated into machine instructions. Instead they provide instructions to the assembler itself. Examples: RESB and RESW instruct the assembler to reserve memory locations without generating data values.
11. The assembler must write the generated object code onto some output device. This object program will later be loaded into memory for execution.

Object program format contains three types of records:

- **Header record**: Contains the program name, starting address and length.
- **Text record**: Contains the machine code and data of the program.

- **End record**: Marks the end of the object program and specifies the address in the program where execution is to begin.

Record format is as follows:

Header record:

Col. 1 H

Col.2-7 Program name

Col.8-13 Starting address of object program

Col.14-19 Length of object program in bytes

Text record:

Col.1 T

Col.2-7 Starting address for object code in this record

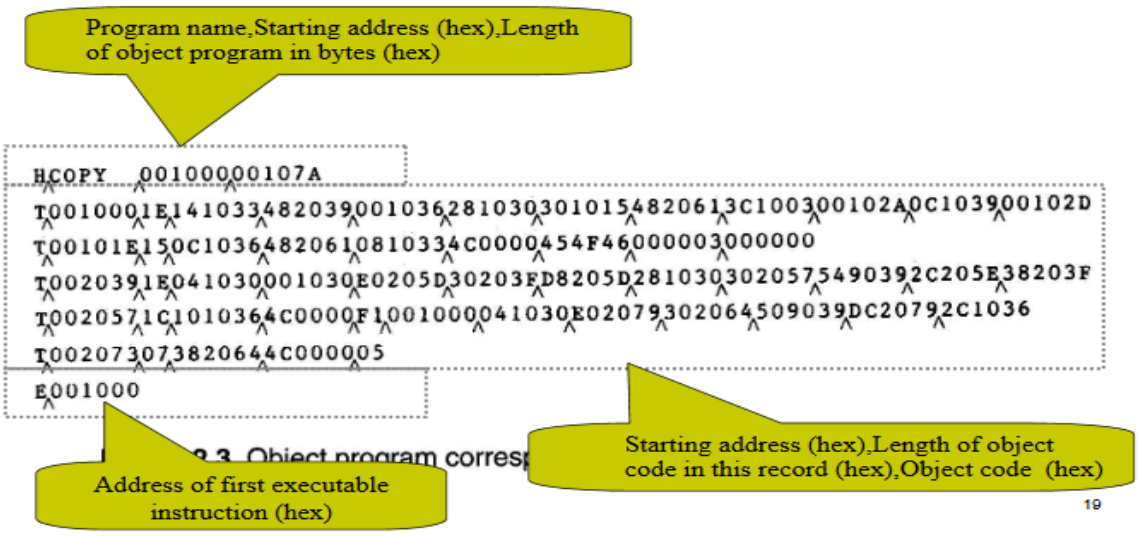
Col.8-9 Length of object code in this record in bytes

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.



The assembler design can be done:

- Single pass assembler

- Multi-pass assembler

Single-pass Assembler:

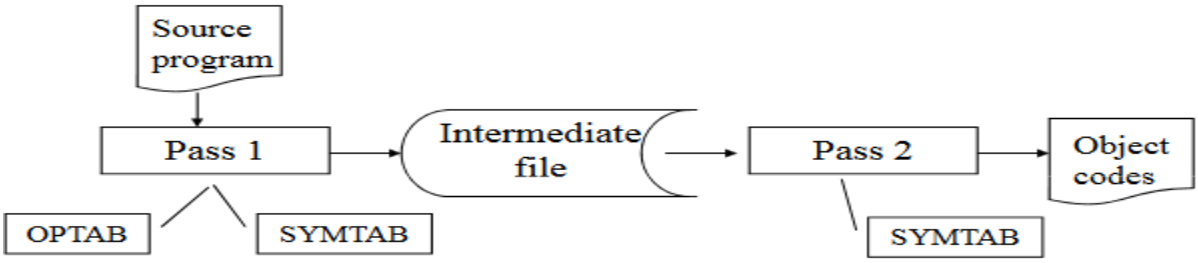
- In Single-pass assembler the whole process of scanning, parsing, and object code conversion is done in single pass.
- The only problem with this method is resolving forward reference.
- The problem of forward reference is shown with an example below:

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 in 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 is valid then it has to be converted to its equivalent object code. For this the object code is generated by adding the opcode of STL and the value for the symbol RETADR, 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:



Data Structures:

Operation Code Table (OPTAB)
Symbol Table (SYMTAB)
Location Counter(LOCCTR)

A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references (the problem of forward referencing) and assemble code in one pass

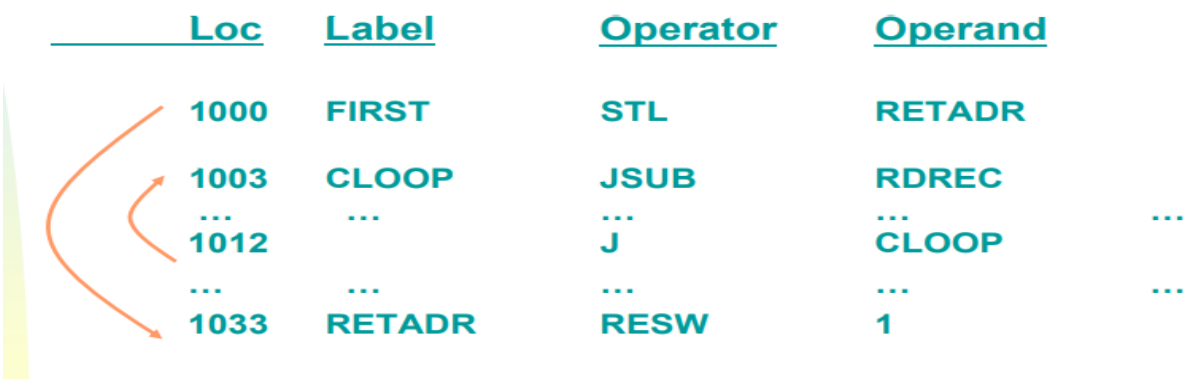
Assembler Design:

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.
- 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.

- **Forward reference: reference to a label that is defined later in the program.**



Functions of the two passes of assembler:

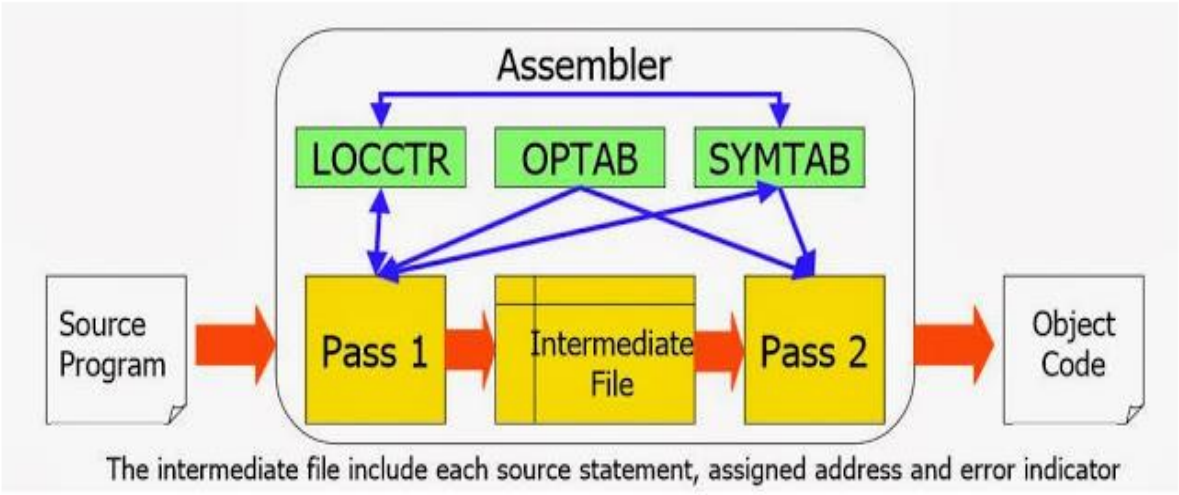
Pass 1 (Define symbols)

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

Pass 2 (Assemble instructions and generate object programs)

1. Assemble instructions (translating operation codes and looking up addresses).
2. Generate data values defined by BYTE,WORD etc.
3. Perform processing of assembler directives not done in Pass 1.
4. Write the object program and the assembly listing.

Assembler Algorithm and Data Structures



Assembler uses two major internal data structures:

1. **Operation Code Table (OPTAB) :** Used to lookup mnemonic operation codes and translate them into their machine language equivalents.
2. **Symbol Table (SYMTAB) :** Used to store values(Addresses) assigned to labels.

Location Counter (LOCCTR) :

- It is a variable used to help in the assignment of addresses.
- It is initialized to the beginning address specified in the START statement.
- After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR.
- Whenever a label is reached in the source program, the current value of LOCCTR gives the address to be associated with that label.

Operation Code Table (OPTAB) :

Looked up for the translation of mnemonic code

- key: mnemonic code
- result: bits

Hashing is usually used

- once prepared, the table is not changed
- efficient lookup is desired
- since mnemonic code is predefined, the hashing function can be tuned *a priori*

The table may have the instruction format and length

- to decide where to put op code bits, operands bits, offset bits
- for variable instruction size
- used to calculate the address

- In Pass 1, OPTAB is used to lookup and validate operation codes in the source program.
- In Pass 2, it is used to translate the operation codes to machine language program.
- During Pass 2, the information in OPTAB tells which instruction format to use in assembling the instruction and any peculiarities of the object code instruction.
- In most cases, OPTAB is a static table – i.e entries are not added or deleted from it.

Symbol Table (SYMTAB) :

Stored and looked up to assign address to labels

- efficient insertion and retrieval is needed
- deletion does not occur

Difficulties in hashing

- non random keys

Problem

    - the size varies widely
- Includes the name and value(address) for each label in the source program and flags to indicate error conditions(ex – symbol defined in two different places).
- During Pass 1 of the assembler, labels are entered into SYMTAB as they are encountered in the source program along with their assigned addresses.
- During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address, error indicators. This file is used as the input to Pass 2. This copy of the source program can also be used to retain the results of certain operations that may be performed during Pass 1 such as scanning the operand field for symbols and addressing flags, so these need not be performed again during Pass 2.

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
                        end {if symbol}
                        search OPTAB for OPCODE
                        if found then
                                add 3 (instruction 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  {if BYTE}
            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}
```

Explanation – PASS I Algorithm

1. Read in a line of assembly code
2. Assign an address to this line
    - increment N (word addressing or byte addressing)
3. Save address values assigned to labels
    - in symbol tables
4. Process assembler directives
    - constant declaration
    - space reservation

- 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 3 * 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 from intermediate file
        if OPCODE =  'START' then
        begin
                write listing line
                read next input line
        end {if START}

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
            if found then
                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
    end {if opcode found}
    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 {while not END}
Write last Text record to object program
Write End record to object program
Write last listing line.
End {Pass 2}
```

Explanation – PASS II Algorithm

- 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 assembled 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.

MACHINE DEPENDENT ASSEMBLER FEATURES

- Instruction formats and addressing modes
- Program relocation

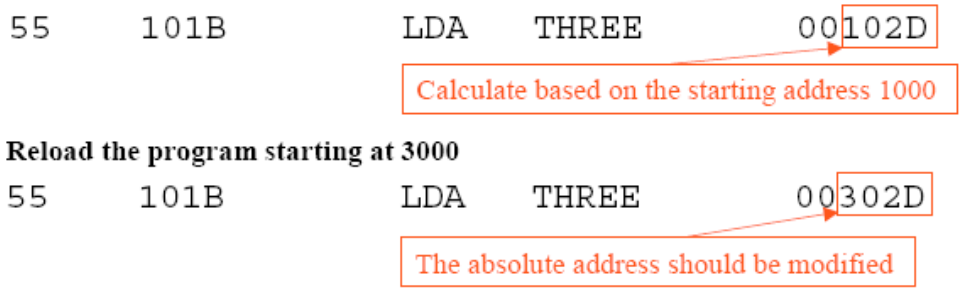
Program Relocation

The need for program relocation

- It is desirable to load and run several programs at the same time.
- The system must be able to load programs into memory wherever there is room.
- The exact starting address of the program is not known until load time.

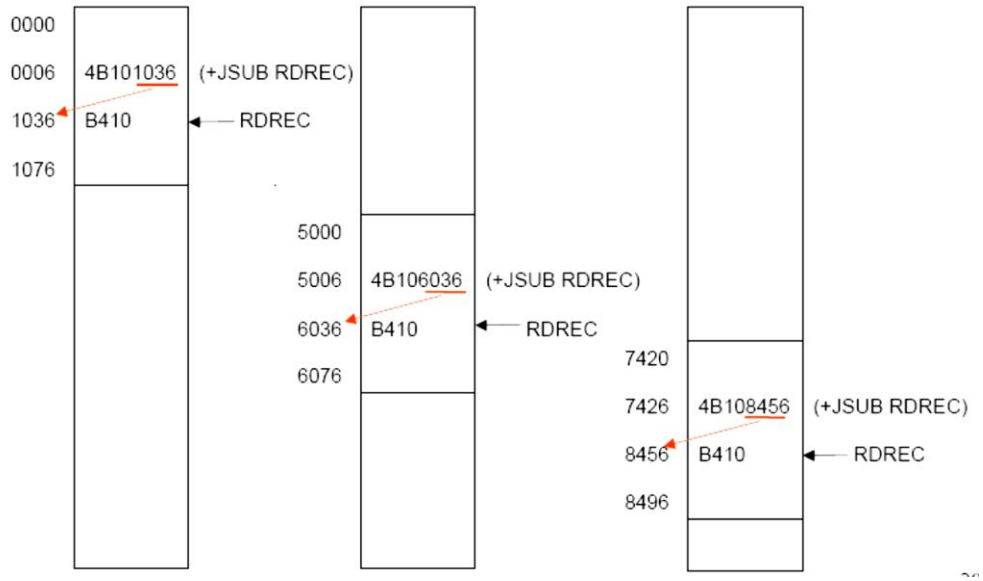
Absolute Address

- Program with starting address specified at assembly time
- The address may be invalid if the program is loaded into somewhere else.
- Example:



- ❖ The above 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 3000 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 3000.
- ❖ 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.

Example: Program Relocation



1) The above diagram shows the concept of relocation.
   a) Initially the program is loaded at location 0000.
   b) The instruction JSUB is loaded at location 0006.
   c) The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC.
2) 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.
3) 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 parts of the program that require modification at load time are those that specify direct addresses.
   - The rest of the instructions need not be modified.
     - Not a memory address (immediate addressing)

- PC-relative, Base-relative

- 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.

The way to solve the relocation problem

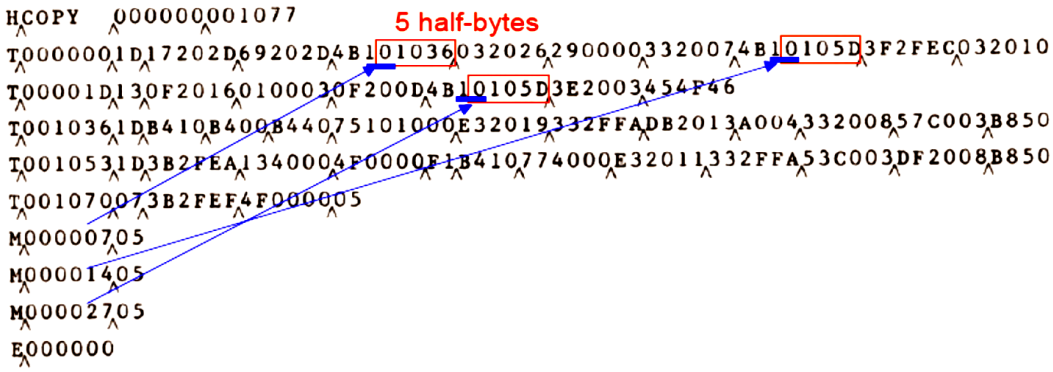
- For an address label, its address is assigned relative to the start of the program(START 0)
- Produce a Modification record to store the starting location and the length of the address field to be modified.
- The command for the loader must also be a part of the object program.

