BANGALORE INSTITUTE OF TECHNOLOGY
K.R. Road, V.V.Puram, Bengaluru-560 004

DEPARTMENT OF COMPUTER SCIENCE & ENGG

SYSTEM SOFTWARE AND COMPILER DESIGN

NOTES

SUBJECT CODE: 15CS63

By

Mrs. Hemavathi. P
Assistant Professor
Department of CSE

SYSTEM SOFTWARE AND COMPILER DESIGN
[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2016 -2017)
SEMESTER – VI

Subject Code	15CS63	IA Marks	20
Number of Lecture Hours/Week	4	Exam Marks	80
Total Number of Lecture Hours	50	Exam Hours	03

CREDITS – 04

Course objectives: This course will enable students to

- Define System Software such as Assemblers, Loaders, Linkers and Macroprocessors
- Familiarize with source file, object file and executable file structures and libraries
- Describe the front-end and back-end phases of compiler and their importance to students

Module – 1	**Teaching Hours**
Introduction to System Software, Machine Architecture of SIC and SIC/XE. **Assemblers:** Basic assembler functions, machine dependent assembler features, machine independent assembler features, assembler design options. **Macroprocessors:** Basic macro processor functions, **Text book 1: Chapter 1: 1.1,1.2,1.3.1,1.3.2, Chapter2 : 2.1-2.4,Chapter4: 4.1.1,4.1.2**	**10 Hours**
Module – 2	
Loaders and Linkers: Basic Loader Functions**,** Machine Dependent Loader Features, Machine Independent Loader Features, Loader Design Options, Implementation Examples. **Text book 1 : Chapter 3 ,3.1 -3.5**	**10 Hours**
Module – 3	
Introduction: Language Processors, The structure of a compiler, The evaluation of programming languages, The science of building compiler, Applications of compiler technology, Programming language basics **Lexical Analysis:** The role of lexical analyzer, Input buffering, Specifications of token, recognition of tokens, lexical analyzer generator, Finite automate. **Text book 2:Chapter 1 1.1-1.6 Chapter 3 3.1 – 3.6**	**10 Hours**
Module – 4	
Syntax Analysis: Introduction, Role Of Parsers, Context Free Grammars, Writing a grammar, Top Down Parsers, Bottom-Up Parsers, Operator-Precedence Parsing **Text book 2: Chapter 4 4.1 4.2 4.3 4.4 4.5 4.6 Text book 1 : 5.1.3**	**10 Hours**
Module – 5	
Syntax Directed Translation, Intermediate code generation, Code generation **Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2**	**10 Hours**

Course outcomes: The students should be able to:

- Explain system software such as assemblers, loaders, linkers and macroprocessors
- Design and develop lexical analyzers, parsers and code generators
- Utilize lex and yacc tools for implementing different concepts of system software

Question paper pattern:
The question paper will have TEN questions.
There will be TWO questions from each module.
Each question will have questions covering all the topics under a module.
The students will have to answer FIVE full questions, selecting ONE full question from each module.

Text Books:

1. System Software by Leland. L. Beck, D Manjula, 3rd edition, 2012
2. Compilers-Principles, Techniques and Tools by Alfred V Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Pearson, 2nd edition, 2007

Reference Books:

1. Systems programming – Srimanta Pal , Oxford university press, 2016
2. System programming and Compiler Design, K C Louden, Cengage Learning
3. System software and operating system by D. M. Dhamdhere TMG
4. Compiler Design, K Muneeswaran, Oxford University Press 2013.

BANGALORE INSTITUTE OF TECHNOLOGY
K R ROAD, V V PURAM, BENGALURE-04

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

COURSE OBJECTIVES AND OUTCOMES-2015-19

Course Title : System Software and Compiler Design	Course Code : : 15CS63
No. of Lecture Hrs./Week : 04	Exam Hours : 03
Total No. of Lecture Hrs. : 52	Exam Marks : 80

Prerequisites

1. Microprocessors and Microcontrollers(15CS44)
2. Automata Theory and Computability (15CS54)

Course Learning Objectives

This course will help students to achieve the following objectives:

1. To understand the concepts of System software, Application Software and different hypothetical machine architectures.
2. Familiarize with source file, symbol table creation (pass-1), object file creation (pass-2), loaders and linkers.
3. To know the fundamental concepts of translators.
4. To identify the methods and strategies for parsing techniques.
5. Devise and perform syntax-directed translation schemes for compiler.
6. Devise intermediate code generation schemes and analyze the optimized code generated after the synthesis phase.

Course Outcomes

At the end of the course students should be able to:

1. Apply the knowledge of System Software such as Assemblers, Loaders, Linkers and Macro processors to build an application.
2. Understand the basic principles of compiler in high level programming language.
3. Analyze and design the analysis phase using different techniques.
4. Build the system software by associating synthesis phase with analysis phase for better optimization and performance.

MODULE-1

TEXTBOOK: System Software by Leland.L. Beck, D.Manjula, 3rd Edition, 2012

CHAPTER 1: **Introduction to System Software and Machine Architecture**

- 1.1 Introduction
- 1.2 System Software and Machine Architecture
- 1.3 The Simplified Instructional Computer (SIC)
  - 1.3.1 SIC Machine Architecture
  - 1.3.2 SIC/XE Machine Architecture
  - 1.3.3 SIC Programming Examples

CHAPTER 2: **Assemblers**

- 2.1 Basic Assembler Functions
  - 2.1.1 A Simple SIC Assembler
  - 2.1.2 Assembler Algorithm and Data Structures
- 2.2 Machine-Dependent Assembler Features
  - 2.2.1 Instruction Formats and Addressing Modes
  - 2.2.2 Program Relocation
- 2.3 Machine-Independent Assembler Features
  - 2.3.1 Literals
  - 2.3.2 Symbol-Defining Statements
  - 2.3.3 Expressions
  - 2.3.4 Program Blocks
  - 2.3.5 Control Sections and Program Linking
- 2.4 Assembler Design Options
  - 2.4.1 One-Pass Assemblers
  - 2.4.2 Multi-Pass Assemblers

CHAPTER 4: **Macro Processors**

- 4.1 Basic Macro Processor Functions
  - 4.1.1 Macro Definition and Expansion
  - 4.1.2 A Simple Bootstrap Loader

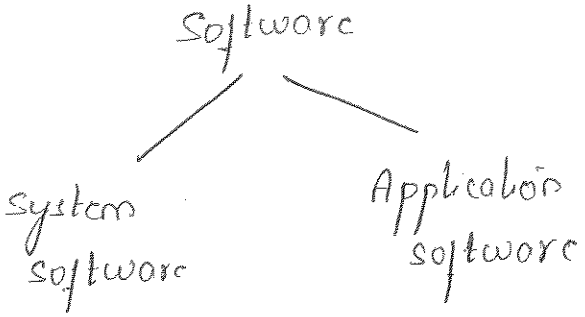
CHAPTER 1

Introduction to System Software and Machine Architecture

18/12/2017

The term "software" refers to the set of electronic program instructions or data a computer processor reads in order to perform a task.

"Hardware" refers to the physical components that you can see and touch, such as the computer hard drive, mouse and keyboard.



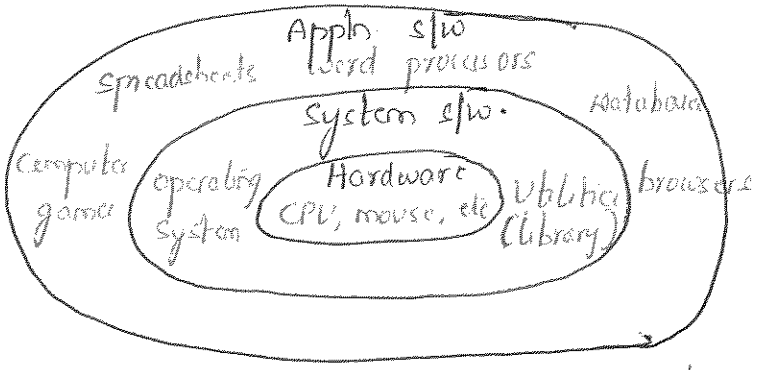


Fig: Relationship b/w system software, Appln software and Hardware

Defn: "System software" is a set of programs that are dedicated to manage the computer itself, such as operating system, file management utilities,

Application software are a set of productivity programs or end-user programs to perform their specific tasks

Difference between system software and Application software

System software	Application software
1 System software is a set of programs that are dedicated to manage the computer itself (mem. mgmt, process mgmt, protection security	Application software is a set of Computer programs designed to permit the user to perform a group of functions, tasks or activities.
2. Is written in a low level language ie assembly language	Is written in a high level language like C, C++, Java, .net, VB etc
3. Starts running when the sfm is turned on and runs till the system is shut down	Runs as and when the user requests
4 A system is unable to run without system software	Appln. software is even not required to run the system ∴ it is user specific
5. system software is general purpose	Appln. software is specific purpose software
6. Ex: operating system, assembler, compiler, loader or linker, text editor, debugger, macro processors,	Ex: web browser, word processing, spreadsheet, database, Adobe creative suite, Audio creator suite, games
7 not machine dependent (machine architecture)	not machine dependent

12. System software and machine architecture

- Machine dependency of system software
  - → System programs are intended to support the operation and use of the computer.
  - → machine architecture differs in :
    - machine code,
    - instruction formats
    - Addressing mode
    - Registers

- machine independency of system software
  - → general design and logic is basically the same :
    - code optimization
    - subprogram linking

1.3. Simplified Instructional Computer.

As we know different systems have different features and different features are difficult to study one by one. so to avoid this problem we study Simplified Instructional Computer.

SIC is a hypothetical computer system introduced in system software. Due to the fact that most modern microprocessors include complex functions for the purpose of efficiency, it is very difficult to learn systems programming using a real-world system. The SIC solves this problem by abstracting away these complex behaviours in favour of an architecture that is clear and accessible for those wanting to learn systems programming.

- SIC comes in two versions
  - → standard model
  - → XE version (Extra Equipment or Extra Expensive)
- The two versions has been designed to be upward compatible.

1.3.1 SIC machine architecture

Every machine architecture include

a) memory
b) Registers
c) Data formats
d) Instruction formats
e) Addressing modes
f) Instruction set
g) Input and output

a) Memory
→ memory consists of 8 bit bytes
→ Any 3 consecutive byte form a word (24 bits)
→ All addresses on SIC are byte address
→ words are addressed by the location of their lowest numbered byte
→ Total of 32,768 (2^{15}) byte in the computer memory
∴ 15 - line address bus

b) Registers
→ Five registers, all of which have special uses
→ Each register is 24 bits in length
→ table shows the mnemonic, number and use of each register.

Mnemonic	Number	Use
A Accumulator	0	Used for arithmetic operations
X Index Register	1	Used for addressing
L Linkage Register	2	JSUB – Jump to subroutine instruction store return addres
PC Program counte	8	Contain the address of the next instruction to be fetched for execution
SW status word	9	Contains variety of information including a condition code in comp instruction

c) Data formats

→ Integers are stored as 24-bit binary number.

($0 - 2^{24} - 1 \Rightarrow 0$ to $16^6 - 1$)

→ Negative value are represented as 2's complement

Ex: -3_H is represented as (8 bit representation)

3_H = 0 0 0 1 1 0 0 0

1's complement 1 1 1 0 0 1 1 1

$+ \quad 1$

1 1 1 0 1 0 0 0 $\Rightarrow 232$

→ characters are stored using their 8-bit ASCII codes

→ There is no floating-point hardware on the standard version of SIC

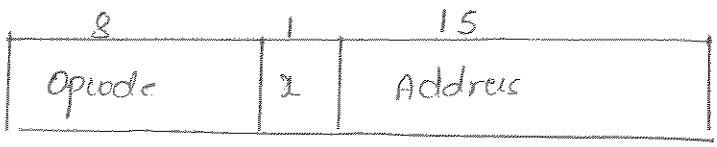
Ex:- 5 = 0000 0000 0000 0000 0000 0101

−5 = 1111 1111 1111 1111 1111 1011

A = 0100 0001 (65)

d) Instruction formats

→ All machine instructions on the standard version of SIC ~~are~~ have 24-bit format



x → indicates indexed addressing mode

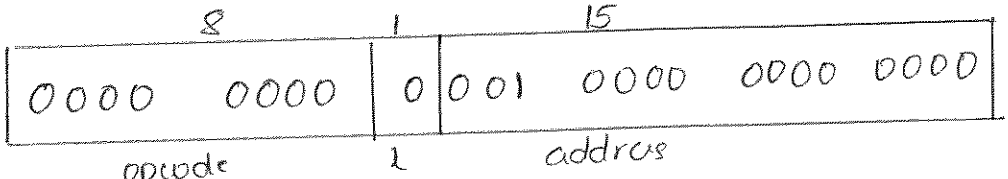
e) Addressing modes

→ Two addressing modes based on x bit

- Direct Addressing
- Indexed addressing

mode	Indication	Target address (TA)
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x) parentheus indicate the content of a register or memory location

→ Ex:- LDA TEN ; LDA = 00 (opcode)

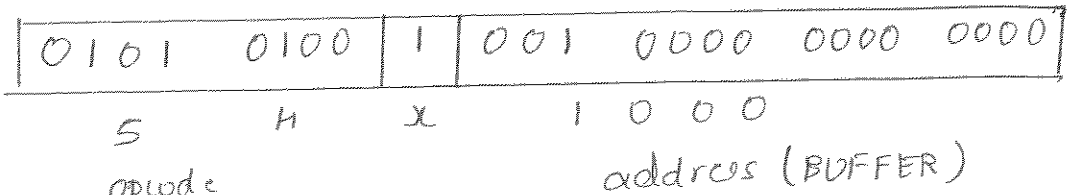


Target address / Effective address = 1000 ie contents of the address 1000 is loaded to accumulator

→ Indexed addressing mode

Ex: STCH BUFFER, X ; opcode for STCH = 5H

BUFFER = 1000



TA = address + (x)

= 1000 + content of the index register X

ie the Accumulator content, the character is loaded to the effective address.

1) Instruction set

(i) load and store: LDA, LDX, STA, STX

(ii) Integer Arithmetic operations: ADD, SUB, MUL, DIV

- Arithmetic operation involve register A and a word in memory with the result being left in the register

Ex: ADD WORD ; A ← A + WORD

ie adds register A contents with ~~word~~ WORD and result is stored in register A

iii) Comparision operations: COMP

- Compares the value in register A with memory and sets a condition code (cc) accordingly

Ex: COMP WORD ; compares A's contents with WORD and sets cc as < = >

(iv) Conditional jump instructions : JLT, JEQ, JGT

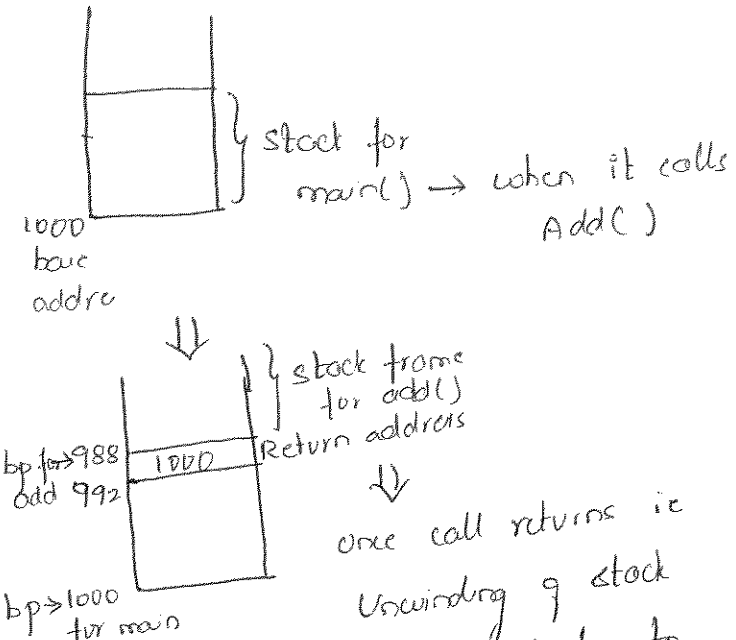
- these instructions will test the setting of CC and jump accordingly

(v) Subroutine linkage instructions ; JSUB, RSUB

- JSUB – jumps to the subroutine by placing the return address in register L. (Fn. call)
- RSUB – returns by jumping to the address contained in register L (Fn. return to the caller)

Ex:

```
void main()
{
    Add(1,2);
}
Fn call
Add(int x, int y)
{ return (x+y);
Fn Return }
```



once call returns it Unwinding of stack happens it has to go back to caller making base pointer ~~add~~ point to main so 1000 is stored.

(vg) Input and output

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

→ Each device is assigned a unique 8-bit code

→ There are three I/O instructions which specify the device code as an operand

(i) TD (Test Device) : checks whether the addressed device is ready to send or receive a byte of data. CC (condition code in SW register) is set according (< =)

* < → device is ready to send/receive
= → device is not ready.

→ A program has to wait until the device is ready, then execute a Read Data (RD) and write Data (WD) instructions.

→ This sequence should be continued for each byte of data. (I/O).

→ RD : Transfers a byte of data from i/p device into rightmost byte of register A (RD INDEV STA DATA)

→ WD : Byte of data is loaded into rightmost byte of reg. A and then written to output device (LDA DATA & WD OUTDEV)

Ex: 1) SIC instructions for data movement operations (no memory - memory move instructions)

```
        LDA     FIVE    ; Load constant 5 into register A
        STA     ALPHA   ; Store in Alpha
        LDCH    CHAR2   ; load character '2' into reg. A
        STCH    C1      ; store in character variable C1
        .
        .
        .
ALPHA   RESW    1       ; one-word variable
FIVE    WORD    5       ; one-word constant
CHAR2   BYTE    C '2'   ; one-byte constant
C1      RESB    1       ; one-byte variable
```

⇓ same can be written as

```
        LDX     FIVE
        STX     ALPHA
```

Or

```
        LDL     FIVE
        STL     ALPHA
```

A – Accumulator
X – indexed register
L – Linkage register

2) SIC instructions for arithmetic operations

Delta = BETA = ALPHA + INCR − 1

DELTA = GAMMA + INCR − 1

```
        LDA     ALPHA   ; loads Alpha into register A
        ADD     INCR    ; A ← (A) + (INCR)
        SUB     ONE     ; A ← (A) − 1
        STA     BETA    ; BETA ← (A)
        LDA     GAMMA   ; A ← (GAMMA)
        ADD     INCR    ; A ← (A) + (INCR)
        SUB     ONE     ; A ← (A) − 1
        STA     DELTA   ; DELTA ← (A)
        :
ONE     WORD    1
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

3) SIC instructions for looping and indexed operations

(program to copy 11-byte character string to another string)

```
// LDX  ZERO ; initialize index register
for (i=0; j=0; s1[i] != '\0' ; i++)
        s2[j++] = s1[i];
s2[j] = '\0';
```

```
        LDX   ZERO   ; initialize index register to 0
LOOP    LDCH  STR1,X ; copies the first character of str1 to reg. A
                       (TA : (address) → content of first byte
                       of str1)
        STCH  STR2,X ; store the first character into STR2
        TIX   ELEVEN ; Add 1 to index, compare to 11
                       X = 0+1 = 1 ; 1 ⇔ 11 CC will
                       be set as <
        JLT   LOOP   ; repeats if index is < 11
                       (X)
        .
        .
        .
STR1    BYTE  C 'HELLO WORLD'      (space)
STR2    RESB  11
ZERO    WORD  0
ELEVEN  WORD  11
```

TIX compare X with 11

h) Program to add 2 arrays of 100 words each and store it in another array. Each word is 3 byte ∴ 100 words : 3 x 100 = 300 byte

Q = A + B ;

```
ADDLOOP  LDX   INDEX   ; initialize index value X = 0
         LDA   ALPHA,X ; A ← (ALPHA)
         ADD   BETA,X  ; A ← (A) + (BETA) at 0th byte (index)
         STA   GAMMA,X ; Q ← (A) at 0th byte (index value)
         LDA   INDEX   ; A ← 0
         ADD   THREE   ; A ← (A) + 3 = 3 → n
         STA   INDEX   ; INDEX = 3
         COMP  K300    ; A ↔ K300 ie 3 ⇔ 300 CC = <
         JLT   ADDLOOP ; repeat loop, now x = 3rd byte

K300     WORD  300     THREE  WORD  3
INDEX    RESW  1       BETA   RESW  100
ALPHA    RESW  100     GAMMA  RESW  100
```

5) To read one byte of data from input device and copies it to device 05

```
INLOOP  TD      INDEV   ; Tests input device
        JEQ     INLOOP  ; cc := then loop until device ready
        RD      INDEV   ; once ready, read a byte into reg A
        STCH    DATA    ; store it in data (memory)
        :
OUTLOOP TD      OUTDEV  ; Test output device
        JEQ     OUTLOOP ; cc := then loop until device ready
        LDCH    DATA    ; load data byte into reg. A
        WD      OUTDEV  ; write one byte to output device
        :
INDEV   BYTE    X 'F1'
OUTDEV  BYTE    X '05'
DATA    RESB    1
```

6) Subroutine call to read a 100-byte record from an input device into memory.

```
        JSUB    READ    ; call Read subroutine wherein
        :               it stores the return address in
                        linkage register
READ    LDX     ZERO    ; X ← 0
RLOOP   TD      INDEV   ; Test input device
        JEQ     RLOOP   ; cc:=, loop until device is ready
        RD      INDEV   ; read one byte into reg. A
        STCH    RECORD,X ; store A into RECORD at Xth addr
        TIX     K100    ; X = (X)+1 & 1 <-> 100 compare
        JLT     RLOOP   ; cc: < then loop back
        RSUB            ; Exit from subroutine; it
        :               returns to the address stored in
                        linkage register
```

