# Chp 1: Introduction to System Software

Date | Jan 4, 2018
Page |

* System Software
- System software consist of a variety of programs that support the operation of computer.
- It is a set of programs to perform a variety of system functions such as resource management, I/O management and storage management
- System software defters from application software in machine dependency.
- An application program is primarily concern with the solution of some problem using computer as a tool.
- System Program are intended to support the operation & use of computer itself. rather than any particular application.
- System programs are usually related to the architecture of machine on which they are run.
- Like assemblers translates mnemonics code into machine code so the instruction formats and addressing modes are of direct concern in assembler design.
- There are some aspects of system software that donot directly depend upon the type of commuting system being supported which are known as machine independent feature like the general design and logic of an assembler.

Typeo of System Softwoare

1. Operating System
2. ~~Trans~~ language Translator
   - Complea
   - Interpreter
   - Assembler
3. Loader
4. Linker
5. Macro - processor.

Date
Page

* Simple Instructional Computer (SIC)

1) ~~Simple~~ SIC Simple

Memory:-

8 bit = 1 byte

1 word = 3 bytes

Memory size = 32 kB

$= 2^{15}$ byte

∴ 15 ~~bytes~~ bits needed to represent memory location.

byte addresable.

i.e. Memory location to each byte.

Registers.

| | | | |
|---|---|---|---|
| (Accumulator) | A | → 0 | 24 bits |
| (index) | X | → 1 | |
| (linkage) | L | → 2 | |
| (Program Counter) | PC | → 8 | |
| (Status word) | SW | → 9 | |

→ A is a special purpose register used for arithmetic operation.

→ X is used for addressing

→ L Stores the return address of the jump to subroutine. (JSUB)

→ PC Contain address of current instruction being executed.

→ SW contains variety of info including conditional code.

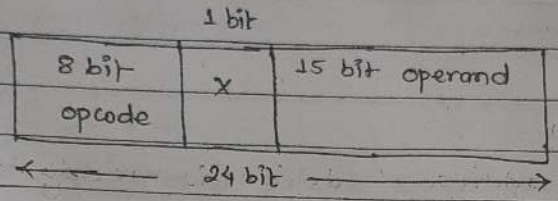
Data Formats.

- Size of integer = 24 bit
- Negative numbers are represented in 2's complement format.

Date
Page

- characters are represented by using 8-bit ASCII code.
- float point data can not be represented.

## Instruction Formats:-




x = 0 → Direct addressing mode. here TA = address

x = 1 → Indexed " "

here, Target Address(TA) = address + (X)

↓ represent content of index register.

eg:- if address = 450, x = 10

∴ TA = 450 + 10

= 460

## Instruction Set

- includes instructions like

i) Data management instruction

LDA, LDX, STA, STX

ii) Arithmatic operating instruction

ADD, SUB, MUL, DIV

iii) Branching instruction

JLT, JEQ, JGT

iv) Subroutine linkage

RSUB, JSUB

## Input and Output.

→ I/O is performed by transferring one byte at a time to or from rightmost 8 bit of register A.

→ each device is assigned unique 8 bit code.

→ 3 I/O instructions.

i) Test Device (TD)

- Test whether addressed device is ready to send or receive a byte of data

ii) a program must wait until the device is ready and then execute Read Data (RD) or write Data (WD)

iii) Sequence must be repeated for each byte of data to be read or written

2) SIC XE

### Memory

Memory size = 1 MB = $2^{20}$ byte

i.e 20 bits are used to represent memory location.

### Registers

- Same as that of SIC simple
- additional registers

B → 3
S → 4
T → 5
} 24-bit

F → 6 } 48-bit

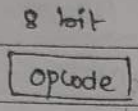
Date
Page

## Data Format :

Size of float point number = 48 bit

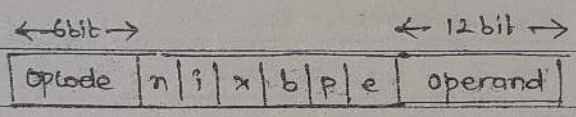
## Instruction Format :-

1) 1 byte instruction

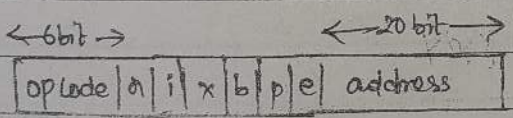



2) 2 byte instruction


3) 3 byte instruction




4) 4 byte instruction




Note :- if e=0, 3 byte instruction

if e=1, 4 " "

immediate नहीं होनी TA = displacement.

Date
Page

n=0, i=0 } Direct addressing mode
n=1, i=1 }

n=0, i=1 → immediate addressing mode.
n=1, i=0 → indirect addressing mode

x=0 → No indexing
x=1 → Indexed addressing

b=0 } absolute addressing mode
p=0 }

p=1 → PC relative
TA = PC + displacement

Note:
displacement = उहाँको value.
operand or addr

b=1 → base relative
TA = base + displacement

{ 0 ≤ base ≤ FFF
{ -800 ≤ PC ≤ 7FF } Hex

OR

{ 0 ≤ Base ≤ 4095
{ -2048 ≤ PC ≤ 2047 } decimal

e=0, 3 byte instruction
e=1, 4 " "

Date
Page

eg:-

given

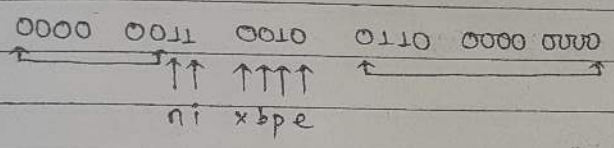
(B) : 006000

(PC) : 003000

(X) : 000090

Now, calculate TA and value located into accumulator

① 032600




here, n=1, i=1 → Direct addressing mode

e=0, 3 byte

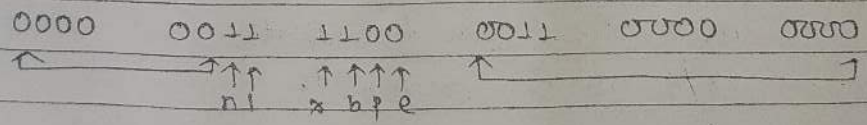
p=1 → PC relative

= 003000 + 600

= 003600

value store is 103000

② 03C300




n=1, i=1 → Direct

e=0, 3 byte

~~p=0 → absolute~~

b=1, base relative

TA = 006000 + 300

= ~~006300~~

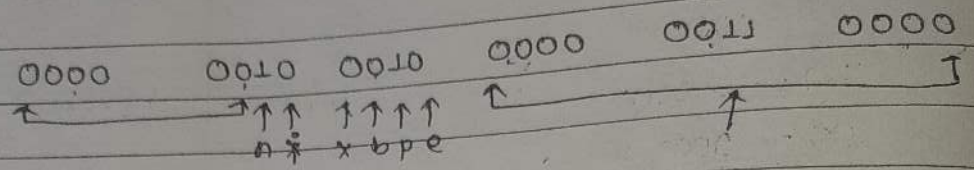
006300 + 000090

006390

value store is 00C303

| 3030 | 003600 |
|---|---|
| 3600 | 103000 |
| 6390 | 00C303 |
| C303 | 003030 |

③ 022030



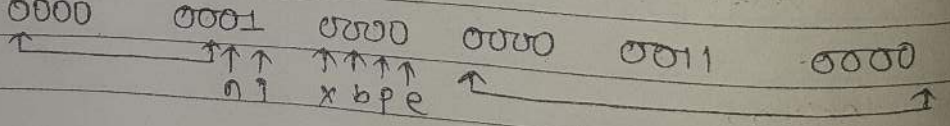

n=1, i=0 → Indirect
e=0, 3 byte
p=1, pc relatative

TA = pc + displ
= 3000 + 30
= 3030

value stone is ~~003600~~ 103000

④ 010030
⑤ 003600
⑥ 0310C303

⑦ →




n=0, i=1 → immediate
p=0, b=0 → absolute
e=0, format 3
x=0 no indexing

TA = displacement
= 030

∴ value = 030

Instructions

1. load store (LDA, STA, LDX, STX etc)
2. integer arithmetic operations (ADD, SUB, MUL, DIV)
3. Compare (Comp)
4. Conditional Jumps (JCT, JEQ, JGT)
5. subroutine linkage (JSUB, RSUB)
6. Input output Control (RD, WD, TD)
7. Floating point arithmetic (ADDF, SUBF, MULF, DIVF)
8. COMR, SHIFTR, ADDR, SUBR, MULR, DIVR

Input/Output.

→ $2^8$ (256) I/O devices may be attached, each has its own unique 8-bit address.

→ 1 byte of data will be transfered to 1 from rightmost 8 bit of register A.

→ There I/O instructions are provided

1. RD - read data from I/O device into A.
2. WD - write data to I/O device from A.
3. TD - Test data determine if addressed I/O ~~is read~~ device is ready to send/Receive byte of data

Other instructions.

SIO — Start I/O

HIO — Halt I/O

TIO — Test I/O

Date
Page

Deadline:- Jan 30th

Assignment 1.

1. Differentiate between RISC and CISC architecture.
2. Write about architecture of VAX.
3. Write about registers and data format of pentium Pro architecture.
4. Write about ULTRA SPARC architecture.
5. Write about Data format and addressing modes of power pc architecture.
6. Write about Cray T3E architecture.