Modification record

- One modification record 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.

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)

Relocatable Object Program



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.

Module 3
Assembler Features and Design Options

Machine Dependent Assembler Features

1. **Instruction format and Addressing modes**
    - PC-relative or Base-relative addressing: op m
    - Indirect addressing: op @m
    - Immediate addressing: op #c
    - Extended format: +op m
    - Index addressing: op m,x
    - register-to-register instructions
2. **Program Relocation**

Moving program from one memory location to another memory location is known as Program Relocation. The only parts of the program that require modification at load time are those that specify **direct addresses**. Except for **absolute address**, the rest of the instructions need not be modified.

The below given diagram is an example for program relocation.

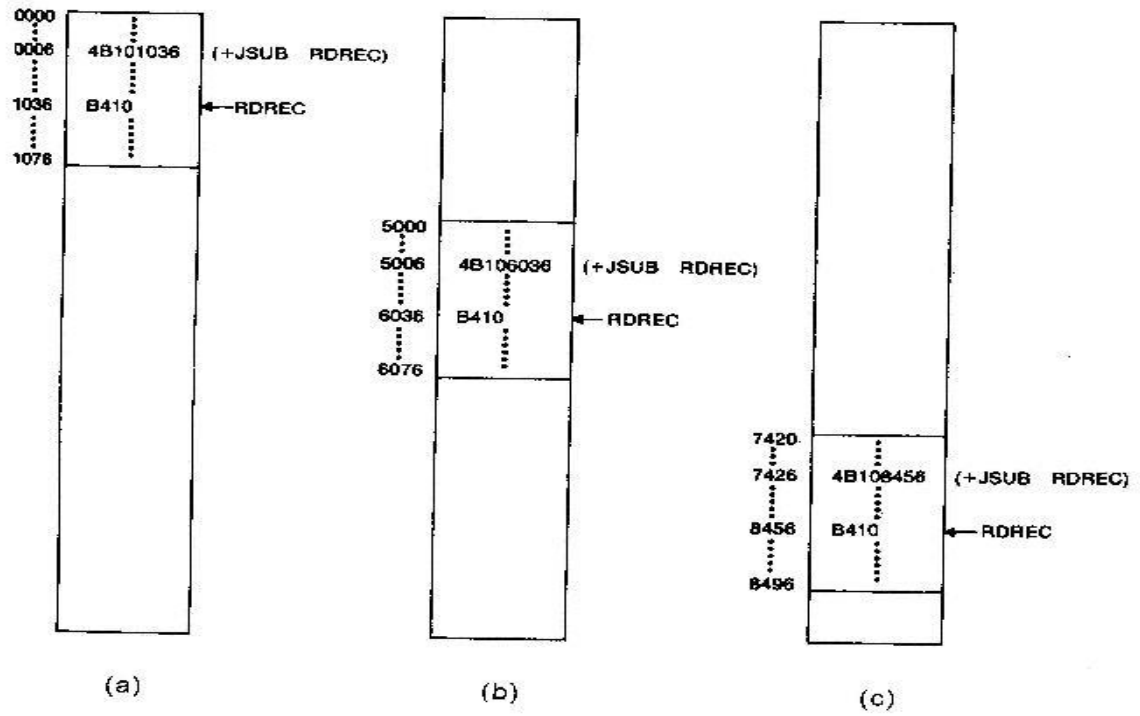


圖2.7 程式重定位範例

Machine-Independent Assembler Features

Machine Independent Assembler features are some common assembler features that are not closely related to machine architecture.The main such features are

- Literals
- Symbol Defining Statement
- Expressions
- Program Blocks
- Control Sections and Program Linking

1. Literals

- One feature is Literal which is an operand and is part of an instruction.
- In Literal the value is stated literally in the instruction
- In an assembler language, a literal is identified with the prefix = which is followed by a specification of the literal value similar to that in a BYTE statement.
- Literal pool is a combination of literal operands.

Design idea

- Let programmers to be able to write the value of a constant operand as a part of the instruction that uses it.This avoids having to define the constant elsewhere in the program and make up a label for it.

Consider the following example

```
        :
        LDA     FIVE
        :
FIVE    WORD    5
        :
```

It is convenient to write the value of a constant operand as a part of instruction

```
        :
        LDA     =X'05'
        :
```

Literals vs. Immediate Operands

- **Immediate addressing**
  - The operand value is assembled as part of the machine instruction.

    e.g. 55 0020 LDA #3 010003

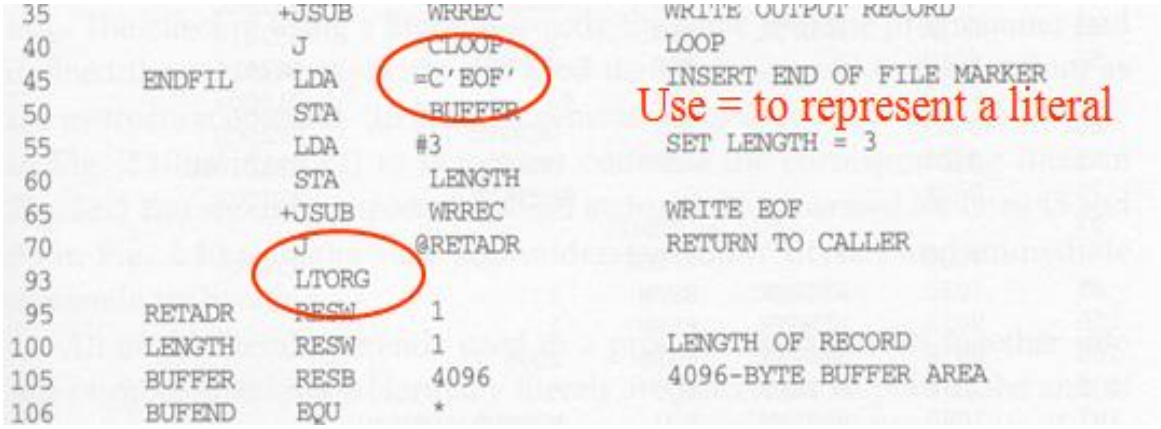
Literal

- The assembler generates the specified value as a constant at some other memory location.
- The address of this generated constant is used as the target address for the machine instruction.
- The effect of using a literal is exactly the same as if the programming had defined the constant explicitly and used the label assigned to the constant as the instruction operand.

e.g. 45 001A ENDFIL LDA =C'EOF' 032010

Literal Pool

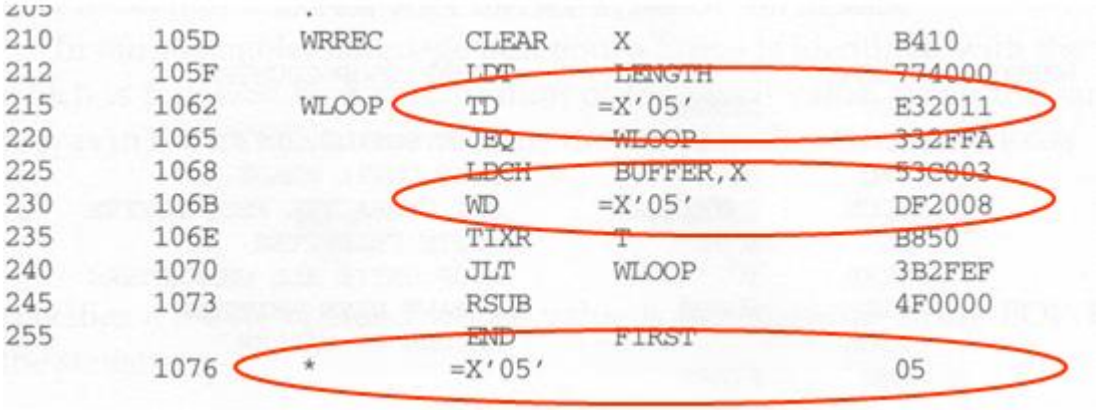
- All of the literal operands used in the program are gathered together into one or more literal pools.Normally literals are placed into a pool at the end of the program.Sometimes, it is desirable to place literals into a pool at some other location in the object program.
  - **LTORG** directive is introduced for this purpose.
  - When the assembler encounters a **LTORG**, it creates a pool that contains all of the literals used since the previous LTORG.



If we do not use a LTORG on line 93, the literal =C'EOF' will be placed at the end of the program.This operand will then begin at address 1073 and be too far away from the instruction that uses it. PC-relative addressing mode cannot be used. LTORG thus is used when we want to keep the literal operands close to the instruction that uses it.

Duplicate literals – the same literal used in more than one place in the program.For duplicate literals, we should store only one copy of the specified data value to save space.Most assembler can recognize duplicate literals.

- E.g., There are two uses of =X'05' on lines 215 and 230 respectively.
- However, just one copy is generated on line 1076.



```
210     105D    WRREC   CLEAR   X           B410
212     105F            LDT     LENGTH      774000
215     1062    WLOOP   TD      =X'05'      E32011
220     1065            JEQ     WLOOP       332FFA
225     1068            LDCH    BUFFER,X    53C003
230     106B            WD      =X'05'      DF2008
235     106E            TIXR    T           B850
240     1070            JLT     WLOOP       3B2FEF
245     1073            RSUB                4F0000
255                     END     FIRST
        1076    *       =X'05'              05
```

A literal table LITTAB is needed for literal processing. For each literal used, the table contains:

- Literal name.
- The operand value and length.
- The address assigned to the operand when it is placed in a literal pool.
- LITTAB is often organized as a hash table, using the literal name or value as the key

LITTAB

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

Implementation of Literals

Pass 1

- Build LITTAB with literal name, operand value and length, leaving the address unassigned
- When LTORG or END statement is encountered, assign an address to each literal not yet assigned an address
- The location counter is updated to reflect the number of bytes occupied by each literal

Pass 2

- Search LITTAB for each literal operand encountered
- Generate data values using BYTE or WORD statements
- Generate Modification record for literals that represent an address

in the program

2. Symbol Defining Statements

EQU directive is used to define a symbol's value.The value assigned to a symbol may be a constant, or any expression involving constants and previously defined symbols.

E.g., +LDT #4096 can be changed to :

```
MAXLEN EQU 4096

+LDT  #MAXLEN
```

- When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB (with value 4096). During assembly of the LDT instruction, the assembler searches SYMTAB for the symbol MAXLEN, using its value as the operand in the instruction.

- Define mnemonic names for registers

```
A       EQU     0
X       EQU     1
L       EQU     2
```

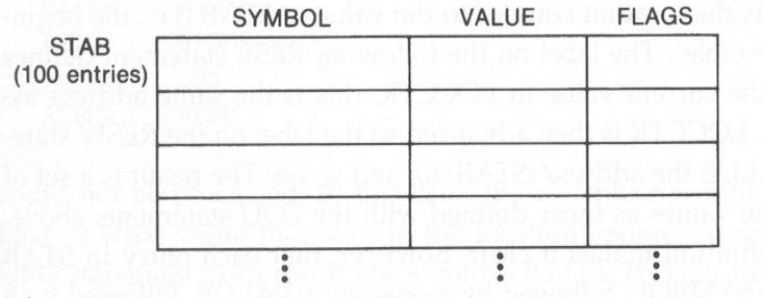
ex: RMO A,X

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

ORG Directive

This can be used to indirectly assign values to symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter to the specified value.The ORG statement will thus affect the values of all labels defined until the next ORG. Normally when an ORG without specified value is encounter, the previously saved location counter value is restored,

Suppose that we have the following data structure and want to access its fields:



SYMBOL: 6 bytes

VALUE: 3 bytes (one word)

FLAGS: 2 bytes

- We want to refer to every field of each entry
- If EQU statements are used

```
STAB    RESB 1100
SYMBOL  EQU  STAB
VALUE   EQU  STAB+6
FLAG    EQU  STAB+9
```

Offset from STAB

If ORG statements are used

```
STAB        RESB   1100
            ORG    STAB        <- Set LOCCTR to STAB
SYMBOL      RESB   6
VALUE       RESW   1           <- Size of each field
FLAGS       RESB   2
            ORG    STAB+1100   <- Restore LOCCTR
```

We can fetch the VALUE field by

```
        LDA     VALUE,X
```

- X = 0, 11, 22, ... for each entry

For EQU and ORG, all symbols used on the right hand side of the statement must have been defined previously in the program.This is because in the two-pass assembler, we require that all symbols must be defined in pass 1.

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

Allowed

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

Not allowed

3. Expression

The assembler evaluates the expressions and produces a single operand address or value. Expressions consist of

- **Operator**
  - +,-,*,/ (division is usually defined to produce an integer result)
- **Individual terms**
  - Constants
  - User-defined symbols
  - Special terms, e.g., *, the current value of LOCCTR

Regarding program relocation, a symbol's value can be classified as

Relative

- Its value is relative to the beginning of the object program, and thus its value is dependent of program location.
- E.g., labels or reference to location counter (*)

Absolute

- Its value is independent of program location
- E.g., a constant. None of the relative terms may enter into a multiplication or division operation
- Errors:
  - BUFEND+BUFFER
  - 100-BUFFER
  - 3*BUFFER

The type of an expression keep track of the types of all symbols defined in the program. Therefore, we need a flag in the symbol table to indicate type of value (absolute or relative) in addition to the value itself.

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

- Absolute value

```
BUFEND - BUFFER
```

- Illegal

```
BUFEND + BUFFER
100 - BUFFER
3 * BUFFER
```

4. Program blocks

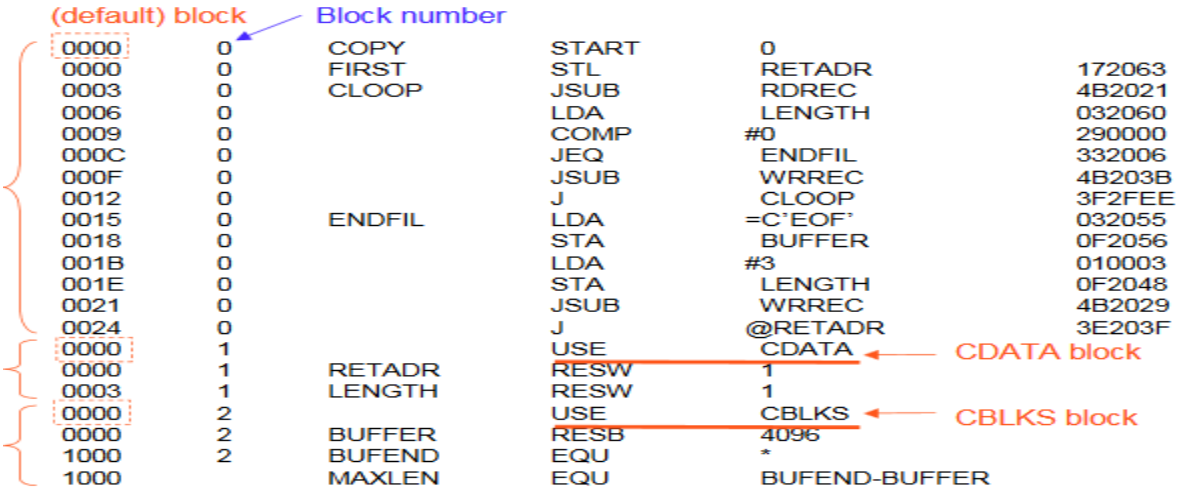
Program blocks Refer to segments of code that are rearranged within a single object program unit. Assembler directive USE is used to denote Program blocks.