```
INDEX    BYTE    X'F1'
RECORD   RESB    100
ZERO     WORD    0
K100     WORD    100
```

1.3.2 : SIC/XE machine Architecture

→ SIC/XE : Simple Instructional Computer with Extra Equipments

a) Memory
b) Registers
c) Data formats
d) Instruction formats
e) Addressing modes
f) Instruction set
g) Input and output

a) Memory

→ memory consists of 8 bit bytes
→ 3 consecutive bytes form a word (24 bits)
→ All addresses are byte addresses
→ words are addressed by the location of their lowest numbered byte
→ Total of 1 MB (2^{20} bytes) in the memory. (20 bit address bus) which leads to change in instruction formats and addressing modes.

b) Registers

→ There are 9 registers
→ Each register is 24 bits in length except Floating Point reg.
→ The registers are A, X, L, B, S, T, F, PC & SW

Mnemonic	Number	Uses
A Accumulator	0	Used for arithmetic operations
X Index Register	1	Used for addressing (Indexed)
L Linkage register	2	used to store the return address for JSUB instruction
B Base register	3	used for addressing
S General Register	4	General working register - no special use
T General Register	5	General working register - no special use
F Floating Point Accumulator	6	~~General working register~~ Floating point accumulator (48 bits)
PC Program Counter	8	Contains the address of the next instruction to be fetched for execution
SW status word	9	Contains a variety of information including a condition code (CC)

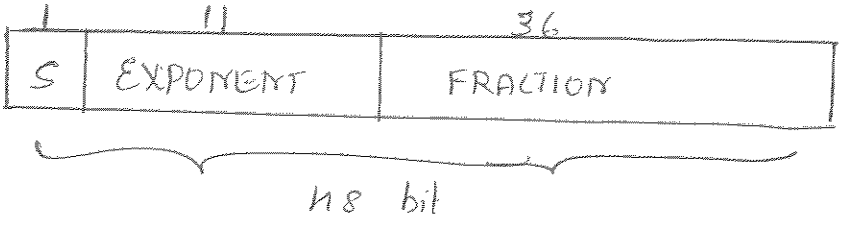
c) Data formats :

→ Integers are stored as 24-bit binary numbers

→ negative value are represented as 2's complement (1's complement +1)

→ characters are stored using their 8-bit ASCII codes

→ There is a 48-bit floating point data type



→ The fraction is interpreted as a value between 0 and 1

→ The assumed binary point is immediately before the higher order bit

→ For normalized floating point numbers, the higher order bit of the fraction must be 1

→ The exponent is interpreted as an unsigned binary number between 0 and 2047 $(0 - (2^{11} - 1))$

→ If the expression has value e, fraction f and the absolute value of number is represented is

$$f * 2^{(e-1023)}$$

→ The sign of floating point number is indicated by S (S = 0 (+ve) and 1 (-ve))

Ex:- 5 = 0000 0000 0000 0000 0000 0101

−5 = 1111 1111 1111 1111 1111 1011

A = 0100 0001 (65)

Ex:- 4.89 representation

As we know from computer organization it is represented as $\pm mB^{\pm E}$

↓ ↳ Base (2)

Fraction (mantissa)

1) Represent 4 in binary form → 100

2) Convert 0.89 into binary form until it repeats or until we get 36 bits which represents the fraction part

100.111000111101011100001010001111010111

000010100

3) normalization has to be done but not always ∵ they have specified that binary point is immediately before the higher order bit

i.e 100.111000...

0.100111000111101011100001010001111010 $\times 2^3$ → Exponent

fraction

Note: for normalized floating point number it will be as $1.00111000111\ldots \times 2^2$

i.e $\pm 1 * 2^{e \pm 1024} = 0.100111\ldots \times 2^{3+1027}$

$= 0.1001111\ldots \times 2^{1029}$

$0.89 \times 2 \rightarrow$

$0.78 \times 2 \rightarrow 1$ i.e 1.78

$0.56 \times 2 \rightarrow 1$

$0.12 \times 2 \rightarrow 1$

$0.24 \times 2 \rightarrow 0$

$0.48 \times 2 \rightarrow 0$

$0.96 \times 2 \rightarrow 0$

$0.92 \times 2 \rightarrow 1$

$0.84 \times 2 \rightarrow 1$

$0.68 \times 2 \rightarrow 1$

$0.36 \times 2 \rightarrow 1$

$0.72 \times 2 \rightarrow 0$

$0.44 \times 2 \rightarrow 1$

$0.88 \times 2 \rightarrow 0$

$0.76 \times 2 \rightarrow 1$

$0.52 \times 2 \rightarrow 1$

$0.04 \times 2 \rightarrow 1$

$0.08 \times 2 \rightarrow 0$

$0.16 \times 2 \rightarrow 0$

$0.32 \times 2 \rightarrow 0$

$0.64 \times 2 \rightarrow 0$

$0.28 \times 2 \rightarrow 1$

$0.56 \times 2 \rightarrow 0$

$0.12 \times 2 \rightarrow 1$

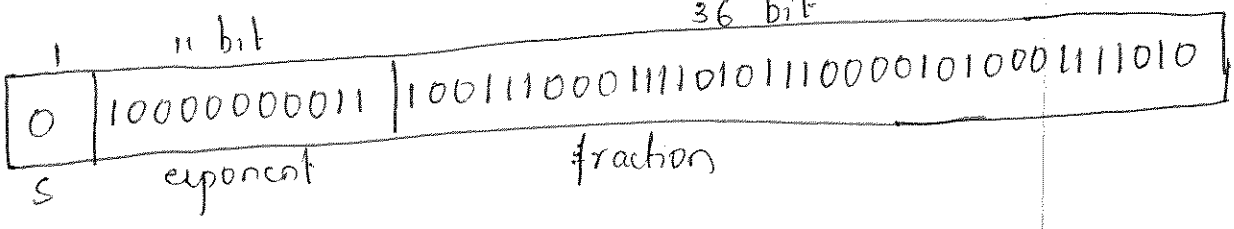
$0.24 \times 2 \rightarrow 0$

$0.48 \times 2 \rightarrow 0$

$0.96 \times 2 \rightarrow$ continue as above

01111010111000010100

$(1027)_2 \rightarrow 10000000011$



c) Represent $\xrightarrow{s=1 \text{ (negative)}}$ $-0.000h89$ in binary format

Given $-0.000h89$

↓

↔ Represent 0 as binary 0

→ Represent $0.000h89$ as binary

$= 0.000000000010000000011000000$
$11110000001111110,$

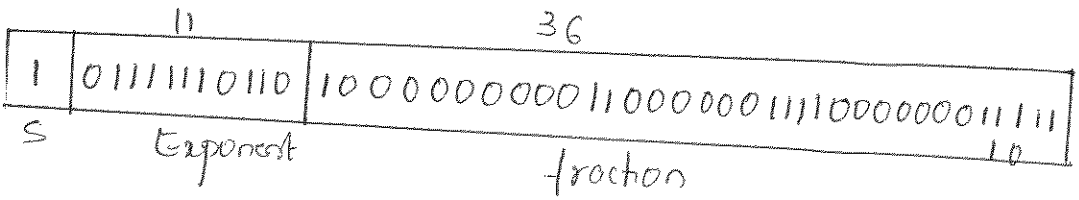
$= .10000000001100000001111100000001$
111111×2^{-10} (fraction)

⇒ $f * 2^{e+1024} = .10000\ldots \times 2^{-10+1024}$

$= .10000\ldots \times 2^{1014 \to E}$ $\quad (1014)_2 = 01111110110$

fraction

$.000h89 \times 2$	
0.000978×2	$\to 0$
0.001956×2	$\to 0$
0.003912×2	$\to 0$
0.007824×2	$\to 0$
0.015648×2	$\to 0$
0.031296×2	$\to 0$
0.062592×2	$\to 0$
0.125184×2	$\to 0$
0.250368×2	$\to 0$
0.500736×2	$\to 0$
0.001472×2	$\to 01$
0.002944×2	$\to 0$



d) Instruction formats.

→ Since the memory used by SIC/XE may be 2^{20} bytes, the instruction format of SIC is not enough.

→ There are two possible options

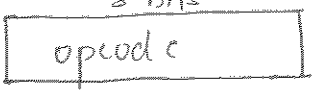
(i) Either use some form of relative addressing

(ii) Extend the address field to 20 bits

→ if $e=0$, then format 3

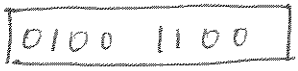
→ if $e=4$, then format 4

1) Format 1 (1 byte)



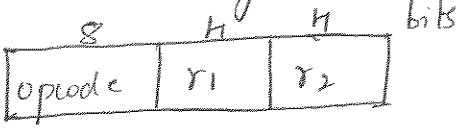
Ex: RSUB (return to subroutine) → HC

⇒ it returns to the address stored in linkage register



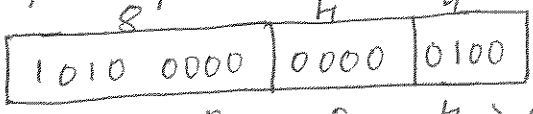
→ object code

2) Format 2 (2 byte)

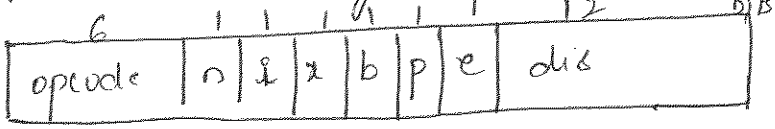


Ex: COMPR A,S (Compare the contents of registers A & S)

opcode of COMPR = A0

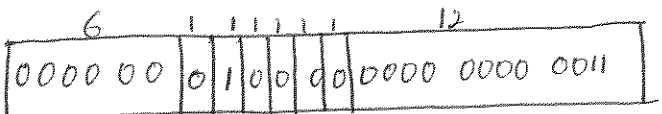


3) Format 3 (3 byte)



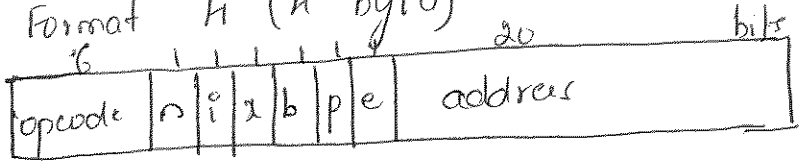
note: e=0

Ex: LDA #3 (load 3 to A)



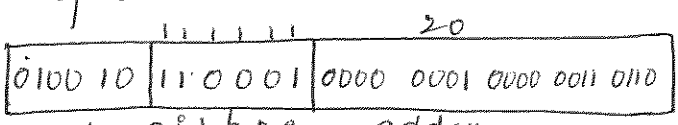
01003 → object code

4) Format 4 (4 byte)



Ex: +JSUB RDREC (Jump to address 1036)

opcode JSUB → 48



object code is 4B101036

e) Addressing modes

	MODE	INDICATION	TARGET ADDRESS CALCULATION
1.	Base Relative	$b=1, p=0$	TA = (B) + displacement ($0 <= disp \leq 4095$)
2.	Program Counter Relative	$b=0, p=1$	TA = (P) + displacement ($-2048 \leq disp \leq 2047$)
3.	Direct Addressing	$b=0, p=0$ (for format 3)	TA = displacement
		$b=0, p=0$ (for format 4)	TA = Address field
4.	Base Relative Indexed addressing	$b=1, p=0$ $x=1$	TA = (B) + (X) + displacement
5.	Program Relative Indexed addressing	$b=0, p=1$ $x=1$	TA = (P) + (X) + displacement
6.	Immediate addressing	$i=1, n=0$	TA = Target address itself used as operand value (no memory reference)
7.	Indirect addressing	$i=0, n=1$	TA = displacement value
8.	Simple addressing	$i=0, n=0$ OR $i=1, n=1$	TA = location of operand

Note:

Format 3 :

↳ in Base relative, disp is interpreted as 12 bit unsigned integer (1)

↳ in PC relative, disp is interpreted as 12-bit signed integer & negative numbers in 2's complete (2)

Special symbols indication

1) # : Immediate address
2) @ : Indirect address
3) + : Format 4
4) * : The current value of PC
5) c' ' : character string
6) op m, x : x - denote the index register
7) base : Base - register

Instruction set :

Note: Immediate addressing (i=1, n=0) → Target address itself is used as the operand value (no memory reference is performed)

→ Indirect addressing (i=0, n=1) → the word at the location given by the target address is fetched and the value contained in this word is then taken as the address of the operand value.

→ Indexing cannot be used with immediate or indirect addressing mode.

** → we can't set both b=1 & p=1 which is invalid instruction set.

examples of SIC/XE instructions and addressing mode

(B) = 006000
(PC) = 003000
(X) = 000090

Machine instruction

Hex	Binary: 6 opcode	n	i	x	b	p	e	12/20 disp/address	Target address	Value loaded into Reg. A	Mode
032600	000000 (0)	1 (3)	1	0	0	1 (2)	0	0110 0000 0000 (6 0 0)	3600	103000	Program Counter Relative TA = (PC) + displacement = 003000 + 600 = 3600
03C300	0000 00	1	1	1	1	0	0	0011 0000 0000	6390	00C303	Base relative Indexed TA = (B) + (X) + disp = 006000 + 000090 + 300 = 6390
022030	0000 00	1	0	0	0	1	0	0000 0011 0000	3030 ↓contain 3600) contain	103000 ↑	Indirect + Program relative TA = (PC) + disp = 003000 + 030 = 3030
010030	0000 00	0	1	0	0	0	0	0000 0011 0000	30	000030	Immediate addressing TA is used as operand value
003600	0000 00	0	0	0	0	1	1	0110 0000 0000	3600	103000	PC relative TA = (PC) + disp = 003000 + 600 = 3600
0310C303	0000 00	1	1	0	0	0	1	0000 1100 0011 0000 0011	C303	003030	Simple addressing TA = location of operand

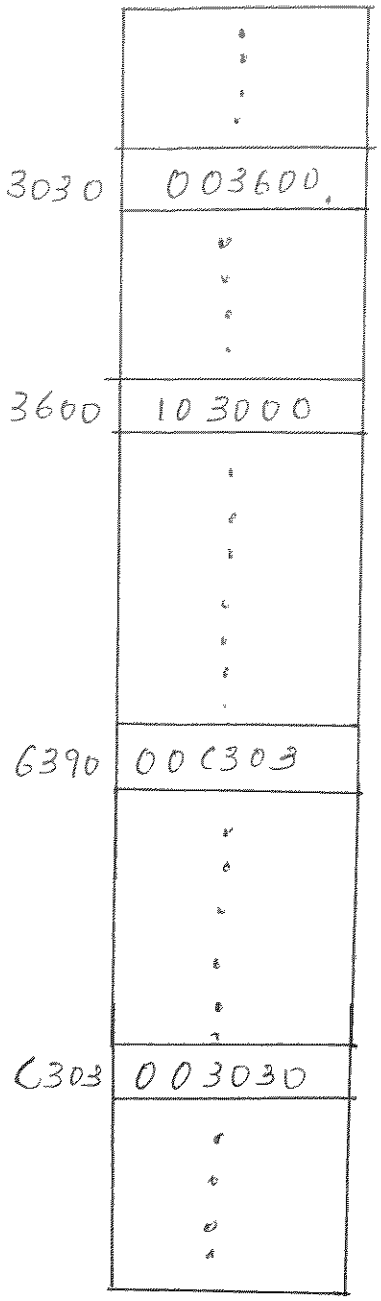


Fig: contents of registers B, PC and X & memory locations

1) Instruction set

* → Load and store instruction : LDA, LDX, STA, STX, LDB, STB
* → Integer and floating point arithmetic operations :
  ADD, SUB, MUL, DIV, ADDF, SUBF, MULF, DIVF
* → Register move instructions (RMO) ⇒ register to-register operations such as ADDR, SUBR, MULR, DIVR
* → A special supervisor call (SVC) instruction is provided. Executing this instruction generates an interrupt that can be used for communication with the operating system.
  → Comparision instruction : COMP, COMPR, COMPF
  → Conditional jump instructions : JLT, JEQ, JGT
  → subroutine linkage instructions : JSUB, RSUB

2) Input and output

→ Input and output is performed by transferring 1 byte of data at a time to or from, the rightmost 8 bits of register A.
→ Each device is assigned a unique 8-bit code
→ Three I/O instructions which specify the device code as an operand
  (i) TD (Test Device) → Tests whether the addressed device is ready to send or receive a byte of data and sets the condition code (CC)
    < : Device is ready to send/receive
    = : Device is not ready

→ Test continues until the device is ready.

→ Once ready, either RD (Read Data): Transfers data ~~reads~~ from input device or keyboard into rightmost byte of register A and stored in buffer if required. (RD INDEV & STA DATA)

→ WD (write data): a byte of data is loaded into the rightmost byte of register A and then written to the addressed device (LDA DATA & WD OUTDEV)

* → There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. This allows overlap of computing and I/O, resulting in more efficient system operation.

↳ SIO ↔ start I/O

↳ TIO ↔ Test I/O

↳ HIO ↔ Halt I/O

SIC/XE programs

1) Data movement operations

```
        LDA     #5          ; loads value 5 into register A
        STA     ALPHA       ; store in alpha : A ← (A) + (ALPHA)
                                                   5 + (ALPHA)
        LDA     #90         ; load ascii code for '2' into A
        STCH    C1          ; store in character variable C1
        :
ALPHA   RESW    1           ; one-word variable
C1      RESB    1           ; one byte variable
```

2) Arithmetic operations (Beta = alpha + incr − 1)

```
        LDS     INCR        ; load value of incr to S
        LDA     ALPHA       ; load value of alpha to A
        ADDR    S, A        ; A ← (A) + (S)
                source  destination
        SUB     #1          ; A ← (A) − 1
        STA     BETA        ; BETA ← (A)
        :
```

3) Looping and indexed operations

GAMMA = ALPHA + BETA where ALPHA and BETA are arrays of 100 words each. (100 × 3 = 300 bytes)

```
        LDS     #3          ; S = 3
        LDT     #300        ; T = 300
        LDX     #0          ; X = 0 → which specifies index value
ADDLOOP LDA     ALPHA, X    ; A ← (ALPHA) at the specified address of
                                                          index register
        ADD     BETA, X     ; A ← (A) + (BETA)
        STA     GAMMA, X    ; GAMMA ← (A) at specified index value (0)
        ADDR    S, X        ; X ← (X) + (S) = 0 + 3 = 3 (index register
                                                        address is 3)
        COMPR   X, T        ; (X) ↔ (T) compared i.e. 3 < 300, CC = <
        JLT     ADDLOOP     ; repeat loop till 300 = 300
```

```
        :
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
```

5) To read one byte of data from input device F1 and copies it to device 05

```
INLOOP  TD      INDEV
        JEQ     INLOOP
        RD      INDEV
        STCH    DATA
        :
OUTLOOP TD      OUTDEV
        JEQ     OUTLOOP
        LDCH    DATA
        WD      OUTDEV
        :
INDEV   BYTE    X'F1'
OUTDEV  BYTE    X'05'
DATA    RESB    1
```

6) subroutine call to read 100-byte record from an input device into memory

```
        JSUB    READ
        :
READ    LDX     #0
        LDT     #100
RLOOP   TD      INDEV
        JEQ     RLOOP
        RD      INDEV
```

```
        STCH    RECORD,X
        TIXR    T
        JLT     RLOOP
        RSUB
        :
        :
INDEV   BYTE    X'F1'
RECORD  RESB    100
```

Write a SIC and SIC/XE program to copy 'SYSTEM SOFTWARE' to another string.

a) SIC program

```
        LDX     ZERO
LOOP    LDCH    STR1,X
        STCH    STR2,X
        TIX     FIFTEEN
        JLT     LOOP
        :
        :
STR1    BYTE    C'SYSTEM SOFTWARE'
STR2    RESB    15
ZERO    WORD    0
FIFTEEN WORD    15
```

b) SIC/XE program

```
        LDX     #0
        LDT     #15
LOOP    LDCH    STR1,X
        STCH    STR2,X
        TIXR    T
        JLT     LOOP
        :
        :
STR1    BYTE    C'SYSTEM SOFTWARE'
STR2    RESB    15
```

Exercises 1.3

1. Write a sequence of instructions for SIC to set ALPHA equal to the product of BETA and GAMMA. Assume ALPHA, BETA and GAMMA are 1 word (ALPHA = BETA * GAMMA)

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

2. Write a sequence of instructions for SIC/XE to set ALPHA equal to H * BETA - 9. ALPHA, BETA and GAMMA are 1 word. Use immediate addressing for the constants (A = H * B - 9)

```
LDA   BETA
LDS   #H
MULR  S, A
SUB   #9
STA   ALPHA
:
ALPHA RESW  1
```

3. Write SIC instructions to swap the values of ALPHA and BETA.

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

4. Write a sequence of instructions for SIC to set ALPHA equal to the integer portion of BETA ÷ GAMMA. ALPHA, BETA, GAMMA are 1 word each

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

5. 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.

Ex:- B = 5
G = 2

```
LDA     BETA        ; A = 5
DIVF    GAMMA
LDS     GAMMA       ; S = 2
DIVR    S, A        ; A = A/S = 5/2 = 2
STA     ALPHA       ; A = 2
MULR    S, A        ; A = A * S = 2 * 2 = 4
LDS     BETA        ; S = 5
SUBR    A, S        ; S = S - A = 5 - 4 = 1
STS     DELTA       ; DELTA = 1
.
ALPHA   RESW  1
BETA    RESW  1
GAMMA   RESW  1
DELTA   RESW  1
```

Note:

// To find the remainder

Quotient = Dividend / Divisor

Remainder = Dividend − (Quotient * Divisor)

Ex:- Dividend = 10, Divisor = 3, Q = 10/3 = 3 ; R = 10 − (3 * 3) = 10 − 9 = 1

Dividend = 15, Divisor = 3, Q = 15/3 = 5 ; R = 15 − (5 * 3) = 15 − 15 = 0

6. Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the value of the quotient, rounded to the nearest integer. Use register-to-register instructions to make the calculation as efficient as possible.

```
        LDF     BETA
        DIVF    GAMMA
        FIX
        STA     ALPHA
        :
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
```

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

```
        LDX     ZERO
LOOP    LDCH    BLANK
        STCH    STR1,X
        TIX     TWENTY      ; ADD 1 to index and compare
        JLT     LOOP        ; LOOP if index<100 with 20 & set
        :                                   CC: < =>
STR1    RESB    20
BLANK   BYTE    C' '
ZERO    WORD    0
TWENTY  WORD    20
```

8. Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

```
        LDT   #20
        LDX   #0
CLOOP   LDCH  #0
        STCH  STR1,X
        TIXR  T
        JLT   CLOOP
        :
STR1    RESW  20
```

9. Suppose that ALPHA is an array of 100 words. Write a sequence of SIC instructions to set all 100 elements of the array to 0.

```
        LDA   ZERO
        STA   INDEX
LOOP    LDX   INDEX
        LDA   ZERO
        STA   ALPHA,X
        LDA   INDEX
        ADD   THREE
        STA   INDEX
        COMP  K300
        TIX   TWENTY
        JLT   LOOP
        :
INDEX   RESW  1
ALPHA   RESW  100
ZERO    WORD  0
K300    WORD  300
THREE   WORD  3
```

10. ALPHA is an array of 100 words. Write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

```
        LDS     #3
        LDT     #300
        LDX     #0
LOOP    LDA     #0
        STA     ALPHA, X
        ADDR    S, X
        COMPR   X, T
        JLT     LOOP
        :
ALPHA   RESW    100
```

11 ALPHA is an array of 100 words. Write a sequence of SIC/XE instructions to arrange the 100 words in ascending order and store the result in an array BETA of 100 words.

```
        LDS     #3
        LDT     #300
        LDX     #0
LOOP    LDA     ALPHA, X
        MUL     #H
```

12. ALPHA and BETA are the two arrays of 100 words. Another array of GAMMA elements are obtained by multiplying the corresponding ALPHA element by H and adding the corresponding BETA elements. Write the SIC/XE instructions for the same.

```
        LDS   #3
        LDT   #300
        LDX   #0
LOOP    LDA   ALPHA, X
        MVL   #H
        ADD   BETA, X
        STA   GAMMA, X
        ADDR  S, X      ; X ← X+3
        COMPR X, T      ; X ≥ 300
        JLT   LOOP
        :
        :
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100
```

13. ALPHA is an array of 100 words. write a sequence of SIC/XE instructions to find the maximum element in the array and store results in MAX.

```
        LDS   #3
        LDT   #300
        LDX   #0
LOOP    LDA   ALPHA, X
        COMP  MAX
        JLT   NXMAX
        STA   MAX
NXMAX   ADDR  S, X
        COMPR X, T
        JLT   LOOP
        :
ALPHA   RESW  100
MAX     WORD  -32768
```

note: COMP MAX ⇒ indicates Accumulator value is compared with MAX and set the CC (condition code) ie CC ← < = > of (A) ? (MAX). Based on cc value check the condition ie JLT, JGT, JEQ

→ COMPR X, T ⇒ Register value are compared X : (X) ? (T) and cc is set : < = > and Jump instruction is called

Explanation

ALPHA = {10, 20, 30, 40, ... 32768, ... }
↑ ↑
0 3
index

Each value is 1 word = 3 byte

100 words = 3 x 100 = 300 byte

index has to be incremented by 3.

ie initially x = 0, 3, 6, 9 ... 300

According to code ; S = 3, T = 100, X = 0

1st iteration LOOP ; Accumulator (A) = 10 at 0th position

COMP MAX ; 10 ? 32768 sets CC : <

JLT NOMAX

NOMAX ; ADDR S, X ; X ← (X) + S = 0+3 → Increment by 3 (next element)

element is 20

COMPR X, T ; X ← (X) ? (T)

← 300 ? 300 CC ; = <

JLT LOOP

↓

To check whether the array index has come to an end.

iteration

2) LOOP ; A ← 20 which is at 3rd position

LDA ALPHA 3 → value at 3rd position

COMP MAX ; 20 ? 32768 - CC ; <

JLT NOMAX

:

Continued

14. A RECORD contains a 100-byte record. Write a subroutine for SIC that will write this record onto device 05.

```
        JSUB    WRREC
          :
WRREC   LDX     ZERO        ; Initialize index register = 0
LOOP    TD      OUTPUT      ; Test output device
        JEQ     WLOOP       ; Loop if device is busy
        LDCH    RECORD,X    ; load one byte to Accumulator
        WD      OUTPUT      ; write one byte to device
        TIX     LENGTH      ; Add 1 to index & compare to 100
        JLT     LOOP        ; Loop if index is < 100
        RSUB                ; Exit from subroutine
          :
ZERO    WORD    0
LENGTH  WORD    1
OUTPUT  BYTE    X '05'
RECORD  RESB    100
```

Note: To read and write the data between the devices, the device has to be ready to perform. This is done by using TD (Test device) instruction; status of the device is tested and cc is set to either { < (ready) = (not ready) }
If ready then RD is executed. ⇒ reads 1 byte of data from device into rightmost byte of register A. If the input device is character-oriented (keyboard), the value placed in reg. A is the ASCII code for the character that was read. While WD → o/p device byte is loaded into the rightmost byte of register A and

15. Write a subroutine for SIC/XE to write a RECORD of 100 byte onto output device 05.

```
        JSUB    WRREC   ; Jump to subroutine

WRREC   LDX     #0
        LDT     #100
LOOP    TD      OUTPUT
        JEQ     LOOP
        LDCH    RECORD, X
        WD      OUTPUT
        TIXR    T
        JLT     LOOP
        RSUB
        :
OUTPUT  BYTE    X '05'
RECORD  RESB    100
```

16. Write a subroutine for SIC that will read a record into a buffer. The record may be any length from 1 to 100 byte. The end of the record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH.

```
        JSUB    RDREC
        :
RDREC   LDX     ZERO
RLOOP   TD      INDEV
```

```
        JEQ     RLOOP
        RD      INDEV
        COMP    NULL
        JEQ     EXIT
        STCH    BUFFER,X
        TIX     K100
        JLT     RLOOP
EXIT    STX     LENGTH
        RSUB
        :
        :
        :
ZERO    WORD    0
NULL    WORD    0
K100    WORD    100
INDEV   BYTE    X'F1'
LENGTH  RESW    1
BUFFER  RESB    100
```

17> SIC/XE :

```
        JSUB    RDREC
        :
RDREC   LDX     #0
        LDT     #100
        LDS     #0
RLOOP   TD      INDEV
        JEQ     RLOOP
        RD      INDEV
        COMPR.  A, S
        JEQ     EXIT
        STCH    BUFFER,X
        TIXR    T
        JLT     RLOOP
EXIT    STX     LENGTH
        RSUB
        :
INDEV   BYTE    X'F1'
LENGTH  RESW    1
BUFFER  RESB    100
```

11 To sort an array of 10 words in an ascending order.

```
OUTER   LDX   INDEX ;
        LDS   ARR1, X ;
        LDX   #0
INNER   LDT   ARR1, X
        COMR  S, T
        JLT   LOOP
        JEQ   LOOP
        RMO   S, A
        RMO   T, S
        RMO   A, T
        RMO   X, A
        LDX   INDEX
        STS   ARR1, X
        RMO   A, X
        STT   ARR1, X
LOOP    RMO   X, A
        ADD   #3
        COMP  LENGTH
        RMO   A, X
        JLT   INNER
        LDA   INDEX
        ADD   #3
        COMP  LENGTH
        STA   INDEX
        JLT   OUTER
        :
INDEX   WORD  0
ARR1    RESW  10
LENGTH  WORD  30
```

OR

1. Write a SIC program to copy string 'SYSTEM SOFTWARE' to another string.