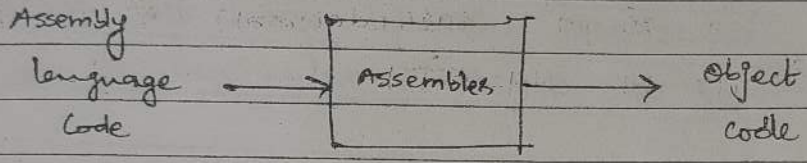
Chp:- Assembler

Date
Page

→ Assembler is a system s/w which is used to convert an assembly language program into its equivalent object code.

→ The input to the assembler is a source code written in assembly language (using mnemonic codes) and output is object code.

→ The design of an assembler depends upon machine architecture or the language used is mnemonic language




Fundamental functions of Assembler

→ Translating mnemonic operation code to their machine language equivalents.

→ ~~Assemb~~ Assigning machine address to symbolic labels used by programmer.

→ in simple SIC Assembler program

① Operand with 'X' modifier indicates Indexed addressing

② "." in line represent comments only.

* Assembler Directives.

→ directives gives information to assembler.

→ No memory location, no object code

④ START :- Specify name and start address of program.

⑪ END :- indicate end of program and specify first executable instruction in prog.

(III) BYTE :- to reserve memory.

(IV) WORD :- to reserve one word of memory and store constant in memory.

(V) RESW :- to reserve specified word of memory but do not store constant in memory.

(VI) RESB :- to reserved specified byte of memory.

chart SIC simple (First program)

No. 105 मा 4096 (decimal) लाई Hex मा लाने आनि त्यो Loc मा add गर्ने।

Hex मा 2 digit 1 byte हुन्छ।

225 मा 50 1039

0101 0000 0001 0000 0011 1001

Buffer, x

x present भको x=0 , change गर्ने आनि 1 लैजाने

1010 0000 1001 0000 0011 1001

∴ 509039 //

Date
Page

→ The given program contains a main routine that read records from an input device (code F1) and copies them to an output device (code 05).

→ The main routine calls ~~instruction~~ subroutines

① RDREC:- to read a record into a buffer.

② WRREC:- to write record from buffer to output device.

→ each of each record is marked with null character.

* SIC simple Assembler functions

→ Convert mnemonic operation codes into their machine language equivalent (like translate STL to 14 in line 10)

→ Convert symbolic operands to their equivalent machine address (like translate RETADR to 1033)

→ Build machine instruction in proper format.

→ Convert data constant specified in SIC program into informed machine representation. (like translate EOF to 454F46)

→ write object program and assembly listing.

→ Among these functions, only member 2 function can't be ~~easily~~ easily accomplished by sequential processing of source program are line at a time.

→ in line 10,

10 1000 first STL RETADR 141033

- here, for label RETADR reference is defined later in program. This is called forward referencing, which can't be solved by ~~simple~~ single pass.

- So, must assembler makes two passes over the program. Here first pass scans the source for Label definition and assigns address (LOC). Second pass performs most of actual translation.

→ The assembler must write the generated object code onto some output device. This object program will later be loaded into memory for execution.

→ object program format contains three types records.

1) Header record
   - Contains program name, starting address and length.

2) Text record
   - Contains machine code and data of program.

3) End record
   - marks end of object program and specifies address in program where execution is to begin

Record Format

1) Header record

| | |
|---|---|
| Col 1 | H |
| Col 2 - 7 | program name |
| Col 8 - 13 | Starting address of object program. |
| Col 14 - 19 | length of object program. |

2) Text record

| | |
|---|---|
| Col 1 | T |
| Col 2 - 7 | Starting address of object code in this record. |
| Col 8 - 09 | length of object code in records. |
| Col 10 - 69 | object codes. |

3) End record

| | |
|---|---|
| Col 1 | E |
| Col 2 - 7 | Address of first executable instruction in object program. |

## SIC simple (First Program)

| Line | LOC | SYMBOL | OPCODE | OPERAND | OBJECT CODE |
|---|---|---|---|---|---|
| 5 | 1000 | COPY | START | 1000 | |
| 10 | 1000 | FIRST | STL | RETADR | 141033 |
| 15 | 1003 | CLOOP | JSB | RDREC | 482039 |
| 20 | 1006 | | LDA | LENGTH | 001036 |
| 25 | 1009 | | COMP | ZERO | 281030 |
| 30 | 100C | | JEQ | ENDFIL | 301015 |
| 35 | 100F | | JSUB | WRREC | 48 2061 |
| 40 | 1012 | | J | CLOOP | 3C1003 |
| 45 | 1015 | ENDFIL | LDA | EOF | 00102A |
| 50 | 1018 | | STA | BUFFER | 0C1039 |
| 55 | 101B | | LDA | THREE | 00102D |
| 60 | 101E | | STA | LENGTH | 0C1036 |
| 65 | 1021 | | JSUB | WRREC | 482061 |
| 70 | 1024 | | LDL | RETADR | 081033 |
| 75 | 1027 | | RSUB | | 4C0000 |
| 80 | 102A | EOF | BYTE | C'EOF' | 454F46 |
| 85 | 102D | THREE | WORD | 3 | 000003 |
| 90 | 1030 | ZERO | WORD | 0 | 000000 |
| 95 | 1033 | RETADR | RESW | 1 | |
| 100 | 1036 | LENGTH | RESW | 1 | |
| 105 | 1039 | BUFFER | RESB | 4096 | |
| 110 | | Subroutine to read record into buffer | | | |
| 125 | 2039 | RDREC | LDX | ZERO | 041030 |
| 130 | 203C | | LDA | ZERO | 001030 |
| 135 | 203F | RLOOP | TD | INPUT | E02050 |
| 140 | 2042 | | JEQ | RLOOP | 302030 |
| 145 | 2045 | | RD | INPUT | D8205D |
| 150 | 2048 | | COMP | ZERO | 281030 |
| 155 | 204B | | JEQ | EXIT | 302057 |
| 160 | 204E | | STCH | BUFFER | 541039 |
| 165 | 2051 | | TIX | MAXLEN | 2C205E |
| 170 | 2054 | | JLT | RLOOP | 382053 |
| 175 | 2057 | EXIT | STX | LENGTH | 101036 |
| 180 | 205A | | RSUB | | 4C0000 |
| 185 | 205D | INPUT | BYTE | X'F1' | F1 |
| 190 | 205E | MAXLEN | WORD | 4096 | 001000 |
| 195 | | Subroutine to write record from buffer | | | |
| 210 | 2061 | WRREC | LDX | ZERO | 041030 |
| 215 | 2064 | WLOOP | TD | OUTPUT | E02079 |
| 220 | 2067 | | JEQ | WLOOP | 302064 |
| 225 | 206A | | LDCH | BUFFER, X | 509039 |
| 230 | 206D | | WD | OUTPUT | DC2079 |
| 235 | 2070 | | TIX | LENGTH | 2C1036 |
| 240 | 2073 | | JLT | WLOOP | 382064 |
| 245 | 2076 | | RSUB | | 4C0000 |
| 250 | 2079 | OUTPUT | BYTE | X'05' | 05 |
| 255 | 207A | | END | FIRST | |

SIC simple ①

... पिछले सबै लेख्ने 69/2

16 (Hex) Date | Page |

Eg:- SIC simple.

H^COPY^001000^00107A

T^001000^1E^141033^482039^.....^00102D

T^00101E^15^0C1036^482061^.....^000000

T^002039^1E^041030^001030^.....^38203F

T^002057^1C^101036^4C0000^.....^2C1036

T^002073^07^382064^...^05

E^001000

* Function of two pass assembler.

Pass 1

→ Assign address to all ~~statements~~ in program.

→ Save addresses assigned to all labels.

→ perform some processing of assembler directives.

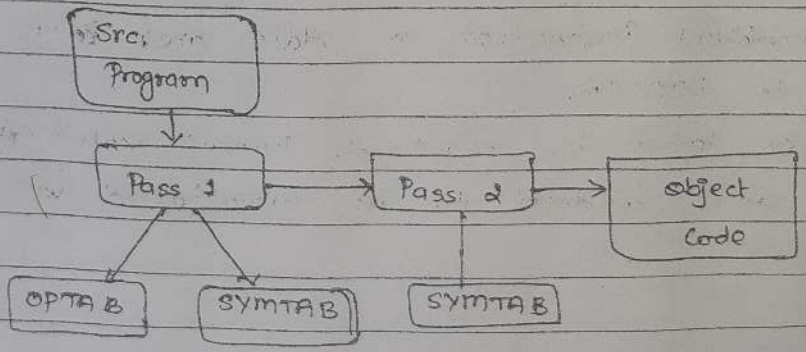
Pass 2

→ Assemble instruction (translating operation code and looking up address)

→ generate data values defined by BYTE, WORD.

→ perform processing of directions.

→ Write object program.



Date
Page

Assembler Algorithm and Data structure

→ Two major internal data structures are

① Operation code table (OPTAB)

used to lookup mnemonic operation code and translate them into their machine equivalent.

② Symbol Table (SYTAB)

used to store values assigned to tabels.