- **USE [blockname]**

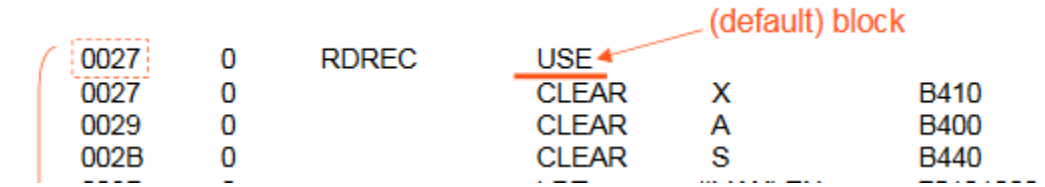
If no USE statements are included, the entire program belongs to this single block. At the beginning, statements are assumed to be part of the unnamed (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program.

- **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**

Assembler rearranges 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.



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	

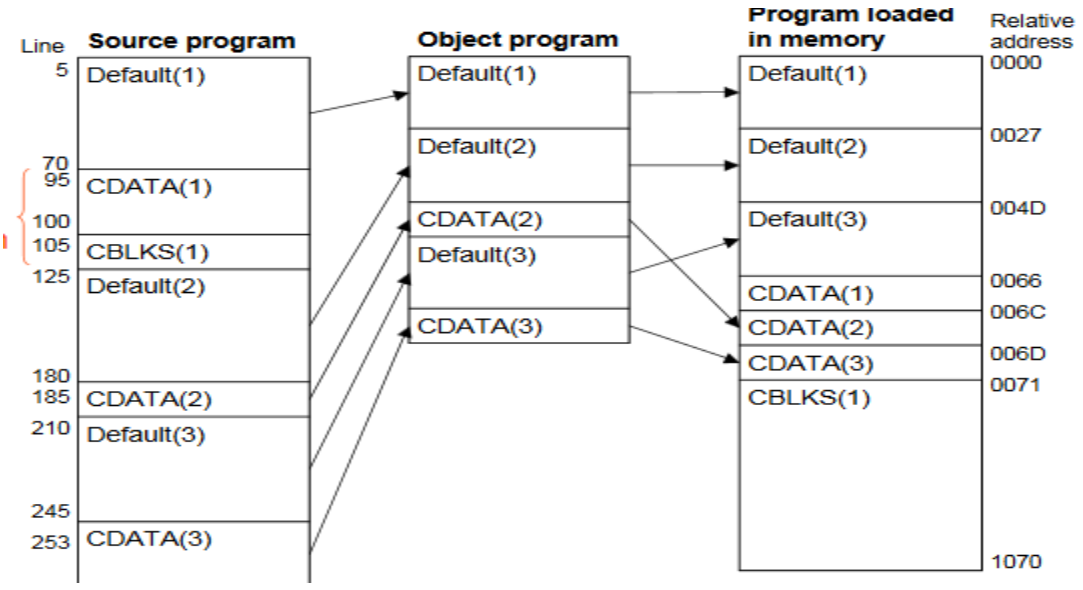


0027	0	RDREC	USE		
0027	0		CLEAR	X	B410
0029	0		CLEAR	A	B400
002B	0		CLEAR	S	B440

- Pass 1
  - A separate location counter for each program block
    - 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

Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

- 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



It is not necessary to physically rearrange the generated code in the object program . The assembler just simply insert the proper load address in each Text record.The loader will load these codes into correct places

```
HCOPY  000000001071
T0000001E1720634B2021032060290000332006 4B203B3F2FEE0320550F2056010003
T00001E090F20484B20293E203F
T0000271DB410B400B44075101000E32038332FFADB2032A00433200857A02FB850
T000044093B2FEA13201F4F0000
T00006C01F1
T00004D19B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000
T00006D04454F4605
E000000
```

5. Control sections

It can be loaded and relocated independently of the others are most often used for subroutines or other logical subdivisions of a program .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. Assembler Directive is **CSECT**

secname CSECT

secname is the control section name. Instructions in one control section may need to refer to instructions or data located in another section.

1.External definition

EXTDEF name [, name]

EXTDEF names symbols that are defined in this control section and may be used by other sections

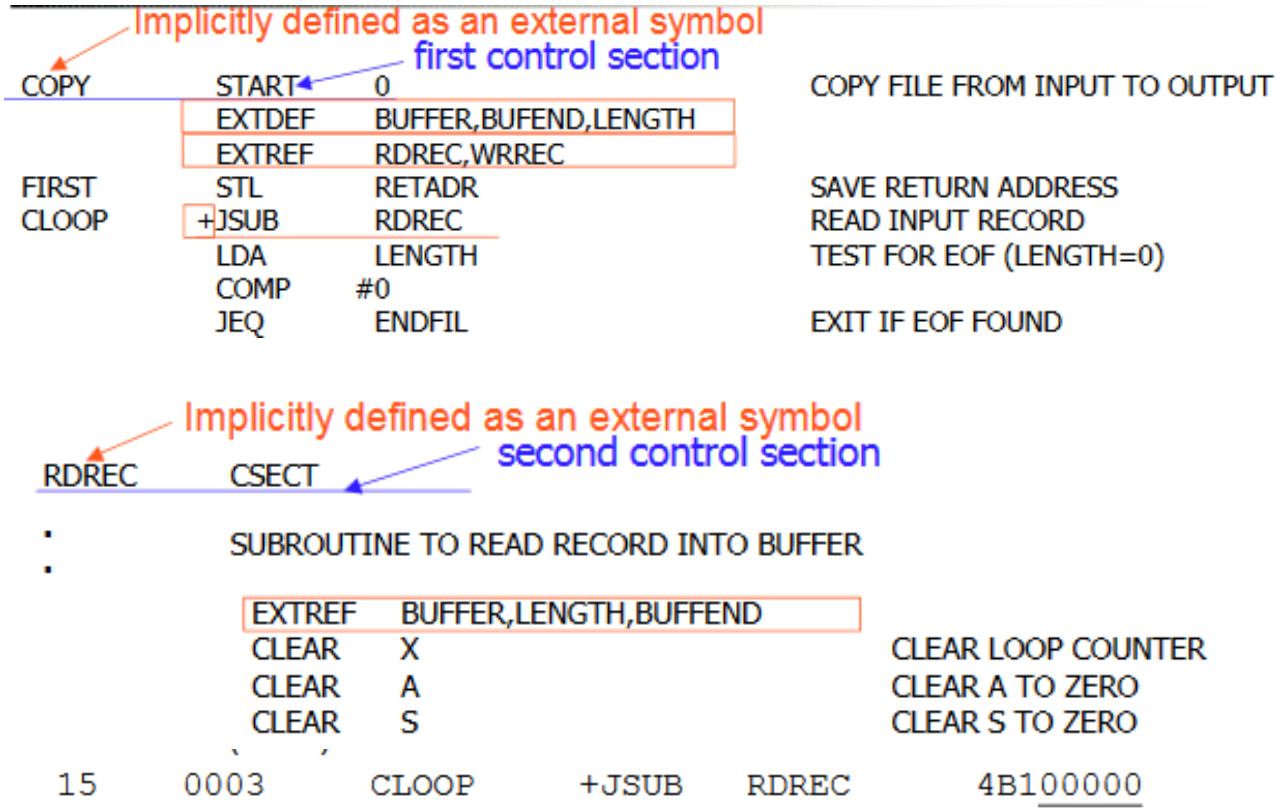
Ex: EXTDEF BUFFER, BUFEND, LENGTH

2. External reference

EXTREF name [,name]

EXTREF names symbols that are used in this control section and are defined elsewhere

Ex: EXTREF RDREC, WRREC



- 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.

The assembler must include information in the object program that will cause the loader to insert proper values where they are required .The assembler communicate to loader through 3 types of records.

1. **Define record**

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

2. **Refer record**
   - Col. 1 D
   - 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 (hexiadecimal)
   - Col. 8-9 Length of the field to be modified, in half-bytes (hexadeccimal)
   - Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field
   - Note: control section name is automatically an external symbol, i.e. it is available for use in Modification records.

Example

- M00000405+RDREC
- M00000705+COPY

```
COPY
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC  WRREC
T000000 1D 172027 4B100000 032023 290000 332007 4B1000003F2FEC032016 0F2016
T00001D 0D 0100030F200A 4B1000003E2000
T000030 03 454F46
M000004 05+RDREC
M000011 05+WRREC
M000024 05+WRREC
E000000
```

Assembler Design Options

1. **One-pass assemblers**
2. **Multi-pass assemblers**

1.One-pass assembler

One-pass assemblers are used when

a. it is necessary or desirable to avoid a second pass over the source program
b. the external storage for the intermediate file between two passes is slow or is inconvenient to use

Main problem: forward references to both data and instructions.

- One simple way to eliminate this problem: require that all areas be defined before they are referenced.(will be placing all storage reservation statements before they are referenced.)

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

here all storage reservations are given at the starting .

Types of one-pass assembler

- Type 1: Load-and-go
  - Produces object code directly in memory for immediate execution
- Type 2:
  - Produces usual kind of object code for later execution

Type 1:Load-and-go

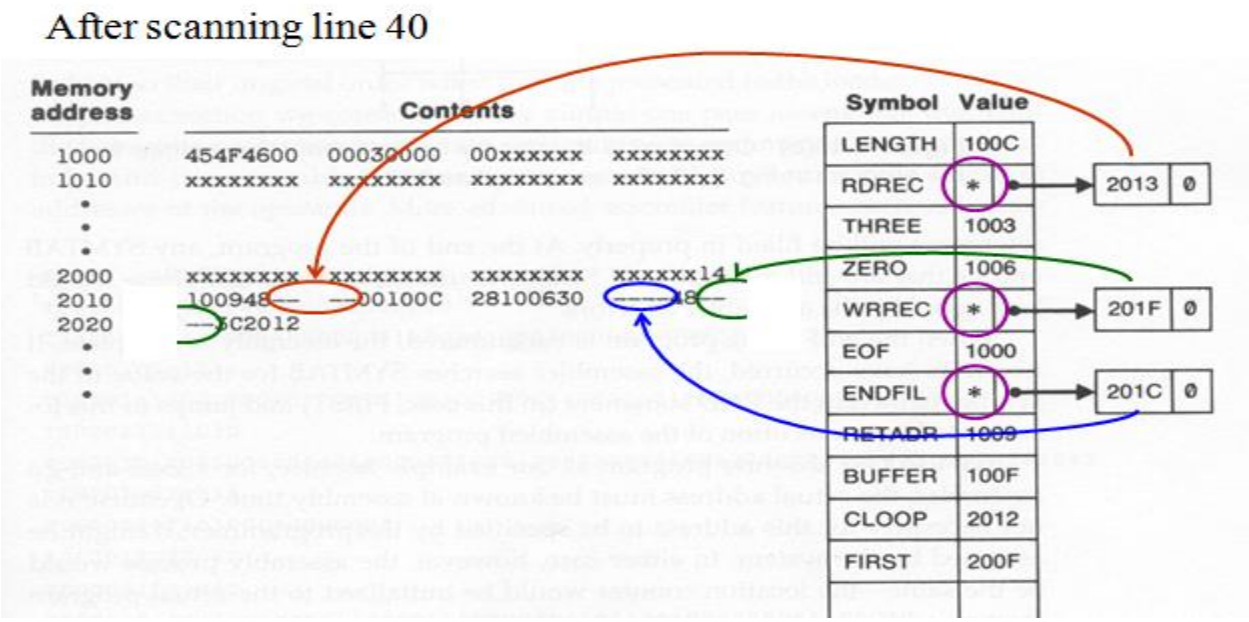
Load-and-go assembler generates their object code in memory for immediate execution.For a load-and-go assembler, the actual address must be known at assembly time.No object program is written out, no loader is needed.

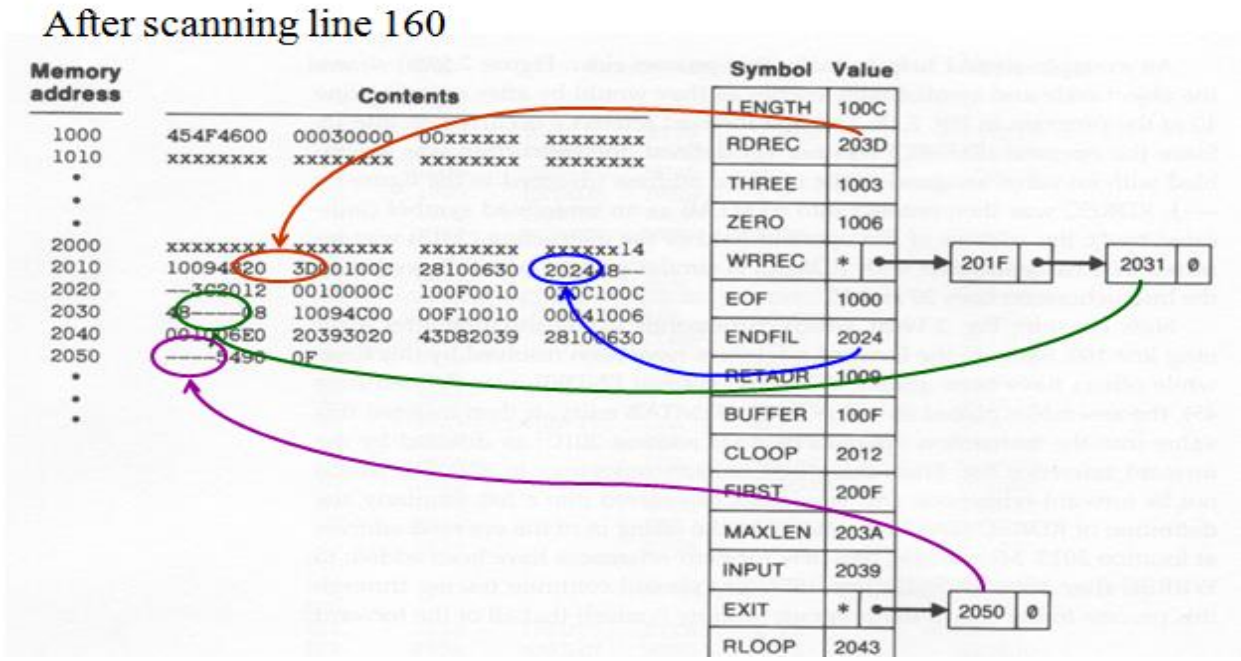
Assembler operations:

- 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
- Scans the reference list and inserts the address when the definition for the symbol is encountered.
- Reports the error if there are still SYMTAB entries indicated undefined symbols at the end of the program
- Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

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	
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	3C2012
45	2024	ENDFIL	LDA	EOF	001000

Figure 2.18





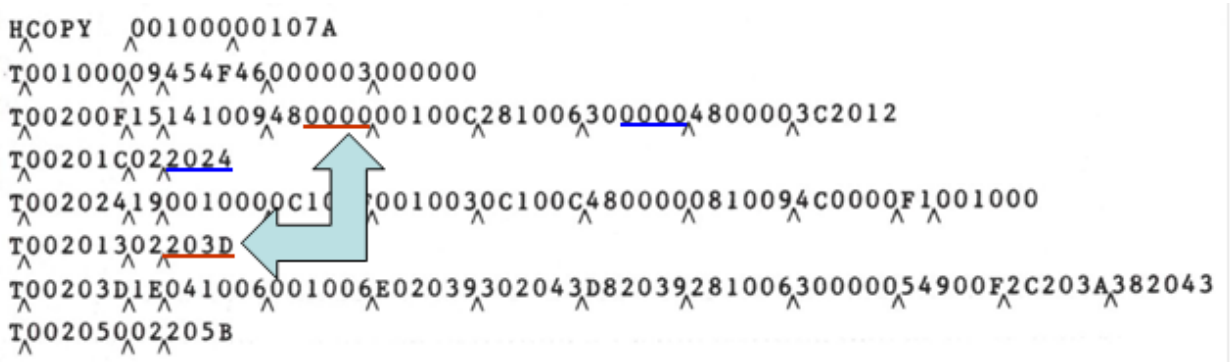
Type 2: Assembler

It will produce object code.