```
        LDX   ZERO      ; Initialize X to zero
MOVECH  LDCH  STR1,X    ; X specific indexing
        STCH  STR2,X
        TIX   FIFTEEN   ; increment X and compare
                        with 15
        JLT   MOVECH
        :
STR1    BYTE  C 'SYSTEM SOFTWARE'
STR2    RESB  15
ZERO    WORD  0
FIFTEEN WORD  15
```

Comparison Chart of SIC and SIC/XE machine

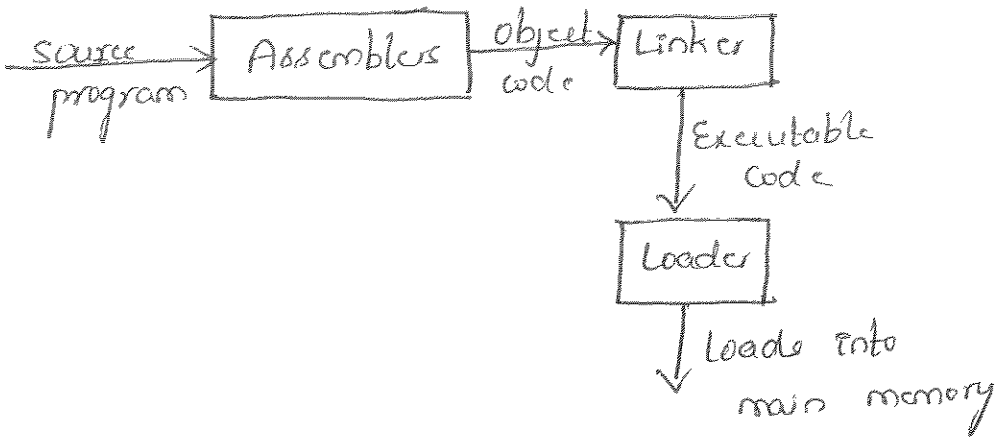
Specification	SIC	SIC/XE
Memory	• **Word size:** 3 bytes (24 bits) • **Total size:** 32,768 bytes (215). Thus any memory address will need at most 15 bits to be referenced ('almost' four hex characters).	• **Word size:** 3 bytes (24 bits) • **Total size:** 32,768 bytes (215). Thus any memory address will need at most 15 bits to be referenced ('almost' four hex characters).
Register	• **Total Registers:** 5 • **Accumulator (A):** Used for most of the operations (number 0) • **Index (X):** Used for indexed addressing (number 1) • **Linkage (L):** Stores return addresses for JSUB (number 2) • **Program Counter (PC):** Address for next instruction (number 8) • **Status Word (SW):** Information and condition codes (number 9).	• **Total Registers:** 9 , same 5 from SIC plus 4 additional ones. • **Base (B):** Used for base-relative addressing (number 3) • **General (S and T):** General use (numbers 4 and 5 resp.) • **Floating Point Accumulator (F):** Used for floating point arithmetic, 48 bits long (number 6)
Instruction Formats	• Only one instruction format of 24 bits (3 bytes / 1 word) • Opcode: first 8 bits, direct translation from the Operation Code Table • Flag (x): next bit indicates address mode (0 direct - 1 indexed) • Address: next 15 bits, indicate address of operand according to address mode.	• Four instruction formats • **Format 1** (1 byte): contains only operation code (straight from table) • **Format 2** (2 bytes): first eight bits for operation code, next four for register 1 and following four for register 2. • The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5). • If the operation uses only one register the last hex digit becomes \0" (ie,TIXR T becomes B850) • **Format 3** (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. • Operation code uses only 6 bits, thus the second hex digit will be a affected by the values of the first two flags (n and i) • The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. • The last flag e indicates the instruction format (0 for 3 and 1 for 4) • **Format 4** (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented

Specification	SIC	SIC/XE
Addressing Modes	• Only two possible addressing modes • **Direct** (x = 0): operand address goes as it is Indexed (x = 1): value to be added to the value stored at the register x to obtain real address of the operand.	• five possible addressing modes plus combinations (see page 11 for examples) • **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format • **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1) • **Immediate** (i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction) • **Indirect** (i = 0, n = 1): The operand value points to an address that holds the address for the operand value • **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.
Assembler Considerations	• Operation code gets translated directly from table (no need to check other bits) • x bit dependent on the addressing mode of the operand. If indexed the code will have to indicate it with \,X" after the operand name (ie. BUFFER,X) • The last 3 hex digits of the address will remain the same, the first hex digit (leftmost) will change if the address is indexed (first bit becomes one, thus the hex digit increases by 8). Ie, if the address of the operand is 124A and the addressing is indexed, the object code will indicate 924A.	• Operation code gets translated directly from table. While the first hex digit remains the same, the second one can change according to the values of the n and i flags. Thus, we can add 1, 2 or 3 to the operation code. • **Direct addressing** is mainly used in extended format (format 4) and is indicated with a \+" before the operand (an indication that the format is 4, which will also make the e flag to be 1). • **Relative:** for Base relative, the instruction BASE will precede the current instruction. • Any other format, except immediate, will be considered Program Counter relative. If the displacement with respect to the PC does not t into the 12 bits, the assembler should try to compute the displacement with respect to the Base register. If neither case works, the instruction should be extended to format 4, where the addressing mode becomes direct.

Specification	SIC	SIC/XE
		• **Immediate addressing** will be indicated by the use of \#" before the operand name/value (ie. #1) • **Indirect addressing** will be indicated by adding the prex \@" to the operand name (ie. @RETADR) • **Indexed addressing** will be indicated the same way as it was for the SIC machine, \,X" after the operand name (ie. BUFFER,X) • Hex digits for the address are not affected by the content of the flags, since the first two flags affect the second digit of the operation code, and the following four make up its own hex digit.

CHAPTER 2
Assemblers

Chapter 2 : Assemblers



Assembler does two functions

1) It converts the mnemonic operation code into their machine language equivalent
2) Converts symbolic labels into their machine address

The design of assembler can be of

1. Convert mnemonic operation codes to their machine language equivalent. Ex: Translate STL to 14
2. Convert symbolic operands to their equivalent machine addresses. Ex: Translate RETADR to 1033
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into their internal machine representation.
   Ex: Translate 'EOF' to 454F46
5. Write the object program and the assembly listing

Different datastructures for assemblers

1. Operation code Table (OPTAB)
2. Symbol Table (SYMTAB)
3. Location Counter Variable (LOCCTR)

I OPERATION CODE TABLE (OPTAB)

a) Contents
- Mnemonic operation codes
- machine language equivalent
- Instruction format and length

b) During pass-1
- Validate op codes
- Find instruction length to increase location counter value (LOCCTR)

c) During pass-2
- Determines the instruction format (3 or 4)
- Translates the operation codes to their machine language equivalents

d) Implementation
- static hash table, easy for searching

Mnemonic name	op code	Format
ADD m	18	3/4
.		
.		
.		
.		

II SYMBOL TABLE (SYMTAB)

a) contents
- label name
- label address
- Flags to indicate error conditions
- data type or length

b) During pass-1
- store label name and assigned address (from LOCCTR) in SYMTAB

c) During pass-2
- symbols used as operands are looked up in SYMTAB

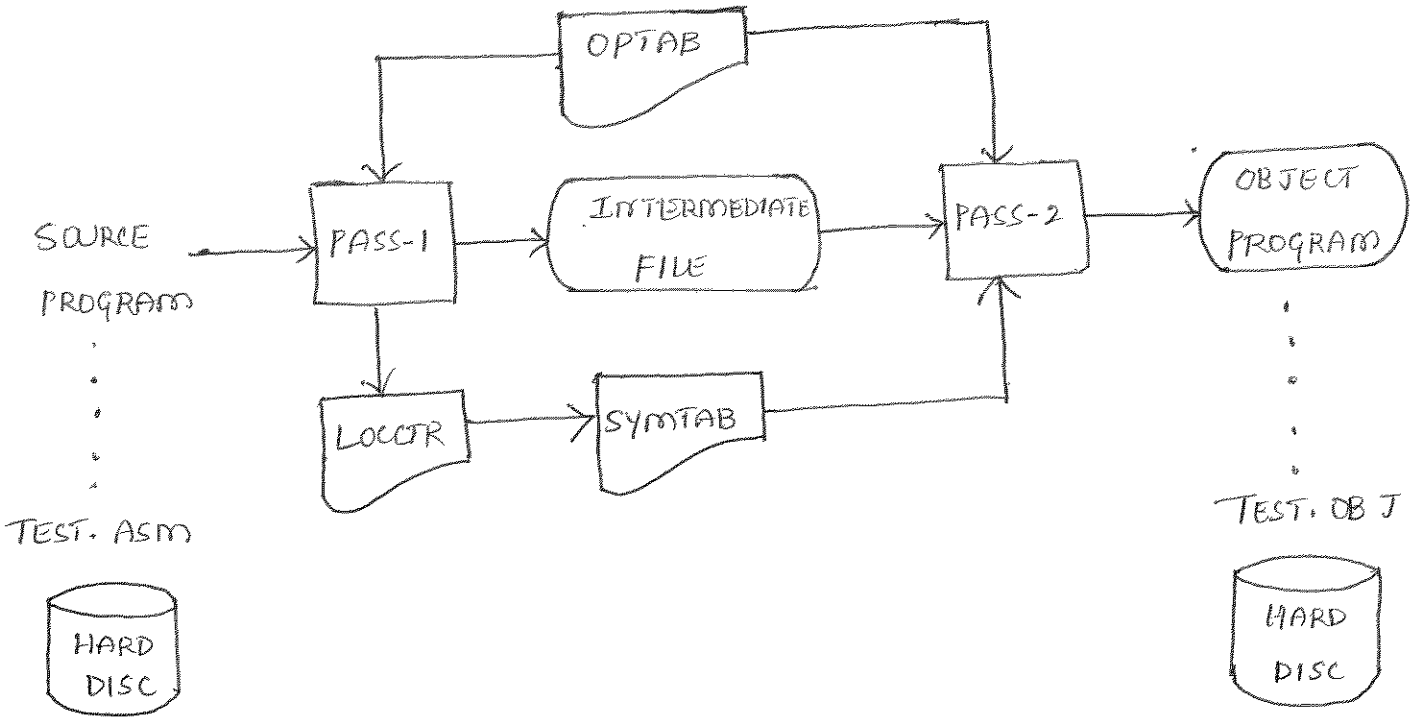
d) Implementation
- A dynamic hash table for efficient insertion and retrieval
- should perform well with non-random keys (LOOP1, LOOP2...)

Label name	value/	Flags	Length
CLOOP	0003		

iii LOCATION COUNTER VARIABLE (LOCCTR)

- Variable accumulated for address assignment ie LOCCTR give the address of the associated labels
- LOCCTR is initialized to be the beginning address specified in the "START" statement
- After each source stmt is processed during pass-1, the instruction length or data area is added to LOCCTR

→ The functionality of assembler looks like this



note : During pass-1, the address of labels is not known ie called forward reference. To resolve this we go for pass-2.

Eg. JEQ RETADR

2.1 Assembler Directives :

1) START — specify name and starting address for the program

2) END — Indicates the end of the source program and optionally specify the first executable instruction in the program

3) BYTE — Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant

4) WORD — Generate one-word integer constant

5) RESB — Reserve the indicated number of bytes for a data area

6) RESW — Reserve the indicated number of words for a data area.

7) LTORG — create a literal pool that contains all of the literal operands used since the previous LTORG or the beginning of the program

8) EQU — Establishes symbolic names that can be used for improved readability inplace of numeric values and also used to define mnemonic names for registers.

9) ORG — Used to indirectly assign values to symbols

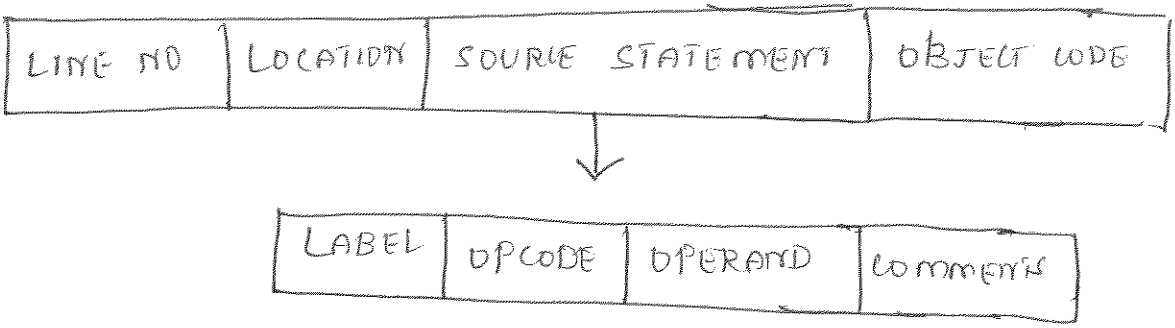
10) USE — Indicates which portion of the source program belong to the various blocks and also indicates a continuation of a previously begun block

11) BASE — Indicates that the base register will contain the address of operand

12) NOBASE — Indicates that the contents of the base register can no longer be relied upon for addressing.

2.1.1 A simple SIC assembler

The usual (general) format to represent the assembly language program for SIC machine with generated assembly code :



where

→ LABEL : An identifier and optional. labels are used to reduce reliance upon programmers remembering where data or code is located. The length of label differs between assemblers.

Ex:- FIRST STL #4096.

→ OPCODE : Is a machine code instruction. It may require additional information like operands (optional)

Ex:- COMP ZERO ; with operand

OR

RSUB ; without operands

→ OPERAND : Is an additional data or information that the opcode requires. Operands are used to specify constants, labels, immediate data, data contained in another register, an address etc

Advantages and disadvantages of assembly language

Advantages :
- → Reduced Errors
- → Faster Translation time
- → changes could be made easier and faster

Disadvantages :
- → many instructions are required to achieve small tasks
- → Source program tend to be large and difficult to follow
- → Programs are machine dependent, thus the complete program has to be rewritten if the hardware is changed
- → The programmer has to have the complete knowledge of the process architecture and instruction set.

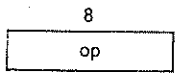
Mnemonic	Format	Opcode	Effect	Notes
ADD m	3/4	18	A ← (A) + (m..m+2)	
ADDF m	3/4	58	F ← (F) + (m..m+5)	X F
ADDR r1,r2	2	90	r2 ← (r2) + (r1)	X
AND m	3/4	40	A ← (A) & (m..m+2)	
CLEAR r1	2	B4	r1 ← 0	X
COMP m	3/4	28	(A) : (m..m+2)	C
COMPF m	3/4	88	(F) : (m..m+5)	X F C
COMPR r1,r2	2	A0	(r1) : (r2)	X C
DIV m	3/4	24	A ← (A) / (m..m+2)	
DIVF m	3/4	64	F ← (F) / (m..m+5)	X F
DIVR r1,r2	2	9C	r2 ← (r2) / (r1)	X
FIX	1	C4	A ← (F) [convert to integer]	X F
FLOAT	1	C0	F ← (A) [convert to floating]	X F
HIO	1	F4	Halt I/O channel number (A)	P X
J m	3/4	3C	PC ← m	
JEQ m	3/4	30	PC ← m if CC set to =	
JGT m	3/4	34	PC ← m if CC set to >	
JLT m	3/4	38	PC ← m if CC set to <	
JSUB m	3/4	48	L ← (PC); PC ← m	
LDA m	3/4	00	A ← (m..m+2)	
LDB m	3/4	68	B ← (m..m+2)	X
LDCH m	3/4	50	A [rightmost byte] ← (m)	
LDF m	3/4	70	F ← (m..m+5)	X F
LDL m	3/4	08	L ← (m..m+2)	
LDS m	3/4	6C	S ← (m..m+2)	X
LDT m	3/4	74	T ← (m..m+2)	X
LDX m	3/4	04	X ← (m..m+2)	
LPS m	3/4	D0	Load processor status from information beginning at address m (see Section 6.2.1)	P X
MUL m	3/4	20	A ← (A) * (m..m+2)	

Mnemonic	Format	Opcode	Effect	Notes
MULF m	3/4	60	F ← (F) * (m..m+5)	X F
MULR r1, r2	2	98	r2 ← (r2) * (r1)	X
NORM	1	C8	F ← (F) [normalized]	X F
OR m	3/4	44	A ← (A) | (m..m+2)	
RD m	3/4	D8	A [rightmost byte] ← data from device specified by (m)	P
RMO r1,r2	2	AC	r2 ← (r1)	X
RSUB	3/4	4C	PC ← (L)	
SHIFTL r1,n	2	A4	r1 ← (r1); left circular shift n bits. {In assembled instruction, r2 = n–1}	X
SHIFTR r1,n	2	A8	r1 ← (r1); right shift n bits, with vacated bit positions set equal to leftmost bit of (r1). {In assembled instruction, r2 = n-1}	X
SIO	1	F0	Start I/O channel number (A); address of channel program is given by (S)	P X
SSK m	3/4	EC	Protection key for address m ← (A) (see Section 6.2.4)	P X
STA m	3/4	0C	m..m+2 ← (A)	
STB m	3/4	78	m..m+2 ← (B)	X
STCH m	3/4	54	m ← (A) [rightmost byte]	
STF m	3/4	80	m..m+5 ← (F)	X F
STI m	3/4	D4	Interval timer value ← (m..m+2) (see Section 6.2.1)	P X
STL m	3/4	14	m..m+2 ← (L)	
STS m	3/4	7C	m..m+2 ← (S)	X
STSW m	3/4	E8	m..m+2 ← (SW)	P
STT m	3/4	84	m..m+2 ← (T)	X
STX m	3/4	10	m..m+2 ← (X)	
SUB m	3/4	1C	A ← (A) – (m..m+2)	
SUBF m	3/4	5C	F ← (F) – (m..m+5)	X F

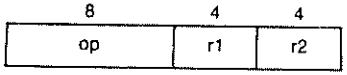
Mnemonic	Format	Opcode	Effect	Notes
SUBR r1,r2	2	94	r2 ← (r2) – (r1)	X
SVC n	2	B0	Generate SVC interrupt. {In assembled instruction, r1 = n}	X
TD m	3/4	E0	Test device specified by (m)	P C
TIO	1	F8	Test I/O channel number (A)	P X C
TIX m	3/4	2C	X ← (X) + 1; (X): (m..m+2)	C
TIXR r1	2	B8	X ← (X) + 1; (X): (r1)	X C
WD m	3/4	DC	Device specified by (m) ← (A) [rightmost byte]	P

Instruction Formats

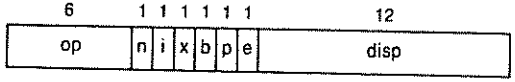
Format 1 (1 byte):



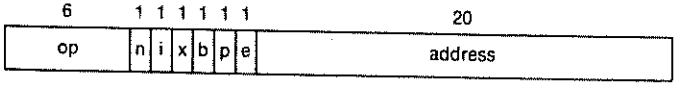
Format 2 (2 bytes):



Format 3 (3 bytes):



Format 4 (4 bytes):



Addressing Modes

The following addressing modes apply to Format 3 and 4 instructions. Combinations of addressing bits not included in this table are treated as errors by the machine. In the description of assembler language notation, *c* indicates a constant between 0 and 4095 (or a memory address known to be in this range); *m* indicates a memory address or a constant value larger than 4095. Further information can be found in Section 1.3.2.

The letters in the Notes column have the following meanings:

- 4 Format 4 instruction
- D Direct-addressing instruction
- A Assembler selects either program-counter relative or base-relative mode
- S Compatible with instruction format for standard SIC machine. Operand value can be between 0 and 32,767 (see Section 1.3.2 for details).

Addressing type	Flag bits n i x b p e	Assembler language notation	Calculation of target address TA	Operand	Notes
Simple	110000	op c	disp	(TA)	D
	110001	+op m	addr	(TA)	4 D
	110010	op m	(PC) + disp	(TA)	A
	110100	op m	(B) + disp	(TA)	A
	111000	op c,X	disp + (X)	(TA)	D
	111001	+op m,X	addr + (X)	(TA)	4 D
	111010	op m,X	(PC) + disp + (X)	(TA)	A
	111100	op m,X	(B) + disp + (X)	(TA)	A
	000- - -	op m	b/p/e/disp	(TA)	D S
	001- - -	op m,X	b/p/e/disp + (X)	(TA)	D S
Indirect	100000	op @c	disp	((TA))	D
	100001	+op @m	addr	((TA))	4 D
	100010	op @m	(PC) + disp	((TA))	A
	100100	op @m	(B) + disp	((TA))	A
Immediate	010000	op #c	disp	TA	D
	010001	+op #m	addr	TA	4 D
	010010	op #m	(PC) + disp	TA	A
	010100	op #m	(B) + disp	TA	A

Line	Source statement			
5	COPY	START	1000	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	ZERO	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	THREE	SET LENGTH = 3
60		STA	LENGTH	
65		JSUB	WRREC	WRITE EOF
70		LDL	RETADR	GET RETURN ADDRESS
75		RSUB		RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
85	THREE	WORD	3	
90	ZERO	WORD	0	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		
120	.			
125	RDREC	LDX	ZERO	CLEAR LOOP COUNTER
130		LDA	ZERO	CLEAR A TO ZERO
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		JEQ	RLOOP	LOOP UNTIL READY
145		RD	INPUT	READ CHARACTER INTO REGISTER A
150		COMP	ZERO	TEST FOR END OF RECORD (X'00')
155		JEQ	EXIT	EXIT LOOP IF EOR
160		STCH	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIX	MAXLEN	LOOP UNLESS MAX LENGTH
170		JLT	RLOOP	HAS BEEN REACHED
175	EXIT	STX	LENGTH	SAVE RECORD LENGTH
180		RSUB		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
190	MAXLEN	WORD	4096	
195	.			
200	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.			
210	WRREC	LDX	ZERO	CLEAR LOOP COUNTER
215	WLOOP	TD	OUTPUT	TEST OUTPUT DEVICE
220		JEQ	WLOOP	LOOP UNTIL READY
225		LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
230		WD	OUTPUT	WRITE CHARACTER
235		TIX	LENGTH	LOOP UNTIL ALL CHARACTERS
240		JLT	WLOOP	HAVE BEEN WRITTEN
245		RSUB		RETURN TO CALLER
250	OUTPUT	BYTE	X'05'	CODE FOR OUTPUT DEVICE
255		END	FIRST	

Figure 2.1 Example of a SIC assembler language program.

Line	Loc		Source statement			Object code
			LABEL	OPCODE	OPERAND	
5	1000		COPY	START	1000	
10	1000	3	FIRST	STL	RETADR	141033
15	1003	3	CLOOP	JSUB	RDREC	482039
20	1006	3		LDA	LENGTH	001036
25	1009	3		COMP	ZERO	281030
30	100C	3		JEQ	ENDFIL	301015
35	100F	3		JSUB	WRREC	482061
40	1012	3		J	CLOOP	3C1003
45	1015	3	ENDFIL	LDA	EOF	00102A
50	1018	3		STA	BUFFER	0C1039
55	101B	3		LDA	THREE	00102D
60	101E	3		STA	LENGTH	0C1036
65	1021	3		JSUB	WRREC	482061
70	1024	3		LDL	RETADR	081033
75	1027	3		RSUB		4C0000
80	102A	3	EOF	BYTE	C'EOF'	454F46
85	102D	3	THREE	WORD	3	000003
90	1030	3	ZERO	WORD	0	000000
95	1033	3	RETADR	RESW	1	
100	1036	3	LENGTH	RESW	1	
105	1039	1000	BUFFER	RESB	4096	
110			.			
115			.	SUBROUTINE TO READ RECORD INTO BUFFER		
120			.			
125	2039	3	RDREC	LDX	ZERO	041030
130	203C	3		LDA	ZERO	001030
135	203F	3	RLOOP	TD	INPUT	E0205D
140	2042	3		JEQ	RLOOP	30203F
145	2045	3		RD	INPUT	D8205D
150	2048	3		COMP	ZERO	281030
155	204B	3		JEQ	EXIT	302057
160	204E	3		STCH	BUFFER,X	549039
165	2051	3		TIX	MAXLEN	2C205E
170	2054	3		JLT	RLOOP	38203F
175	2057	3	EXIT	STX	LENGTH	101036
180	205A	3		RSUB		4C0000
185	205D	1	INPUT	BYTE	X'F1'	F1
190	205E	3	MAXLEN	WORD	4096	001000
195			.			
200			.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205			.			
210	2061	3	WRREC	LDX	ZERO	041030
215	2064	3	WLOOP	TD	OUTPUT	E02079
220	2067	3		JEQ	WLOOP	302064
225	206A	3		LDCH	BUFFER,X	509039
230	206D	3		WD	OUTPUT	DC2079
235	2070	3		TIX	LENGTH	2C1036
240	2073	3		JLT	WLOOP	382064
245	2076	3		RSUB		4C0000
250	2079	1	OUTPUT	BYTE	X'05'	05
255	207A			END	FIRST	

Figure 2.2 Program from Fig. 2.1 with object code.

The following program contains a main routine that reads records from an input device (code: F1) and copies them to output device (code: 05).

main function calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write record from the buffer to output device.

Each subroutine transfers ~~one~~ record one character at a time because only I/O instructions available are RD and WD.

Since the I/O rates of two devices (disk and a printing terminal) may be different, a buffer is used. The end of each record is marked with a null character ie 00 (in hexadecimal). If a record is longer than length of buffer (4096 bytes) then only the first 4096 bytes are copied. The end of file to be copied is indicated by a zero length record.

The program indicates EOF (End of File) on output device when the zero length record (ie end of file) is detected. The program terminates by executing the RSUB instruction since it was called by JSUB instruction.

Procedure to generate object code and object Program (Intermediate File).

Note: We have assumed that the program starts at address 1000.

1) First and foremost write the LOCCTR address
   → START 1000
   → Add 3 byte for each instructions. (∵ instruction format for SIC mc is 24bits. ie 3 byte)
   → BYTE C 'EOF' : count the length of constant and add those many byte
   → RESW 2000 : then it should be 2000x3 byte = 6000B = 1770 (H) added to previous address
   → RESW 1 : add just 3 byte
   → RESB 2000 : convert 2000 to hexadecimal (~~7D0~~ ie 7D0 byte and add
   RESB 4096 : 4096 → $1000_{(16)}$ is added to previous value
   → WORD 3 or WORD 0 → 3 byte added.

2) Start creating the object code.
   → Convert mnemonic operation code to their machine language equivalent. Ex: STL to 14
   → Convert symbolic operands to their equivalent machine address Ex BUFADR to 1033 (forward reference)

→ Build machine instruction in proper format

a) Direct addressing : X=0 : TA = address

b) Indexed addressing : X=1 : TA = address + (X)

→ indicated by symbol 'X'

Ex:- STCH BUFFER, X :→ Line no. 160

→ Convert the data constants into their machine representation. Ex:- EOF to 454F46 (Line no 80)

(A = 65, a = 97)

↳ $(41)_{16}$ ↳ $(61)_{16}$

3) Write the object program (Intermediate File)

→ object program contains three type of records.

a) Header Record b) Text Record c) End Record.

a) Header Record : Contains program name, starting address and length of program.

column 1	H
col. 2-7	Program name
col. 8-13	starting address of object Program (Hexadecimal)
col. 14-19	Length of object program in bytes (Hexadecimal)

Ex:- S 1 → name of program COPY START 1000 ↳ starting address

:

256 207A END

Length of program = Last address - startingaddress

= 207A - 1000 = 107A

∴ H^COPY^001000^00107A (Header Record)

b) Text Record :

Text record contains the translated instructions (machine code) and data of the program together with an indication of addresses where these are to be loaded.

col. 1	T
col. 2-7	starting address for object code in this record (Hexadecimal)
col 8-9	Length of object code in this record in bytes (hexadecimal)
col 10-69	object code represented in hexadecimal (2 columns per byte of object code)

↓ note :

60 columns
⇒ 10 words ⇒ 30 bytes ⇒ $(1E)_{16}$ ← Length of object code

Ex:

10	10000	FIRST	STL	RETADDR	141033 (3 bytes each) } 10 words
⋮					
55	101B		LDA	THREE	001020

Text record →

T^001000^1E^141033^ . . . . . ^00102D

↳ marker for separation

c) End Record :

End record marks the end of the object program and specifies the address in the program where execution is to begin. If no operand is ~~Used~~ specified then the address of the first executable instruction is used.

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

Ex: 10 1000 FIRST STL RETADR 141033

:

255 END FIRST

End record → E^001000.

Let us start for the given program in Fig. 2.1

Given opcodes

STL - 14	J - 3C	LDX - 04	JLT - 38	F - 56
JSUB - 48	STA - 0C	TD - E0	LDCH - 50	
LDA - 00	STX - 10	RD - D8	WD - DC	
COMP - 28	LDL - 08	STCH - 54	E - 45	
JEQ - 30	RSUB - 4C	TIX - 2C	0 - 4F	

(1) start incrementing LOCCTR

Initially it is 1000.

→ start adding 3 byte each time from line no. 5 to 105

→ 105 1039 BUFFER RESB 4096

↳ convert to Hexadecimal ie $(4096)_{16} = 1000$

∴ add 1000 byte to 1039 = 2039

∴ line no. 125 starts at 2039th address continue till line no. 195.

→ 185 205D INPUT BYTE X 'F1' F1 → 1 Byte

∴ add only 1 byte to 205D ⇒ 205E at line no. 190.

→ 190 205E MAXLEN WORD 4096 001000

↳ word so 3 byte not 1000 byte

∴ 3 byte added to 205E ⇒ 2061 at line no. 210

→ 210 2061 CLOOP LDX ZERO 041030.

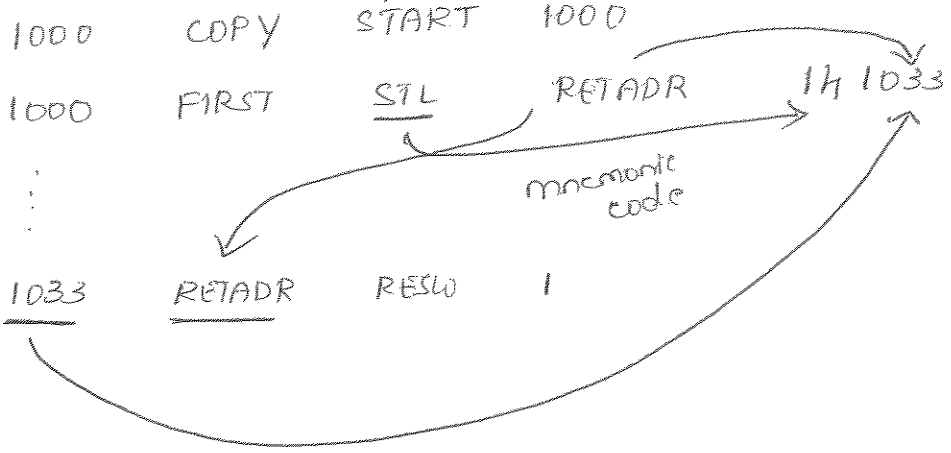
continue the same till end.

→ 255 207A END FIRST

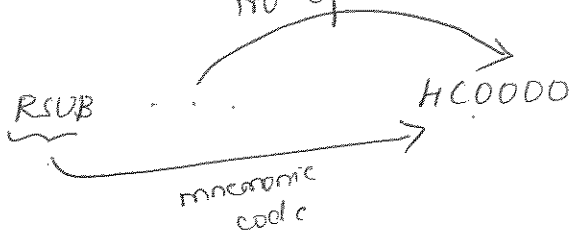
③ object code for each line.

↳ Every line is object addressing except line 00 160 and 225

	Line	LOCCTR	label	opcode	operand	Object code
(i)	5	1000	COPY	START	1000	
	10	1000	FIRST	STL	RETADR	17 1033
	⋮					
	95	1033	RETADR	RESW	1	



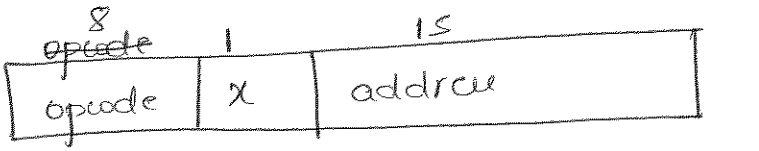
(ii) 75 1027 RSUB ... HC0000

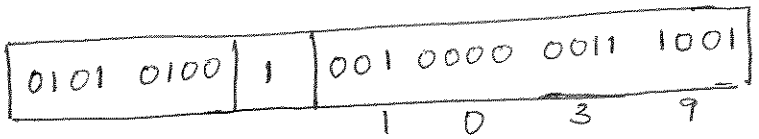


(iii) 160 20HE STCH BUFFER, X ↳ indicates indexed addressing

5H

address of buffer = 1039





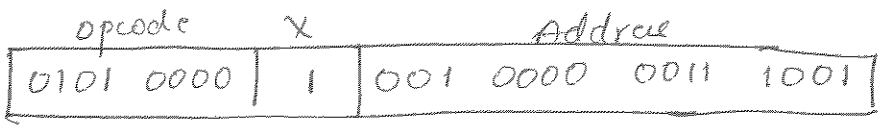
5 H 9 0 3 9

∴ 160 20HE STCH BUFFER, X 5H9039

(iv) Same for line number 225

225 206A LDCH BUFFER, X

LDCH → 50 BUFFER, X → 1039



5 0 9 0 3 9

∴ 225 206A LDCH BUFFER, X 509039

(b) Object program for Fig 2.2

H^COPY. ^001000^00107A

T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^ ··· ^001022D

T^00101E^15^0C1036^482061^081033^4C0000^4545F6^000003^000000

T^002039^1E^0H1030^001030^ED205D^30203F^D8205D^281030^302057^ ··· ^382031F

T^00205T^1C^101036^4C0000^F1^001000^0H1030^ED2079^302064^ ··· ^2C1036

T^002073^07^382064^4C0000^05

E^001000.

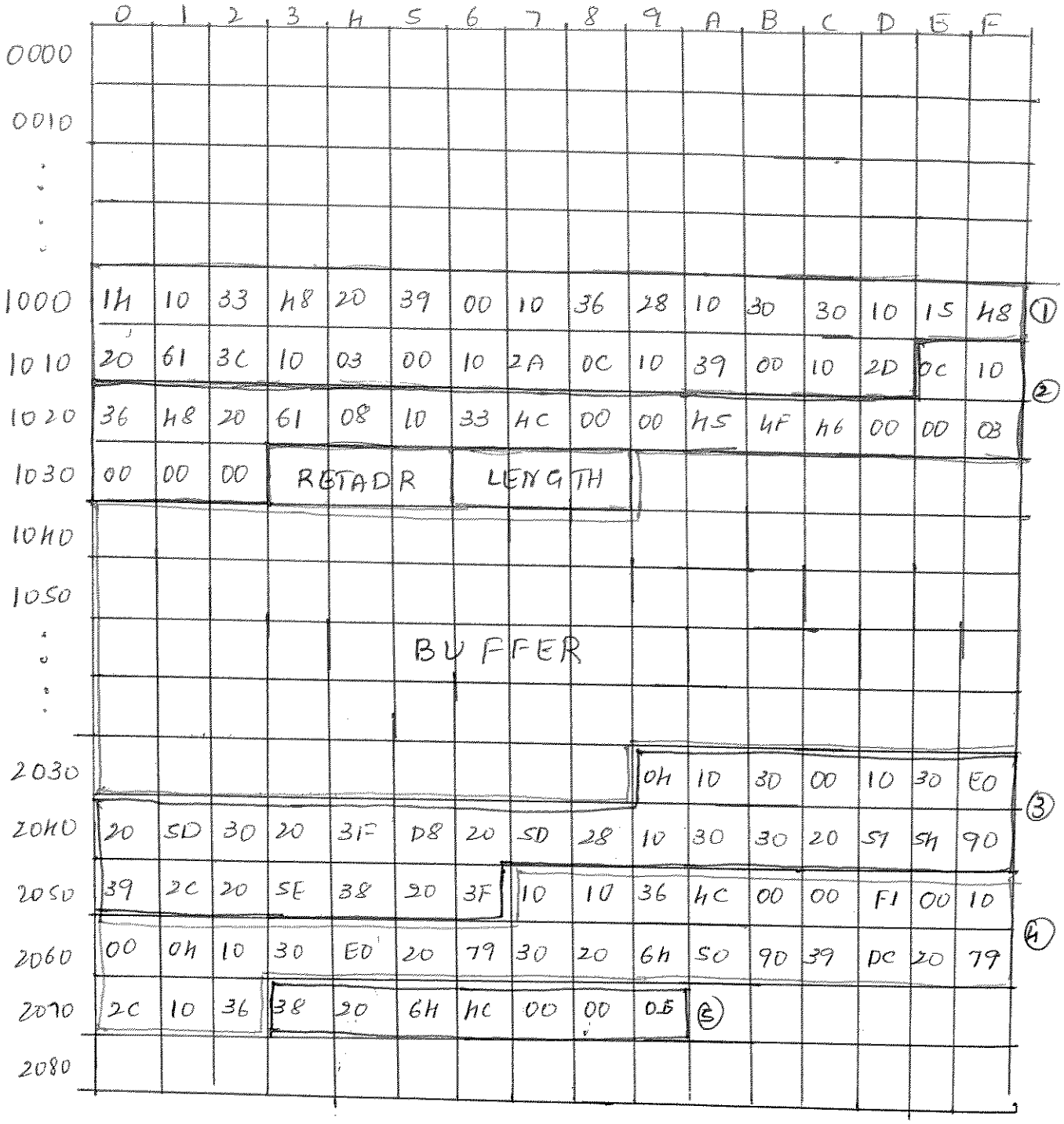
(2)

SYMBOL TABLE

Symbol name	value of the symbol
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	102A
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039
RLOOP	203F

Symbol Name	Value
EXIT	2057
INPUT	205D
MAXLEN	205E
WRREC	2061
WLOOP	2064
OUTPUT	2079

loader loads into main memory



Functions of Pass-I and Pass-II

Pass 1 :

→ Assign addresses to all statements in the program

→ Save the values (addresses) assigned to all labels for use in Pass 2

→ Perform some processing of assembler directives (includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW etc)

Pass 2 :

→ Assemble instructions (Translating operation codes and looking up addresses)

→ Generate data values defined by BYTE, WORD etc

→ Perform processing of assembler directives not done during pass-1

→ Write the object program and the assembly listing.

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
                        insert (LABEL,LOCCTR) into SYMTAB
                  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}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

Pass 2:

```
begin
  read first 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 first Text record
  while OPCODE ≠ 'END' do
     begin
        if this is not a 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 will not fit into the current Text record then
                 begin
                    write Text record to object program
                    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}
```

Figure 2.4(b) Algorithm for Pass 2 of assembler.

2.9 Machine Dependent Assembler Features

• Here we consider an example of SIC/XE machine.

→ As we know already, SIC/XE has

a) Registers : A X L B S T F PC SW
(0 1 2 3 4 5 6 8 9)

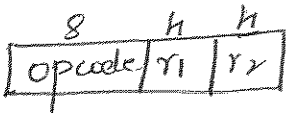
b) Data formats : Integer : 3 byte
Character : 1 byte
Float : 6 byte

c) Instruction Formats :

Format 1 : 1 byte

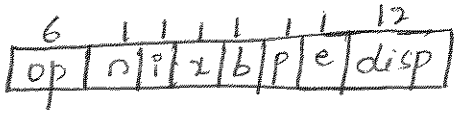
Ex: FLOAT, FIX

Format 2 : 2 byte



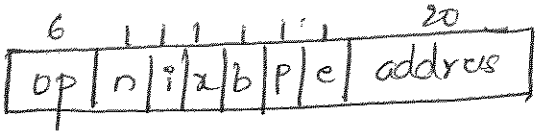
Ex: ADDR A,X

Format 3 : 3 byte



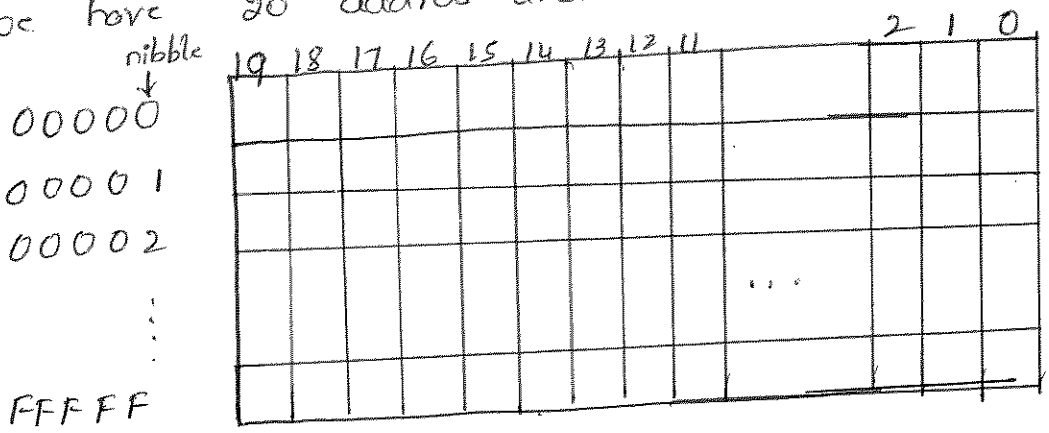
Ex: STL RETADR

Format 4 : 4 byte



Ex: +JSUB RDREC

→ we have 20 address lines ∴ we can have 2^{20} addresses



d) Addressing modes are determined based on 6 bits

n i x b p e

(i) →

n	i	x	Addressing mode
1	0		Indirect addressing
0	1		Immediate
1	1		not immediate, not indirect
0	0		Simple addressing
		1	Indexed addressing
		0	Direct addressing

**

(ii)

b	p	e	Addressing mode
0	1		Program Counter Relative
1	0		Base relative
1	1		Invalid (can't be set)
0	0		NO PC relative, no base relative
		1	Format 4 instruction
		0	Format 3 instructions

**

Different addressing mode notations

1) Indirect Addressing : @
2) Immediate Addressing : #
3) Extended Format : +
4) Indexed Addressing : operand, X
5) character string : C' '
6) Base - Register : BASE
7) Current value of PC : *

→ The addressing priority are as follows

a) PC relative addressing : $-2048 \le disp \le 2047$
$(FFFFF800 \le disp \le 7FF)$

b) Base relative addressing : $0 \le disp \le 4095$
$(0 \le disp \le FFF)$

c) Extended Instruction Format :

note: negative numbers are represented in 2's compliment.

Procedure to create object code for SIC/XE program

1) Write the LOCCTR address for each instruction in the program.

→ if operand field is
(i) memory address → Format 3 ⇒ Add 3 byte
(ii) Register - Register → Format 2 ⇒ Add 2 byte
(iii) + before operand → Format 4 ⇒ Add 4 byte

→ if it is RESW 2000
. 2000 x 3 byte $= (6000)_d = (1770)_H$ ⇒ Add these many byte to previous address.
. Multiplication by 3 ∵ each word is 3 byte

→ RESW 1 ⇒ Add just 3 byte

→ RESB 2000 ⇒ Add 2000 byte in hexadecimal
ie $(2000)_d = (7D0)_H$

→ RESB 4096

$(4096)_d = (1000)_H$ ⇒ Add 1000 byte

→ BYTE C 'EOF' ⇒ Count the length of constant and add those many bytes

→ Enter the labels onto SYMTAB (pass 1)

2) Once we are done with LOCCTR calculation and then finding program length = End Address − start address

3) Now start creating the object code (Pass 2) based on different addressing mode and set corresponding bits and calculate displacement

→ For extended format, displacement = address

→ For Reg-to-Reg instruction, write the opcode address followed by register numbers.

Ex: CLEAR X ⇒ B410 (Format 2)

(1) → Number of X register in the list

→ For PC relative, disp = TA − PC

→ For Base relative, disp = TA − (B)

```
Line        Source statement

  5     COPY      START   0             COPY FILE FROM INPUT TO OUTPUT
 10     FIRST     STL     RETADR        SAVE RETURN ADDRESS
 12               LDB     #LENGTH       ESTABLISH BASE REGISTER
 13               BASE    LENGTH
 15     CLOOP    +JSUB    RDREC         READ INPUT RECORD
 20               LDA     LENGTH        TEST FOR EOF (LENGTH = 0)
 25               COMP    #0
 30               JEQ     ENDFIL        EXIT IF EOF FOUND
 35              +JSUB    WRREC         WRITE OUTPUT RECORD
 40               J       CLOOP         LOOP
 45     ENDFIL    LDA     EOF           INSERT END OF FILE MARKER
 50               STA     BUFFER
 55               LDA     #3            SET LENGTH = 3
 60               STA     LENGTH
 65              +JSUB    WRREC         WRITE EOF
 70               J       @RETADR       RETURN TO CALLER
 80     EOF       BYTE    C'EOF'
 95     RETADR    RESW    1
100     LENGTH    RESW    1             LENGTH OF RECORD
105     BUFFER    RESB    4096          4096-BYTE BUFFER AREA
110     .
115     .         SUBROUTINE TO READ RECORD INTO BUFFER
120     .
125     RDREC     CLEAR   X             CLEAR LOOP COUNTER
130               CLEAR   A             CLEAR A TO ZERO
132               CLEAR   S             CLEAR S TO ZERO
133              +LDT     #4096
135     RLOOP     TD      INPUT         TEST INPUT DEVICE
140               JEQ     RLOOP         LOOP UNTIL READY
145               RD      INPUT         READ CHARACTER INTO REGISTER A
150               COMPR   A,S           TEST FOR END OF RECORD (X'00')
155               JEQ     EXIT          EXIT LOOP IF EOR
160               STCH    BUFFER,X      STORE CHARACTER IN BUFFER
165               TIXR    T             LOOP UNLESS MAX LENGTH
170               JLT     RLOOP           HAS BEEN REACHED
175     EXIT      STX     LENGTH        SAVE RECORD LENGTH
180               RSUB                  RETURN TO CALLER
185     INPUT     BYTE    X'F1'         CODE FOR INPUT DEVICE
195     .
200     .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC     CLEAR   X             CLEAR LOOP COUNTER
212               LDT     LENGTH
215     WLOOP     TD      OUTPUT        TEST OUTPUT DEVICE
220               JEQ     WLOOP         LOOP UNTIL READY
225               LDCH    BUFFER,X      GET CHARACTER FROM BUFFER
230               WD      OUTPUT        WRITE CHARACTER
235               TIXR    T             LOOP UNTIL ALL CHARACTERS
240               JLT     WLOOP           HAVE BEEN WRITTEN
245               RSUB                  RETURN TO CALLER
250     OUTPUT    BYTE    X'05'         CODE FOR OUTPUT DEVICE
255               END     FIRST
```

Figure 2.5 Example of a SIC/XE program.

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
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 2.6 Program from Fig. 2.5 with object code.

Consider the example of figure 2.5

1) Add the length of each instruction and add it to LOCCTR and find the program length

Program length = end address - start address

= 1077 - 0000 = 1077

2) Create the symbol table

pass 1

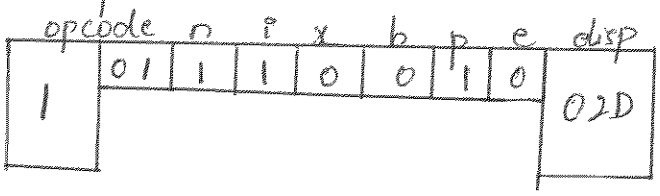
Symbol name	PC value
FIRST	0000
CLOOP	0006
ENDFIL	001A
EOF	002D
RETADR	0030
LENGTH	0033
BUFFER	0036
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WRREC	105D
WLOOP	1062
OUTPUT	1076

3) start creating object code (pass-2)

10 0000 FIRST STL RETADR (Format 3 ∵ oprd is memory address)

14 0030

By default assembler uses PC relative addressing



→ TA = PC + disp

↳ Displacement = TA − PC

= RETADR − LOCCTR (location of next instn to be executed)

= 0030 − 0003 = 002D ∵ format 3 displacement is 12 bits

↳ 02D is within range of $-2048 \le disp \le 2047$

↳ opcode is 6 bits (4+2 ⇒ 1 nibble + 2 bits)

⇒ last 2 bits can be represented by 4 bits but always last 2 bits are "zero"

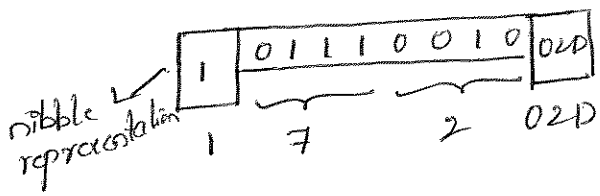
Ex:- 4 ⇒ 0100
only 2 bits

C ⇒ 1100
C

↳ not immediate, not indirect so set n=1, i=1

↳ not indexed x=0, not base relative b=0 but it is pc relative p=1, not format 4 e=0

↳ write the whole instruction's object code ie



STL RETADR 17202D

12 0003 LDB #LENGTH

LDB = 68

LENGTH = 0033

→ opcode for LDB = 68

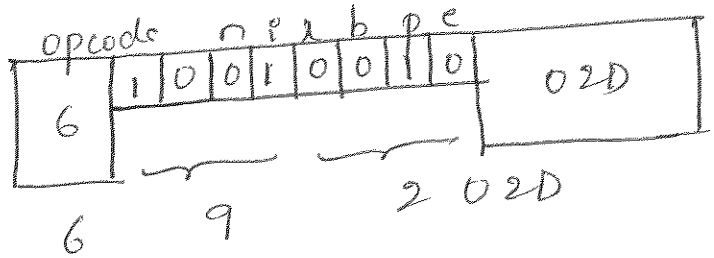
→ it is imm calculate disp.

TA = PC + disp

disp = TA − PC = 0033 − 0006 = ∅02D

→ Pc relative ∵ operand is memory address

→ it is immediate so i=1



LDB #LENGTH ⇒ 69202D

15 0006 CLOOP +JSUB RDREC → Format 4

+JSUB = 48

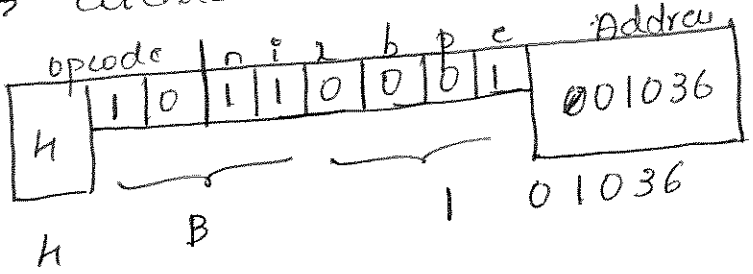
RDREC = 0030

→ disp = (operand) ∵ it is extended format (F4)

= ∅01036 (20 bits)

→ not immediate, not indirect n=1, i=1.

→ extended e=1



CLOOP +JSUB RDREC ⇒ 4B101036

30 000A LDA LENGTH → Format 3

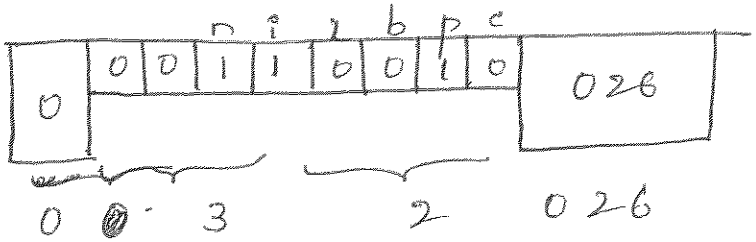
00 0033

→ PC relative, disp = TA − PC

P=1 = 0033 − 000D = 026

→ not immediate, not indirect, not indexed so

n=1, i=1, x=0



LDA LENGTH ⇒ 032026

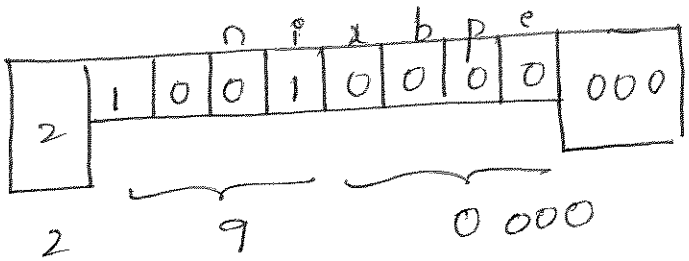
35 000D COMP #0

28

→ ~~immediate~~ not PC relative because operand is direct value but not memory address.

∴ displacement = operand = 000

→ immediate addressing n=0, i=1, b=0, p=0

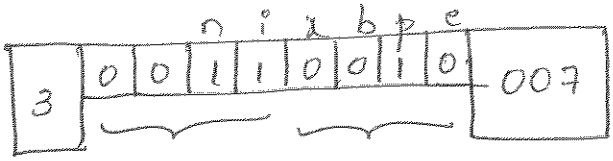


COMP #0 ⇒ 290000

30 0010 JEQ EOFFIL ⇒ Format 3
30 001A

• Pc relative ∴ disp = TA − PC
= 001A − 0013 = 007
within range

• not immediate, not indirect n=1, i=1

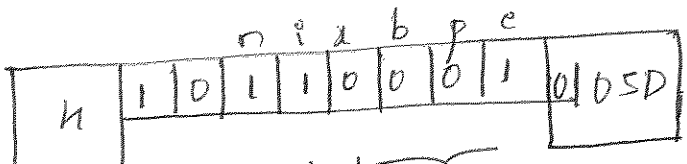


3 3 2 007

∴ JEQ EOFFIL ⇒ 332007

35/65 0013 +JSUB WRREC ⇒ Format 4
4B 1050

• Displacement = address of operand
= 1050
within



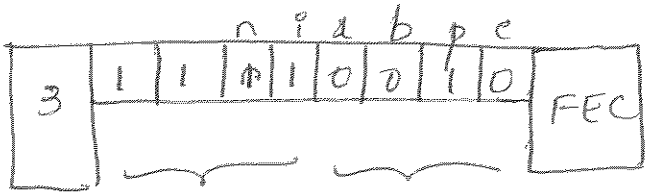
4 B 1050

+JSUB WRREC ⇒ 4B1050

40 0017 J CLOOP ⇒ Format 3
3C 0006

disp = TA − PC = 0006 − 001A
= −14 (it take 2's complement)
= FEC

- it is Pc relative $P=1$
- not immediate, not indirect $n=0, i=1$

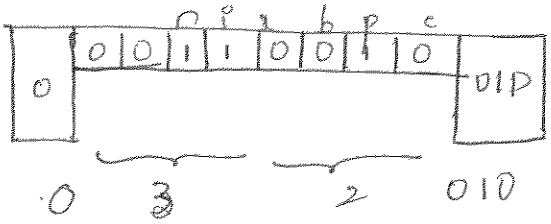


object code : 3 F 2FEC

H5 001A ENDFIL LDA EOF ⇒ format 3 (PC relative)

LDA: 00, EOF: 002D

disp = TA − PC = 002D − 001D = 0010

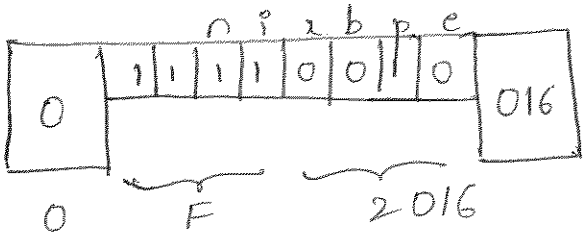


LDA EOF ⇒ 032010

S0 001D STA BUFFER ⇒ format 3 (PC relative)

STA: 0C, BUFFER: 0036

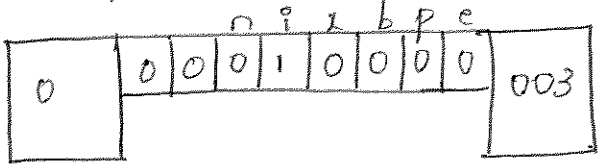
disp = TA − PC = 0036 − 0020 = 0016



∴ STA BUFFER ⇒ 0F2016

S5 0071 LDA #3 ⇒ immediate address

disp = 003

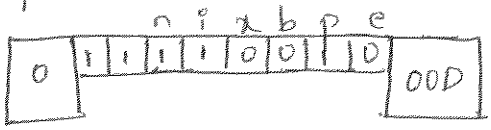


LDA #3 ⇒ 010003

60 0023 STA LENGTH ⇒ Format 3 (pc relative)

0C 0033

disp = TA − PC = 0033 − 0026 = 000D

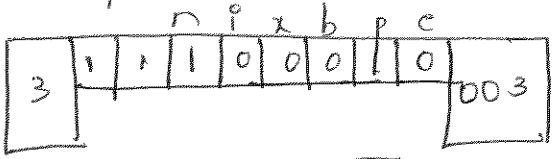


0 F 200D ⇒ 0F200D

70 002A J @RETADR ⇒ Format-3 - Indirect

3C 0030

disp = TA − PC = 0030 − 002D = 0003



3 E 2 003 ⇒ 3E2003

80 002D EOF BYTE C 'EOF'

⇒ Convert EOF to hexadecimal ascii value

E — 45
O — 4F
F — 46

125 1036 RDREC CLEAR X

BH

A X L B S T F PC SW
0 1 2 3 4 5 6 8 9

⇒ BH10 ↳ this is not accumulator

⇒ only 2 bytes since it is register-to-register mode

130 1038 CLEAR A ⇒ BH00

BH

132 103A CLEAR S ⇒ BH40

BH

150 1049 COMPR A,S ⇒ A004

(COMPR = A0)

165 1051 TIXR T ⇒ B850

(TIXR = B8)

235 105E TIXR T ⇒ B850

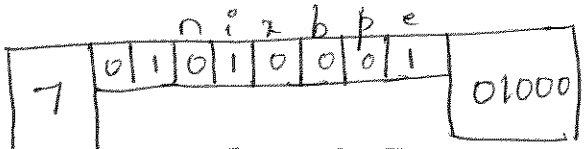
(TIXR = B8)

210 1053 CORREC CLEAR X ⇒ B410

(CLEAR = B4)

133 1036 +LDT #4096 ⇒ Format 4 & Immediate addressing

disp = $(4096)_{16}$ = 01000

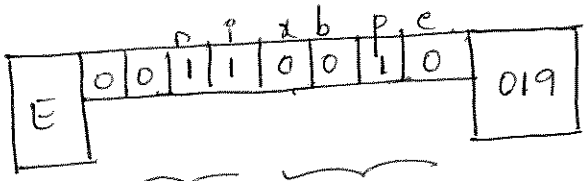


7 5 101000 ⇒ 75101000

135 1040 RLOOP TD INPUT ⇒ Format 3 + PC relative

(TD = E0, INPUT = 105C)

disp = TA − PC = 105C − 1043 = 0019

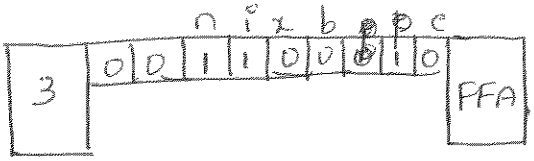


E 3 2019 ⇒ E32019

1H0 1043 JEQ RLOOP ⇒ Format 3 + PC relativ

JEQ: 30 RLOOP: 1040

disp = TA − PC = 1040 − 1046 = −6



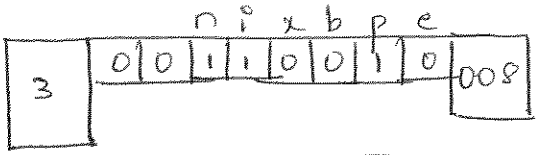
2's complement ⇓ FFA

3 3 2 FFA ⇒ 332FFA

155 104B JEQ EXIT ⇒ Format 3 + PC relativ

JEQ: 30 EXIT: 1056

disp = TA − PC = 1056 − 104E = 008

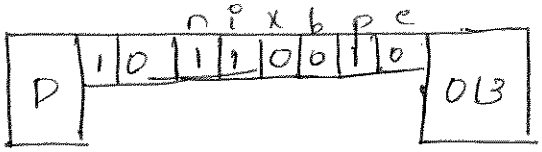


3 3 2 008 ⇒ 332008

1H5 1046 RD INPUT ⇒ Format 3 + PC relativ

RD: D8 INPUT: 105C

disp = TA − PC = 105C − 1049 = 013



D B 2 013 ⇒ DB2013

160 1044 STCH BUFFER,X ⇒ Indexed + PC relative

↑* STCH → 54 BUFFER → 0036

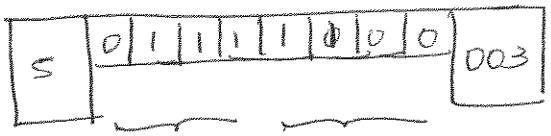
disp = TA − PC = 0036 − 1051 = −101B

= EFF5 (H123)$_{10}$ > 2047

∴ it is not pc relative, go for base relative

disp = TA − B = BUFFER − B (Length is stored in base register at 0033)

= 0036 − 0033 = 003

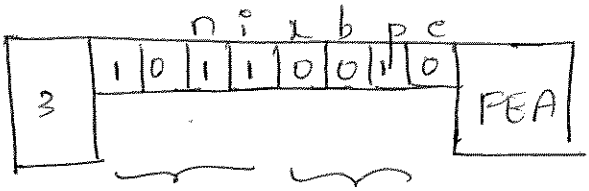


5 7 C 003 ⇒ 57C003

170 1053 JLT RLOOP ⇒ Format 3 + PC relative

JLT → 38 RLOOP → 1040

disp = TA − PC = 1040 − 1056 = FEA



3 B 2 FEA ⇒ 3B2FEA

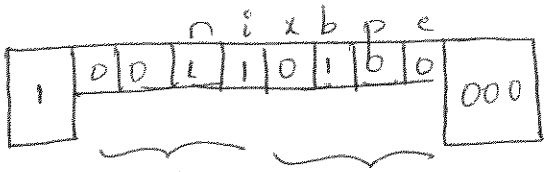
175 1056 EXIT STX LENGTH ⇒ Format 3 + PC relative

** 10 0033

disp = TA − PC = 0033 − 1059 = EFDA > 2047

∴ go for base relative mode

disp = TA − (B) = 0033 − 0033 = 0000

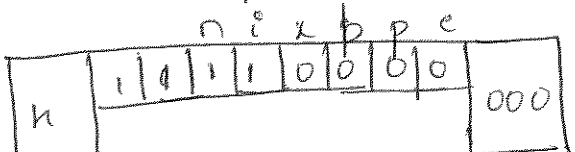


1 3 4000 ⇒ 134000

180 1059 RSUB ⇒ Format 3

4C

no displacement ∴ no operand.



4 F 0000 ⇒ 4F0000

185 105C INPUT BYTE X'F1'

⇒ character string ∴ store as it is

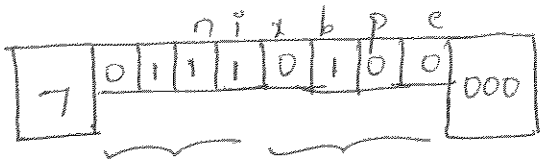
912 105F LDT LENGTH ⇒ Format 3

** 74 0033

disp = TA − PC = 0033 − 1062 = EFD1 (4443) > 2047

∴ go for base relative

disp = TA − LB) = 0033 − 0033 = 0000

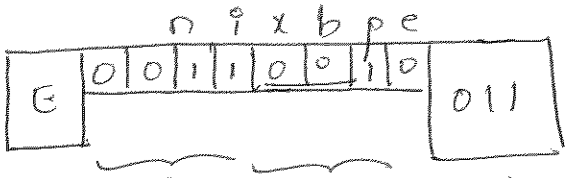


7 7 4000 ⇒ 774000

315 1062 WLOOP TD OUTPUT ⇒ Format 3 + PC relative

EO 1076

disp = TA − PC = 1076 − 1065 = 011

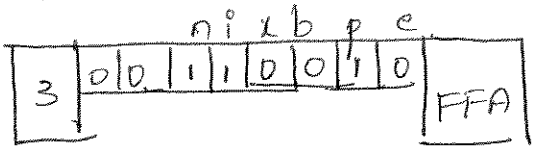


E 3 2011 ⇒ E32011

220 1065 JEQ WLOOP ⇒ Format 3 + PC relative

30 1062

disp = TA − PC = 1062 − 1068 = FFA



3 3 2FFA ⇒ 332FFA

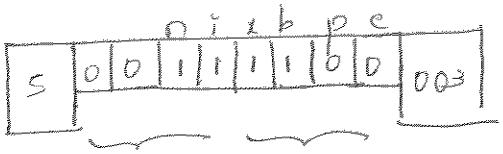
225 1068 LDCH BUFFER, X ⇒ indexed

50 0036

disp = TA − PC = 0036 − 106B = −1035 > 2047

∴ go for base relative mode

disp = TA − B = 0036 − 0033 = 0003

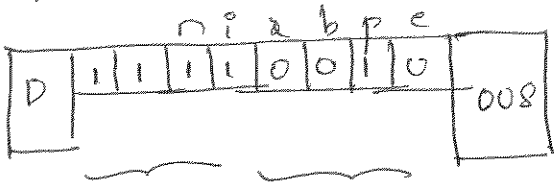


5 3 C003 ⇒ 53C003

230 · 106B LOD OUTPUT ⇒ F3 + Pc relative

LOD: DC
OUTPUT: 1076

disp = TA − PC = 1076 − 106E = 0008

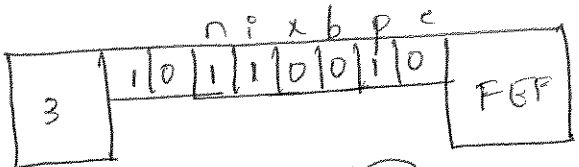


D F 2008 ⇒ DF2008

240 1070 JLT LOOP ⇒ Format 3 + PC relative

JLT: 38
LOOP: 1062

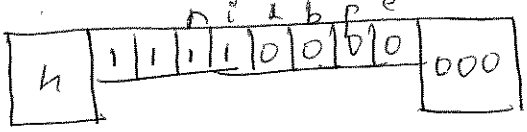
disp = TA − PC = 1062 − 1073 = FEF



3 B 2FEF ⇒ 3B2FEF

245 1073 RSUB ⇒ Format 3

RSUB: 4C



4 F 0000 ⇒ 4F0000

250 1076 OUTPUT BYTE X '05'

⇒ character string ∴ store as it is ⇒ 05

4) object program

H^COPY ^000000^001077

T^000000^1D^172027^692027^4B1010^3F2FEC^032010

T^00001D^13^0F2016^010003^0F2027^4B1010^3E2003^4F4F6

T^001036^1D^B410^B400^B440^754101000^E32019^332FFA^DB2013^A004^332FFA^DB2013^A004^332008^57C003^B850

T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850

T^001070^07^3B2FEF^4F0000^05

E^000000

Loading into memory

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000	17	20	27	69	20	27	4B	10	10	36	03	20	26	29	00	00	①
0010	33	20	07	4B	10	10	5D	3F	2F	EC	03	20	10	0F	20	16	
0020	01	00	03	0F	20	0D	4B	10	10	5D	3E	20	03	4F	4F	46	②
0030	RETADR			LENGTH													
:							BUFFER										
1030							Bh	10	Bh	00	Bh	h0	75	10	10	00	
10H0	E3	20	19	33	2F	FA	DB	20	13	A0	0h	33	20	08	57	C0	③
1050	03	B8	50	3B	2F	EA	13	40	00	4F	00	00	F1	Bh	10	77	
1060	40	00	E3	20	11	33	2F	FA	53	C0	03	DF	20	08	B8	50	④
1070	3B	2F	EF	4F	00	00	05	⑤									

1. Generate the complete object program for the following assembly level program.

CLEAR - B4, LDA - 00, LDB - 68, ADD - 18, TIX - 2C,

JLT - 38 STA - 0C

PASS-I	LENGTH	LABEL	OPCODE	OPERAND	PASS-II
0000		SUM	START	0	
0000	2		CLEAR	X	B410
0002	3		LDA	#0	010000
0005	4		+LDB	#TOTAL	69101789
			BASE	TOTAL	
0009	3	LOOP	ADD	TABLE,X	1BA00D
000C	3		TIX	COUNT	2F2007
000F	3		JLT	LOOP	3F2FF7
0012	4		+STA	TOTAL	0F101789
0016	3	COUNT	RESW	1	
0019	1770	TABLE	RESW	2000 (1770)H	
1789	3	TOTAL	RESW	1	
178C			END	FIRST	

RESW 2000 $\Rightarrow$ 2000x3 = (6000)$_d$ bytes = $(1770)_H$

$\therefore$ 0019 + 1770 = 1789

Program Length = 178C - 0000 = 178C

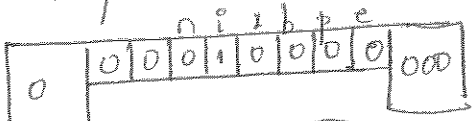
1) 0000 CLEAR X (Register-to-Register)

⇒ directly opcode with register numbers

⇒ B410

2) 0002 LDA #0 ⇒ Immediate Addressing

disp = 000

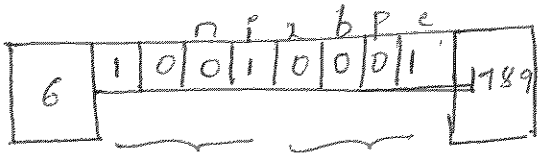


0 01 0 000 ⇒ 010000

3) 0005 +LDB #TOTAL ⇒ Extended & immediate - Format 4 + immediate with PC relative

disp = operandaddress

= 1789



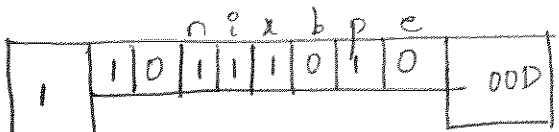
6 9 1 01789 ⇒ 69101789

4) 0009 LOOP ADD TABLE,X ⇒ indexed with PC relative

TA = PC + disp

disp = TA − PC

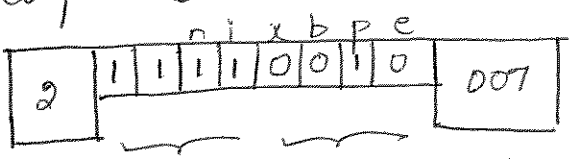
= 0019 − 000C = 00D



1 B A 00D ⇒ 1BA00D

5) 000C TIX COUNT ⇒ Format-3

disp = 0016 − 000F = 007



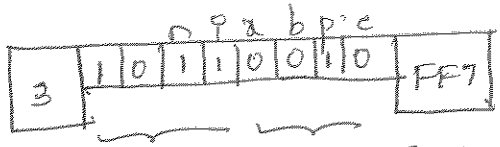
2 F 2 007 ⇒ 2F2007

6) 000F JLT LOOP ⟹ Format-3 PC relative

disp = TA − PC

= 0009 − 0012 = FF7

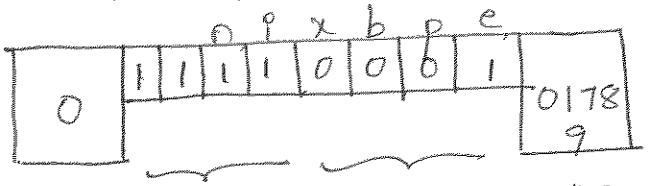
within range ⟵ −2048 ≤ FF7 ≤ 2047



3 B 2 FF7 ⟹ 3F2FF7

7) 0012 +STA TOTAL ⟹ Format 4

disp = operand value = 1789



0 F 1 01789 ⟹ 0F101789

object program (pass-2)

H^ SUM ^0000 ^00178C

T^ 000000 ^16^ BH1D^010000^691017891^1BADOD^2F2007^3F2FF7^0F101789

E^ 000000

SYMTAB → (pass-1)

SYMBOL NAME	VALUE
LOOP	0009
COUNT	0016
TABLE	0019
TOTAL	1789

2. Generate the complete object program for the following assembly level program. Also indicate the contents of symbol table at the end. Assume standard SIC model and assume the following m/c codes in HEX

LDA = 00 STA = 0C TIX = 2C JLT = 38

LDX = 04 ADD = 18 RSUB = 4C

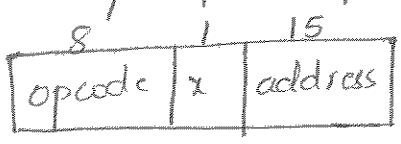
LOCCTR (PASS-1)	LENGTH	LABEL	OPCODE	OPERAND	OBJECT CODE
		SUM	START	4000	
4000	3	FIRST	LDX	ZERO	044788
4003	3		LDA	ZERO	004788
4006	3	LOOP	ADD	TABLE,X	184015
4009	3		TIX	COUNT	2C4785
400C	3		JLT	LOOP	384006
400F	3		STA	TOTAL	0C478B
4012	3		RSUB		4C0000
4015	1770	TABLE	RESW	2000 (1770)H	
4785	3	COUNT	RESW	1	
4788	3	ZERO	WORD	0	000000
478B	3	TOTAL	RESW	1	
478E			END	FIRST	

Program length = END address - starting address

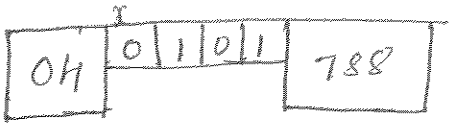
= 478E - 4000 = 178E

→ Since it is SIC program, we have two addressing mode
- direct addressing (x=0)
- indexed addressing (x=1)

- so directly put opcode with operand address



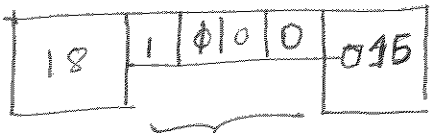
1) H000 FIRST LDX ZERO



⇒ 0h5788

2) H006 ADD TABLE, X ⇒ indexed addressing

18 H015



18 C 015 ⇒ 18C015

SYMTAB

NAME	VALUE
FIRST	H000
LOOP	H006
TABLE	H015
COUNT	5785
ZERO	5788
TOTAL	578B

object program

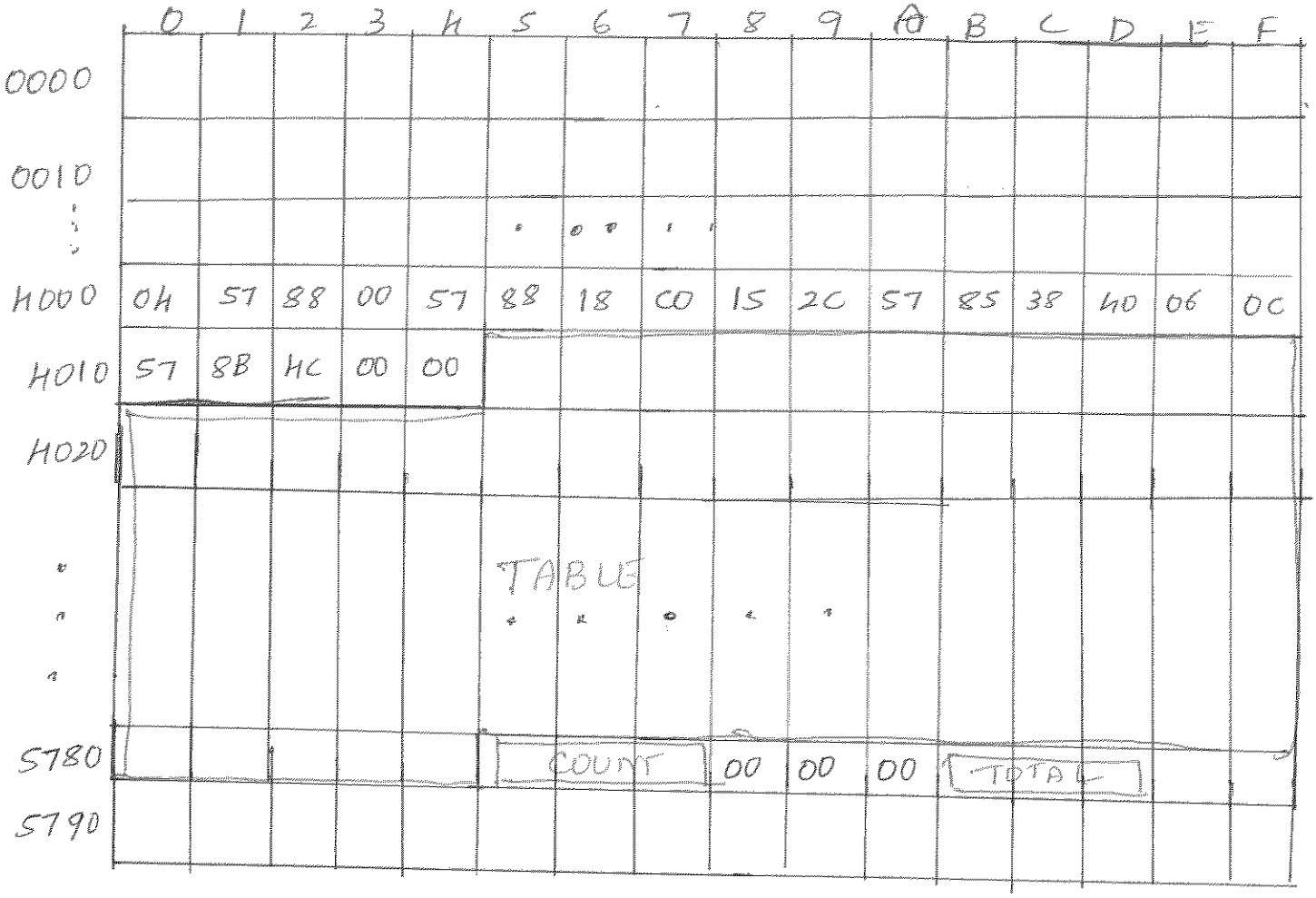
H^ SUM ^00H000 ^001785

T^ 00H000 ^ 15 ^ 0H5788 ^ 005788 ^ 18C015 ^ 2C5785 ^ 38H006 ^ 0C578B ^ H10000

T^ 005788 ^ 03 ^ 000000

E^ 00H000

LOADING INTO MAIN MEMORY



3. Generate the object code for each statement in the following SIC/XE program and generate the object program for the same.

LOCCTR	LENGTH	LABEL	OPCODE	OPERAND	OBJECT-CODE
		SUM	START	0	
0000	3	FIRST	LDX	#0	050000
0003	3		LDA	#0	010000
0006	4		+LDB	#TABLE2	69101790
			BASE	TABLE2	
000A	3	LOOP	ADD	TABLE, X	1BA013
000D	3		ADD	TABLE2, X	1BC000
0010	3		TIX	COUNT	2F200A
0013	3		JLT	LOOP	3B2FFH
0016	4		+STA	TOTAL	0F102F00
001A	3		RSUB		4F0000
001D	3	COUNT	RESW	1	
0020	1770	TABLE	RESW	2000 (1770)H	
1790	1770	TABLE2	RESW	2000 (1770)H	
2F00	3	TOTAL	RESW	1	
2F03		END	FIRST		

LDX = 04 LDB = 68 TIX = 2C STA = 0C

LDA = 00 ADD = 18 JLT = 38 RSUB = 4C

→ keep assigning the length for each instruction based on

(i) 1st operand is memory address — 3 byte
(ii) 1st operand is register — 2 byte
(iv) + (extended format — 4 byte

→ Find the LOCCTR value ; Program length = 2F03 − 0000
= 2F03

→ Create SYMTAB

Name	value
FIRST	0000
LOOP	000A
COUNT	001D
TABLE	0020
TABLE2	1790
TOTAL	2F00

→ object code for each instruction

1) 0000 FIRST LDX #0 → immediate addressing
(LDX = 04)

disp = 000

n	i	x	b	p	e		
0	0	1	0	1	0	0	0	0	000

0 0 S 0 000 ⇒ 050000

2) 0003 LDA #0 → immediate addressing
(LDA = 00)

disp = 000

n	i	x	b	p	e		
0	0	0	0	1	0	0	0	0	000

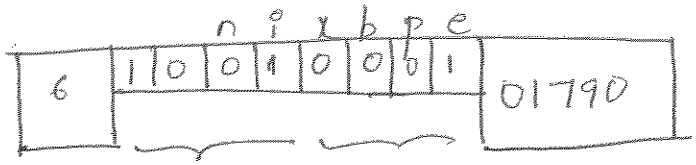
0 1 0000 ⇒ 010000

3) 0006 +LDB #TABLE2 → Extended + Immediate
68

~~TA = PC + disp~~

~~disp = TA = pc = 1790 = 000A~~

disp = Target address = 01790
20bits

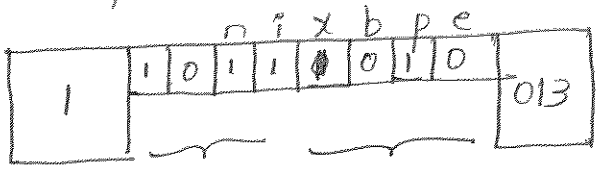


6 9 101790 ⇒ 69101790

4) 000A LOOP ADD TABLE, X → Indexed + PC relative
18

TA = PC + disp

disp = TA − PC = 0020 − 000D = 0013



1 B A 013 ⇒ 1BA013

5) 000D ADD TABLE2, X → Indexed + base relative
18

(∵ TABLE2 is stored in base register)

Initially we can try for PC-relative & checkout whether displacement is within the range.

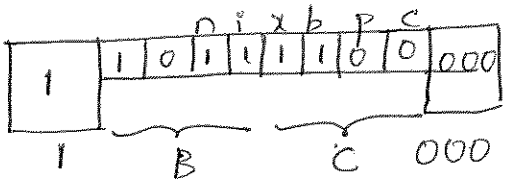
disp = TA − PC

= 1790 − 0010 = $(1780)_H \geq (6016)_d > (2047)_d$

∴ go for base relative

disp = TA − B (look for address of TABLE2 in SYMTAB)

= 1790 − 1790 = 0000

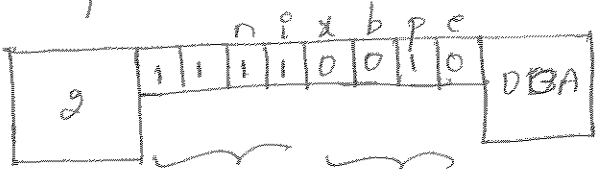


1 B C 000 ⇒ 1BC000

6) 0010 TIX COUNT → Pc relative

TIX: 2C

disp = TA − PC = 001D − 0013 = 000A

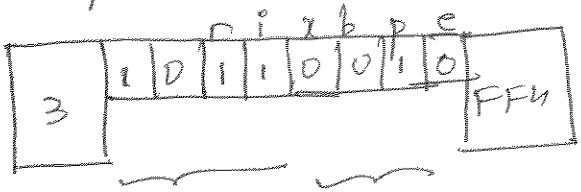


2 F 2 000A ⇒ 2F200A

7) 0013 JLT LOOP ⇒ Pc relative

JLT: 38

disp = TA − PC = 000A − 0016 = FFF4

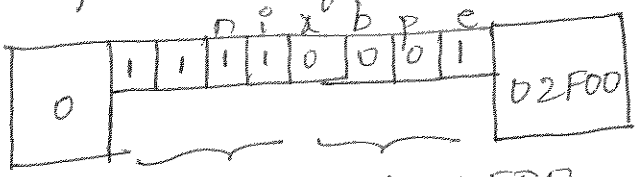


3 B 2FFF4 ⇒ 3B2FFF4

8) 0016 +STA TOTAL ⇒ Extended (Format 4)

STA: 0C

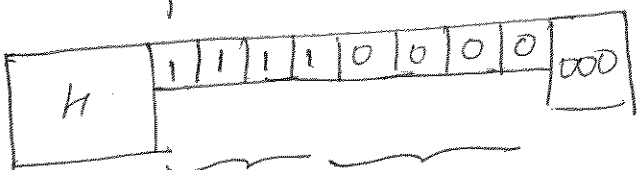
disp = address of TOTAL = 2F00



0 F 102F00

9) RSUB 001A RSUB

⇒ no operand ∴ no displacement



4 F 0000 ⇒ 4F0000

→ object program

H^ sum ^000000^002F03

T^ 000000^1D^050000^010000^69101790^1BA013^1BC000^2F200A^3B2FFh
^0F102F00^hF0000

E^000000

→ loader loads into memory

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	05	00	00	01	00	00	69	10	17	90	1B	A0	13	1B	C0	00
001D	2F	20	0A	3B	2F	Fh	0F	10	2F	00	hF	00	00	COUNT		
002D						TABLE										
1790						TABLE 2										
2F00	TOTAL															

4 Generate the machine code for the following SIC/XE program

Given JSUB = A0, LDA = 80, LDX = 60, STA = 50, COMP = 90,

RSUB = 4C JEQ = B0, J = B8

LOCCTR	LENGTH	LABEL	OPCODE	OPERAND	OBJECT CODE
		COPY	START	1000	
1000	4	CLOOP	+JSUB	RDREC	
	3		LDA	LENGTH	
	3		COMP	ZERO	
	3		JEQ	EXIT	
	3		J	CLOOP	
	3	EXIT	STA	BUFFER	
	3		LDA	THREE	
	3		STA	TOTAL_LENGTH	
	3		RSUB		
		BUFFER	RESB	100	
	3	EOF	BYTE	C'EOF'	
	3	ZERO	WORD	0	
	9	THREE	WORD	3	
	3	LENGTH	RESW	1	
	3	TOTAL_LENGTH	RESW	1	
	3	RDREC	LDX	ZERO	

2.2.2. Program Relocation

a) Absolute Assembly program is one which executes properly, only if program is loaded from specified location.

Ex: All SIC programs are absolute assembly program

Consider the SIC program