Symbol Table

| symbol | value |
|---|---|
| COPY | 1000 |
| FIRST | 1000 |
| CLOOP | 1003 |
| | |

* Location ~~Counter~~ Counter (LOCCTR)

→ variable used to help in assignment of address.

→ it is initialized to the begining address specified in START statement.

→ after each source statement is processed, length of assembled instruction or data area is added to LOCCTR.

→ whenever label is reached, current value of LOCCTR gives address to be associated with that label.

Date
Page

* Operation Code Table (OPTAB)

→ Contains mnemonics operation and its machine language equivalent.

→ also contains information about instruction format and length.

→ in pass 1, OPTAB is used to lookup and validate operation code in source program.

→ in pass 2, used to translate operation code to machine language equivalent.

→ organised as a Hash Table.

* SYMTAB

→ include name and value (address) for each label.

→ include flags to indicate error condition.

→ in pass 1, labels are entered into SYMTAB, along with assigned address from LOCCTR.

→ in pass 2, symbols used as operands are lookup in SYMTAB to get address.

| Line | Symbol | opcode | (operand) Exp |
|---|---|---|---|
| 10 | STRCPY | START | 1000 |
| 20 | FIRST | LDX | ZERO |
| 30 | MOVECH | LDCH | STR1, X |
| 40 | | STCH | STR2, X |
| 50 | | TIX | ELEVEN |
| 60 | | JLT | MOVECH |
| 70 | | BYTE | C'ABCD' |
| 80 | | RESB | 11 |
| 90 | | WORD | 0 |
| 100 | | WORD | 11 |
| 110 | | END | FIRST |

| mnemonic | opcode |
|---|---|
| LDCH | 50 |
| LDX | 04 |
| STCH | 54 |
| JLT | 38 |
| TIX | 2C |

Date
Page

# * Machine Dependent assembler features

→ Consider assembler of SIC/XE version as given.

→ Here,

① indirect addressing
- adding prefix @ to operand.

② Immediate
- adding prefix # to operand.

③ direct
- only operand.

→ direct BASE is used for base relative addressing.

→ PC relative/base relative addressing — op m.

→ indirect addressing — op @m.

→ immediate " — op #m

→ Extended format — top m

→ Index addressing — op m,x

→ register to register — comp r

→ instructions that refer to memory are normally assembled using either the program counter relative or base relative mode.

→ register to register instructions are used wherever possible and are faster than corresponding register to memory operation because they shorter and do not require another memory reference.

## Translation:

→ Register Translation:
- register name (A, X, L, B, S, T, F, PC, SW)
  with values (0, 1, 2, 3, 4, 5, 6, 8, 9)
- preloaded in SYMTAB

→ Address Translation:

- most register-memory instruction use program counter relative or base relative addressing.

- Format 3:
  - 12 bit address field ~~base relative~~
  ~~0 ↔ 4095~~
  base relative 0 ↔ ~~4095~~ 4095
  pc relative - 2048 ↔ 2047

- format 4:
  20 bit address field

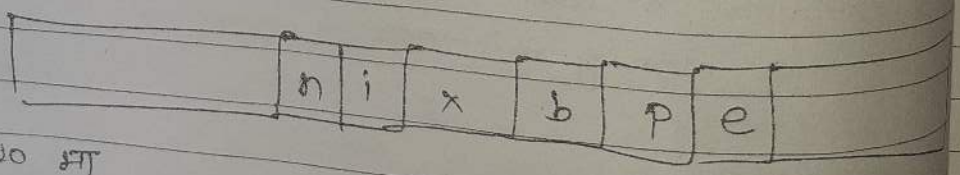
Note:- ( Example 2 (SIC/XE) )

① If simple ~~operand~~ opcode eg: STL, it's format 3.
② If + in opcode eg: +JSUB, it's format 4.
③ If R at last eg: CLEAR, it's format 2.

direct भयो भने n=1, i=1

first assumption pc relative हुन्छ (पहिले)

TA भने को operand को point गरेको,




Line 30 मा

① Step 1 STL को value 14
② Step 2 00ni then step1 + step2
③ TA= ~~Step 3~~ displacement = TA - (PC / Base)

Date
Page

④ Step 4

Step1 + Step2 + Step3 (3 byte only).

यदि Negative छ भने 2's complement गर्ने

तरिका FFF बाट घटाउने अनि 1 जोड्ने //

$$\begin{array}{r} FFF \\ -\ 014 \\ \hline FEB \\ +1 \\ \hline FEC \end{array}$$

line 160

displacement = T.A - PC

= 0036 − 1051

= −101B

Not in PC range

∴ displacement = T.A − Base

= 0036 − 0033

= 0003

Note Base मा First base हेर्ने अनि त्यसको operand को address लिने

0001 0000
0003

1 3

Line 175

x b p e
0 1 0 0

STX
J0
3

13 4 0000

displacement = 0033 − 1059
= −1026
= 0033 + [illegible]
0033
= 0000

relative

→ PC relative or base addressing mode is preferred over direct addressing mode as it can save one byte from using format 3 rather 4.

→ reduce program storage space.

→ relocation will be easier.

* Difference between SIC and SIC/XE program

→ register to register instructions are used whenever possible to increase execution speed.

→ immediate addressing mode indirect addressing mode is used whenever possible.

→ Fetch a value stored in register is much faster than fetch it from memory.

→ Operand is already included in fetched instruction, so no need to fetch operand from memory.

* Program Relocation

→ In previous eg(i.e. of SIC simple) program must be loaded at address 1000 for correct execution.

→ This restriction is loaded at different address, its memory reference will access wrong data.

→ Therefore make program relocatable so that they can be loaded and execute correctly at any place in memory

→ It is desirable to load and run several program at same time.

→ The system must be able to load programs into memory wherever there is room.

→ Exactly starting address of the program is not known until load time.

Date
Page

* Absolute Program

→ Program with starting address specified at assembly time.

→ The address may be invalid if the program is loaded into somewhere else.

eg:-

```
30    100C    JEQ    ENDFIL    30 1015
```

if program is loaded from 5000 then it becomes,

```
30    500C    JEQ    ENDFIL    30 5015
```

Here, address 1015 will not contain value that we expect it may be part of some other user's program.

→ From object code only, in general not possible to tell which values represent address and which represent constant data items.

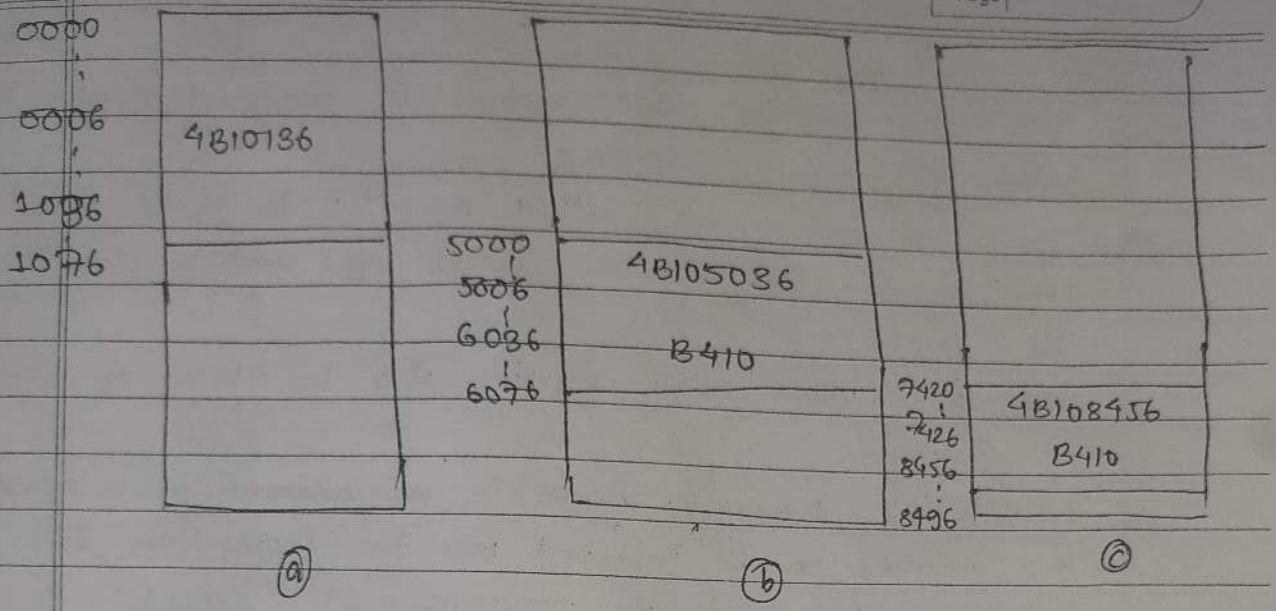
→ Assembler can't make necessary changes in addresses used by program but it can identify for loader the parts that can need to be modified.

→ An object program that contain the information necessary to perform modification is called relocatable program.

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

→ ~~the~~ only

format 4 → modify → modification record.




Here,

If fig ⓐ, program loaded at start address 0000, so for line 15, JSUB loaded at address 0006.

Here address field contain 01036, i.e. address of instruction labelled RDREC.