Assembler operations:

Instruction referencing are written into object file as a Text record, even with incorrect addresses.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.



2.Multi-pass assemblers

A multi-pass assembler that can make as many passes as are needed to process the definitions of symbols. Only the portions of the program that involve forward references in symbol definition are saved for multi-pass reading.

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

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

Reason: symbol definition must be completed in pass 1.

Motivation for using a multi-pass assembler

- DELTA can be defined in pass 1
- BETA can be defined in pass 2
- ALPHA can be defined in pass 3

A symbol table is used

- to store symbol definitions that involve forward references
- to indicate which symbols are dependant on the values of others
- to facilitate symbol evaluation

For a forward reference in symbol definition, we store in the SYMTAB:

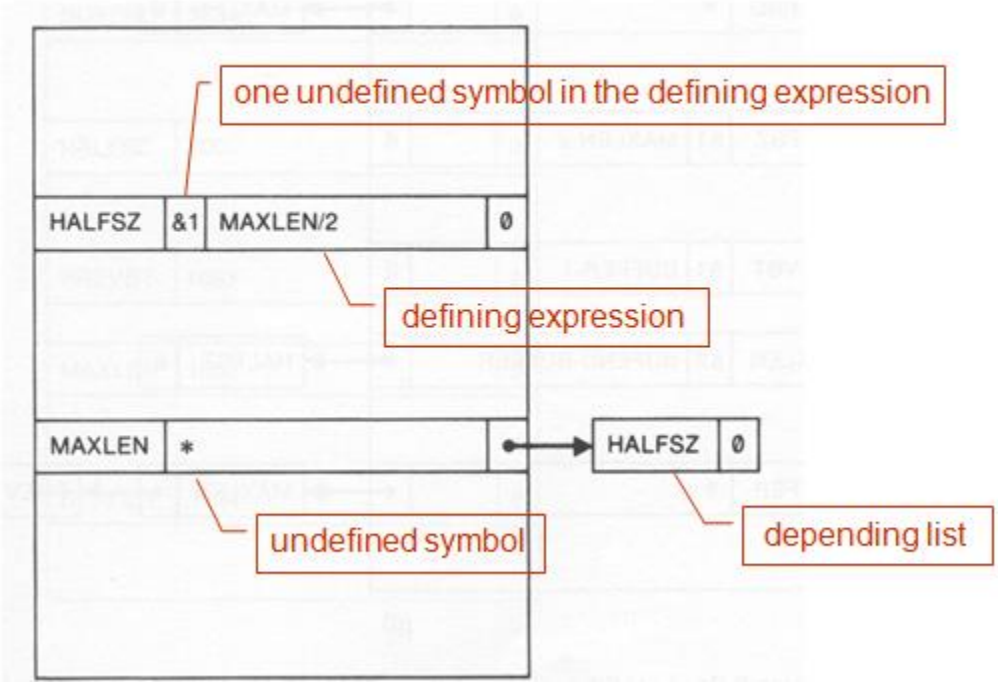
- the symbol name
- the defining expression
- the number of undefined symbols in the defining expression
- the undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.

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

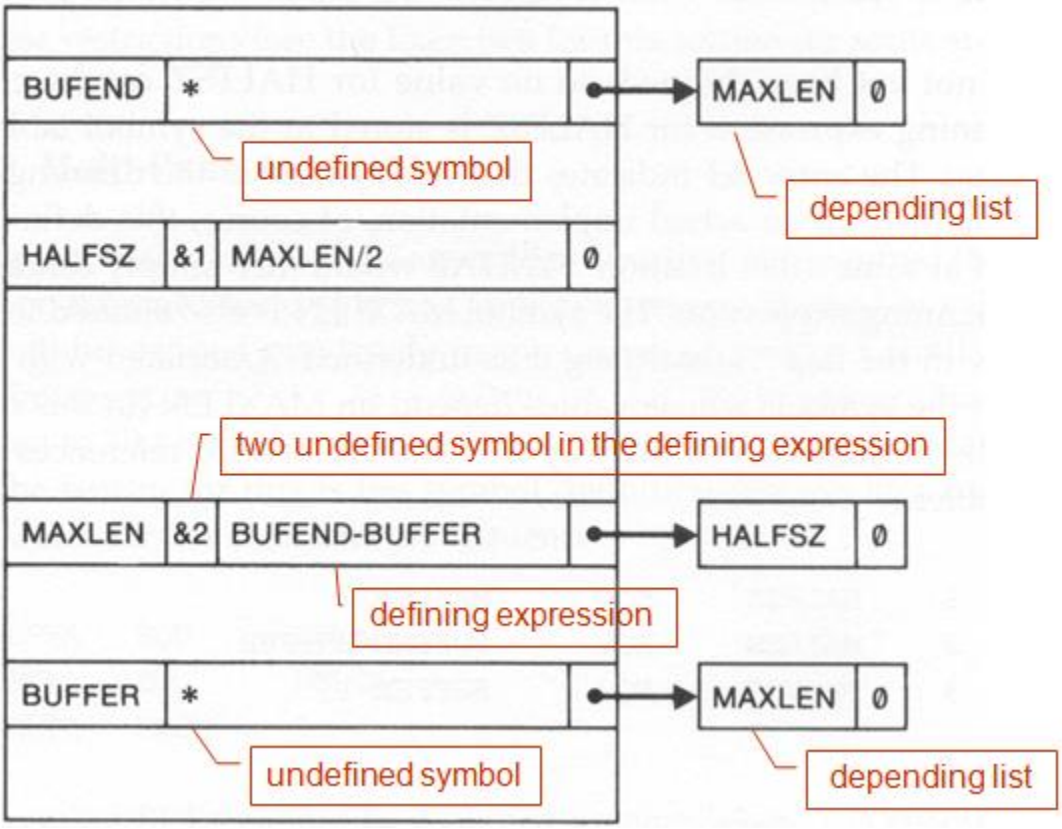
```
1     HALFSZ    EQU     MAXLEN/2
2     MAXLEN    EQU     BUFEND-BUFFER
3     PREVBT    EQU     BUFFER-1
                        .
                        .
                        .
4     BUFFER    RESB    4096
5     BUFEND    EQU     *
```

After Pass 1

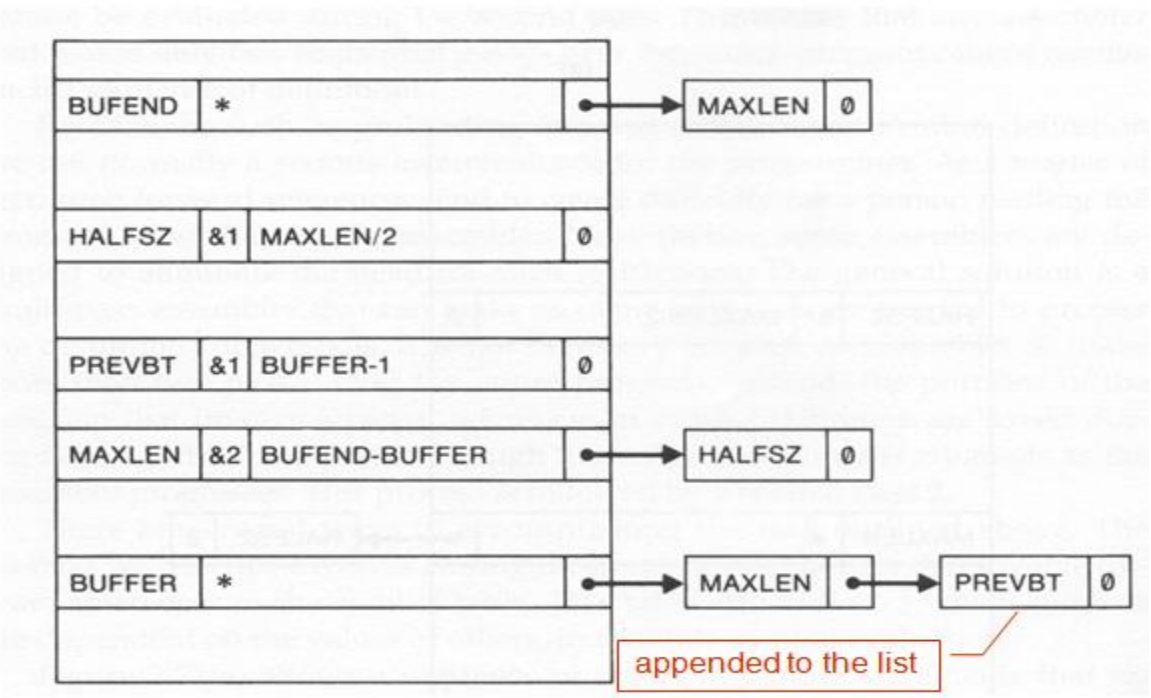
```
1     HALFSZ    EQU     MAXLEN/2
```



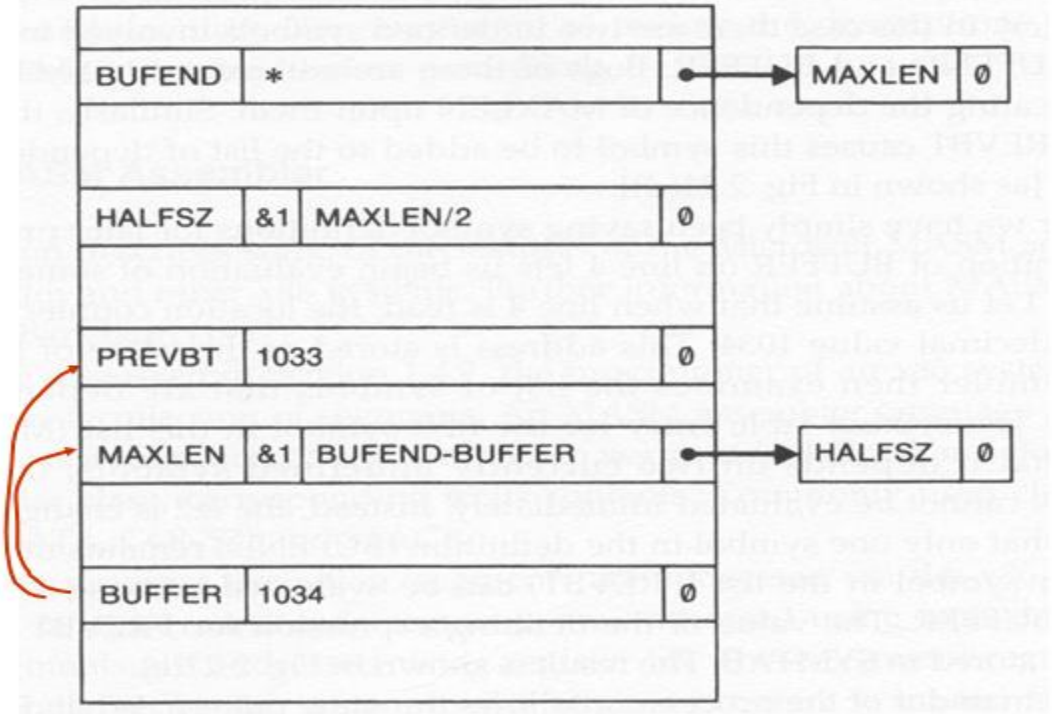
```
2     MAXLEN    EQU    BUFEND-BUFFER
```

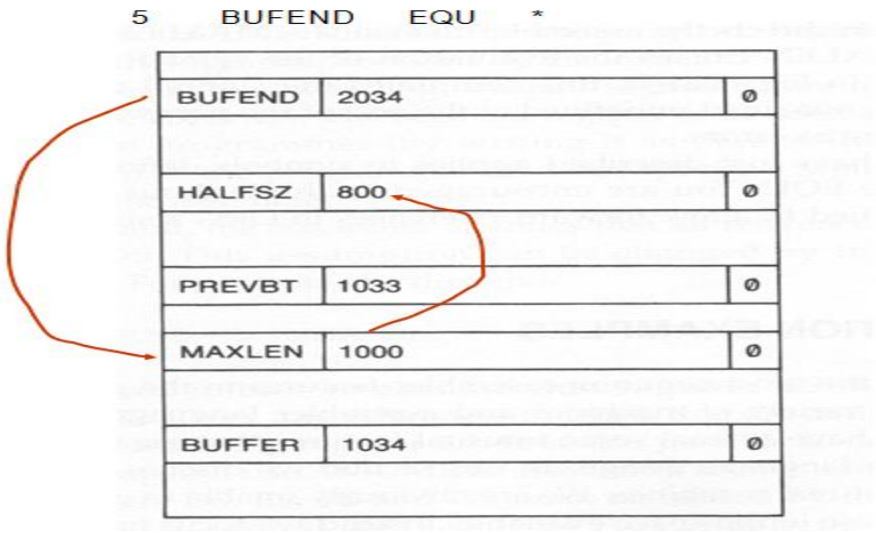


```
3       PREVBT    EQU     BUFFER-1
```




```
4       BUFFER    RESB    4096
```

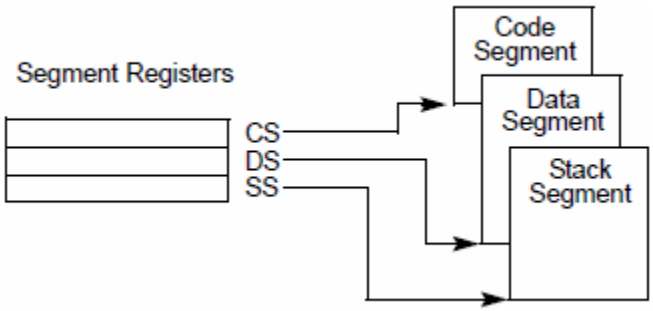




Implementation Examples: MASM Assembler

The Microsoft Macro **Assembler** (**MASM**) is an **assembler** for the x86 family of microprocessors, originally produced Microsoft MS-DOS operating **system.** The 8086 family of processors have a set of 16-bit registers. They are AX,BX,CX,DX (General purpose registers) segment registers like ,SS.CS.DS.ES and pointer registers like SP,BP and Index registers like DI and SI etc…**MASM** assembler language program is written as a collection of segments. Each segment is defined as belonging to a particular class .Common segments are

- **CODE Segment**
- **DATA Segment**
- **CONST Segment**
- **STACK Segment**



Every program written only in MASM has **one** main module, where program execution begins. Main module can contain code, data, or stack segments defined with all of the simplified segment directives. Any additional modules should contain only code and data segments. Every module that uses simplified segments must begin with the .MODEL directive. When you specify a segment in your program, not only must you tell the CPU that a segment is a data segment, but you must also tell the assembler where and when that segment is a data (or code/stack/extra/F/G) segment. The assume directive provides this information to the assembler.