```
5    1000   COPY   START   1000
10   1000   FIRST  STL     RETADDR   141033
15   1003   CLOOP  JSUB    RDREC     482039
       :
55   101B          LDA     THREE     00102D
       :
85   102D   THREE  WORD    3         000003
```

→ Here program is loaded at address 1000.

→ Line no. 55 specifies that the register A is to be loaded from memory address 102D (object code).

→ suppose we attempt to load and execute the program at address 2000 instead of address 1000, the address 102D will not contain the value that we expected, as it might be part of some other user's program.

→ obviously we need to make some changes in the address portion of this instruction so we can load and execute the program at address 2000.

→ At the same time, there are statements like line no. 95. which generate a constant 3, that should remain the same regardless of where the program is loaded.

→ From the object code, we can't it is not possible to tell which values represent addresses and which represent constant data items.

→ This is all because the assembler doesn't know the actual location where the program will be loaded till load time. ∴ it cannot make the necessary changes required.

→ only parts of the program that require modification at load time are those that specify direct addresses.

This is achieved through relocatable program.(for SIC/XE machine)

program relocation

2.2.2. Program Relocation

Program relocation is a process of modifying the addresses used in address sensitive instructions of a program such that program can execute correctly from allocated memory area. It is often needed to have more than one program at some time, sharing the memory and other resources of the machine. Because of this, it is necessary to load a program into memory whenever it is available. Hence relocation of the addresses in the program is required and this will be done during loading time. Assembler only indicates those instructions which need modification and this information is passed to loader.

The Assembler solves the relocation problem as follows:

→ keeping track of operand address relative to start of a program

→ Generating commands for loader which add the beginning address to operand relative address

The An object program that contains the information necessary to perform this kind of modification is called a "relocatable program" - we can accomplish this with a modification record as follows

Modification Record

col. 1 M

col 2-7 starting location of the address field to be modified, relative to the beginning of the program

col 8-9 Length of the address field to be modified, in half-byte (hexadecimal).

→ The length is stored in half-byte (rather than byte) because the address field to be modified may not occupy an integral number of bytes.
Ex:- 20 bits = 5 half-bytes

→ The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If this field occupies an odd number of half-byte, it is assumed to begin in the middle of the first byte at the starting location.

Ex:- SIC/XE program

5	0000	COPY	START		
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
			+JSUB	WRREC	4B1010SD
35	0013		J	CLOOP	3F2FEC
40	0017				
			+JSUB	WRREC	4B1010SD
65	0026				
100	0036	BUFFER	RESB	4096	
					BH10
125	1036	RDREC	CLEAR	X	

program is loaded at address 0000

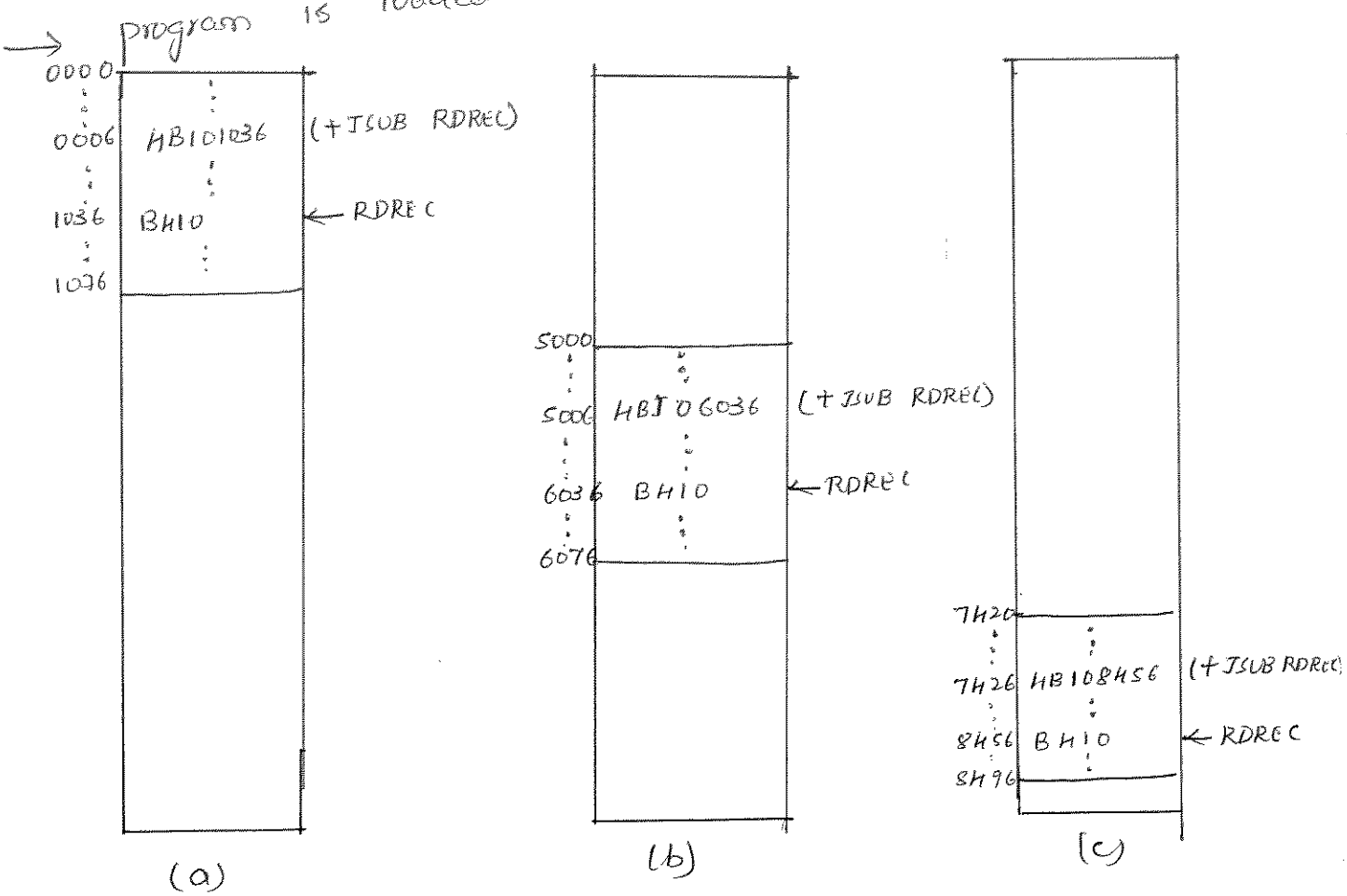


Fig: Example for program relocation

→ JSUB instruction at line 15 is loaded at address 0006

→ The address field contains 01036 (address of RDREC)

is 0006 + JSUB RDREC 4B101036

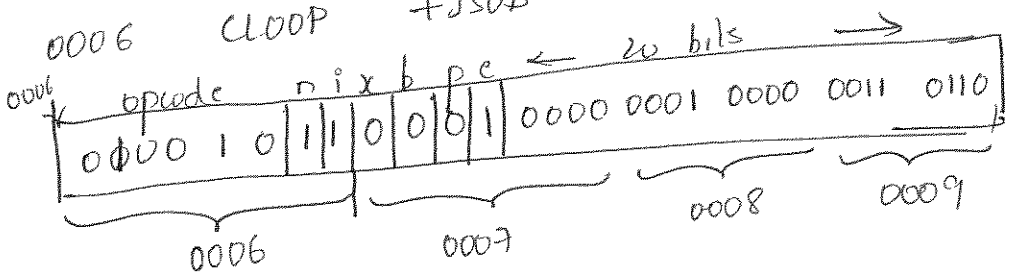
→ Suppose we want to load this program beginning at address 5000, as shown in fig (b), the address of instruction labeled RDREC will be 6036.

→ Likewise if we load at 7A20 as in fig (c), then address of RDREC will be 4B108A56.

→ It means, irrespective of the starting address loaded, RDREC is always 1036 byte past the starting address of the program. This is the reason we initialized the location counter to 0. (ie relative to the starting address)

→ The modification record looks like

0006 CLOOP +JSUB RDREC 4B101036



M^000007^05

note: 05 because 1/2 is 20 bits address ⇒ 05 half byte ⇓ 05 x 4 = 20 bits

Actualy at location 0007, first half byte is part of flag bits x, b, p, e. But length 05 tells loader to modify only last 5 half byte. Hence instruction 4B1 remains unchanged.

Relocation for instruction of the 35 and 65

35	0013	+JSUB	CORREC	4B1010SD
65	0026	+JSUB	CORREC	4B1010SD

M^00001H^05 & M^000027^05

→ if we add 5000 then address should be

4B1/01036
5000
4B1/06036
7H20
4

4B1/01036
+ 7H20 — relocatable address
4B1 08456

→ Some instructions like
CLEAR S } do not need modification ∵ operand is
LDA #3 } not a memory address.

→ 10 STL RETADR ∵ do not need modification ∵ operand is specified using program-counter relative or base-relative addressing. Here the displacement is always 02D. Irrespective of location of program loaded, it is always 2D byte away from the STL instruction.

→ The ten distance between LENGTH and BUFFER will always be 3 byte.

The object program is recognized as (Fig 2.6)

```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^...^4B10105D^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^...          ^B850
T^001053^1D^3B2FEA^134000^4F0000^...             ^B850
T^00107D^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

2.3 Machine Independent Assembler Features

→ machine independent means some assembler features that are not closely related to machine architecture.

This section includes

2.3.1 → The implementation of literals within an assembler

2.3.2 → Two assembler directives EQU and ORG used to define the symbols

2.3.3 → Use of expressions in assembler language statements

2.3.4 → Implementation of program blocks

2.3.5 → Implementation of control sections

2.3.1 Literals

→ Constant operand can be specified as a part of the instruction that use it, instead of using a label which is defined as constant else where. Such an operand is called a literal because the value is stated "literally" in the instruction

Ex:-

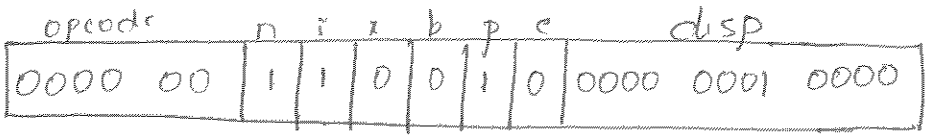
```
45    001A    ENDFIL    LDA    EOF      032010
                                 Label
      :
80    002D    EOF       BYTE   C'EOF'   454F46
```

⇓ can be written as

```
45    001A    ENDFIL    LDA    =C'EOF'  032010
      :
*                       LTORG
                        =C'EOF'          454F46
```

The object code generated for line 45, 915 and 230 in fig 2.6 and fig. 2.10 are identical.