In fig ⓑ,

program loaded at 5000 so, address of instruction labeled RDREC is 6036.

Hence, JSUB instruction must be modified as shown to contain new address.

In fig ⓒ, program loaded from 7420

new address for RDREC 8456.

∴ JSUB instruction changed to 4B108456 to contain new address of RDREC.

## Example 2: (SIC/XE)

| Line | LOC | SYMBOL | OPCODE | OPERAND | OBJECT CODE |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 17202D |
| 12 | 0003 | | LDB | #LENGTH | 692020 |
| 13 | 0006 | | 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 | | Subroutine to read record into buffer | | | |
| 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 |
| | | Subroutine to write record from buffer | | | |
| 210 | 105D | WRREC | CLEAR | X | B410 |
| 212 | 105F | | LDT | LENGTH | 774000 |
| 215 | 1062 | WLOOP | TD | OUTPUT | |
| 220 | 1065 | | JEQ | WLOOP | 332FFA |
| 225 | 1068 | | LDCH | BUFFER, X | |
| 230 | 106B | | WD | OUTPUT | |
| 235 | 106E | | TIXR | T | B850 |
| 240 | 1070 | | JLT | WLOOP | |
| 245 | 1073 | | RSUB | | 4F0000 |
| 250 | 1076 | OUTPUT | BYTE | X '05' | 05 |
| 255 | 1077 | | END | FIRST | |

(4F)

Date
Page

* Difference between base relative and PC relative.

→ Assembler knows the value of PC when it tries to use PC relative mode to assemble instruction.

→ In case of base relative mode to assemble instruction assembler doesn't know value of base register.
∴ program must tell the assemble value of register B using Base directive.

→ also programmer must load appropriate value into register B by himself.

→ Another Base directive can appear later, then assembler need to change the current value of B.

→ NOBASE can also be used to tell assembler that no more base relative addressing should be used.

~~Object Program~~

Example (2) SIC / XE

H^ COPY^ 000000^ 001077

T^ 000000^ 1D^ 17202D^ - - -^ 03 2010

T^ 00001D^ 13^ 0F 2016^ - - - ^ 454F46

T^ 001036^ 1D^ 8410^ - - - ^3850

T^ 001053^ 1D^ 3B2 FEA^ - - - ^ B850

T^ 001070^ 07^ 3B2FEF^ - - - ^ 05

M^ 000007^ 05

M^ 000014^ 05

M^ 000027^05

E^ 000000

Classwork

| line | LOC | symbol | opcode | operand | object code |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | LDT | #3 | 750003 |
| 15 | 0003 | | LDX | #0 | 050000 |
| 20 | 0006 | MOVECH | LDCH | STR1,X | 53A00[illegible] |
| 25 | 0009 | | STCH | STR2,X | 57A0[illegible] |
| 30 | 000C | | TIXR | T | B850 |
| 35 | 000E | | JLT | MOVECH | 3B2FF5 |
| 40 | 0011 | STR1 | BYTE | C'EOF' | 454F46 |
| 45 | 0014 | STR2 | RESB | 3 | |
| 50 | 0017 | | END | FIRST | |

H^ COPY^ 00000 ^ 000017

T^ 000000 ^ 1E ^ 750003 ^ - - - ^ 454F46

E^ 00000

40/2 20/16 14

Date
Page

## Machine Independent Assembler features.

### Literal:-

→ The programmer writes the value of a constant operand as a part of the instruction that use it.
→ this avoids having to define constant else where in program and make label for it.
→ Such an operand is called literal.

Consider following example

```
LDA     five
:
five WORD   5
:
LDA     x='05'
```

- here it is convenient to write value of constant operand as a part of instruction.
- literal is identified with prefix = followed by specification of literal value.

* Difference between immediate operand and literal operand.

→ With immediate addressing, the operand value is assembled as part of machine instruction.
→ With literal, assembler generates specified value as a constant at some other memory location.
→ the address of this generated constant is used as target address of machine instruction, using PC or base relative addressing with memory reference.

→ if we use character string defining a literal to recognize duplicates, we should be careful about literal whose value depends on their location in program.

eg:- * usually denotes the current value of location counter.

```
-BUFFEND    EQU    *
```

→ There may be some literals that have same name but different values

```
,BASE   *
,LDB    =*
```

here, the literal = * repeatedly used in program has same name but different value.

literal = * represents an address in program so assembler must generate appropriate modification record.

→ When there is literals in program, we need a table called literal table LITTAB.

→ for each literal used, the table contains.

① literal name
② operand value.
③ address assigned to operand when it is placed in literal pool.

→ LITTAB is used as hash table using literal name as key.

## Implementation or Processing of literals

### Pass 1.

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

→ When LTORG or END statement is encountered, assign an address to each literal not yet assigned an address.

### Pass 2.

→ Search LITTAB for each literal operand.

→ Generate data values using BYTE or WORD.

→ Generate modification record for literals that represent an address in program.

## Symbol Defining Statements

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

→ EQU is a directive used to define symbol value

→ Syntax

symbol EQU value.

- here, value assigned to symbol may be constant

## Example 3:(Literals)

| Line | LOC | Symbol | Operation | Operand | OBJECT CODE |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 17202D |
| 12 | 0003 | | LDB | #LENGTH | 69202D |
| | | | 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 | 002D | | 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 | BUFFEND - BUFFER | |
| 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 |
| 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 | 1076 | | END | FIRST | |
| | 1076 | * | =X'05' | | 05 |

#maxlen constant मा modification हुदैन ।

Date
Page

Object program

Example 3 (Literals)

H^COPY^000000^001077

T^000000^1D^172027^---^032010

T^00001D^10^0F2016^---^3E2003

T^000027^03^454F46

T^00[1]0036^1D^B410^---^B850

T^001053^1D^

T^001070^07^

M^000007^05

M^000014^05

M^000027^05

E^000000^

~~Loop table~~

Literal table.

| Name | Value | length | LOC |
|---|---|---|---|
| =C'EOF' | 454F ~~E0F~~ 46 | 3 | 002D |
| =X'05' | 05 | 1 | 1076 |

32/2 16

Date
Page

~~Forgot~~

Symbol defining statements continue...

→ Replace +LDT #4096 with

```
Maxlen   EQU   4096
+LDT     #MAXLEN
```

→ helps in defining mnemonic ~~for~~ names for registers

```
A   EQU   0
X   EQU   1
```

→ ~~who~~ also by writing

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

→ Here, programmer can establish and we names that reflect the logical function of registers in program.

→ ORG is another assembler directive that can be used to indirectly assign value to symbols.

Syntax: ORG value

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

→ ORG will affect the values of all labels defined until next ORG as values of symbols used as labels are taken from LOCCTR.

Date
Page

→ No relative terms enters into multiplication or division operation.

→ expressions that do not meet condition of either 'absolute' or 'relative' should be flagged as errors.

BUFFEND + BUFFER
100 − BUFFER
8 * BUFFER
} illegal.

Date
Page

→ A program block may contain several separate segments of the source program.
→ the assembler will rearrange these segments to gather together pieces of each block.

Pass 1

→ A separate location counter for each program block
→ save and restore LOCCTR when switching betwe blocks.
→ at the begining of block, LOCCTR is set to 0
→ Assign each label an address relative to start of b
→ Store the block name or number in the SYMTA along with assigned relative address of label.
→ At end of pass 1, the latest value of LOCCTR indicate block length for each block.
→ Assign to each block a starting address in the object program by concatenating the program blocks in particular order.
→ Construct a table called program block table th contains block name, block number, starting addres and length for all blocks.

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

Date
Page

Pass 2

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

Advantages.

→ No longer need of extended formal, base register (∵ large buffer (CBLKS)) is moved to end of object program.

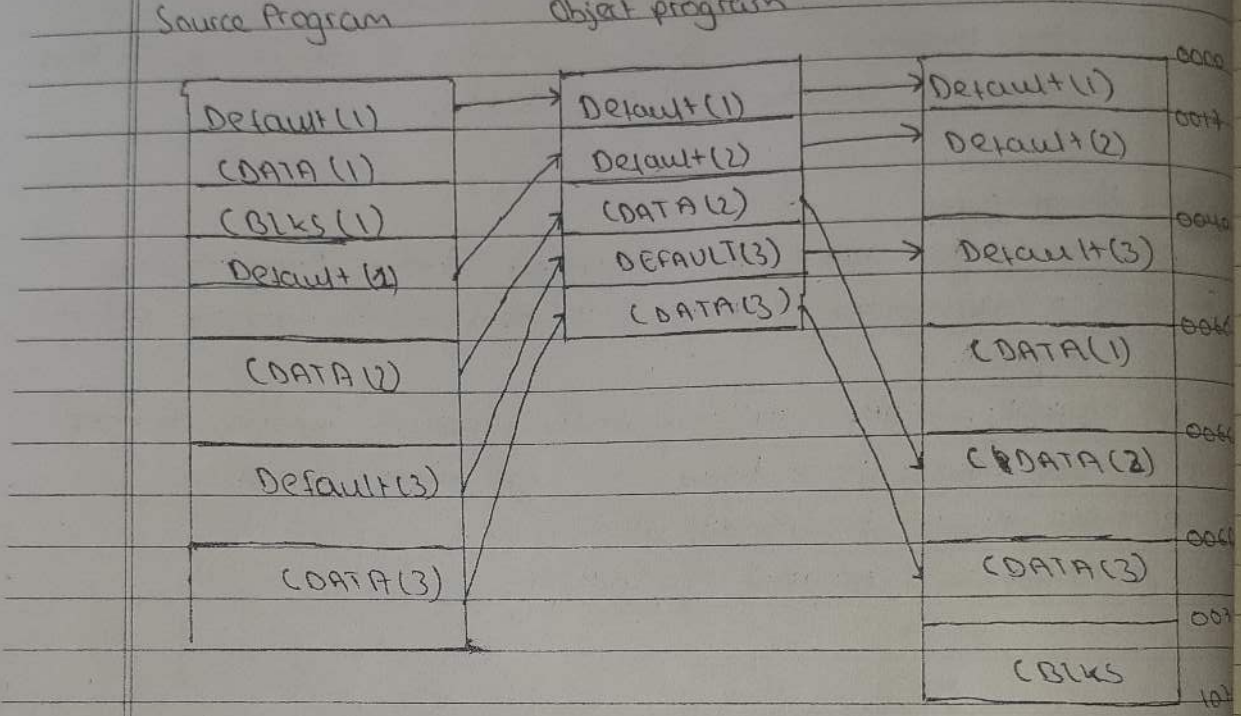
→ simply, LTORG statements is used to ensure literals are placed a head of large data areas.