The **assume** directive takes the following form:

ASSUME ES:DATASEG2

It tells the assembler to assume that register ES indicates the segment DATASEG2. ASSUME tells MASM the contents of a segment register, the programmer must provide instructions to load this register when the program is executed.

ASSUME DS : DATA

This tells the assembler that for any program instruction which refers to the data segment ,it should use the logical segment called DATA.

ENDS-End Segment:

This directive is used with the name of the segment to indicate the end of that logical segment.

Ex: **CODE SEGMENT** : Start of logical segment containing code
CODE ENDS : End of the segment named CODE.

Near jump

A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.Near jump occupies 2 or 3 bytes.

Short jump

A near jump where the jump range is limited to -128 to +127 from the current EIP value.

Far jump

A jump to an instruction located in a different segment than the current code segment, sometimes referred to as an intersegment jump. Far jump occupies 5 bytes.

Consider a jump instruction

JMP TARGET

If the definition of the label TARGET occurs in the program before the JMP instruction, the assembler can tell whether this is a near jump or a far jump, If this is a forward reference to TARGET the assembler does not know how many bytes to reserve for instruction. By default MASM assumes that a forward jump is a near jump.If the jump address is within 128 bytes of the current instruction, the programmer can specify the shorter (2 byte)near jump by writing

JMP SHORT TARGET

If JMP to TARGET is a far jump and the programmer does not specify FAR PTR a problem occurs.Segments in an MASM source program can be written in more than one part .If a SEGMENT directive specifies the same name as a previously defined segment it is considered to be a continuation of that segment.External References and External Definition in MASM is handled using the directive **EXTRN** and **PUBLIC** .PUBLIC has same function as EXTDEF in SIC/XE and EXTRN has same function as EXTREF in SIC/XE.

Module 4- Loaders

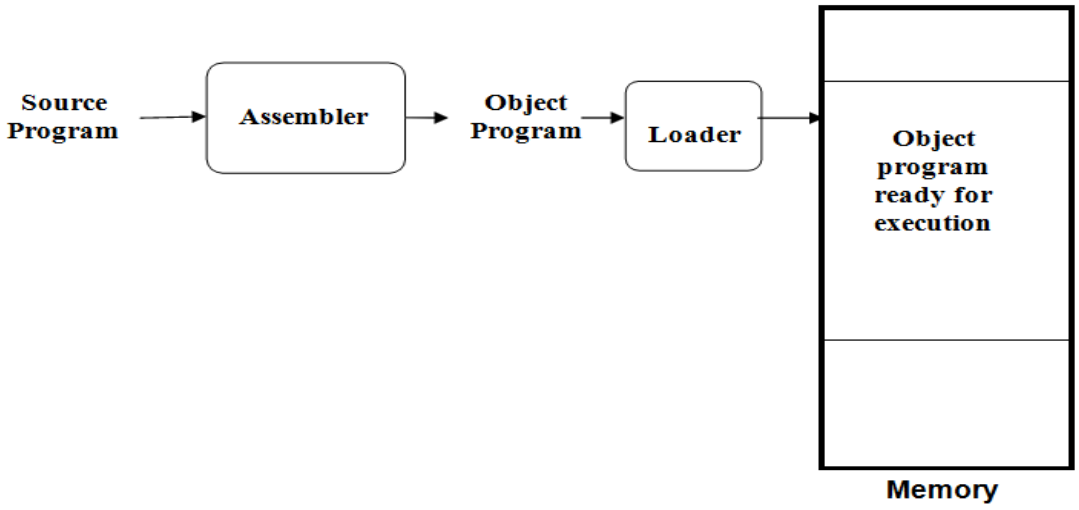
The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies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



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 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 ≠ '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
```

```
HCOPY  001000001007A
T001000 1E 141033482039001036281030301015482061 3C1003 00102A 0C1039 00102D
T001013 1E 150C10364820610810334C0000454F46000003000000
T002039 1E 041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T002057 1C 1010364C0000F1001000041030E02079302064509039DC20792C1036
T002073 07 3820644C000005
E001000
```

(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	← COPY
⋮	⋮	⋮	⋮	⋮	
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	
⋮	⋮	⋮	⋮	⋮	

Machine-Dependent Loader Features

1. **Relocation**
2. **Program Linking**
3. **Algorithm and Data structures**

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

1. **Relocation**

The concept of program relocation is, the execution of the object program using

any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

Methods for specifying relocation

Two methods for specifying relocation as part of the object program.

Modification Record(for SIC/XE)

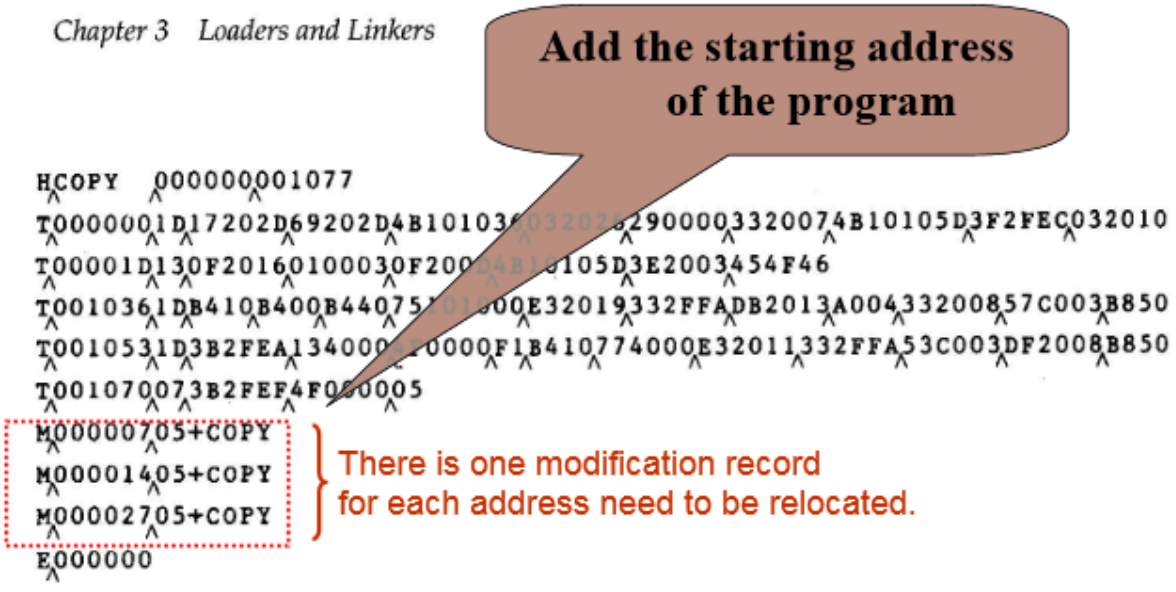
Relocation Bit(for SIC)

1. **Modification Record(for SIC/XE)**

In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks. Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record
col 1: M
col 2-7: relocation address
col 8-9: length (halfbyte)
col 10: flag (+/-)
col 11-17: segment name

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003



The Modification record is not well suited for use with all machine architectures. **Too many modification records will be needed in certain situations.(usually in SIC machine instructions) Hence we use relocation bit mask.**

2. Relocation Bit(for SIC)

The relocation bit method is used for simple machines. This is specified in the columns 10 -12 of text record (T), the format of text record, along with relocation bits is as follows

Relocation bit is either 0 nor 1

0: no modification is necessary

1: modification is needed

Twelve-bit mask is used in each Text record since each text record contains less than 12 words unused words are set to 0

Text record

- col 1: T
- col 2-7: starting address
- col 8-9: length (byte)
- col 10-12: relocation bits
- col 13-72: object code

```
HCOPY  000000001 07A
T0000001EFFC140033481039000036280030300015481061 3C000300002A0C003900002D
T00001E15E000C003648106108003 34C0000454F460000003000000
T001039 1EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

2.Program Linking

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: **EXTDEF** BUFFER, BUFFEND, LENGTH
EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: **EXTREF** RDREC, WRREC
EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of **Define record (D) and, Refer record(R).**

Define record

The format of the Define record (D) along with examples is as shown here**.**

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

Example records

D LISTA 000040 ENDA 000054
D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here**.**

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

Example records

R LISTB ENDB LISTC ENDC R
LISTA ENDA LISTC ENDC R LISTA
ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation.

```
0000  PROGA    START   0
              EXTDEF  LISTA, ENDA
              EXTREF  LISTB, ENDB, LISTC, ENDC
              ……..
              …….
0020  REF1    LDA     LISTA                    03201D
0023  REF2    +LDT    LISTB+4                  77100004
0027  REF3    LDX     #ENDA-LISTA              050014
              ..
0040  LISTA   EQU     *
0054  ENDA    EQU     *
0054  REF4    WORD            ENDA-LISTA+LISTC    000014
0057  REF5    WORD            ENDC-LISTC-10       FFFFF6
005A  REF6    WORD            ENDC-LISTC+LISTA-1  00003F
005D  REF7    WORD            ENDA-LISTA-(ENDB-LISTB) 000014
0060  REF8    WORD            LISTB-LISTA         FFFFC0
              END     REF1
```

CONTROL SECTION- PROGB

0000	**PROGB**	START	0		**External definition and References**
		EXTDEF	LISTB, ENDB		
		EXTREF	LISTA, ENDA, LISTC, ENDC		
		……..			
		…….			
0036	REF1	+LDA	LISTA		03100000
003A	REF2	LDT	LISTB+4		772027
003D	REF3	+LDX	#ENDA-LISTA		05100000
		.			
		.			
0060	LISTB	EQU	*		
0070	ENDB	EQU	*		
0070	REF4	WORD	ENDA-LISTA+LISTC		000000
0073	REF5	WORD	ENDC-LISTC-10		FFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1		FFFFFF
0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)		FFFFF0
007C	REF8	WORD	LISTB-LISTA		000060
		END			

0000	**PROGC**	START	0		**External definition and References**
		EXTDEF	LISTC, ENDC		
		EXTREF	LISTA, ENDA, LISTB, ENDB		
		……..			

```
                    ……..
0018  REF1    +LDA    LISTA                          03100000
001C  REF2    +LDT    LISTB+4                        77100004
0020  REF3    +LDX    #ENDA-LISTA                    05100000
                .
                .
0030  LISTC   EQU    *
0042  ENDC    EQU    *
0042  REF4    WORD          ENDA-LISTA+LISTC         000030
0045  REF5    WORD          ENDC-LISTC-10            000008
0045  REF6    WORD          ENDC-LISTC+LISTA-1       000011
004B  REF7    WORD          ENDA-LISTA-(ENDB-LISTB)  000000
004E  REF8    WORD          LISTB-LISTA              000000
                END
```

```
H PROGA 000000 000063
D LISTA   000040 ENDA   000054
R LISTB   ENDB   LISTC   ENDC
.
.
T 000020 0A 03201D 77100004 050014
.
.
T 000054 0F 000014 FFFF6 00003F 000014 FFFFC0
M000024 05+LISTB
M000054 06+LISTC
M000057 06+ENDC
M000057 06 -LISTC
M00005A06+ENDC
M00005A06 -LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

```
H PROGB 000000 00007F
D LISTB   000060 ENDB   000070
R LISTA   ENDA   LISTC   ENDC
.
T 000036 0B 03100000 772027 05100000
.
T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05+LISTA
M00003E 06+ENDA
M00003E 06 -LISTA
M000070 06 +ENDA
M000070 06 -LISTA
M000070 06 +LISTC
M000073 06 +ENDC
M000073 06 -LISTC
M000073 06 +ENDC
M000076 06 -LISTC
M000076 06+LISTA
M000079 06+ENDA
```