(i) 45 001A ENDFIL LDA = C'EOF' 032010



disp = opaddr - pc TA = (pc) + disp

= 002D - 001D = 01D

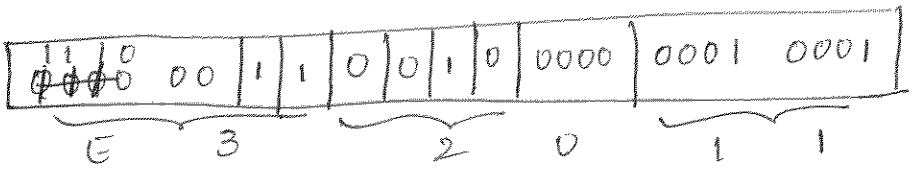
⇒ 032010

(ii) 215 1062 WLOOP TD = X'05' E32011

TD = E0

disp = opaddress - pc

= 1076 - 1065 = 011



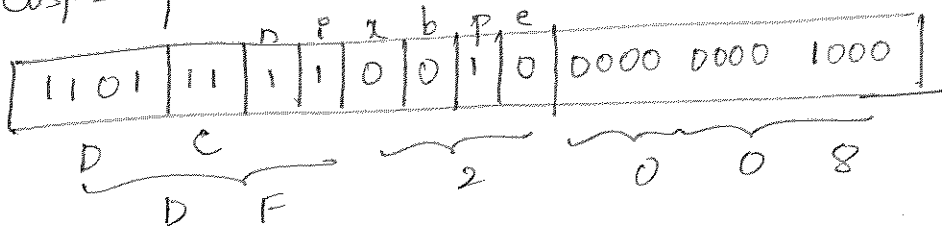
(iii) 230 106B WD = X'05' DF2008

⋮

1076 * = X'05'

WD = DC

disp = opaddress - pc = 1076 - 106E = 008



Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
13		LDB	#LENGTH	ESTABLISH BASE REGISTER
14		BASE	LENGTH	
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	=C'EOF'	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
93		LTORG		
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	
107	MAXLEN	EQU	BUFEND-BUFFER	MAXIMUM RECORD LENGTH
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		
120	.			
125	RDREC	CLEAR	X	CLEAR LOOP COUNTER
130		CLEAR	A	CLEAR A TO ZERO
132		CLEAR	S	CLEAR S TO ZERO
133		+LDT	#MAXLEN	
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		JEQ	RLOOP	LOOP UNTIL READY
145		RD	INPUT	READ CHARACTER INTO REGISTER A
150		COMPR	A,S	TEST FOR END OF RECORD (X'00')
155		JEQ	EXIT	EXIT LOOP IF EOR
160		STCH	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIXR	T	LOOP UNLESS MAX LENGTH
170		JLT	RLOOP	HAS BEEN REACHED
175	EXIT	STX	LENGTH	SAVE RECORD LENGTH
180		RSUB		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
195	.			
200	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.			
210	WRREC	CLEAR	X	CLEAR LOOP COUNTER
212		LDT	LENGTH	
215	WLOOP	TD	=X'05'	TEST OUTPUT DEVICE
220		JEQ	WLOOP	LOOP UNTIL READY
225		LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
230		WD	=X'05'	WRITE CHARACTER
235		TIXR	T	LOOP UNTIL ALL CHARACTERS
240		JLT	WLOOP	HAVE BEEN WRITTEN
245		RSUB		RETURN TO CALLER
255		END	FIRST	

Figure 2.9 Program demonstrating additional assembler features.

```
Line    Loc     Source statement                        Object code
   5    0000    COPY    START   0
  10    0000    FIRST   STL     RETADR          17202D
  13    0003            LDB     #LENGTH         69202D
  14                    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     =C'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
  93                    LTORG
        002D    *       =C'EOF'                 454F46
  95    0030    RETADR  RESW    1
 100    0033    LENGTH  RESW    1
 105    0036    BUFFER  RESB    4096
 106    1036    BUFEND  EQU     *
 107    1000    MAXLEN  EQU     BUFEND-BUFFER
 110                    .
 115                    .       SUBROUTINE TO READ RECORD INTO BUFFER
 120                    .
 125    1036    RDREC   CLEAR   X               B410
 130    1038            CLEAR   A               B400
 132    103A            CLEAR   S               B440
 133    103C            +LDT    #MAXLEN         75101000
 135    1040    RLOOP   TD      INPUT           E32019
 140    1043            JEQ     RLOOP           332FFA
 145    1046            RD      INPUT           DB2013
 150    1049            COMPR   A,S             A004
 155    104B            JEQ     EXIT            332008
 160    104E            STCH    BUFFER,X        57C003
 165    1051            TIXR    T               B850
 170    1053            JLT     RLOOP           3B2FEA
 175    1056    EXIT    STX     LENGTH          134000
 180    1059            RSUB                    4F0000
 185    105C    INPUT   BYTE    X'F1'           F1
 195                    .
 200                    .       SUBROUTINE TO WRITE RECORD FROM BUFFER
 205                    .
 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
```

Figure 2.10 Program from Fig. 2.9 with object code.

Literal Pools:-

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

→ Normally literals are placed into a pool at the end of the program, which shows the assigned address and the generated data values.

→ The drawback of keeping literal pool at the end of the program is use the literal operand is too far away from the instruction referencing it and require a large amount of storage reservation for the buffer too.

→ To avoid this we use an assembler directive LTORG (ORIGIN of LITERALS) which instructs the assembler to assemble the current literals pool immediately.

→ When the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the program). ⇒ ie keep the literal operand close to the instruction.

→ Same literal may be used more than once in the program ie duplicate literals but it stores only one copy of the specified data value.

Ex:- 215 1062 WLOOP TD =X'05'

230 106B WD =X'05'

→ Apart from one copy of data value, it stores only one data area with this value generated. Both instructions refer to the same address in the literal pool for their operand

→ There are two ways of recognizing the duplicate literals

(a) Compare the character strings defining them. Same literal name with different value.

Ex: X 'os'

(b) Compare the generated data value. This is better but increases the complexity of the assembler.

Ex: =C'EOF' and =X'454F46'

→ The problem of using character strings to recognize duplicate literals is, as we see '*' denotes a literal refers to the current value of program counter after line no. 93. There may be some literals that have the same name but different value, for example the statements

BASE * —①

LDB =* —②

① — loads the beginning address of the program into register B. This value will be available later for base relative addressing.

** → it causes a problem if we use at line no. 13

ie 13 0003 LDB =* 692003

it specifies an operand with value 0003.

55 0020 LDA =* 010020

ie literand operands have identical names but they have different values and both must appear in the literal pool.

** → The same problem arise if a literal refers to any other item whose value changes between one point in the program and another.

→ The datastructure used to store literal operands is literal table "LITTAB"

→ Literal Table (LITTAB) : It is a hashtable using literal name or value as the key.

contains
- ↳ literal name
- ↳ operand value
- ↳ operand length
- ↳ address assigned

NAME	OPERAND VALUE	LENGTH	ADDRESS
=C 'EOF'	EOF	03	002D
=X '05'	05	01	1076

pass-1

→ Builds LITTAB with literal name, operand value and length, leaving the address unassigned

→ when LTORG statement is encountered, assign an address to each literal not yet assigned an address. Along with this, location counter is updated to reflect the number of byte occupied by each literal

Pass-2

→ Search LITTAB for each literal operand encountered to generate respective object code
→ Generate data values using BYTE or WORD statements
→ Generate modification record for literals that represent an address in the program.

Difference between literal and an immediate operand

Literal (=)	Immediate operand (#)
1. Literal is an assembler directive	Immediate is a machine recognizable data
2. The assembler generates the specified value as a constant at some other memory location. The address of this generated constant is used as target address for machine instructions	2. Here value is assembled as part of machine instruction.
3 Architectural support is required	Architectural support not required
4. very slow since values are obtained from data memory	faster than literal ∵ data is within the instruction
5. capable of storing larger data	can't store larger data ie Fullword is opcode, registers

55 0020 LDA #3 010003

2.3.2 Symbol-Defining Statements

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.

The assembler directives are a) EQU b) ORG

a) EQU (Equate)

→ allows the programmer to define symbols (i.e enters it into SYMTAB) and assigns to it the specified value.

syntax: Symbol EQU value

value → constant / Expression / previously defined symbol

→ The value may be

(i) constant

(ii) An expression involving constant

(iii) previously defined symbols.

→ Uses of EQU

1) To establish symbolic names that can be used for improved readability in place of numeric value.

Ex: +LDT #4096 ; load the value 4096 into reg. T

⇓ replace with

MAXLEN EQU 4096

+LDT MAXLEN

→ when assembler encounters the EQU statement, it enters MAXLEN to SYMTAB with value 4096.

→ During assembly of LDT instruction, the assembler searches the SYMTAB for the MAXLEN symbol and uses its value as the operand in the instruction.

→ The advantage of doing so is if we want to change the value 4096 to some other value, we need to change in only one place ^ MAXLEN instead of searching or scanning through the program for #4096 for the replacement. (required change) ⇒ #define in c

3) To define mnemonic names for registers.

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

→ The symbols A, X, L has to be entered into SYMTAB with their corresponding values 0, 1, 2.

→ instruction RMO A, X searches the SYMTAB for A and X and their values to assemble the instruction

3) To reflect the logical function of the registers

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

→ implies Register R1 is used as base register, R2 as program counter, R3 as index registers etc

→ Forward reference is not allowed in EQU ie all terms in the value field must have been defined previously during pass-1

Ex:
```
ALPHA  RESW  1          BETA   EQU  ALPHA
BETA   EQU   ALPHA      ALPHA  RESW 1
```
Allowed / not allowed

b) ORG (origin)

→ Assembler directive used to indirectly assign value to symbols

→ syntax :

ORG value

→ value can be

(i) Constant

(ii) expression involving constant

(iii) Previously defined symbols

→ When ORG is encountered, the assembler resets its LOCCTR to the specified value.

→ Location counter is used to control assignment of storage in the object program. Hence altering its value would result in an incorrect assembly.

∴ the directive should be minimum used.

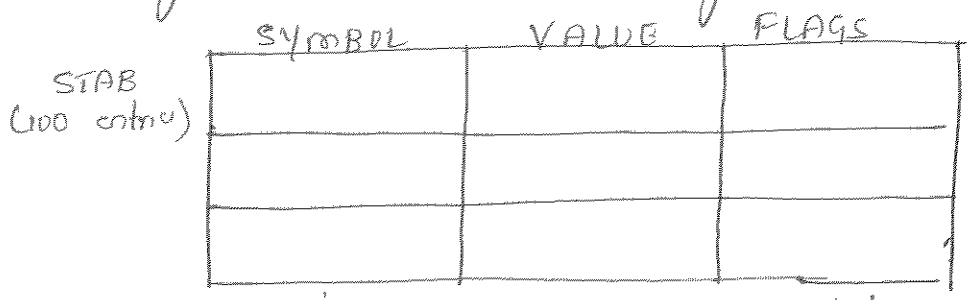
→ The ORG statement will affect the value of all labels defined until the next ORG.

→ If the previous value of LOCCTR is automatically remembered, then we can return to the normal use of LOCCTR just by writing

ORG

→ Example: To define a symbol table with the following structure.

→ Symbol table with the given structure



↳ SYMBOL field contains user defined symbols.
↳ VALUE field represents the value assigned to the symbol
↳ FLAG field specifies symbol type and other information
↳ The space for this table is reserved as

STAB1 RESB 1100 ; 100 (entries) × 11 (each entry) = 1100

↳ We can access the label entries in two ways (Usage of EQU and ORG)
↳ Using EQU ⇒

```
SYMBOL  EQU  STAB
VALUE   EQU  STAB  +6   ← offset from STAB
FLAGS   EQU  STAB  +9
```

(i) To fetch the value field,

LDA VALUE, X ; where X = 0, 11, 22, ... for each entry

↳ index register

*** (ii) This method of definition simply defines the labels, it does not make the structure of the table as clear as it might be.

(iii) Therefore we make use of ORG.

↳ Using ORG ⇒

```
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
```

(i) The first ORG sets the location counter to the value of STAB

(ii) RESB statement defines SYMBOL to have the current value in LOCCTR

(iii) LOCCTR is then advanced, so the label on RESW statement assigns to VALUE to address (STAB+6) and then advanced to assign to FLAGS to address (STAB+9).

(iv) The last ORG statement sets LOCCTR back to its previous value. That is the address of the next unassigned byte of memory after the table STAB.

→ Forward reference is not allowed in ORG ie all symbols used to specify the new location counter value have to be previously defined.

→ Example:

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

```
BYTE3    RESB  1
         ORG
ALPHA    RESB  1
```

↳ cannot processed :' the assemble doesnot know what value has to be assigned to the location counter in response to the first ORG statement

The symbols BYTE1, BYTE2, BYTE3 are not assigned addresses during pass 1.

↳ It has to be written as

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

2.3.3. Expressions

→ The assembler allows the use of expressions as operand.

→ It calculates the expressions and produces a single operand address or value.

→ The expression consists of

(i) operators : + - * / (division is usually defined to produce an integer value)

(ii) Individual terms : Constants, user-defined symbols, special terms like * (current value of the location counter).

Ex:-
```
MAXLEN   EQU   BUFEND - BUFFER
STAB     RESB  (6+3+1) * MAX
BUFEND   EQU   *
```

→ The value of terms can be absolute (independent of program location) such as constants or relative (to the beginning of the program) such as Address labels, data areas, references to the location counter value.

→ Expressions are classified as

(i) Absolute Expressions } based on type of value
(ii) Relative Expressions } produced.

(i) Absolute Expressions :

↳ Absolute means independent of program location and contains absolute terms like constants

↳ It may also contain relative terms provided the relative terms occur in pair and the terms in each such pair have opposite signs.

↳ It is not necessary that the paired terms be adjacent to each other in the expression however, all relative terms must be capable of being paired in this way.

↳ None of the relative terms may enter into a multiplication or division operation.

(ii) Relative Expressions :

↳ Relative means relative to the beginning of the program, such as labels on the instruction, data areas, references to the location counter value.

↳ Here, all of the relative terms except one can be paired as in absolute expressions and the remaining unpaired relative term must have a positive sign.

↳ no relative terms may enter into a multiplication or division operation.

note: If either of absolute expression or relative expression do not meet the conditions, they are flagged as errors

→ A relative term or expression represents some value which is written as (S+r)

- S = starting address of the program
- r = value of term or expression relative to the starting address.

Ex: (1) MAXLEN EQU BUFEND - BUFFER

↳ both BUFEND and BUFFER are relative terms representing an address within the program. The expression represents an absolute value.

(2) Illegal Expressions

BUFEND + BUFFER ; no opposite signs

100 - BUFFER ; both are not relative terms

3 * BUFFER ; * can't be used

→ Type of expression is determined by keeping track of symbol types in the program. This is done by adding a flag (R or A) in the SYMTAB for each symbol defined

Ex:

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

} two symbols of type R

2.3.4 Program Blocks

→ Till now, we have seen that the program being assembled was treated as a single unit, eventhough it had subroutine, data or csect etc resulting in a single block of object code.

→ within this object code (program) the generated machine instructions and data appeared in the same order as they were written in the source program.

→ But sometimes it is required to logically rearrange the statements of the source program so that the large buffer area can be moved to the end of object program, no need of using extended instruction format, the base register usage is not required, the problem of placing literals in program has to be more flexible etc.

→ All these are achieved through some of the assembler feature such as program blocks and control sections.

→ Program blocks: Allows the generated machine instructions and data to appear in the object program in a different order from the corresponding source statement.

or

Program blocks are segment of code that are rearranged within a single object program unit.

Assembler Directive : USE

syntax : USE BLOCKNAME

Fig. 9.11 shows the sample program with program blocks

→ There are three blocks in the program

(i) Unnamed program block contains the executable instructions of the program

(ii) CDATA program block contains all data areas that consists of few block of memory ie few words or less in length

(iii) CBLKS program block contain all data areas that consists of larger blocks of memory.

→ 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.

→ USE on line 92 indicates the beginning of CDATA block

→ USE on line 103 indicates the beginning of CBLKS block

→ USE on line 123 resume the default block

→ Each program block may contain several separate segments of the source program but assembler will logically rearrange these segments to gather together the piece of each block and assign address.

→ Program readability is better if data areas are placed in the source program close to the statements that reference them.

The assembler accomplishes this logical rearrangement of code by maintaing during pass-1 and pass 2

(i) Pass-1

Fig 2.12(b) shows the pass-1 of program blocks

→ A separate location counter for each program block is assigned and is assigned to ZERO when a program block begins

↳ Saving and Restoring the current value of LOCCTR when occurs whenbe switching between blocks

↳ Each label is assigned an address relative to the start of the block.

↳ store the block name and number in the SYMTAB along with the assigned relative address of the label.

↳ At the end of pass-1, indicate the block length as the latest value of LOCCTR for each block.

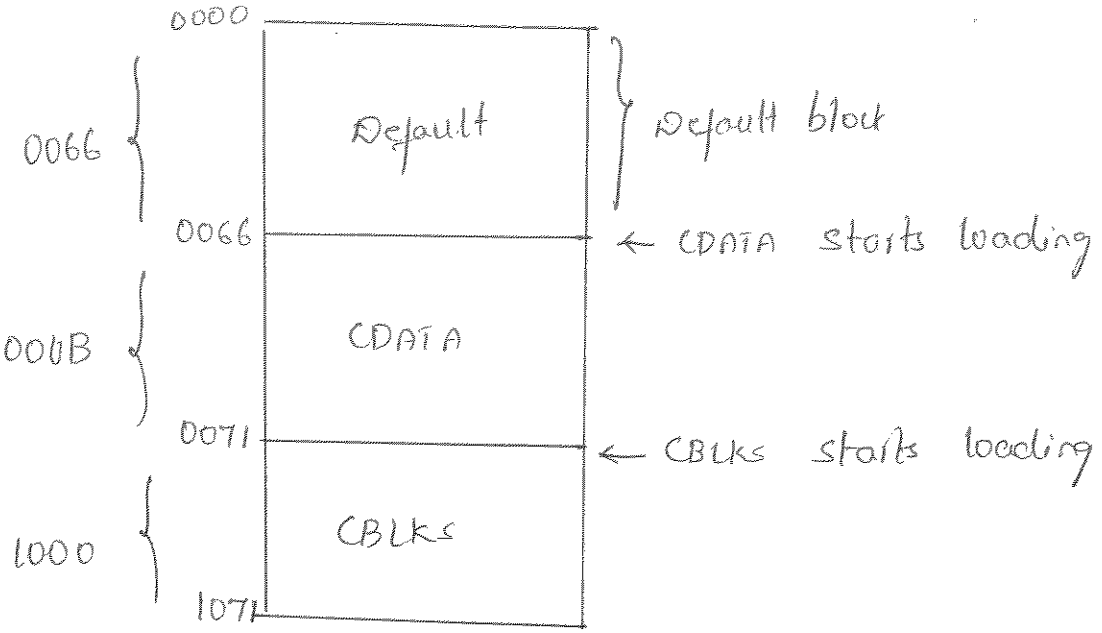
↳ constructs a table which contains the starting address and length for all blocks.

↳ Assembler assigns to each block a starting address in the object program (beginning with relative location 0).

latest value of LC ↓

Block name	Block number	Address	Length	
default	0	0000	0066	0063 to 0003 = 0066
CDATA	1	0066	000B	0066+ 000B = 0071
CBLKS	2	0071	1000	1071 - 0071 = 1000

↳ Flag is also added in this table



ii) Pass 2 :

→ calculates the address for each symbol relative to the start of the object program (not the start of the individual program blocks) by adding

(i) The location of the symbol relative to the start of its block (from SYMTAB)

(ii) The starting address of this block

Example

```
20      0006    0       LDA     LENGTH
:
                                USE     CDATA
92      0000    1
100     0003    1       LENGTH  RESW    1
```

→ The value of the operand LENGTH is 0003 relative to block 1 (CDATA)

∴ address (TA) = 0003 + 0066 = 0069 relative to program when this instruction is executed

Line	Loc	Block		Source statement			Object code
5	0000	0		COPY	START	0	
10	0000	0		FIRST	STL	RETADR	172063
15	0003	0		CLOOP	JSUB	RDREC	4B2021
20	0006	0			LDA	LENGTH	032060
25	0009	0			COMP	#0	290000
30	000C	0			JEQ	ENDFIL	332006
35	000F	0			JSUB	WRREC	4B203B
40	0012	0			J	CLOOP	3F2FEE
45	0015	0		ENDFIL	LDA	=C'EOF'	032055
50	0018	0			STA	BUFFER	0F2056
55	001B	0			LDA	#3	010003
60	001E	0			STA	LENGTH	0F2048
65	0021	0			JSUB	WRREC	4B2029
70	0024	0			J	@RETADR	3E203F
92	0000	1			USE	CDATA	
95	0000	1		RETADR	RESW	1	
100	0003	1		LENGTH	RESW	1	
103	0000	2			USE	CBLKS	
105	0000	2		BUFFER	RESB	4096	
106	1000	2		BUFEND	EQU	*	
107	1000			MAXLEN	EQU	BUFEND-BUFFER	
110				.			
115				.	SUBROUTINE TO READ RECORD INTO BUFFER		
120				.			
123	0027	0			USE		
125	0027	0		RDREC	CLEAR	X	B410
130	0029	0			CLEAR	A	B400
132	002B	0			CLEAR	S	B440
133	002D	0			+LDT	#MAXLEN	75101000
135	0031	0		RLOOP	TD	INPUT	E32038
140	0034	0			JEQ	RLOOP	332FFA
145	0037	0			RD	INPUT	DB2032
150	003A	0			COMPR	A,S	A004
155	003C	0			JEQ	EXIT	332008
160	003F	0			STCH	BUFFER,X	57A02F
165	0042	0			TIXR	T	B850
170	0044	0			JLT	RLOOP	3B2FEA
175	0047	0		EXIT	STX	LENGTH	13201F
180	004A	0			RSUB		4F0000
183	0006	1			USE	CDATA	
185	0006	1		INPUT	BYTE	X'F1'	F1
195				.			
200				.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205				.			
208	004D	0			USE		
210	004D	0		WRREC	CLEAR	X	B410
212	004F	0			LDT	LENGTH	772017
215	0052	0		WLOOP	TD	=X'05'	E3201B
220	0055	0			JEQ	WLOOP	332FFA
225	0058	0			LDCH	BUFFER,X	53A016
230	005B	0			WD	=X'05'	DF2012
235	005E	0			TIXR	T	B850
240	0060	0			JLT	WLOOP	3B2FEF
245	0063	0			RSUB		4F0000
252	0007	1			USE	CDATA	
253					LTORG		
	0007	1		*	=C'EOF'		454F46
	000A	1		*	=X'05'		05
255					END	FIRST	

Figure 2.12(a) Program from Fig. 2.11 with object code.

Line Source statement

```
  5   COPY      START   0               COPY FILE FROM INPUT TO OUTPUT
 10   FIRST     STL     RETADR          SAVE RETURN ADDRESS
 15   CLOOP     JSUB    RDREC           READ INPUT RECORD
 20             LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
 25             COMP    #0
 30             JEQ     ENDFIL          EXIT IF EOF FOUND
 35             JSUB    WRREC           WRITE OUTPUT RECORD
 40             J       CLOOP           LOOP
 45   ENDFIL    LDA     =C'EOF'         INSERT END OF FILE MARKER
 50             STA     BUFFER
 55             LDA     #3              SET LENGTH = 3
 60             STA     LENGTH
 65             JSUB    WRREC           WRITE EOF
 70             J       @RETADR         RETURN TO CALLER
 92             USE     CDATA
 95   RETADR    RESW    1
100   LENGTH    RESW    1               LENGTH OF RECORD
103             USE     CBLKS
105   BUFFER    RESB    4096            4096-BYTE BUFFER AREA
106   BUFEND    EQU     *               FIRST LOCATION AFTER BUFFER
107   MAXLEN    EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
110   .
115   .         SUBROUTINE TO READ RECORD INTO BUFFER
120   .
123             USE
125   RDREC     CLEAR   X               CLEAR LOOP COUNTER
130             CLEAR   A               CLEAR A TO ZERO
132             CLEAR   S               CLEAR S TO ZERO
133            +LDT     #MAXLEN
135   RLOOP     TD      INPUT           TEST INPUT DEVICE
140             JEQ     RLOOP           LOOP UNTIL READY
145             RD      INPUT           READ CHARACTER INTO REGISTER A
150             COMPR   A,S             TEST FOR END OF RECORD (X'00')
155             JEQ     EXIT            EXIT LOOP IF EOR
160             STCH    BUFFER,X        STORE CHARACTER IN BUFFER
165             TIXR    T               LOOP UNLESS MAX LENGTH
170             JLT     RLOOP             HAS BEEN REACHED
175   EXIT      STX     LENGTH          SAVE RECORD LENGTH
180             RSUB                    RETURN TO CALLER
183             USE     CDATA
185   INPUT     BYTE    X'F1'           CODE FOR INPUT DEVICE
195   .
200   .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205   .
208             USE
210   WRREC     CLEAR   X               CLEAR LOOP COUNTER
212             LDT     LENGTH
215   WLOOP     TD      =X'05'          TEST OUTPUT DEVICE
220             JEQ     WLOOP           LOOP UNTIL READY
225             LDCH    BUFFER,X        GET CHARACTER FROM BUFFER
230             WD      =X'05'          WRITE CHARACTER
235             TIXR    T               LOOP UNTIL ALL CHARACTERS
240             JLT     WLOOP             HAVE BEEN WRITTEN
245             RSUB                    RETURN TO CALLER
252             USE     CDATA
253             LTORG
255             END     FIRST
```

Figure 2.11 Example of a program with multiple program blocks.

disp : TA − (pc)

= 0069 − 0009

= 0060

pc = 0000 + 0009 = 0009

(starting address of obj block) + 0009 = 0009

n i x b p e		
0000 00	1 1 0 0 1 0	060

→ 032060

SYMTAB

Label name	Block number	Address	Flag
Length	1	0003	

note: line 107

1000 MAXLEN EQU BUFEND-BUFFER

→ shown without a block number indicates that MAXLEN is an absolute symbol, whose value is not relative to the start of any program block.

object Program :

→ It is not necessary to physically rearrange the generated code in the object program. The assembler just simply inserts the proper load address in

** each text record. The loader will load these code into correct place

→ Header record as before

→ Text records : the first 2 text records generated from line 5 through 70.

→ when USE statement on line 92 is encounted, the assembler writes the new Text record eventhough there is room (espace) in the previous text record.

→ The process continues till the end of the program

H^COPY^ ^ 000000^001071

T^000000^1E^172063^4B2021^ . . . - . . . ^01D0D3

T^00001E^09^0F2048^4B2029^3E203F

T^000027^1D^B410^B400^ . . . . . ^B850^

T^0000 4A^09^3B2FEA^132DFA^4F0000

T^00006C^01^F1 ^4F0000

T^00004D^19^B410^772017^ . . . . . .

T^00006D^04^4B4FF6^05

E^000000

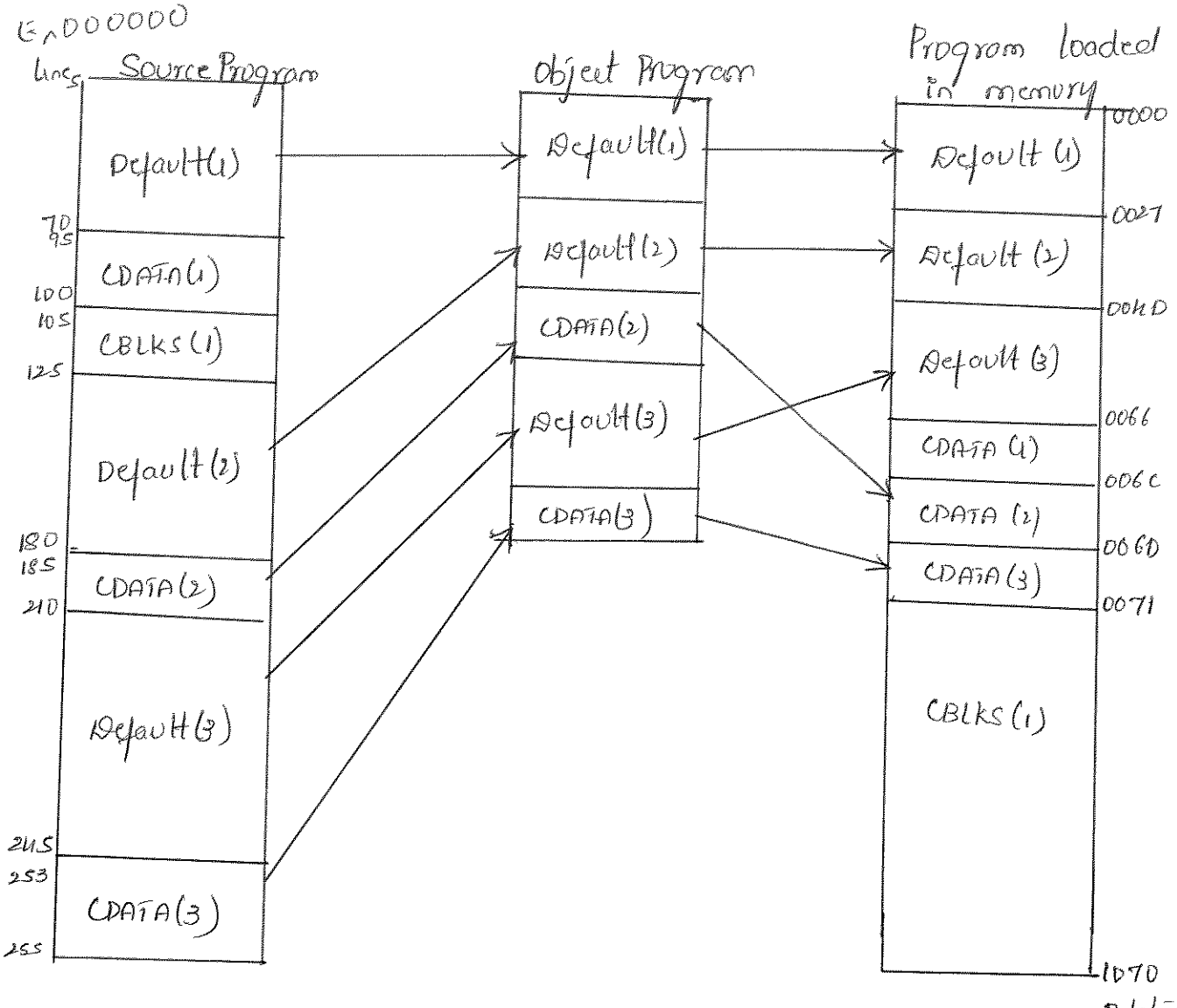


Fig: Program blocks loaded in memory

```
begin
  block number = 0 LOCCTR[i] = 0 for all i
  read the first input line
  if OPCODE = 'START' then
  begin
    write line to intermediate file
    read next input line
  end {if START}
  while OPCODE ≠ 'END' do
  if OPCODE = 'USE'
  begin
    if there is no OPEREND name then
      set block name as default
    else block name as OPERAND name
    if there is no entry for block name then
      insert (block name, block number ++) in block table
    i = block number for block name
    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
          insert (LABEL, LOCCTR[i]) into SYMTAB
        end {if symbol}
      Search OPTAB for OPCODE
      if found then
        add 3 instruction length to LOCCTR[i]
      else if OPCODE = 'WORD' then
        add 3 to LOCCTR[i]
      else if OPCODE = 'RESW' then
        add 3 * #[OPERAND] to LOCCTR[i]
      else if OPCODE = 'RESB' then
        add #[OPERAND] to LOCCTR[i]
      else if OPCODE = 'BYTE' then
      begin
        find length of constant in bytes
        add length to LOCCTR[i]
      end {if byte}
else
```

Figure 2.12(b) Pass 1 of program blocks.