→ No need of modification record.

→ improve program readability.//

Date
Page

# System Programming




→ It doesnot matter that text record of object program are notin sequence of address, the loader simply load object code from each record at the indicated address.

# Control section and program linking

→ Control section can be loaded and relocated independently of other.

→ are most often use for subroutines or other logical subdivision of program.

→ the programmer can assemble, load and manipulate

each of these control section separately.
→ because of this there should be some means of linking them together

→ assembler directive CSECT is used
Syntax
sectionname CSECT

→ Separate location counter for each control section.
→ instruction in one control section might need to refer to instruction or data located in another section.

External Definition and References.

① External Definition
→ EXTDEF name [name]
→ EXTDEF names symbol that are defined in this control section and may be used by other sections.
eg:
EXTDEF BUFFER, BUFFEND, LENGTH

② External References
→ EXTREF name [name]
→ EXTREF name symbols that are used in this control section and are defined elsewhere
eg: EXTREF RDREC, WRREC

→ To reference an external symbol, extended format instruction is needed.

0000 - 0003

FFF
-3
+1

Example 4:

14
3
192

line No 10 मा रहेका Block बाट अर्को Block मा जादा length add गरिन्छ

displacement = (length + T.A) - pc
= (0066 + 0000) - 0003
= 0063

4B203B

4B2021

3F2FEC

0003 - 0015
- 012,
FFF

032

332006

03

(0066 + 0007) - 0018

0C 032 0055

OF2 04B

OF2FF0

OF2056

Line no. 50 मा रहेका BLK बाट फेरी अर्को Block मा जादा अर्को address जोडिन्छ।

displacement = (0066 + 000B + 0000) - PC

Page

Literal continue....

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

→ Normally, literals are placed into a pool at end of the program.

→ It is also desirable to place literals into a pool at some other locations.
- for this LTORG directive is used.
- when LTORG is encountered by assembler, it creates a pool that contains all of literals used before LTORG.

→ LTORG is used when we want to keep the literal operands close to the instruction that uses it.

→ There may be same literal used in more than one place in program.
- if which such then it is called duplicate literal.
- for duplicate literals we should store only one copy of specified data value to save space.

→ Most assembler recognize duplicate literals by
① Comparing the character string defining them like =X'05' and =X'05'

② Comparing generated data volume like

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

Date
Page

## To solve Relocation Problem.

→ For an address label, address is assigned relative to start of program ( START 0).

→ produce a modification record to store the starting location and length of address field to be modified.

→ command for loader must also be part of object program.

→ When the assembler generate an address for a symbol the address to be inserted into the instruction is relative to start of program.

→ The assembler also produces a modification record, in which the address and length need to be modified address field are stored.

→ The loader when seeing record, will add begining address of loaded program to address field stored in record.

## Modification Record.

Col 1, M

Col 2-7, Starting location of address field to be modified, relative to begining of program.

Col 8-9, length of address field to be modified.

1013 + 11 (B)

Date
Page

| ine | LOC | Symbol | opcode | operand | object code |
|---|---|---|---|---|---|
| 10 | 1000 | STRCPY | START | 1000 | |
| 20 | 1000 | FIRST | LDX | ZERO | 041016 |
| 30 | 1003 | MOVECH | LDCH | STR1,X | 509000 |
| 40 | 1006 | | STCH | STR2,X | 549013 |
| 50 | 1009 | | TIX | ELEVEN | 2C1021 |
| 60 | 100C | | JLT | MOVECH | 381003 |
| 70 | 100F | STR1 | BYTE | C'ABCD' | 41424344 |
| 80 | 1013 | STR2 | RESB | 11 | — |
| 90 | 101E | ZERO | WORD | 0 | 000000 |
| 100 | 1021 | ELEVEN | WORD | 11 | 00000B |
| 110 | 1024 | | END | FIRST | — |

Data Structure

SYMTAB

| | |
|---|---|
| STRCPY | 1000 |
| FIRST | 1000 |
| MOVECH | 1003 |
| STR1 | 100F |
| STR2 | 1013 |
| ELEVEN | 1021 |
| ZERO | 101E |

Object Program

H^STRCPY^001000^000024

T^001000^13^041016^---^41424344

T^00101E^06^000000^00000B

E^001000

Date
Page

or any expression involving any constant and previously defined symbols.

⇒ used to improve program readability.

Example 8: (Literals)

line no. 45

1000 –

74

× 75

E0

E32011

B8

3B2FE8A

74

30 3

3

774

332

50

3

33C803

Date
Page

→ If previous value of LOCCTR can be automatically remembered, we can return to normal use of LOCCTR by simply writing ORG.

→ Forward reference is not allowed in either EQU or ORG.

→ All terms in the value field must have been defined previously in program.

| | | | | | | |
|---|---|---|---|---|---|---|
| ALPHA | RESW | 1 | ✓ | BETA | EQU | ALPHA |
| BETA | EQU | ALPHA | | ALPHA | RESW | 1 |

✗

Expressions

→ Assembler allows the use of expressions as operand

→ Assembler evaluates the expressions and produce a single operand address or value.

→ Expression consists of

(i) operators +, −, *, /

(ii) individual terms:- constant, symbols, special terms

eg:- * the current value of LOCCTR

eg:- MAXLEN EQU ~~BUFFEND~~ BUFFEND − BUFFER.

→ Regarding program relocation, a symbols value can be classified as

@ Relative

→ its value is relative to the begining of the object program and thus its value is independent of program location.

eg:- labels or reference to location counter (*)

Date
Page

ⓑ Absolute

→ its value is independent of program location

eg:- constant.

→ Depending on the type of value they produce, expressions are classified as

ⓐ Absolute expressions

→ An expression that contain only absolute terms like:-

```
MAXLEN   EQU   1000
```

→ An expression that contain relative terms but relative terms occur in pair and have opposite signs

like

```
MAXLEN   EQU   BUFFEND - BUFFER
```

ⓑ Relative expression

→ An expression in which all relative terms except one can be paired and the remaining unpaired term must have a positive sign

like:- + STAB

```
STAB   EQU   OPTAB + (BUFFEND - BUFFER)
```

Date
Page

## * Program Blocks.

→ The source programs logically contained main, subroutines, data areas etc but assemblers handle them as a single entity resulting in a single block of object code.

→ here generated machine instructions and data appeared in the same order as in source program.

→ Some assembler provide flexible handling by allowing the generated machine instruction and data to appear in object program in different order.

→ Program blocks

→ Segments of code that are rearranged with the single object program unit.

→ Control sections

→ Segment of code that are translated into independent object program units.

→ Here, ~~there are three blocks~~ three blocks are used

(A) default :- contain executable instruction

(B) CDATA :- Contain data areas ~~that~~ are less in length.

(C) CBLKS :- Contain data areas that consist of larger block of memory.

→ Assembler directive 'USE' is used.

→ at the begining, statements are assumed to be part of default block.

→ if no 'USE' statements are included, the entire program belongs to single block.

→ The 'USE' statement signals the begining of block

→ The 'USE' statement may also indicate a ~~contain~~ ~~continuous~~ continuation of previously begun block.

## External Reference Handling

Case I

15 0003 CLOOP +JSUB RDREC 4B100000

→ Here,
- Operand RDREC is an external reference.
- assembler has no idea where RDREC is, so insert address of zero.
- assembler can only use extended format to provide enough room i.e. relative addressing for external reference is invalid

Case II

160 0017 +STCH BUFFER,X 57900000

→ Here,
- Operand Buffer is an external reference.
- So, instruction is assembled using extended format with address of zero.
- x bit is set to 1 to indicate index addressing.

Case III

190 0028 MAXLEN WORD BUFFEND-BUFFER 000000

→ Here,
- BUFFEND and BUFFER both are external references are in given expression.
- Assembler inserts value of zero
- When the program is loaded, the loader will add to this data area the address of BUFFEND & subtract from it address of BUFFER, which then results desired value.