```
H PROGC 000000 000051
D LISTC   000030 ENDC   000042
R LISTA   ENDA  LISTB  ENDB
.
T 000018 0C 03100000 77100004 05100000
.
T 000042 0F 000030 000008 000011 000000 000000
M000019 05+LISTA
M00001D 06+LISTB
M000021 06+ENDA
M000021 06 -LISTA
M000042 06+ENDA
M000042 06 -LISTA
M000042 06+PROGC
M000048 06+LISTA
M00004B 06+ENDA
M00004B 006-LISTA
M00004B 06-ENDB
M00004B 06+LISTB
M00004E 06+LISTB
M00004E 06-LISTA
```

Two Passes Logic

Pass 1: assign addresses to all external symbols

Pass 2: perform the actual loading, relocation, and linking

Consider the case in which the program is loaded in the memory location 4000 .Then the control section PROGA is loaded in 4000 and PROGB is loaded at 4063(4000+ 63(length of program A))

Load address for control sections

- » PROGA 004000 63
- » PROGB 004063 7F
- » PROGC 0040E2 51

Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB	4000+0063=	4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC	4063+007F=	40E2	0051
	LISTC	4112	
	ENDC	4124	

Load address for symbols

- LISTA: PROGA+0040=4040
- LISTB: PROGB+0060=40C3
- LISTC: PROGC+0030=4112

REF4 in PROGA

- ENDA-LISTA+LISTC=14+4112=4126
- T0000540F000014FFFFF600003F000014FFFFC0
- M00005406+LISTC
-

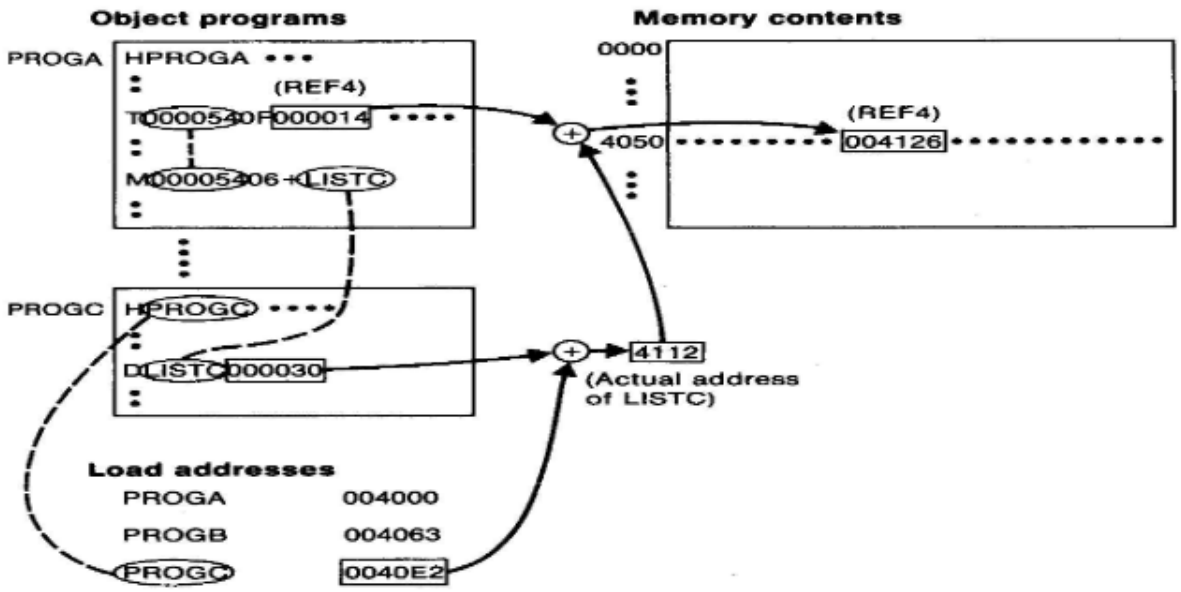


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

After these control sections are linked, relocated, and loaded

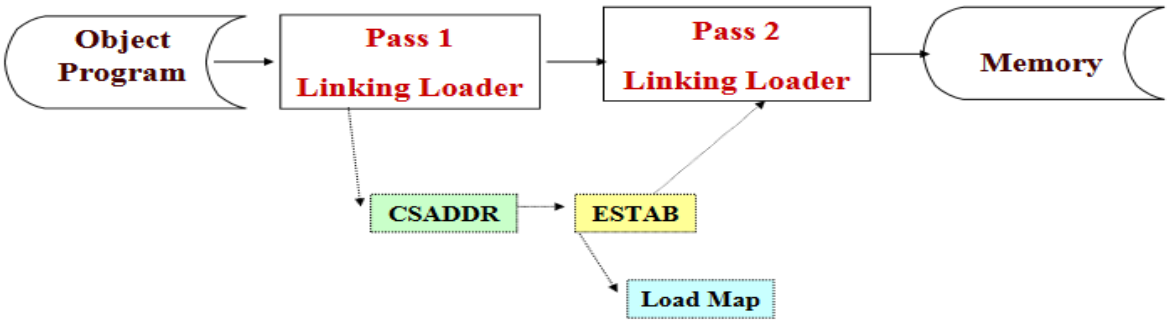
3. Algorithm and Data structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program, which is already given. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism. Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader. Input is a set of object programs, i.e., control sections. A linking loader usually makes two passes over its input, just as an assembler does

Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.



Data Structures

- External Symbol Table (ESTAB)
  - For each external symbol, ESTAB stores
    - its ***name***
    - its ***address***
    - in which ***control section*** the symbol is defined
  - *Hashed organization*
- Program Load Address (PROGADDR)
  - PROGADDR is the ***beginning address*** in memory where the ***linked program*** is to be loaded (supplied by ***OS***).
- Control Section Address (CSADDR)
  - CSADDR is the ***starting address*** assigned to the ***control section*** currently being scanned by the ***loader***.
- Control section length (CSLTH)

Control section	Symbol	Address	Length
Progam A		4000	63
	LISTA	4040	
	ENDA	4054	
Program B		4063	7F
	LISTB	40C3	
	ENDB	40D3	
Program C		40E2	51
	LISTC	4112	
	ENDC	4124	

Pass 1 Program Logic

Assign addresses to all external symbols

- Loader is concerned only with Header and Define records in the control sections

To build up ESTAB

- Add control section name into ESTAB
- Add all external symbols in the Define record into ESTAB

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
   begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag {duplicate external symbol}
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end {for}
         end {while ≠ 'E'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Pass 2 Program Logic

- Perform the actual loading, relocation, and linking
- When Text record is encountered read into the specified address (+CSADDR)
- When Modification record is encountered

  Lookup the symbol in ESTAB

  This value is then added to or subtracted from the indicated location in memory
- When the End record is encountered

  Transfer control to the loaded program to begin execution

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record {Header record}
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end  {if 'M'}
         end {while ≠ 'E'}
      if an address is specified {in End record} then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

A Simple Bootstrap Loader

The bootstrap itself begins at address 0 in the memory of the machine

It loads the OS (or some other program) starting address 0x80

- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.

After all the object code from device F1 has been loaded, the bootstraps jumps to address 80

- Begin the execution of the program that was loaded.

```
BOOT     START     0        BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
         CLEAR     A        CLEAR REGISTER A TO ZERO
         LDX       #128     INITIALIZE REGISTER X TO HEX 80
LOOP     JSUB      GETC     READ HEX DIGIT FROM PROGRAM BEING LOADE
         RMO       A,S      SAVE IN REGISTER S
         SHIFTL    S,4      MOVE TO HIGH-ORDER 4 BITS OF BYTE
         JSUB      GETC     GET NEXT HEX DIGIT
         ADDR      S,A      COMBINE DIGITS TO FORM ONE BYTE
         STCH      0,X      STORE AT ADDRESS IN REGISTER X
         TIXR      X,X      ADD 1 TO MEMORY ADDRESS BEING LOADED
         J         LOOP     LOOP UNTIL END OF INPUT IS REACHED
```

```
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC    TD      INPUT   TEST INPUT DEVICE
        JEQ     GETC    LOOP UNTIL READY
        RD      INPUT   READ CHARACTER
        COMP    #4      IF CHARACTER IS HEX 04 (END OF FILE),
        JEQ     80          JUMP TO START OF PROGRAM JUST LOADED
        COMP    #48     COMPARE TO HEX 30 (CHARACTER '0')
        JLT     GETC    SKIP CHARACTERS LESS THAN '0'
        SUB     #48     SUBTRACT HEX 30 FROM ASCII CODE
        COMP    #10     IF RESULT IS LESS THAN 10, CONVERSION IS
        JLT     RETURN      COMPLETE. OTHERWISE, SUBTRACT 7 MORE
        SUB     #7          (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN  RSUB            RETURN TO CALLER
INPUT   BYTE    X'F1'   CODE FOR INPUT DEVICE
        END     LOOP
```

MACHINE-INDEPENDENT LOADER FEATURES

Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler. They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

1. Automatic Library Search

Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. Linking loaders that support *automatic library search* must keep track of external symbols that are referred to, but not defined, in the primary input to the loader. At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.The subroutines fetched from a library in this way may

themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved. If unresolved external references remain after the library search is completed, these must be treated as errors.

2. Loader Options

Many loaders allow the user to specify options that modify the standard processing.

- **Typical loader option 1**: Allows the selection of alternative sources of input.
  Ex : INCLUDE program-name (library-name) might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.
- **Loader option 2:** Allows the user to delete external symbols or entire control sections.
  Ex : DELETE csect-name might instruct the loader to delete the named control section(s) from the set of programs being loaded.

CHANGE name1, name2 might cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

- **Loader option 3:** Involves the automatic inclusion of library routines to satisfy external references.
  Ex. : LIBRARY MYLIB

Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

NOCALL STDDEV, PLOT, CORREL

To instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

LOADER DESIGN OPTIONS

Linking loaders perform all linking and relocation at load time.

- There are two alternatives:

1. **Linkage editors**, which perform linking prior to load time.
2. **Dynamic linking**, in which the linking function is performed at execution time.

- Precondition: The source program is first assembled or compiled, producing an object program.
- A **linking loader** performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.
- A **linkage editor** produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

Linkage Editors

The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required.If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required. Linkage editors can perform many useful functions besides simply preparing an object program for execution. Ex., a typical sequence of linkage editor commands used:

INCLUDE PLANNER (PROGLIB)

DELETE PROJECT **{delete from existing PLANNER}**

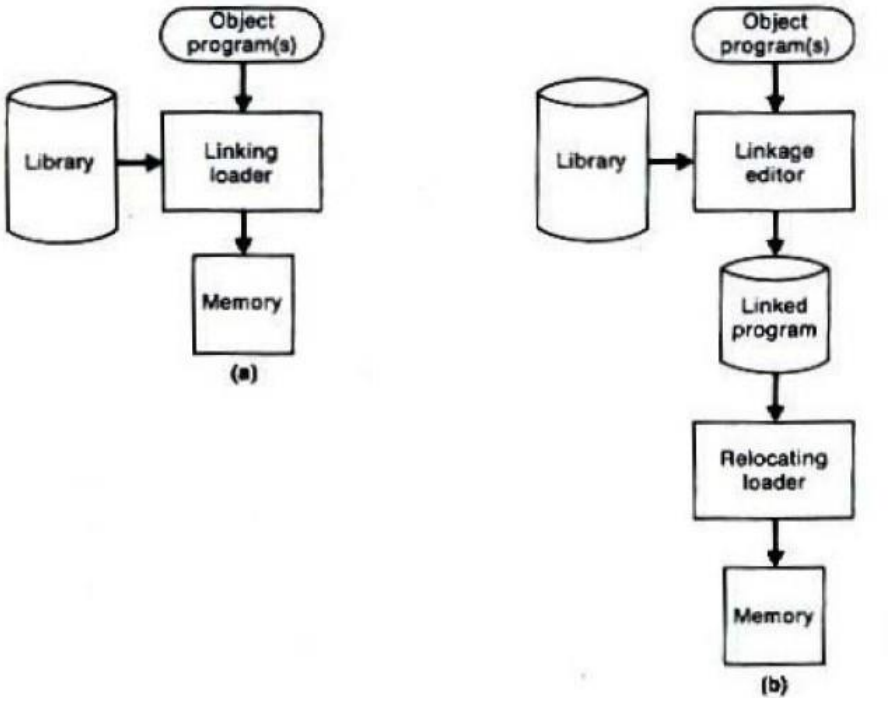
INCLUDE PROJECT (NEWLIB) {**include new version**}

REPLACE PLANNER (PROGLIB)

Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages.Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

Processing of an object program using (a) Linking loader and (b) Linkage editor

Dynamic Linking



Linkage editors perform linking operations before the program is loaded for execution.Linking loaders perform these same operations at load time.Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C.With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines

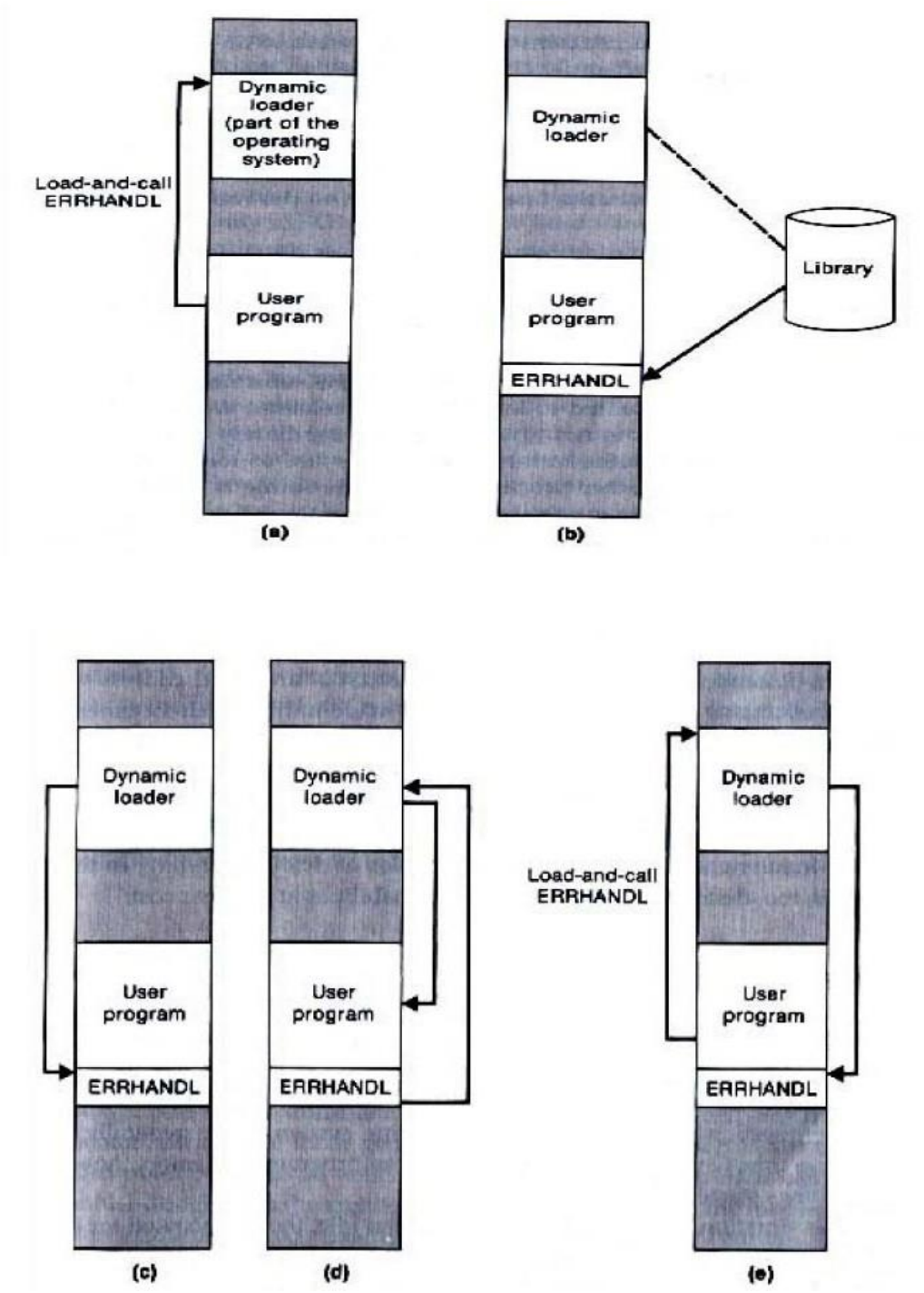


Fig (a): Instead of executing a JSUB instruction referring to an external symbol, the

program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.

Fig (b): OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.

Fig (c): Control is then passed from OS to the routine being called

Fig (d): When the called subroutine completes it processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

Fig (e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine

Module 5

MACRO PROCESSOR

A *Macro* represents a commonly used group of statements in the source programming language. A macro instruction (macro) is a notational convenience for the programmer It allows the programmer to write shorthand version of a progra (moduleprogramming)

The macro processor replaces each macro instruction with the corresponding group of source language statements (*expanding*) . Normally, it performs no analysis of the text it handles. It does not concern the meaning of the involved statements during macro expansion. The design of a macro processor generally is *machine independent!* ·

Two new assembler directives are used in macro definition

MACRO: identify the beginning of a macro definition

MEND: identify the end of a macro definition

Prototype for the macro each parameter begins with '&'

macroname MACRO parameters
:
body (the statements that will be generated as the expansion of the macr).
:
MEND

Basic Macro Processor Functions:

1. Macro Definition and Expansion
2. Macro Processor Algorithms and Data structures

Macro Definition and Expansion:

In Figure the left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction. M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the

parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

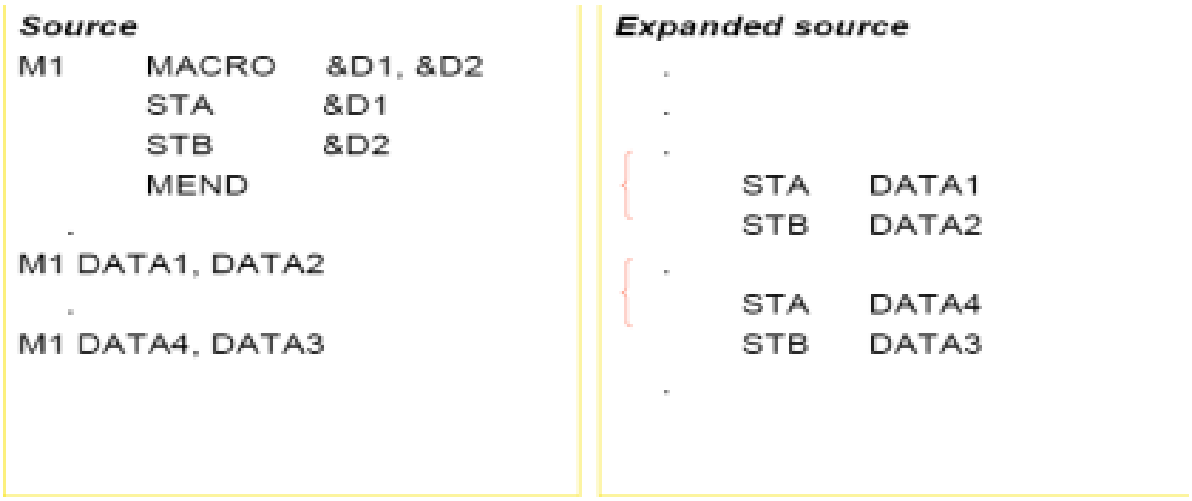


Fig 6.1: macro call

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

2 **. *Macro Expansion*:**

The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

After *macro processing* the expanded file can become the input for the *Assembler*. The *Macro Invocation* statement is considered as comments and the statement generated

from expansion is treated exactly as though they had been written directly by the programmer. The difference between *Macros* and *Subroutines* is that the statement s from the body of the Macro is expanded the number of times the macro invocation is encountered, whereas the statement of the subroutine appears only once no matter how many times the subroutine is called. Macro instructions will be written so that the body of the macro contains no labels.

Problem of the label in the body of macro:

If the same macro is expanded multiple times at different places in the program …

There will be *duplicate labels*, which will be treated as errors by the assembler.

Solutions:

Do not use labels in the body of macro.

Explicitly use PC-relative addressing instead.

The following program shows the concept of Macro Invocation and Macro Expansion.