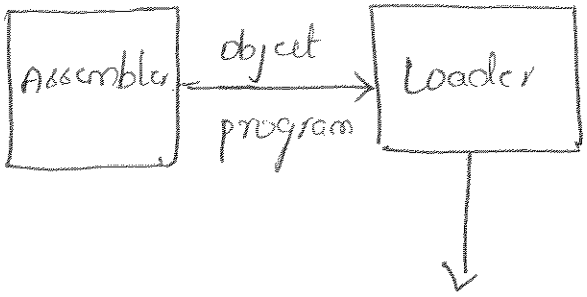
```
    Set error flag
    end {if not a comment}
  write line to intermediate file
  read Text input line
  end {while not END}
write last line to intermediate file
save Length[i] as LOCCTR[i] for all i
Address[o] = starting address
Address[i] = address(i - 1) + Length(i - 1)
             [for i = 1 to max(block number)]
insert(address[i], Length[i]) in block table for all i
end {Pass 1}
```

Figure 2.12(b) (*cont'd*)

```
If OPCODE = 'USE' then
  set block number for block name with OPERAND field
  search SYMTAB for OPERAND
  store symbol value + address [block number] as operand address
end {Pass 2}
```

Figure 2.12(c) Pass 2 of program blocks.

Loading



contacts OS & OS says start loading at particular address

Assembler generates the object program (Header Record, Text record, modify record and End record). Assembler interact with loader through object program. loader contacts operating system to load at particular address. Then OS checks if that particular ~~address~~ space is free if so it starts loading. If not it will tell the loader to either wait or remove unnecessary space

∴ "loader loads the object program residing in hard disk to main memory and start executing".

→ when there is no enough space, somebody has to instruct the loader to change its address and tay loading. It is done by assembler not by OS.

∴ Assembler instructs the loader to change the address.

ie line 15/35/65 - + JSUB RDREC

line 15 → address is HB101036 at 0006.

01036 starts from 0007 (middle). ie we can access 1 byte but not 1 nibble.

0006 - HB

0007 → 10 → go here and modify the record. This is done by assembler ∴ we have modification record M^000007^05

→ loader should listen to both assembler and os

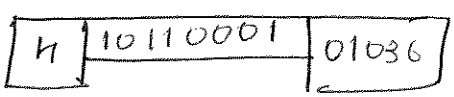
→ Assembler says goto 0007 and modify but os says it is loaded at 5000 ∴ modify at 5007.

** → + JSUB #4096 → doesnot need modification record ∵ it is immediate addressing. Irrespective of relocation it remains same.

→ Execution is part of microprocessor

Relocation → All instructions works except F4 instructions.

Ex + JSUB RDREC HB101036



TA = 01036 → loaded at 0000

** if it is loaded at 5000, it will not work properly. Loader stops functioning ∵ TA = 01036.

+ JSUB 06036 ✗ ∴ go for modification record

→ start adding length as
- memory address - 3 byte
- Register - to - register - 2 byte
- Extended - 4 byte

→ note: All literals should be placed where LTORG appears in the program. If LTORG is not present all literals will be inserted at the end of the program. (Line no 253)

→ At line 95, block 1 (CDATA) starts ∴ it stores the LOCCTR value ie 0027 in LOCCTR-0. Then starts assigning 0000 to block 1.

→ At line 105, block 2 (CBLCK) starts ∴ it saves the LOCCTR value - 0006 in LOCCTR-1 column.

→ line 125, block 0 starts again. It restore the LOCCTR value 0027 and starts over till line no. 185. (LOCCTR = 004D)

→ line 185, block-1 (CDATA) restarts by restoring the saved LOCCTR value & saves LOCCTR = 0007

→ line 210, restore 004D and starts over till line no 215 having LOCCTR = 0066 which is stored in LOCCTR-0 column

THREE LOCATION COUNTERS

LOCCTR-0 (Default block)	LOCCTR-1 (CDATA Block)	LOCCTR-2 (CBLKS)
0000	0000	0000
0027	0006	1000
002D	0007	
0066	000B	

LITERAL TABLE

Literal name	Value of Literal	Length of literal	Address of literal
=C'EOF'	454F46	03	0007
=X'05'	05	01	000A
			000B

BLOCK TABLE

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

} program length = 66 + 0B + 1000 = 1071

SYMBOL TABLE (with block number)

Symbol name	value	Block no
FIRST	0000	0
CLOOP	0003	0
ENDFIL	0015	0
RETADR	0000	1
LENGTH	0003	1
BUFFER	0000	2
BUFEND	1000	2

Symbol name	value	Block no
MAXLEN	1000	
RDREC	0027	0
RLOOP	0031	0
EXIT	003F	0
INPUT	0006	1
WRREC	002D	0
WLOOP	0052	0

→ line 952, we have USE CDATA (Block 1) and

952 → LTORG ∴ store LOCCTR = 0007 at line

953

ie 953 0007 LTORG

↓

all literals should be placed where LTORG appears in the program. we have two literals here ie =C'EOF' and =X'05' ⇒ 4 bytes

3 Byte 1 Byte

starts at 0007, 0008, 0009, 000A

(E) (O) (F) (05)

So it stores 000B in LOCCTR-1 column

→ line 105, block 2 starts and it reserve 1000 byte of memory ∴ it save 1000 in LOCCTR-2 column.

note → BUFEND EQU 100 ⇒ value of buffer is 100

BUFEND EQU * ⇒ value will be current location

value = 0000 + 1000 = 1000

Store these value in literal table.

10 0000 0 - STL RETADR

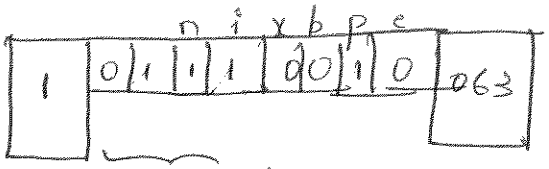
↳ present in block 1 ∴ addr wrt of block 0 (default block)

x^e displacement = sizeof (previous block) + TA - PC

= sizeof (B0) + RETADR - PC

= 0066 + 0000 − 0003

= 0063



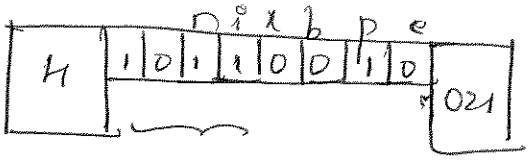
1 7 2 063 ⟹ 172063

15 0003 0 JSUB RDREC

Block (0)

disp = sizeg previous block + TA - PC

= 0 + 0027 − 006 = 021

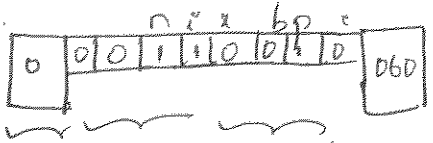


4 B 2 021 ⇒ 4B2021

20 0006 0 LDA LENGTH
block 1

disp: size of previous block + TA - PC

= 0066 + 0003 - 0006 = 0060



0 3 2060 = 032060

35 ~~0ADF~~ JSUB RDREC
48 belongs to block 0

⟹ As before

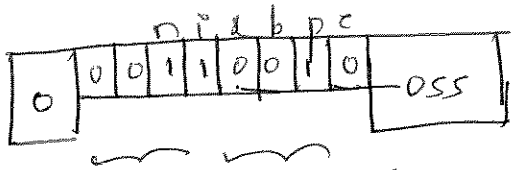
** 45 0015 LDA =C'EOF'

belongs to CDATA not to default block since it is literal which is placed after LTORG.

disp = size of previous block + TA - PC

= size of B0 + C'EOF' - PC

= 0066 + 0007 - 0018 = 0055



0 3 2055 ⟹ 032055

50 0018 STA BUFFER

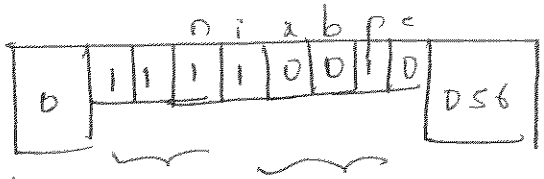
0c belongs to block 2 (03 Lks)

disp = size of (B0 + B1) + BUFFER - PC

= size of (Default block + CDATA) + BUFFER - PC

= 0066 + 0003 + 0000 - 0018

= 0071 - 0018 = 0056



0 F 2 0 56 => 0F2056

object Program

HACOPY ^ 000000 ^ 001071

1 T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEF
^032055^0F2056^010003

2 T^00001E^09^0F2048^4B2029^3E203F

3 T^000027^1D^B410^B410^B400^B440^75101000^E32038^332FFA^
^DB2032^A004^332008^57A02F^B850

4 T^000044^09^3B2FEA^13201F^4F0000

5 T^00006C^01^F1

6 T^000047^19^B410^772017^E32011^332FFA^53A016^DF2012^B850
^3F2FEE^4F0000

7 T^00006D^04^4548^0

E^000000

Loading the object program into memory

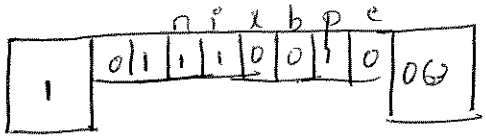
Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000	17	20	63	4B	20	21	03	20	60	29	00	00	33	20	06	4B	default(1) T1 T2
0010	20	3B	3F	2F	EE	03	20	55	0F	20	56	01	00	03	0F	20	
0020	48	4B	20	29	3E	20	3F	BH	10	BH	00	B4	40	75	10	10	default(2) T3 + TH
0030	00	63	20	38	33	2F	FA	DB	20	32	A0	0H	33	20	08	57	
00H0	A0	2F	B8	50	3B	2F	EA	13	20	1F	4F	00	00	BH	10	17	default(3) T6
0050	20	17	63	20	1B	33	2F	FA	53	A0	16	0F	20	12	B8	50	
0060	3B	2F	EE	4F	00	00	RETADR			LENGTH			F1	h5	4F	46	
0070	05								CDATA(1)				CDATA(2)				
0080																	
0090							BUFFER										
⋮																	
1050																	
1060																	
1070																	

How does microprocessor execute an instruction

Ex: 10 0000 STL RETADR 172063 L = 666600

⇒ store the contents of linkage register into RETADR location

⇒ opcode for STL = 14 (known by microprocessor)



1 7 2063

TA = PC + disp

= 0003 + 063 = 0066

RETADR

ie STL 0066 ⇒ copy the contents of linkage register (666600) into address 0066

note:- If starting address (location) is changed from 0000 to 5000, it works as usual. i.e. for object have format of address. ∴ no need of modification record as before. ⟶ advantage of program block

2.3.5 Control sections and Program linking

→ Control section is a part of the program that maintains its identity after assembly.

→ Each control section can be loaded and relocated independently of the other.

→ Control sections are usually used for subroutines or other logical subdivisions of a program.

→ The programmer can assemble, load and manipulate each of these control sections separately

* → It used a assembler directive : CSECT which indicates the beginning of the control section. where each control section starts its location counter separately

** → when control sections form logically related parts of a program, it is necessary to provide some means for linking them together. This is because instructions in one control section may need to refer to instructions or data located in another control section. and assembler has no idea where exactly the control sections will be located at execution time.

→ Such reference between control sections are called external references.

→ The assembler generate information for each external reference that will allow the loader to perform the required linking.

→ There are two types of ~~extr~~ reference external symbols

(1) External Definition (EXTDEF)

- Symbols that are defined in one section and are used by other sections

syntax: EXTDEF name [,name]

Ex: EXTDET BUFFER, BUFEND

(2) External Reference (EXTREF)

- Symbols that are used in this control sections but are defined in some other control sections.

syntax: EXTREF name [,name]

Ex: EXTREF RDREC, WRREC

Note: To reference an external symbols, extended format instruction (Format 4) is needed.

```
Line    Loc     Source statement                    Object code
   5    0000    COPY    START   0
   6                    EXTDEF  BUFFER,BUFEND,LENGTH
   7                    EXTREF  RDREC,WRREC
  10    0000    FIRST   STL     RETADR              172027
  15    0003    CLOOP   +JSUB   RDREC               4B100000
  20    0007            LDA     LENGTH              032023
  25    000A            COMP    #0                  290000
  30    000D            JEQ     ENDFIL              332007
  35    0010            +JSUB   WRREC               4B100000
  40    0014            J       CLOOP               3F2FEC
  45    0017    ENDFIL  LDA     =C'EOF'             032016
  50    001A            STA     BUFFER              0F2016
  55    001D            LDA     #3                  010003
  60    0020            STA     LENGTH              0F200A
  65    0023            +JSUB   WRREC               4B100000
  70    0027            J       @RETADR             3E2000
  95    002A    RETADR  RESW    1
 100    002D    LENGTH  RESW    1
 103                    LTORG
       0030    *       =C'EOF'                     454F46
 105    0033    BUFFER  RESB    4096
 106    1033    BUFEND  EQU     *
 107    1000    MAXLEN  EQU     BUFEND-BUFFER

 109    0000    RDREC   CSECT
 110                    .
 115                    .       SUBROUTINE TO READ RECORD INTO BUFFER
 120                    .
 122                    EXTREF  BUFFER,LENGTH,BUFEND
 125    0000            CLEAR   X                   B410
 130    0002            CLEAR   A                   B400
 132    0004            CLEAR   S                   B440
 133    0006            LDT     MAXLEN              77201F
 135    0009    RLOOP   TD      INPUT               E3201B
 140    000C            JEQ     RLOOP               332FFA
 145    000F            RD      INPUT               DB2015
 150    0012            COMPR   A,S                 A004
 155    0014            JEQ     EXIT                332009
 160    0017            +STCH   BUFFER,X            57900000
 165    001B            TIXR    T                   B850
 170    001D            JLT     RLOOP               3B2FE9
 175    0020    EXIT    +STX    LENGTH              13100000
 180    0024            RSUB                        4F0000
 185    0027    INPUT   BYTE    X'F1'               F1
 190    0028    MAXLEN  WORD    BUFEND-BUFFER       000000

 193    0000    WRREC   CSECT
 195                    .
 200                    .       SUBROUTINE TO WRITE RECORD FROM BUFFER
 205                    .
 207                    EXTREF  LENGTH,BUFFER
 210    0000            CLEAR   X                   B410
 212    0002            +LDT    LENGTH              77100000
 215    0006    WLOOP   TD      =X'05'              E32012
 220    0009            JEQ     WLOOP               332FFA
 225    000C            +LDCH   BUFFER,X            53900000
 230    0010            WD      =X'05'              DF2008
 235    0013            TIXR    T                   B850
 240    0015            JLT     WLOOP               3B2FEE
 245    0018            RSUB                        4F0000
 255                    END     FIRST
       001B    *       =X'05'                      05
```

Figure 2.16 Program from Fig. 2.15 with object code.

Line Source statement

```
  5    COPY      START   0            COPY FILE FROM INPUT TO OUTPUT
  6              EXTDEF  BUFFER,BUFEND,LENGTH
  7              EXTREF  RDREC,WRREC
 10    FIRST     STL     RETADR       SAVE RETURN ADDRESS
 15    CLOOP    +JSUB    RDREC        READ INPUT RECORD
 20              LDA     LENGTH       TEST FOR EOF (LENGTH = 0)
 25              COMP    #0
 30              JEQ     ENDFIL       EXIT IF EOF FOUND
 35             +JSUB    WRREC        WRITE OUTPUT RECORD
 40              J       CLOOP        LOOP
 45    ENDFIL    LDA     =C'EOF'      INSERT END OF FILE MARKER
 50              STA     BUFFER
 55              LDA     #3           SET LENGTH = 3
 60              STA     LENGTH
 65             +JSUB    WRREC        WRITE EOF
 70              J       @RETADR      RETURN TO CALLER
 95    RETADR    RESW    1
100    LENGTH    RESW    1            LENGTH OF RECORD
103              LTORG
105    BUFFER    RESB    4096         4096-BYTE BUFFER AREA
106    BUFEND    EQU     *
107    MAXLEN    EQU     BUFEND-BUFFER

109    RDREC     CSECT
110    .
115    .         SUBROUTINE TO READ RECORD INTO BUFFER
120    .
122              EXTREF  BUFFER,LENGTH,BUFEND
125              CLEAR   X            CLEAR LOOP COUNTER
130              CLEAR   A            CLEAR A TO ZERO
132              CLEAR   S            CLEAR S TO ZERO
133              LDT     MAXLEN
135    RLOOP     TD      INPUT        TEST INPUT DEVICE
140              JEQ     RLOOP        LOOP UNTIL READY
145              RD      INPUT        READ CHARACTER INTO REGISTER A
150              COMPR   A,S          TEST FOR END OF RECORD (X'00')
155              JEQ     EXIT         EXIT LOOP IF EOR
160             +STCH    BUFFER,X     STORE CHARACTER IN BUFFER
165              TIXR    T            LOOP UNLESS MAX LENGTH
170              JLT     RLOOP          HAS BEEN REACHED
175    EXIT      +STX    LENGTH       SAVE RECORD LENGTH
180              RSUB                 RETURN TO CALLER
185    INPUT     BYTE    X'F1'        CODE FOR INPUT DEVICE
190    MAXLEN    WORD    BUFEND-BUFFER

193    WRREC     CSECT
195    .
200    .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
207              EXTREF  LENGTH,BUFFER
210              CLEAR   X            CLEAR LOOP COUNTER
212             +LDT     LENGTH
215    WLOOP     TD      =X'05'       TEST OUTPUT DEVICE
220              JEQ     WLOOP        LOOP UNTIL READY
225             +LDCH    BUFFER,X     GET CHARACTER FROM BUFFER
230              WD      =X'05'       WRITE CHARACTER
235              TIXR    T            LOOP UNTIL ALL CHARACTERS
240              JLT     WLOOP          HAVE BEEN WRITTEN
245              RSUB                 RETURN TO CALLER
255              END     FIRST
```

Figure 2.15 Illustration of control sections and program linking.

In Fig 2.16, there are three control sections.

1) main program → COPY from line 5 to line 107
2) read subroutine → RDREC from line 109 to line 190
3) write subroutine → WRREC from line 193 to 255.

→ Assembler establishes a separate location counter (beginning at 0) for each control section

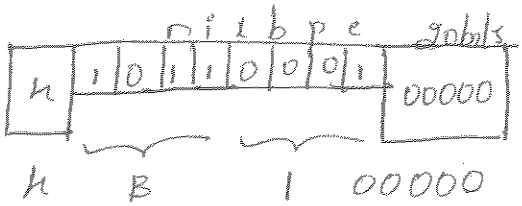
→ Control section names COPY, RDREC, WRREC are not named in EXTDEF because they are automatically considered to be external symbols.

→ Assembler handles the external references as follows

a) 15 0003 CLOOP +JSUB RDREC
(48) (EXTREF)

. Assembler is unaware of RDREC address, so it inserts an address of 3000 and pass this to loader, which is taken care during loading.

. The address of RDREC will have no predictable relationship to anything in the control section by name COPY, therefore relative addressing is not possible. Thus an extended format instruction must be used to provide room for the actual address to be inserted.



b) 160 0017 +STCH BUFFER, X 57900000

→ BUFFER is used in RDREC control section but defined in COPY control section.

c) 190 0028 MAXLEN WORD BUFFEND-BUFFER 000000

→ two external references in the expression BUFEND and BUFFER in WRREC section.

→ The assembler inserts an address of 3000, it passes information to the loader to add to this data area the address of BUFEND and subtract from this data area the address of BUFFER, which results in the desired value.

d) 107 1000 MAXLEN EQU BUFEND-BUFFER

→ Both expressions looks same but the (107 & 190)
difference is here BUFEND and BUFFER are defined and used in the same control section. so value can be calculated immediately.

MAXLEN EQU 1033 - 0033 .

= 1000

* A reference to MAXLEN in the COPY control section will use the definition on line 107, whereas a reference to MAXLEN in RDREC control section will use the definition on line 190.

→ object program

. Along with header record, text record, modification record, two more records are added.

a) Define Record → Information of symbols defined in this control section

col. 1	D
col. 2-7	name of external symbol defined in this control section
col. 8-13	Relative address within this control section (Hexadecimal)
col. 14-73	Repeat information in col. 2-13 for other external symbol

b) Refer Record → symbols that are used as external references in this control section

col. 1	R
col 2-7	name of external symbol refered in this control section
col. 8-73	name of other external reference symbol

c) Modification Record

col. 1	M
col 2-7	starting address of the field modified, in half-byte (hexadecimal
col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field

COPY

H^COPY ^000000^001033

D^BUFFER^000033^BUFEND^001033^LENGTH^00002D

R^RDREC ^WRREC

T^000000^1D^172027^HB100000^03 2023^290000^332007^HB100000^3F2FEC^
032016^0F2016

T^00001D^0D^010003^0F200A^HB100000^3E2000

T^000030^03^454F46

M^000004^05^+RDREC

M^000011^05^+WRREC

M^000024^05^+WRREC

E^000000

RDREC

H^RDREC ^000000^00002B

R^BUFFER^LENGTH^BUFEND

T^000000^1D^B410^B400^B440^77201F^E32018^332FFA^DB2015^A004^332009^
57900000^13850

T^00001D^0E^3B2FE9^131000^4F0000^F1^000000

M^000018^05^+BUFFER

M^000021^05^+LENGTH

M^000028^06^+BUFEND

M^000028^06^-BUFFER

E

WRREC

H^WRREC ^000000^00001C

R^LENGTH^BUFFER

T^000000^1C^B410^771000^E32012^332FFA^53900000^DF2008^B850^
^3B2FEE^4F000005

M^000003^05^+LENGTH

M^00000D^05^+BUFFER

E

2.4 Assembler Design Options

. we will learn two alternatives of the standard two-pass assembler.

a) One-pass Assemblers
b) multi-pass Assemblers

a) One-pass Assemblers

→ As we know already, assembling the forward references is very difficult ∵ we don't know the address. This can be eliminated very easily for data items. That is data items are defined in the source program before they are referenced. (Ex. ~~Assembler~~ declaration of data variable in c).

→ It is not the same for labels on instructions.

Ex:- If the program has a forward jump ie escaping from a loop after testing some condition → here we can't define before itself. Therefore the assembler has to provide the way to handle forward references.

→ Two types of one-pass assemblers

(i) load-go-assembler : Assembler produces object program code directly in memory for immediate execution.

→ Here object program is not written out and no loader is needed.

→ Application : program development and testing.

→ Ex :- A university computing system for student. Here a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

→ load-and-go assembler avoids the overhead of writing the object program out (secondary storage) and reading it back in. ∴ forward references can be handled easily.

→ Avoids usage of forward references

→ Used on systems where external working-storing devices are either slow or not available.

```
Line    Loc     Source statement                Object code

   0    1000    COPY    START   1000
   1    1000    EOF     BYTE    C'EOF'          454F46
   2    1003    THREE   WORD    3               000003
   3    1006    ZERO    WORD    0               000000
   4    1009    RETADR  RESW    1
   5    100C    LENGTH  RESW    1
   6    100F    BUFFER  RESB    4096
   9            .
  10    200F    FIRST   STL     RETADR          141009
  15    2012    CLOOP   JSUB    RDREC           48203D
  20    2015            LDA     LENGTH          00100C
  25    2018            COMP    ZERO            281006
  30    201B            JEQ     ENDFIL          302024
  35    201E            JSUB    WRREC           482062
  40    2021            J       CLOOP           302012
  45    2024    ENDFIL  LDA     EOF             001000
  50    2027            STA     BUFFER          0C100F
  55    202A            LDA     THREE           001003
  60    202D            STA     LENGTH          0C100C
  65    2030            JSUB    WRREC           482062
  70    2033            LDL     RETADR          081009
  75    2036            RSUB                    4C0000
 110            .
 115            .       SUBROUTINE TO READ RECORD INTO BUFFER
 120            .
 121    2039    INPUT   BYTE    X'F1'           F1
 122    203A    MAXLEN  WORD    4096            001000
 124            .
 125    203D    RDREC   LDX     ZERO            041006
 130    2040            LDA     ZERO            001006
 135    2043    RLOOP   TD      INPUT           E02039
 140    2046            JEQ     RLOOP           302043
 145    2049            RD      INPUT           D82039
 150    204C            COMP    ZERO            281006
 155    204F            JEQ     EXIT            30205B
 160    2052            STCH    BUFFER,X        54900F
 165    2055            TIX     MAXLEN          2C203A
 170    2058            JLT     RLOOP           382043
 175    205B    EXIT    STX     LENGTH          10100C
 180    205E            RSUB                    4C0000
 195            .
 200            .       SUBROUTINE TO WRITE RECORD FROM BUFFER
 205            .
 206    2061    OUTPUT  BYTE    X'05'           05
 207            .
 210    2062    WRREC   LDX     ZERO            041006
 215    2065    WLOOP   TD      OUTPUT          E02061
 220    2068            JEQ     WLOOP           302065
 225    206B            LDCH    BUFFER,X        50900F
 230    206E            WD      OUTPUT          DC2061
 235    2071            TIX     LENGTH          2C100C
 240    2074            JLT     WLOOP           382065
 245    2077            RSUB                    4C0000
 255                    END     FIRST
```

Figure 2.18 Sample program for a one-pass assembler.

Assembler generating object codes:
Working process of load-go-assembler

↳ Fig 2.18 shows an eiample for one-pass-assembler

↳ Here all data item definitions are placed ahead of the code that references them.

```
Ex:   01   1000   EOF     BYTE   C 'EOF'
      02   1003   THREE   WORD   3
      03   1006   ZERO    WORD   0
                  :
      06   100F   BUFFER  RESB   H096.
```

↳ The assembler generates object code as it scans the source program

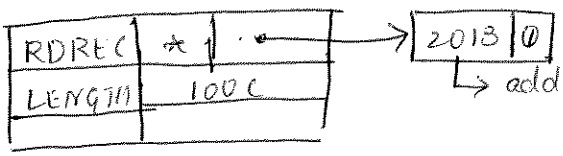
↳ If an instruction operand is a symbol that has not yet been defined (forward reference), the operand address is ommitted during assembly.

↳ The symbol is entered into symbol table if not exists along with a flag indicating undefined.

*1 ↳ The address of the operand field of the instruction that refers to the undefind symbol is added to a list of forward references auociated with the symbol table entry.

```
Ex:   15   2012   CLOOP   JSUB   RDREC   H80000
```

2012 → H8
2013 → ....



↳ address where it has to start later

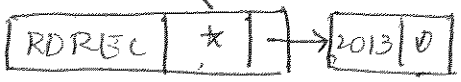
↳ when the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if exists) and the proper address is inserted into any instructions previously generated.

↳ Fig 2.19 shows the object code and symbol table entries as they were scanned. till line no: two

↳ 15 2012 CLOOP JSUB RDREC

↳ undefined

∴ symbol table entry is



2013 is the address location where it has to load once found further.

↳ Same for line nos 30, 35.

Memory Address	Contents			
1000	4SHFH600	00030000	00XXXXXX	XXXXXXXX
1010	XXXXXXXX	XXXXXXX	XXXXXXXX	XXXXXXXX
⋮				
2000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXX1H
2010	1009H8--	--001006	28100630	----H8--
2020	-- 3C2012			
⋮				

SYMBOL TABLE

LENGTH	100C
RDREC	* → 2013 | 0
THREE	1003
ZERO	1006
WRREC	* → 201F | 0
EOF	1000
BYNDFIL	* → 201C | 0
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	2001

Fig 2.19: Object code in memory and symbol table entries after scanning line no

Memory Address	Contents			
1000	45484600	00030000	00XXXXXX	XXXXXXXX
1010	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
:				
2000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXX1H
2010	1009H820	3D00100C	28100630	202HH8--
2020	--3C2012	00100000	100F0010	030C100C
2030	H8------08	1009HC00	00F10010	0004100C
20H0	00100GE0	20393020	H3D82039	28100630
2050	----5H90	0F		

SYMBOL TABLE

LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	* → 201F → 2031
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	* → 2050
RLOOP	20H3

Fig: object code in memory and symbol table entries after scanning line 160 of fig 8.18

1000	45484600	00030000	00XXXXXX	XXXXXXXX
1010	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
:				
2000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXX1H
2010	1009H820	3D00100C	28100630	202HH820
2020	623C2012	00100000	100F0010	030C100C
2030	H8206208	1009HC00	00F10010	0004100C
20H0	00100GE0	20393020	H3D82039	28100630
2050	205B5H90	0F2C203A	38204310	100CHC00
2060	0005OH10	06ED2061	30206550	9001FDC20
2070	612C100C	38206SHC	0000	
2080				

LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	2062
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	205B
RLOOP	2043
OUTPUT	2061
WLOOP	2065

Fig: complete object program in memory and symbol table entries

note: If any symbols in the SYMTAB are still marked with *, should be flagged by the assembler as errors. (undefined symbol error in C long after compiling completely)

↳ The assembler searches SYMTAB for the value of the symbol named END statement and jumps to this location to begin execution of the assembled program.