Case IV

107 1000 MAXLEN EQU BUFFEND-BUFFER ←→

→ Here,

BUFFEND and BUFFER are defined in the same control section, so expression can be calculated immediately.

In object program, we need two new record types and change in previously defined modification record type.

1) Define Record

Gives information about external symbols that are defined in this control section.

2) Reference Record

List symbols that are used as external references by this control section.

Define Record.

| | |
|---|---|
| Col 1 | D |
| Col 2-7 | name of external symbol defined in this control section. |
| Col 8-13 | relative address within this control section. |
| Col 14-73 | repeat information in col 2-13 for other external symbols. |

Reference Record.

| | |
|---|---|
| Col 1 | R |
| Col 2-7 | name of external symbol referred in this section. |
| Col 8-13 | name of other external reference symbols. |

Date
Page

## Modification Record.

| | | |
|---|---|---|
| Col | 1 | M |
| Col | 2-7 | Starting address of field to be modified. |
| Col | 8-9 | length of field to be modified in half byte. |
| Col | 10 | modification flag (+ or −) |
| Col | 11-16 | external symbol whose value is to be added or subtracted.// |

@Asf Akash shrestha//

## Example 4:(Program Blocks)

| Line | LOC | Symbol | Operation | Operand | OBJECT CODE |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 172063 |
| 15 | 0003 | CLOOP | JSUB | RDREC | 4B2021 |
| 20 | 0006 | | LDA | LENGTH | 032060 |
| 25 | 0009 | | COMP | #0 | 290000 |
| 30 | 000C | | JEQ | ENDFIL | 332006 |
| 35 | 000F | | JSUB | WRREC | 4B203B |
| 40 | 0012 | | J | CLOOP | 3F2FEE |
| 45 | 0015 | ENDFIL | LDA | =C'EOF' | 032055 |
| 50 | 0018 | | STA | BUFFER | 0F2056 |
| 55 | 001B | | LDA | #3 | 010003 |
| 60 | 001E | | STA | LENGTH | 0F2048 |
| 65 | 0021 | | JSUB | WRREC | 4B2029 |
| 70 | 0024 | | J | @RETADR | 3E203F |
| 92 | 0000 | | USE | CDATA | |
| 95 | 0000 | RETADR | RESW | 1 | |
| 100 | 0003 | LENGTH | RESW | 1 | |
| 103 | 0000 | | USE | CBLKS | |
| 105 | 0000 | BUFFER | RESB | 4096 | |
| 106 | 1000 | BUFEND | EQU | * | |
| 107 | 1000 | MAXLEN | EQU | BUFFEND - BUFFER | |
| | | Subroutine to read record into buffer | | | |
| 123 | 0027 | | USE | | |
| 125 | 0027 | RDREC | CLEAR | X | B410 |
| 130 | 0029 | | CLEAR | A | B400 |
| 132 | 002B | | CLEAR | S | B440 |
| 133 | 002D | | LDT | #MAXLEN | 75101000 |
| 135 | 0031 | RLOOP | TD | INPUT | E3203A |
| 140 | 0034 | | JEQ | RLOOP | 332FFA |
| 145 | 0037 | | RD | INPUT | DB2032 |
| 150 | 003A | | COMPR | A,S | A004 |
| 155 | 003C | | JEQ | EXIT | 332008 |
| 160 | 003F | | STCH | BUFFER,X | 57A02F |
| 165 | 0042 | | TIXR | T | B850 |
| 170 | 0044 | | JLT | RLOOP | 3B2FFA |
| 175 | 0047 | EXIT | STX | LENGTH | 18201F |
| 180 | 004A | | RSUB | | 4F0000 |
| 183 | 004D | | USE | CDATA | |
| 185 | 0006 | INPUT | BYTE | X'F1' | F1 |
| | | Subroutine to write record from buffer | | | |
| 208 | 004D | | USE | | |
| 210 | 004D | WRREC | CLEAR | X | B410 |
| 212 | 004F | | LDT | LENGTH | 772017 |
| 215 | 0052 | WLOOP | TD | =X'05' | E3201B |
| 220 | 0055 | | JEQ | WLOOP | 332FFA |
| 225 | 0058 | | LDCH | BUFFER,X | 53A016 |
| 230 | 005B | | WD | =X'05' | DF2012 |
| 235 | 005E | | TIXR | T | B850 |
| 240 | 0060 | | JLT | WLOOP | 3B2FEF |
| 245 | 0063 | | RSUB | | 4F0000 |
| 252 | 0007 | | USE | CDATA | |
| 253 | 0007 | | LTORG | | |
| | 0007 | * | =C'EOF' | | 454F46 |
| | 000A | * | =X'05' | | 05 |
| 255 | 000B | | END | FIRST | |

line No. 133 मा maxlen को point गरेको address हालिन्छ value हो।

Object Program (Example 4)

H^ COPY ^ 000000 ^ 001071
T^ 00000 ^ 1E ^ 172068 ^ . . . . ^ 010003
T^ 00001E ^ 09 ^ 0F2048 ^ . . . ^ 3E203F
T^ 000027 ^ 1D ^ B410 ^ . . . . ^ B850
T^ 000044 ^ 09 ^ B2FFA ^ . . . . ^ 4F0000
T^ 00006C ^ 01 ^ F1
T^ 00004D ^ 19 ^ B410 ^ 4F0000
T^ 00006D ^ 04 ^ 454F46 ^ 05
E^ 000000

format 4 भए पनि operand मा constant (#maxlen) & अर्को modification record हुँदैन।

## Example 5:(Control Sections)

| Line | LOC | Symbol | Operation | Operand | 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 | 0030 | | LTORG | | |
| | 0030 | * | =C'EOF' | | 454F46 |
| 105 | 0033 | BUFFER | RESB | 4096 | |
| 106 | 1033 | BUFEND | EQU | * | |
| 107 | 1000 | MAXLEN | EQU | BUFFEND - BUFFER | |
| | | Subroutine to read record into buffer | | | |
| 109 | 0000 | RDREC | CSECT | | |
| 122 | | | EXTREF | BUFFER, BUFEND, LENGTH | |
| 125 | 0000 | | CLEAR | X | B410 |
| 130 | 0002 | | CLEAR | A | B400 |
| 132 | 0004 | | CLEAR | S | B440 |
| 133 | 0006 | | LDT | #MAXLEN | 75201F |
| 135 | 0009 | RLOOP | TD | INPUT | E32019 |
| 140 | 000C | | JEQ | RLOOP | 332FFA |
| 145 | 000F | | RD | INPUT | DB2013 |
| 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 | BUFFEND - BUFFER | 000000 |
| | | Subroutine to write record from buffer | | | |
| 193 | 0000 | WRREC | CSECT | | |
| | | | EXTREF | BUFFER, LENGTH, BUFEND | |
| 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 | 001B | | END | FIRST | |
| | 001B | * | =X'05' | | 05 |

Date
Page

Example 5: Control sections

Copy Section.

H^ COPY ^ 000000 ^ 001033
D^ BUFFER ^ 000033 ^ BUFFEND ^ 001033 ^ LENGTH ^ 00002D
R^ RDREC ^ WRREC
T^ 000000 ^ 1D ^ 172027 ^ - - - ^ 0F2016
T^
T^
M^000004 ^ 05 ^ + ^ + RDREC
M^ 000011 ^ 05 ^ + ^ WRREC
M^ 000024 ^ 05 ^ + ^ WRREC
E^ 000000

RDREC Section

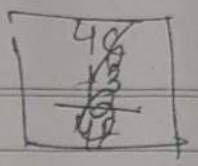
H^ RDREC ^ 000000 ^ 000023
R^ BUFFER ^ BUFFEND ^ LENGTH
T^ 000000 ^ 1D ^ B410 ^ - - - ^ B850
T^ 000001D ^ 0E ^ 3B2FE9 ^ - - - ^ 000000
M^ 000018 ^ 05 ^ + ^ BUFFER
M^ 000021 ^ 05 ^ + ^ LENGTH
M^ 000028 ^ 06 ^ + ^ BUFFEND
M ^ 000028 ^ 06 ^ - ^ BUFFER
E^ 000000

Date
Page

~~Note no. 133 में master के pointer होते हैं.~~
~~address होता value होती~~

WRREC section

H^WRREC^000000^00001C
R^BUFFER^LENGTH^BUFFEND
T^000000^1C^00B410^ - - - ^05
M^000003^05^+^LENGTH
M^00000D^05^+^BUFFER.
E^00000



Date
Page

Expression in multiple Control section,

⇒ Incase of control section, both terms in each pair of an expression must be within same control section.

→ BUFFEND – BUFFER ✓

→ RDREC – COPY

→ When an expression involves external references, the assembler cannot determine whether or not expression is illegal.

→ Assemble evaluates all the term first, combines to form initial expression value and generate modification records.

→ loader checks the expression for error and finishes the evaluation.//

Date
Page

## Assembler Design

→ Most assembler are two pass assemblers that process source program into two passes.

→ SYMTAB, LITTAB, OPTAB are used by both passes.

→ The main problem to assemble a program in one pass involves forward references.