```
170  .                    MAIN PROGRAM
175  .
180         FIRST   STL     RETADR          SAVE RETURN ADDRESS
190         CLOOP   RDBUFF  F1,BUFFER,LENGTH READ RECORD INTO BUFFER
195                 LDA     LENGTH          TEST FOR END OF FILE
200                 COMP    #0
205                 JEQ     ENDFIL          EXIT IF EOF FOUND
210                 WRBUFF  05,BUFFER,LENGTH WRITE OUTPUT RECORD
215                 J       CLOOP           LOOP
220         ENDFIL  WRBUFF  05,EOF,THREE    INSERT EOF MARKER
225                 J       @RETADR
230         EOF     BYTE    C'EOF'
235         THREE   WORD    3
240         RETADR  RESW    1
245         LENGTH  RESW    1               LENGTH OF RECORD
250         BUFFER  RESB    4096            4096-BYTE BUFFER AREA
255                 END     FIRST
```

```
5        COPY     START   0                 COPY FILE FROM INPUT TO OUTPUT
180      FIRST    STL     RETADR            SAVE RETURN ADDRESS
190      .CLOOP   RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
190a     CLOOP    CLEAR   X                 CLEAR LOOP COUNTER
190b              CLEAR   A
190c              CLEAR   S
190d             +LDT     #4096             SET MAXIMUN RECORD LENGTH
190e              TD      =X'F1'            TEST INPUT DEVICE
190f              JEQ     *-3               LOOP UNTIL READY
190g              RD      =X'F1'            TEST FOR END OF RECORD
190h              COMPR   A, S              TEST FOR END OF RECORD
190i              JEQ     *+11              EXIT LOOP IF EOR
190j              STCH    BUFFER, X         STORE CHARACTER IN BUFFER
190k              TIXR    T                 LOOP UNLESS MAXIMUN LENGTH
190l              JLT     *-19              HAS BEEN REACHED
190M              STX     LENGTH            SAVE RECORD LENGTH
```

Fig 6.2: concept of Macro Invocation and Macro Expansion.

Macro Processor Algorithm and Data Structure:

Design can be done as two-pass or a one-pass macro. In case of two-pass assembler.

Two-pass macro processor

· You may design a two-pass macro processor

Pass 1: Process all macro definitions

Pass 2: Expand all macro invocation statements

However, one-pass may be enough because all macros would have to be defined during the first pass before any macro invocations were expanded. The definition of a macro must appear before any statements that invoke that macro.

Moreover, the body of one macro can contain definitions of the other macro · Consider the example of a Macro defining another Macro.

In the below given example the body of the first Macro (MACROS) contains statement that define RDBUFF, WRBUFF and other macro instructions for SIC machine. · The body of the second Macro (MACROX) defines the se same macros for SIC/XE machine.

· A proper invocation would make the same program to perform macro invocation to run on either SIC or SIC/XEmachine.

MACROS for SIC machine

```
1        MACROS   MACOR      {Defines SIC standard version macros}
2        RDBUFF   MACRO      &INDEV,&BUFADR,&RECLTH
                  .
                  .          {SIC standard version}
                  .
3                  MEND      {End of RDBUFF}
4        WRBUFF   MACRO      &OUTDEV,&BUFADR,&RECLTH
                  .
                  .          {SIC standard version}
5                  MEND      {End of WRBUFF}
                  .
                  .
                  .
6                  MEND      {End of MACROS}
```

Fig 4.3(a)

MACROX for SIC/XE Machine

```
1        MACROX   MACRO      {Defines SIC/XE macros}
2        RDBUFF   MACRO      &INDEV,&BUFADR,&RECLTH
                  .
                  .          {SIC/XE version}
                  .
3                  MEND      {End of RDBUFF}
4        WRBUFF   MACRO      &OUTDEV,&BUFADR,&RECLTH
                  .
                  .          {SIC/XE version}
                  .
5                  MEND      {End of WRBUFF}
                  .
                  .
6                  MEND      {End of MACROX}
```

Fig 4.3(b)

A program that is to be run on SIC system could invoke MACROS whereas a program to be run on SIC/XE can invoke MACROX. However, defining MACROS or MACROX does not define RDBUFF and WRBUFF.These definitions are processed only when an invocation of MACROS or MACROX is expanded.

One-Pass Macro Processor: A one-pass macro processor that alternate between *macro definition* and *macro expansion* in a recursive way is able to handle recursive macro definition.

Restriction -The definition of a macro must appear in the source program before any statements that invoke that macro. This restriction does not create any real inconvenience.The design considered is for one-pass assembler.

The data structures required are:

1. DEFTAB (Definition Table) -Stores the macro definition including macro prototype and macro body o Comment lines are omitted. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
2. NAMTAB (Name Table) - Stores macro names and Serves as an index to DEFTAB . Pointers to the beginning and the end of the macro definition (DEFTAB)
3. ARGTAB (Argument Table) -Stores the arguments according to their positions in the argument list. o As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.

   The figure below shows the different data structures described and their relationship.

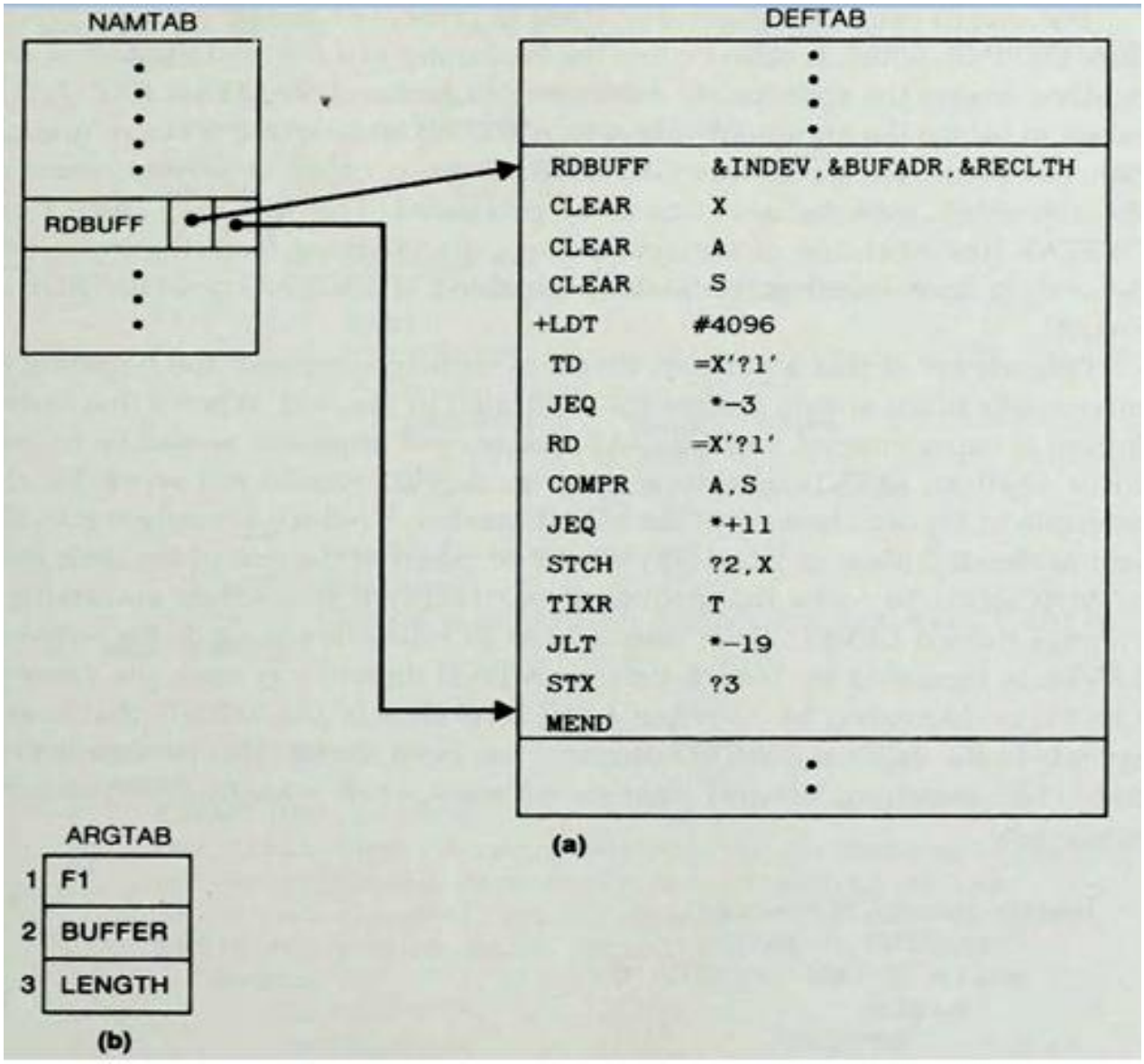


Fig 6.4: data structures and their relationship.

The above figure shows the portion of the contents of the table during the processing of the program in page no. 3. In fig 4.4(a) definition of RDBUFF is stored in DEFTAB, with an entry in NAMTAB having the pointers to the beginning and the end of the definition. The arguments referred by the instructions are denoted by their positional notations.

For example,

TD =X'?1'

The above instruction is to test the availability of the device whose number is given by the parameter & INDEV. In the instruction this is replaced by its positional value? 1. Figure 4.4(b) shows the ARTAB as it would appear during expansion of the RDBUFF statement as given below:

CLOOP RDBUFF F1, BUFFER, LENGTH

For the invocation of the macro RDBUFF, the first parameter is F1 (input device code), second is BUFFER (indicating the address where the characters read are stored), and the third is LENGTH (which indicates total length of the record to be read). When the ?n notation is encountered in a line fro DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

The algorithm of the Macro processor is given below. This has the procedure DEFINE to make the entry of *macro name* in the NAMTAB, *Macro Prototype* in DEFTAB. EXPAND is called to set up the argument values in ARGTAB and expand a *Macro Invocation* statement. Procedure GETLINE is called to get the next line to be processed either from the DEFTAB or from the file itself.

When a macro definition is encountered it is entered in the DEFTAB. The normal approach is to continue entering till MEND is encountered. If there is a program having a Macro defined within another Macro.

While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the definition of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL.

Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

Most macro processors allow the definitions of the commonly used instructions to appear in a standard system library, rather than in the source program. This makes the use of macros convenient; definitions are retrieved from the library as they are needed during macro processing.

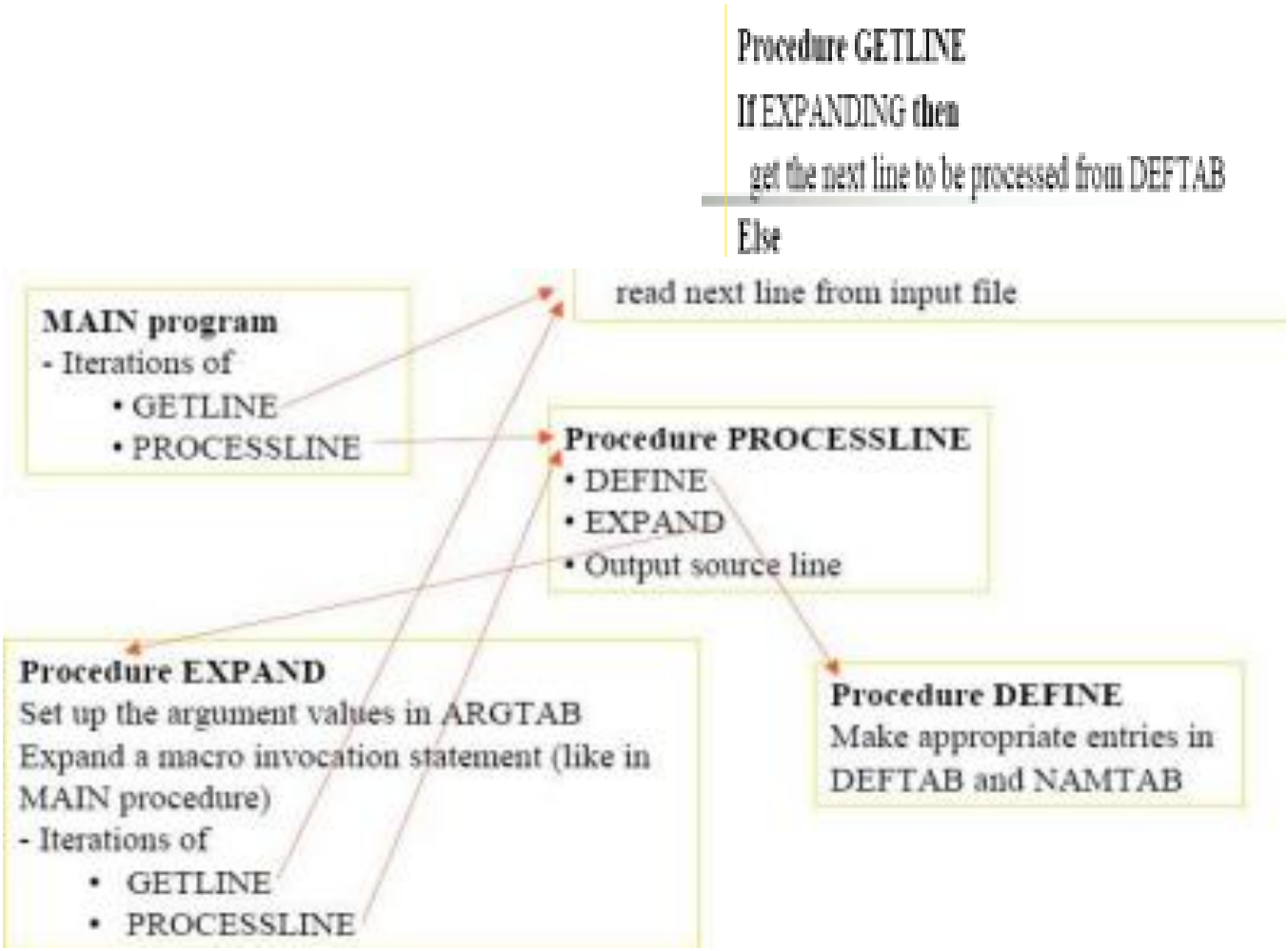


Fig 6.5: Macro library

Algorithms