↳ In load-and-go assembler, the actual address must be known at assembly time.

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR as starting address
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
      if there 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
              begin
                  if symbol value as null
                  set symbol value as LOCCTR and search
                    the linked list with the corresponding
                    operand
                  PTR addresses and generate operand
                    addresses as corresponding symbol
                    values
                  set symbol value as LOCCTR in symbol
                    table and delete the linked list
              end
              else
                insert (LABEL, LOCCTR) into SYMTAB
          end
            search OPTAB for OPCODE
              if found then
                begin
                  search SYMTAB for OPERAND address
              if found then
                if symbol value not equal to null then
                  store symbol value as OPERAND address
                else
                  insert at the end of the linked list
                    with a node with address as LOCCTR
                else
                  insert (symbol name, null)
```

Figure 2.19(c) Algorithm for One pass assembler.

```
            add 3 to LOCCTR
          end
          else if OPCODE = 'WORD' then
            add 3 to LOCCTR & convert comment to
              object code
          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
              convert constant to object code
            end
        if object code will not fit into current
          text record then
          begin
            write text record to object program
            initialize new text record
          end
        add object code to Text record
      end
    write listing line
    read next input line
  end
write last Text record to object program
write End record to object program
write last listing line
end {Pass 1}
```

Figure 2.19(c) *(cont'd)*

references that could not be handled by the assembler. Of course, the objec

iii) Assembler generating object code.

↳ This type of one-pass assembler also works in the same manner as before (load-and-go) except where the definition of symbol is encountered

↳ when a symbol definition is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. ⟹ means they have already been written out as part of a Text Record in the object program.

↳ In such situations, assembler generates another Text record with the correct operand address.

↳ when the program is loaded, this address will be inserted into the instructions by the loader.

↳ The object program for fig 2.18 is shown below during one pass-one.

H^COPY ^ 001000 ^ 00107A

1 T^001000^09^45^F6^000003^000000

2 T^00200F^15^1H1009^48<u>0000</u>^001003^281006^300000^48<u>0000</u>^3C2012

3 T^00201C^02^2024

4 T^002024^19^001000^0C1003^001003^0C1003^48<u>0000</u>^081009^4C<u>0000</u>^F1^001000

5 T^002013^02^203D

6 T^00203D^1E^041006^001006^6E02039^302043^D82039^281006^300000^54900F^2C203A^382043

7 T^002050^02^205B

8 T^00205B^07^10100C^4C0000^05

9 T^00201F^02^2062

10 T^002031^02^2062

11 T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000

E^00200F

- The second text record contains the object code generated from lines 10 through 40 in fig 2.18.
- The operand addresses for instructions on line 15, 30 and 45 has been generated as 0000.
- When definition of ENDFIL on line 45 is encountered, the assembler generates the third Text Record. It indicates that the value 2024 (address of ENDFIL) has to be loaded at location 201C.
- This continues for all the forward references encountered.

b) multi-pass Assemblers

↳ we know that whenever we use 'EQU' assembly directive, any symbol used on the right-hand side should be defined previously in the source program. This is not true always.

↳ Ex:-
```
ALPHA   EQU   BETA
BETA    EQU   DELTA
DELTA   RESW  1
```

↳ As we see above, we have multiple forward references ie Alpha depends on value of Beta, Beta depends on value of delta.

↳ Any assembler that make only two sequential pass over the source program cannot resolve such a sequential sequence of definitions.

↳ To overcome this we go for multi-pass assembler which make as many pass as needed to process the definitions of symbols.

↳ It is not necessary for multi-pass assembler to make more than two pass over the entire program.

** ↳ Instead, the portions of the program that involve forward references in symbol definitions are saved

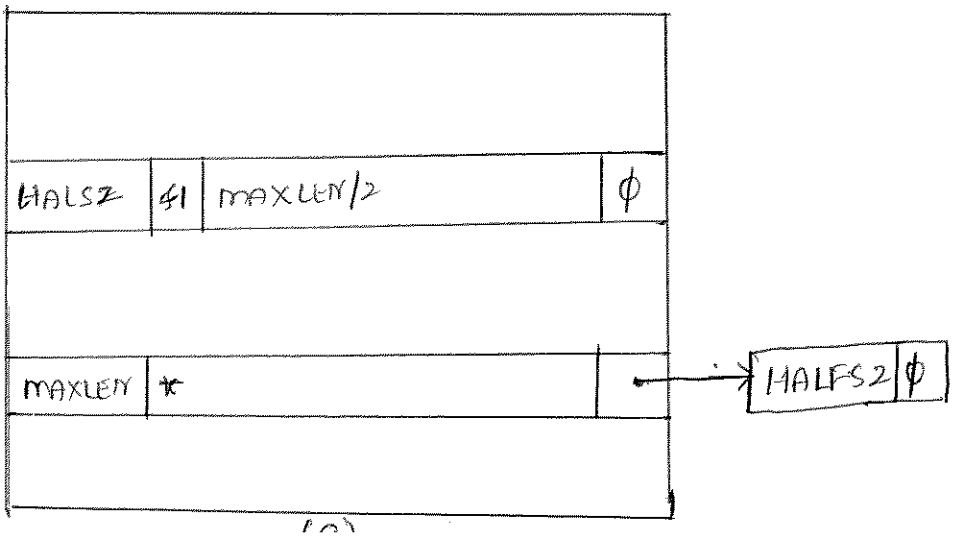
during pass-1, Additional pass through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass-2.

↳ SYMTAB stores the symbol, definition, symbols which are dependent on this, no of symbols dependent on this symbol.

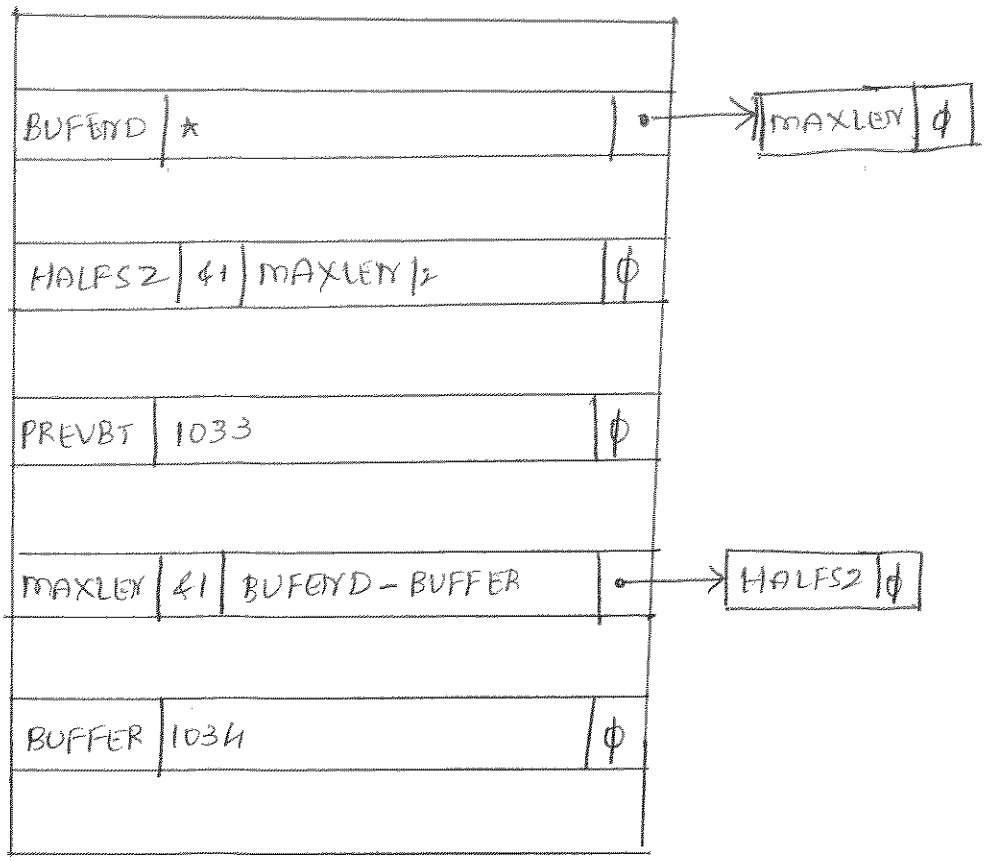
Ex:-

	LOC			
1		HALFSZ	EQU	MAXLEN/2
2		MAXLEN	EQU	BUFEND - BUFFER
3		PREVBT	EQU	BUFFER - 1
4	103H	BUFFER	RESB	4096 ⇒ $(1000)_{16}$
5	203H	BUFEND	EQU	* - indicate the current location counter value

→ below fig shows the symbol table entry when it reads line no. 1 indicating that Halfsz depends on maxlen value

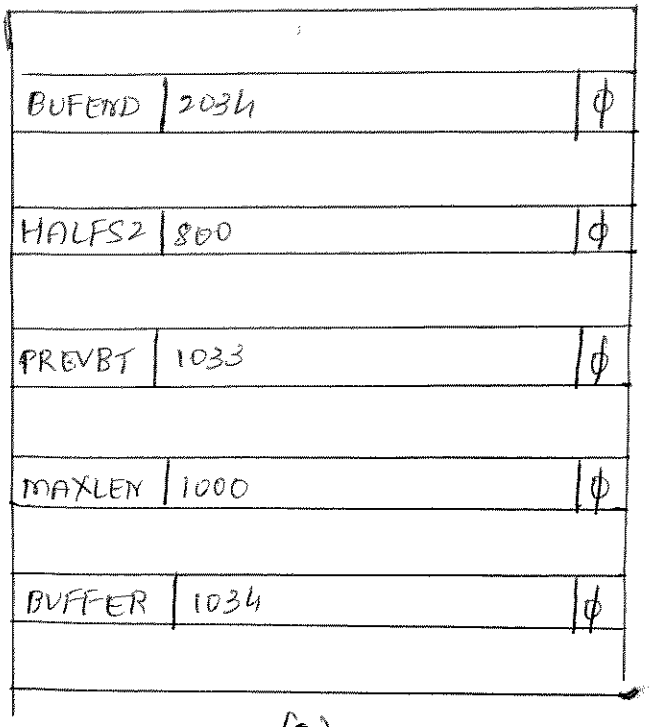


(a)



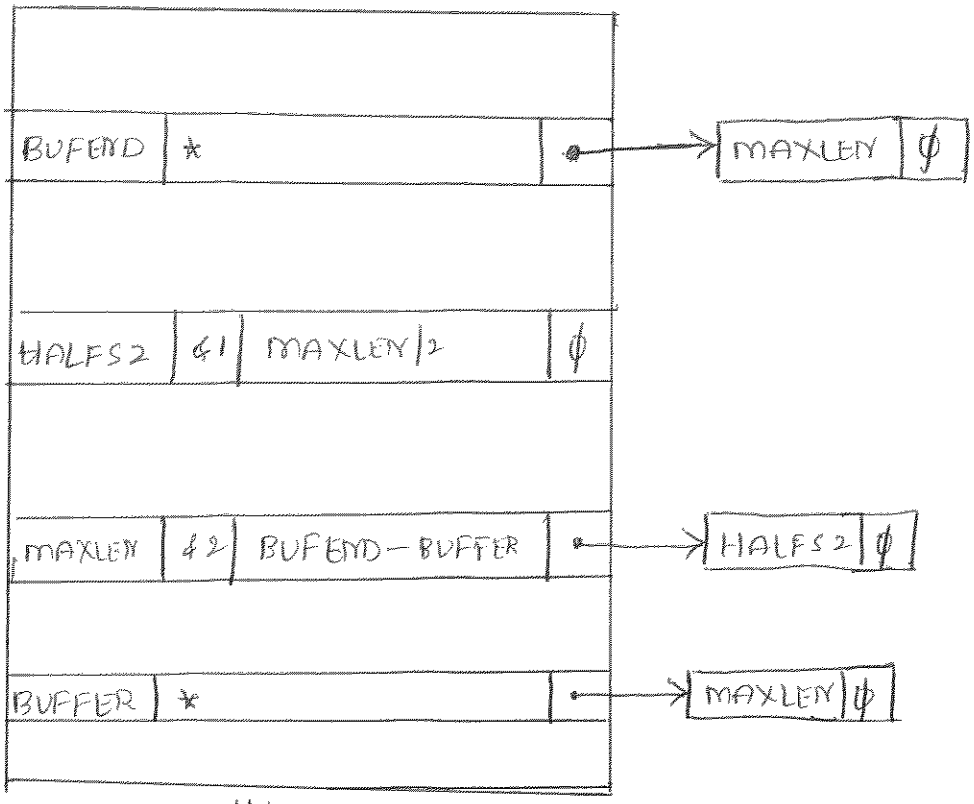
(d)

↳ Fig (d) shows the symbol table entry after scanning line no. (11), whose location counter value is 1034. MAXLEN dependency value &2 is reduced to &1.



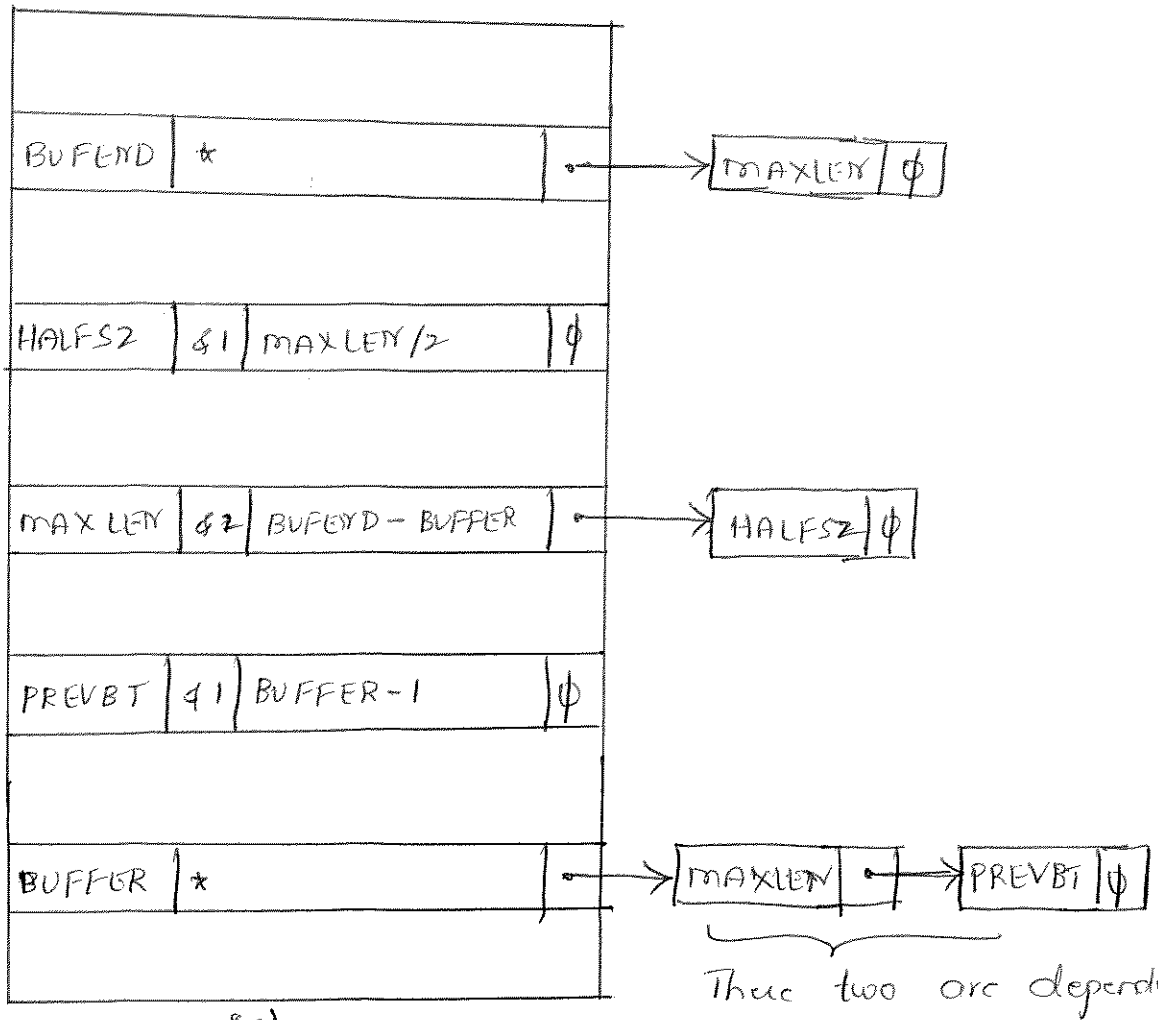
(e)

↳ fig (e) indicates the complete symbol table entry proces



(b)

↳ Fig (b) shows symbol table entry after reading line no. 9
As we see, MAXLEN depends on 2 symbols BUFEND & BUFFER . ∴ MAXLEN &9.



(c)

CHAPTER 4

Macro Processors

Chapter 4 : Macroprocessor

→ We are going to study definition of macro
→ what is the need for macro
→ Data structure used in macro invocation and expansion

Macro : It is a single instruction that expands automatically into a set of instructions to perform a particular task. Thus macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macroprocessor.

Ex: In SIC/XE, 7 instructions (STA, STB, etc) is required to save the contents of all registers before calling a subprogram, but by using a macro instruction, the programmer can write a single statement like SAVEREGS. The SAVEREGS macro instruction would be expanded into seven instruction required to save the contents of all registers

↳ LOADREGS macro instruction would be used to reload the register contents after returning from the subprogram

* → Macro processor performs no analysis of the text it handles and is not concerned about the meaning of the involved statements during macro expansion. ∴ The design of a macro processor is machine independent.

4.1 Basic macro processor Functions.

→ Macros refers to a set of statements which will replace every invocation to it. The two concepts associated with macros are :

(i) Macro definition
(ii) Macro Expansion

(i) Macro Definition :

→ Consists of macro prototype, one or more module and macro preprocessor.

→ macro definition is a set of statements present in between a macro header statement (MACRO) and a macro end statement (MEND). MACRO and MEND are two assembly directives used in macro definition

* Syntax of macro prototype :

<macro name> [< formal parameter specification>[,...]]

where <macroname> : The mnemonic field of a statement

<formal parameter> : <parmater name>[<parameter kind>]

In each parameter begins with '&'

Syntax: Macro call

<macro name> [<actual parameter specification> [, ...]]

→ In general, macro definition is given as

~~NAME MACROPARAMETERS~~

```
NAME    MACRO    PARAMETERS
  :
  :
body; the statements which are generated as the expansion
      of the macros
  :
  :
MEND
```

// macro invocation is as

```
  :
  :
NAME   PARAMETERS
  :
  :
```

→ As in fig 4.1, macro definition is at line 10

```
10    RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH
  :
  :
95    MEND
```

→ macro invocation in fig 4.2

```
190   RDBUFF   F1, BUFFER, LENGTH
```

(ii) Macro Expansion

→ Macro invocation statements are the statements of the macro body that are expanded each time the macro is invoked.

→ The program in fig 4.1 is supplied as input to a macro processor.

→ Fig 4.2 shows the output that would be generated.

→ In expanding the macro invocation on line 190, argument F1 is substituted for the parameter &INDEV, BUFFER is substituted for &BUFADR, LENGTH is substituted for &RECLTH.

→ Line 190a through 190m show the complete expansion of the macro invocation on line 190.

→ Same in line 210a through 210h for WRBUF MACRO

→ As we see the macros body does have any labels ; for example, line 140 → JEQ *-3 and line 155 JLT *+14. If we put label, it would be generated twice on line 210d and 220d resulting in an error (duplicate label definition) when the program is assembled.

→ *-3, *+9, ... indicate PC-relative addressing

```
Line    Source statement
        LABEL     OPCODE    OPERAND.
5       COPY      START     0                 COPY FILE FROM INPUT TO OUTPUT
10      RDBUFF    MACRO     &INDEV,&BUFADR,&RECLTH
15      .
20      .         MACRO TO READ RECORD INTO BUFFER
25      .
30                CLEAR     X                 CLEAR LOOP COUNTER
35                CLEAR     A
40                CLEAR     S
45                +LDT      #4096             SET MAXIMUM RECORD LENGTH
50                TD        =X'&INDEV'        TEST INPUT DEVICE
55                JEQ       *-3               LOOP UNTIL READY
60                RD        =X'&INDEV'        READ CHARACTER INTO REG A
65                COMPR     A,S               TEST FOR END OF RECORD
70                JEQ       *+11              EXIT LOOP IF EOR
75                STCH      &BUFADR,X         STORE CHARACTER IN BUFFER
80                TIXR      T                 LOOP UNLESS MAXIMUM LENGTH
85                JLT       *-19                HAS BEEN REACHED
90                STX       &RECLTH           SAVE RECORD LENGTH
95                MEND
100     WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
105     .
110     .         MACRO TO WRITE RECORD FROM BUFFER
115     .
120               CLEAR     X                 CLEAR LOOP COUNTER
125               LDT       &RECLTH
130               LDCH      &BUFADR,X         GET CHARACTER FROM BUFFER
135               TD        =X'&OUTDEV'       TEST OUTPUT DEVICE
140               JEQ       *-3               LOOP UNTIL READY
145               WD        =X'&OUTDEV'       WRITE CHARACTER
150               TIXR      T                 LOOP UNTIL ALL CHARACTERS
155               JLT       *-14                HAVE BEEN WRITTEN
160               MEND
165     .
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
```

Figure 4.1 Use of macros in a SIC/XE program.


```
Line    Source statement
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 MAXIMUM RECORD LENGTH
190e              TD        =X'F1'            TEST INPUT DEVICE
190f              JEQ       *-3               LOOP UNTIL READY
190g              RD        =X'F1'            READ CHARACTER INTO REG A
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 MAXIMUM LENGTH
190l              JLT       *-19                HAS BEEN REACHED
190m              STX       LENGTH            SAVE RECORD LENGTH
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
210a              CLEAR     X                 CLEAR LOOP COUNTER
210b              LDT       LENGTH
210c              LDCH      BUFFER,X          GET CHARACTER FROM BUFFER
210d              TD        =X'05'            TEST OUTPUT DEVICE
210e              JEQ       *-3               LOOP UNTIL READY
210f              WD        =X'05'            WRITE CHARACTER
210g              TIXR      T                 LOOP UNTIL ALL CHARACTERS
210h              JLT       *-14                HAVE BEEN WRITTEN
215               J         CLOOP             LOOP
220     .ENDFIL   WRBUFF    05,EOF,THREE      INSERT EOF MARKER
220a    ENDFIL    CLEAR     X                 CLEAR LOOP COUNTER
220b              LDT       THREE
220c              LDCH      EOF,X             GET CHARACTER FROM BUFFER
220d              TD        =X'05'            TEST OUTPUT DEVICE
220e              JEQ       *-3               LOOP UNTIL READY
220f              WD        =X'05'            WRITE CHARACTER
220g              TIXR      T                 LOOP UNTIL ALL CHARACTERS
220h              JLT       *-14                HAVE BEEN WRITTEN
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
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

```
1  MACROS   MACRO    {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}
```

(a)

```
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}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.

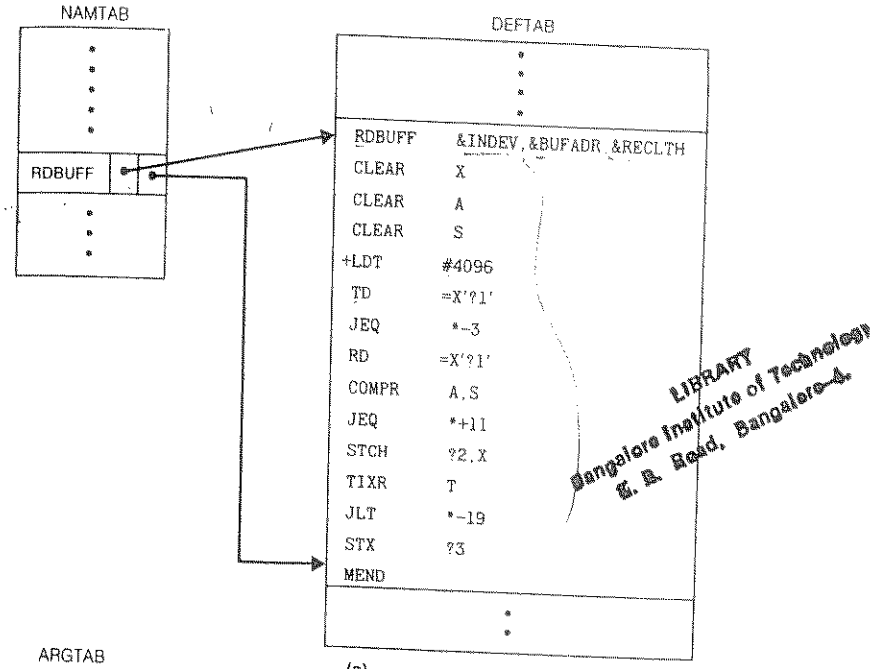


Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

4.1.3. Macro Processor Algorithm and Data structures

→ For designing two pass macro processor, all macro definitions are processed during pass-1 and all macro invocation statements are expanded during pass-2.

→ The two-pass macro processor would not allow the body of one macro instruction to contain definition of other macros, ∴ all macros defined during the pass1 before any macro invocation were expanded.

→ Example of recursive macro definition is shown in Fig 4.3 (a) for SIC machine and Fig 4.3 (b) for SIC/XE machine

→ The same program can be run on either a SIC machine or SIC/XE machine. Invocation of MACROS or MACROX is only changed for use.

→ A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition provided that a macro definition of a macro should appear before the invocation.

Datastructures for one-pass macro processor. (Fig 4.4)

→ There are three main data-structures involved

(i) Definition table (DEFTAB)

↳ It stores the macro definitions which contains the macro prototype and the statements that make up the macro body.

↳ Comment lines are omitted ∵ they are not part of the macro expansion

↳ References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

(ii) Name Table (NAMTAB)

↳ it stores the macro name, which serves as index to DEFTAB.

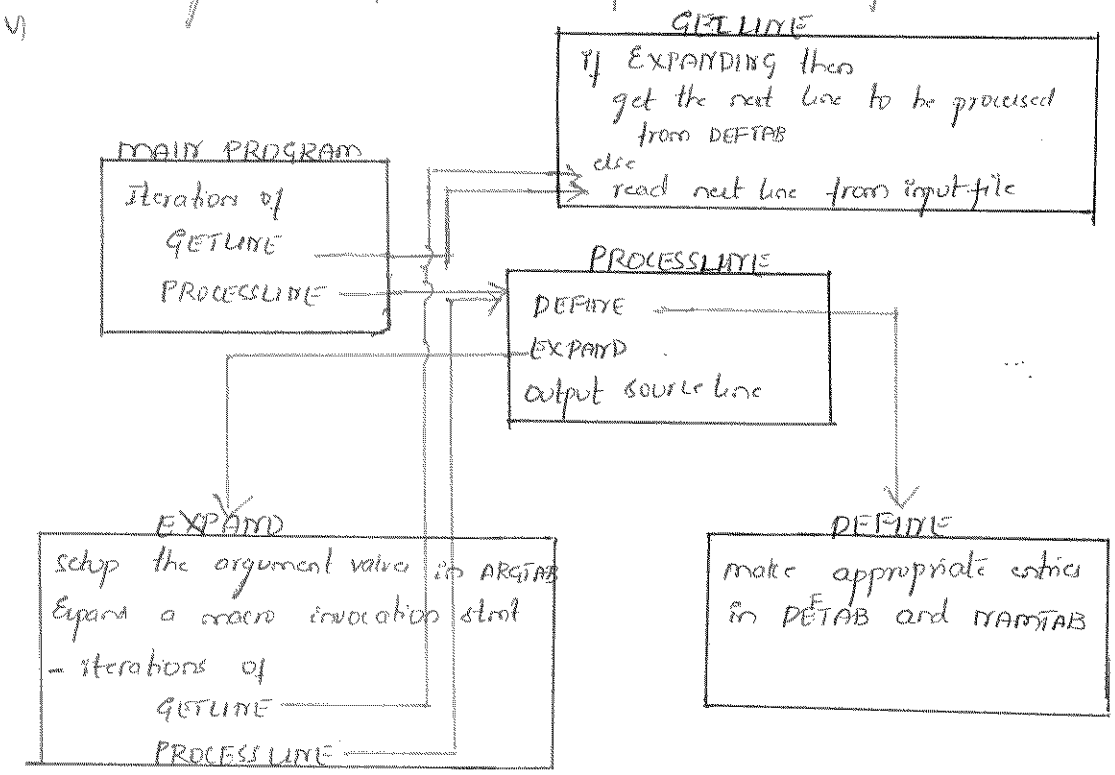
↳ For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in ~~DEFTAB~~ DEFTAB.

(iii) Argument Table (ARGTAB)

↳ used during the expansion of macro invocations.

↳ when a macro invocation stmt is recognized, the arguments are stored in ARGTAB according to their position in the argument list.

↳ when it is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.




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


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}
```

Figure 4.5 Algorithm for a one-pass macro processor.


```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL := 1
        while LEVEL > 0 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
        write 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}
```

Figure 4.5 *(cont'd)*

Handling nested macro definition within macros

→ In DEFTAB (define procedure), when a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached. This will not work for nested macro defn : the first MEND encountered in the inner macro will terminate the whole macro definition proces

→ To solve this problem, a DEFINE procedure is used which maintains a counter named LEVEL. The LEVEL value is incremented by 1 when MACRO directive is read. The value is determined by 1 when MEND directive is read. when LEVEL value becomes 0, the MEND that corresponds to the original MACRO directive has been found. This process is very much like matching left and right parantheis when scanning an arithmetic expression.