→ Assembler design deals with
i) one pass assembler
ii) multi " "

1) One pass Assembler.

→ to elimate forward reference to data items, they can be defined before they are referenced

→ But forward reference to labels on instructions can't be eliminated. as easily because logic of program often requires forword jump,

→ to reduce size of the problem, many one pass assembler prohibit forward references to data items.

→ there are two types of one pass assembler
(a) one type produce object code directly in memory for immediate execution.
(b) other type produce usual kind of object program for later execution.

(a) Load and Go one pass assembler.

→ generates object code in memory for immediate execution

→ No object program is written out, no loader in needed.

→ Useful in a system with frequent program development and testing.

→ As program are re-assembled nearly every time they run, efficiency of assembler process is an important consideration.

→ avoids overhead of writing object program out and reading back in.

→ if an instruction operand is a symbol that has not yet been defined, the operand address is admitted when instruction is assembled.

→ The symbol used as an operand is entered into symbol table.
- it is flagged to indicate that symbol is undefined.
- when definition of symbol is encountered, forward reference list for that symbol is scanned and proper address is inserted into any instruction previously generated.

→ at the end of program, reports the error if there are still SYMTAB entries indicated undefined symbols.

Object code in memory and SYMTAB.

| Memory | Contents. | | | |
|---|---|---|---|---|
| 1000 | 454F4600 | 00030000 | 00xxxxxx | xxxxxxxx |
| 1010 | xxxxxxxx | xxxxxxxx | xxxxxxxx | xxxxxxxx |
| ⋮ | ⋮ | | | |
| 2000 | xxxxxxxx | xxxxxxxx | xxxxxxxx | xxxxxxxx |
| 2010 | 100948-- | --00100C | 28100630 | ----48-- |
| 2020 | --BC2012, | | | |
| ⋮ | ⋮ | | | |

| SYMBOL | VALUE |
|---|---|
| LENGTH | 100C |
| RDREC | *\| → 2013 \| ϕ |
| THREE | 1003 |
| ZERO | 1006 |
| WRREC | *\| → 201F \| ϕ |
| EOF | 1000 |
| ENDFIL | *\| → 201C \| ϕ |

Memory Contents,

| | |
|---|---|
| 1000 | . . . . . |
| 1010 | . . . . . |
| ! | |
| ! | |
| ! | |
| 2000 | |
| | - - - - - - - - - - |
| 2010 | 10094820 3D 00100C 28100630 20244 8- - |
| 2020 | - - 3C2012 |

| SYMBOL | VALUE |
|---|---|
| LENGTH | 100C |
| RDREC | 203D |
| THREE | 1003 |
| ZERO | 1006 |
| WRREC | *\| → 201F \| → 2031 \| ϕ |
| EOF | 1000 |
| ENDFIL | 2024 |

(6) another one pass assembler that generates object program.

→ if the operand contains an undefined symbol, use 0 as the address and write the text record to the object program.

→ forward references are entered into list as in load and go assembler.

→ When the definition of symbol is encountered, the assembler generates another text record with correct operand address of each entry in the reference list.

→ When loaded, the incorrect address 0 will be updated by the later text record containing symbol definition.

H^COPY^001000^001078

T^001000^09^454F46^...^000000

T^00200F^15^141009^480000^.....^300000^.....
^3C2012

T^00201C^02^2024

T^002024^19^001000^.....^480000^.....^001000

T^002013^02^203D

~~T^002050^~~

T^00203D^1E^041006^.....^300000^382043

T^002050^02^205B

T^00205B^07^101000^.....^05

T^00201F^02^2062

T^002031^02^2062

T^002062^18^041006^.....^4C0000

E^00200F

→ Here the second Text record contains object code generated from line 10 through 40.

→ the operand address for the instruction on lines 15, 30,

35 have been generated as 0000.

→ When the definition of ENDFIL is encountered in line 45, third text record is generated, which specifies value 2024 is to be loaded at location 20

→ When program is loaded, value 2024 will replace previously loaded 0000.

→ This way, forward references in program are handled with the services of loader.

Akash Shrestha

2) Multipass Assembler.

- For a two pass assembler, forward reference in symbol def$^n$ are not allowed

```
Alpha   EQU   BETA
BETA    EQU   DELTA
DELTA   RESW  1
```

- here, symbol BETA can't be assigned a value when it is encountered during pass 1 because DELTA has not yet been defined.
- hence, ALPHA can't be evaluated during Pass 2.
- symbol definitions must be completed in pass 1.
- The general solutions for forward references (which tends to create difficulty for person reading program as well as assembler) is a multipass, that can be make many passes as are needed to process the definition of symbols.
- the portions of program that involve forward references in symbol definition are saved during pass 1.
- additional passes through these stored definitions are made as the assembly progresses.
- For a forward references in symbol definition, we store in SYMTAB
  - Symbol name
  - defining expressions.
  - No. of undefined symbols in the defining expressions
- the undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- when a symbol is defined, we can recursively

evaluate the symbol expressions depending on the newly defined symbol.

Eg:-

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

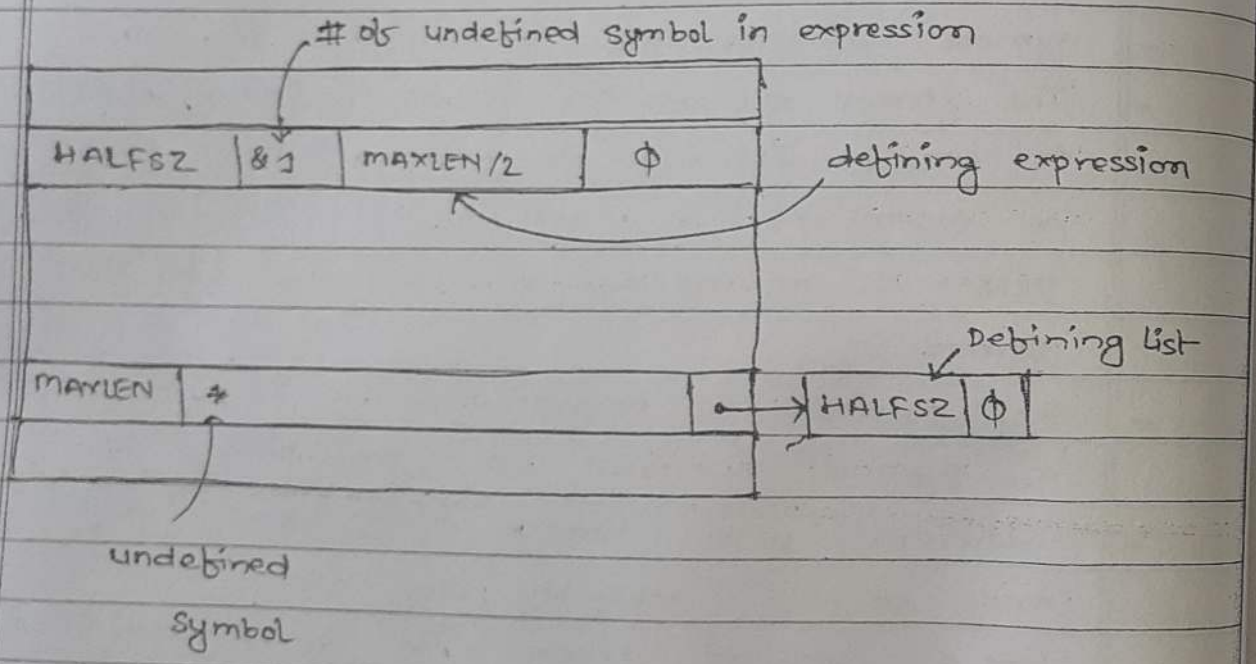



Fig ⑧

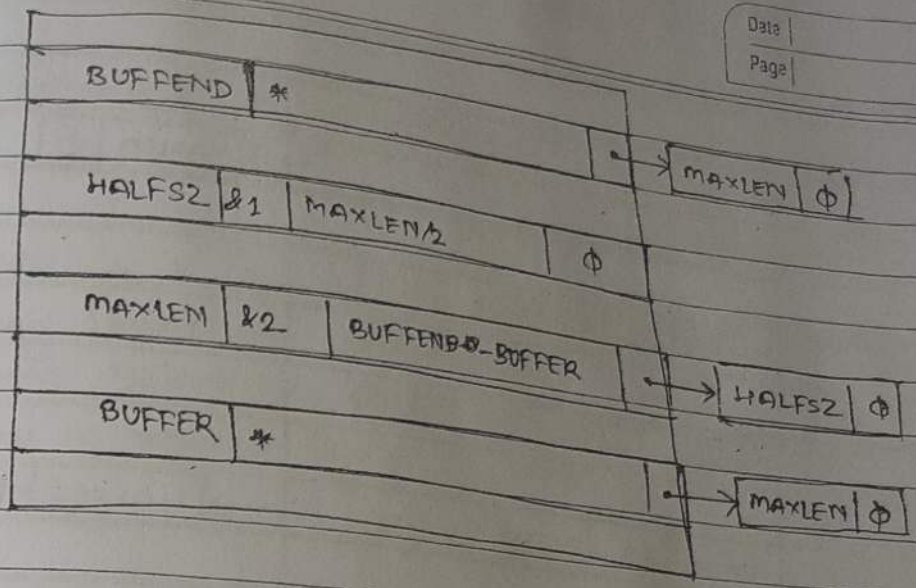

BUFFEND
*
MAXLEN
ϕ
HALFSZ
&1
MAXLEN/2
ϕ
MAXLEN
&2
BUFFEND-BUFFER
HALFSZ
ϕ
BUFFER
*
MAXLEN
ϕ

Fig (b)