```
begin {macro processor}
        EXPANDINF := FALSE
        while OPCODE ≠ 'END' do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
end {macro processor}


Procedure PROCESSLINE
        begin
          search MAMTAB for OPCODE
          if found then
                EXPAND
          else if OPCODE = 'MACRO' then
                DEFINE
          else write source line to expanded file
        end {PROCOESSOR}
Procedure DEFINE
        begin
              enter macro name into NAMTAB
              enter macro prototype into DEFTAB
              LEVEL   :- 1
             while LEVEL > do
                begin
                     GETLINE
                     if this is not a comment line then
                       begin
                          substitute positional notation for parameters
                          enter line into DEFTAB
                          if OPCODE = 'MACRO' then
                              LEVEL := LEVEL +1
                          else if OPCODE = 'MEND' then
                              LEVEL := LEVEL – 1
                       end {if not comment}
                end {while}
             store in NAMTAB pointers to beginning and end of definition
        end {DEFINE}
```

```
Procedure EXPAND
    begin
            EXPANDING := TRUE
            get first line of macro definition {prototype} from DEFTAB
            set up arguments from macro invocation in ARGTAB
            while macro invocation to expanded file as a comment
            while not end of macro definition do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
            EXPANDING := FALSE
    end {EXPAND}

Procedure GETLINE
    begin
            if EXPANDING then
               begin
                  get next line of macro definition from DEFTAB
                  substitute arguments from ARGTAB for positional notation
               end {if}
            else
               read next line from input file
     end {GETLINE}
```

Comparison of Macro Processor Design

One-pass algorithm

- Every macro must be defined before it is called
- One-pass processor can alternate between macro definition and macro expansion
- Nested macro definitions are allowed but nested calls are not allowed.

Two-pass algorithm

- Pass1: Recognize macro definitions
- Pass2: Recognize macro calls
- Nested macro definitions are not allowed

Machine-independent Macro-Processor Features.

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor. These features are:

· **Concatenation of Macro Parameters**
· **Generation of unique labels**
· **Conditional Macro Expansion**
· **Keyword Macro Parameters**

Concatenation of unique labels:

Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction.

The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, Xb1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like LDA X&ID1

& is the starting character of the macro instruction; but the end of the parameter is not

marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended. If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.

Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as LDA X&ID→

```
ID123   MACRO   &ID
        LAD     X&ID→1
        ADD     X&ID→2
        STA     X&ID→3
        MEND
```

```
1   SUM MACRO   &ID
2       LDA     X&ID→ 1
3       ADD     X&ID→ 2
4       ADD     X&ID→ 3
5       STA     X&ID→ S
6       MEND
```

```
SUM     A                   SUM     BETA
 ↓                           ↓
LDA     XA1                 LDA     XBEATA1
ADD     XA2                 ADD     XBEATA2
ADD     XA3                 ADD     XBEATA3
STA     XAS                 STA     XBEATAS
```

The above figure shows a macro definition that uses the concatenation operator as previously described. The statement SUM A and SUM BETA shows the invocation statements and the corresponding macro expansion.

Generation of Unique Labels

It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the

assembler. This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion. During macro expansion each $ will be replaced with $XX, where xx is a two character alphanumeric counter of the number of macro instructions expansion. For example, XX = AA, AB, AC…

This allows 1296 macro expansions in a single program.

The following program shows the macro definition with labels to the instruction.

```
25        RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
30                 CLEAR    X                    CLEAR LOOP COUNTER
35                 CLEAR    A
40                 CLEAR    S
45                 +LDT     #4096                SET MAXIMUM RECORD LENGTH
50        $LOOP    TD       =X'&INDEV'           TEST INPUT DEVICE
55                 JEQ      $LOOP                LOOP UNTIL READY
60                 RD       =X'&INDEV'           READ CHARACTER INTI REG A
65                 COMPR    A, S                 TEST FOR END OF RECORD
70                 JEQ      $EXIT                EXIT LOOP IF EOR
75                 STCH     &BUFADR, X           STORE CHARACTER IN BUFFER
80                 TIXR     $LOOP                HAS BEEN REACHED
90        $EXIT    STX      &RECLTH              SAVE RECORD LENGTH
                   MEND
```

The following figure shows the macro invocation and expansion first time.If the macro is invoked second time the labels may be expanded as

$AB LOOP , $AB EXIT.

```
.        RDBUFF   F1,BUFFER,LENGTH

30               CLEAR    X              CLEAR LOOP COUNTER
35               CLEAR    A
40               CLEAR    S
45               +LDT     #4096          SET MAXIMUM RECORD LENGTH
50       $AALOOP TD       =X'F1'         TEST INPUT DEVICE
55               JEQ      $AALOOP        LOOP UNTIL READY
60               RD       =X'F1'         READ CHARACTER INTO REG A
65               COMPR    A,S            TEST FOR END OF RECORD
70               JEQ      $AAEXIT        EXIT LOOP IF EOR
75               STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80               TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85               JLT      $AALOOP          HAS BEEN REACHED
90       $AAEXIT STX      LENGTH         SAVE RECORD LENGTH
```

Unique labels are generated within macro expansion. Each symbol beginning with $ has been modified by replacing $ with $AA. The character $ will be replaced by $xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded. For the first macro expansion in a program, xx will have the value AA. For succeeding macro expansions, xx will be set to AB, AC etc

Conditional Macro Expansion

There are applications of macro processors that are not related to assemblers or assembler programming.

Conditional assembly depends on parameters provides

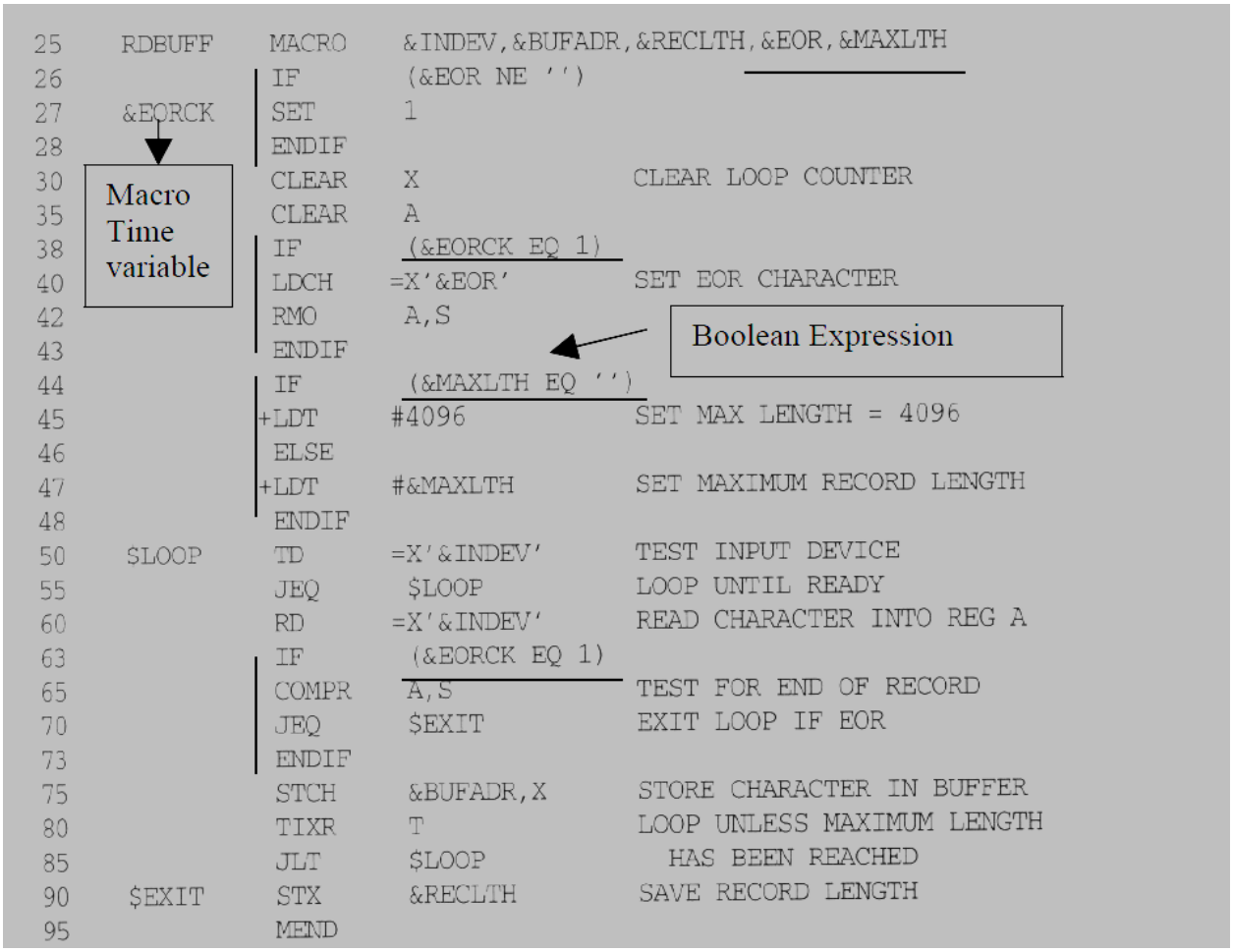
```
MACRO &COND
……..
IF (&COND NE ‘’)
part I
ELSE
part II
ENDIF
```

.....
ENDM

Part I is expanded if condition part is true, otherwise part II is expanded. Compare operators: NE, EQ, LE, GT.

Macro-Time Variables:

Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion. Any symbol that begins with symbol & and not a macro instruction parameter is considered as *macro-time variable*. All such variables are initialized to zero.




```
25    RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
26             IF      (&EOR NE '')
27    &EORCK   SET     1
28             ENDIF
30             CLEAR   X               CLEAR LOOP COUNTER
35             CLEAR   A
38             IF      (&EORCK EQ 1)
40             LDCH    =X'&EOR'        SET EOR CHARACTER
42             RMO     A,S
43             ENDIF
44             IF      (&MAXLTH EQ '')
45             +LDT    #4096           SET MAX LENGTH = 4096
46             ELSE
47             +LDT    #&MAXLTH        SET MAXIMUM RECORD LENGTH
48             ENDIF
50    SLOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
55             JEQ     $LOOP           LOOP UNTIL READY
60             RD      =X'&INDEV'      READ CHARACTER INTO REG A
63             IF      (&EORCK EQ 1)
65             COMPR   A,S             TEST FOR END OF RECORD
70             JEQ     $EXIT           EXIT LOOP IF EOR
73             ENDIF
75             STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
80             TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85             JLT     $LOOP             HAS BEEN REACHED
90    $EXIT    STX     &RECLTH         SAVE RECORD LENGTH
95             MEND
```

Figure gives the definition of the macro RDBUFF with the parameters &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH. According to the above program if &EOR has any value, then &EORCK is set to 1 by using the directive SET, otherwise

it retains its default value 0.

The above programs show the expansion of Macro invocation statements with different values for the time variables. In figure 6.9(b) the &EOF value is NULL. When the macro invocation is done, IF statement is executed, if it is true EORCK is set to 1, otherwise normal execution of the other part of the program is continued.

The macro processor must maintain a symbol table that contains the value of all macro-time variables used. Entries in this table are modified when SET statements are processed. The table is used to look up the current value of the macro-time variable whenever it is required. When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.

If the value of this expression TRUE,

The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement. If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF. Once it reaches ENDIF, it resumes expanding the macro in the usual way.

If the value of the expression is FALSE,

The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion.

The *macro-time* IF-ELSE-ENDIF structure provides a mechanism for either generating(once) or skipping selected statements in the macro body. There is another construct WHILE statement which specifies that the following line until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. The testing of this condition, and the looping are done during the macro is under expansion. The example shown below shows the usage of Macro-Time Looping statement.

WHILE-ENDW structure

When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated .

If True,

The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.

When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.

If false ,

The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.

Keyword Macro Parameters

All the macro instruction definitions used positional parameters. Parameters and arguments are matched according to their positions in the macro prototype and the macro invocation statement. The programmer needs to be careful while specifying the arguments. If an argument is to be omitted the macro invocation statement must contain a null argument mentioned with two commas. Positional parameters are suitable for the macro invocation. But if the macro invocation has large number of parameters, and if only few of the values need to be used in a typical invocation, a different type of parameter specification is required

Ex: XXX MACRO &P1, &P2, …., &P20, ….

XXX A1, A2,,,,,,,,,,…,,A20,…..

Null arguments

Keyword parameters

Each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. Null arguments no longer need to be used.

· Ex: XXX P1=A1, P2=A2, P20=A20.

It is easier to read and much less error-prone than the positional method.

Macro Processor Design Options

1. Recursive Macro Expansion

We have seen an example of the *definition* of one macro instruction by another. But we have not dealt with the *invocation* of one macro by another. The following example shows the invocation of one macro by another macro.

Problem of Recursive Expansion

Previous macro processor design cannot handle such kind of recursive macro invocation and expansion

The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.

o The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, *i.e.*, the macro process would forget that it had been in the middle of expanding an "outer" macro.

· Solutions

o Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.

o If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows:

	Value
	BUFFER
	LENGTH
	F1
	(unused)
	-

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like

Parameter	Value
1	F1
2	(Unused)
--	--

At the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE. Thus the macro processor would 'forget' that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

General-Purpose Macro Processors

· Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages

· **Pros**

o Programmers do not need to learn many macro languages.

o Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language,

thus save substantial overall cost.

· **Cons**

o Large number of details must be dealt with in a real programming language ▪ Situations in which normal macro parameter substitution should not occur, e.g., comments.

▪ Facilities for grouping together terms, expressions, or statements

▪ Tokens, e.g., identifiers, constants, operators, keywords

▪ Syntax had better be consistent with the source programming language

Macro Processing within Language Translators

· The macro processors we discussed are called "Preprocessors".

o Process macro definitions

o Expand macro invocations

o Produce an expanded version of the source program, which is then used as input to an assembler or compiler

· You may also combine the macro processing functions with the language translator: o Line-by-line macro processor

o Integrated macro processor

Line-by-Line Macro Processor

· Used as a sort of input routine for the assembler or compiler

o Read source program

o Process macro definitions and expand macro invocations

o Pass output lines to the assembler or compiler

· Benefits

o Avoid making an extra pass over the source program.

o Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)

o Utility subroutines can be used by both macro processor and the language translator.

▪ Scanning input lines

▪ Searching tables

▪ Data format conversion

o It is easier to give diagnostic messages related to the source statements

Integrated Macro Processor

· An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

o Ex (blanks are not significant in FORTRAN)

▪ DO 100 I = 1,20

· a DO statement

▪ DO 100 I = 1

· An assignment statement

· DO100I: variable (blanks are not significant in FORTRAN)

· An integrated macro processor can support macro instructions that depend upon the context in which they occur.