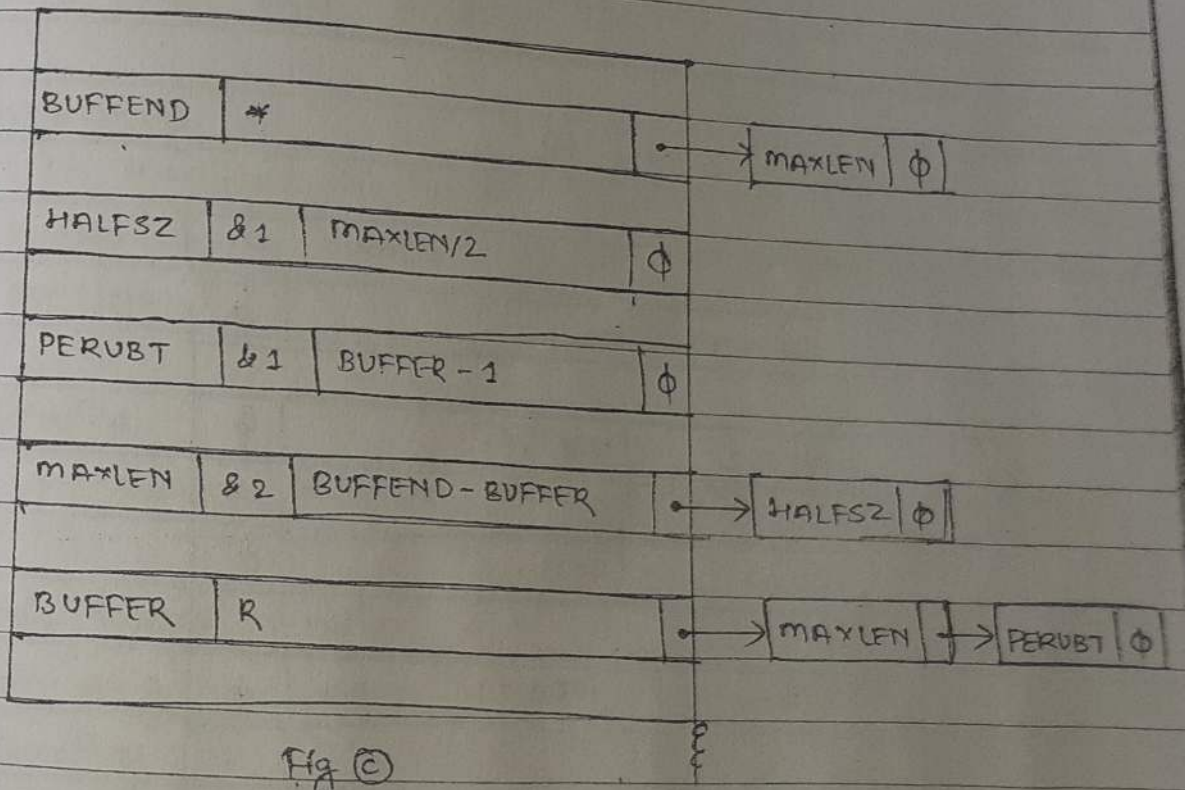

BUFFEND
*
MAXLEN
ϕ
HALFSZ
&1
MAXLEN/2
ϕ
PERUBT
&1
BUFFER - 1
ϕ
MAXLEN
&2
BUFFEND - BUFFER
HALFSZ
ϕ
BUFFER
R
MAXLEN
PERUBT
ϕ

Fig (c)

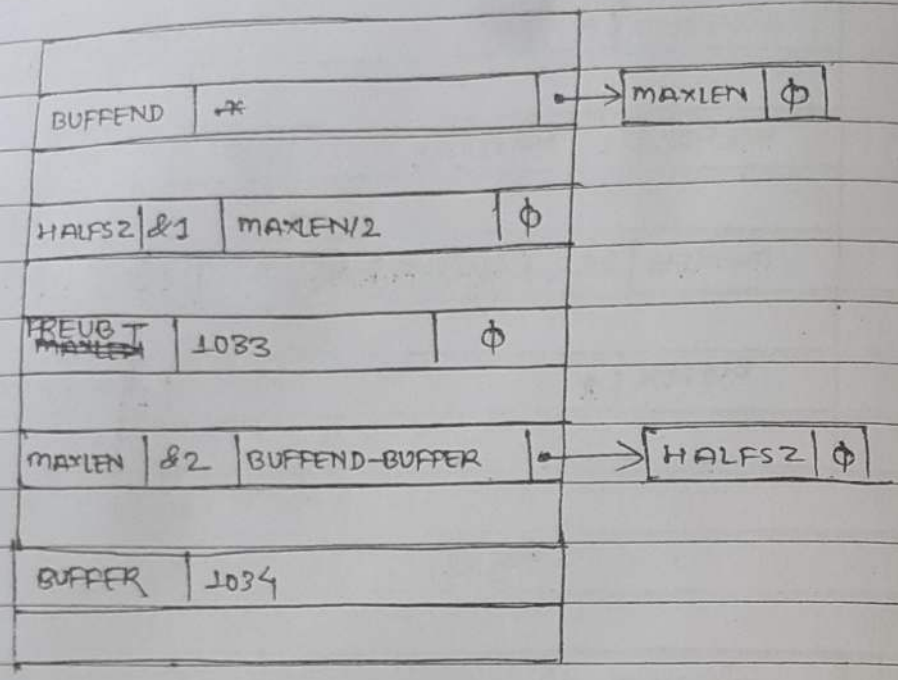

BUFFEND
*
MAXLEN
Φ
HALFS2
&1
MAXLEN/2
Φ
PREVBT
1033
Φ
MAXLEN
&2
BUFFEND-BUFFER
HALFS2
Φ
BUFFER
1034

Fig (d)

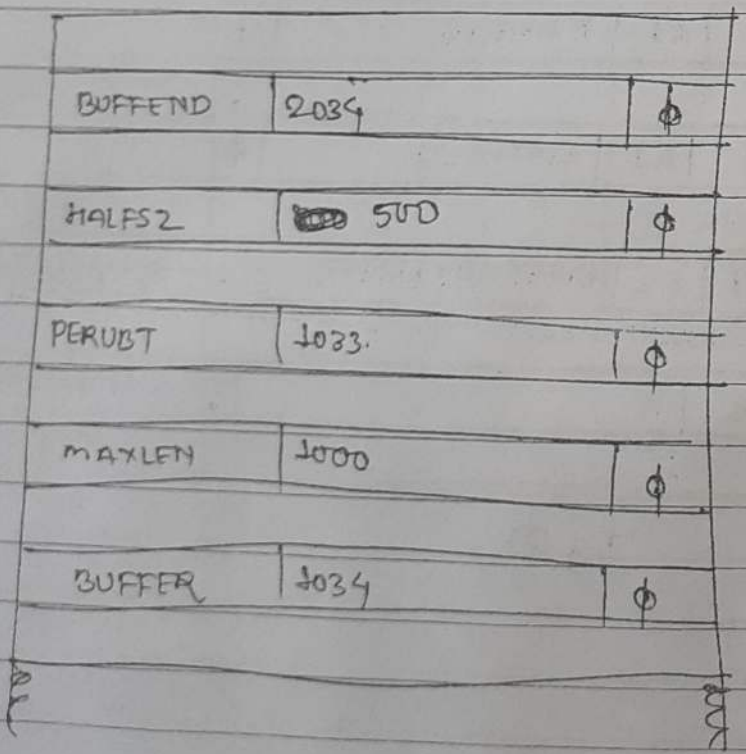
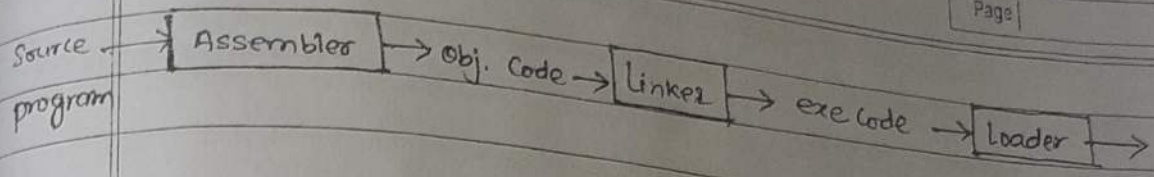

BUFFEND
2034
Φ
HALFS2
500
Φ
PERVBT
1033.
Φ
MAXLEN
1000
Φ
BUFFER
1034
Φ

Fig (e)

# Chapter:-3 Loader and Linker//




→ Loader is a system program that performs loading function.
→ many loaders also support relocation and linking.
→ Some systems have linker to perform linking operation and a separate loader to ~~have~~ handle relocation and loading

→ loading
:- brings object program into memory for execution.

→ Relocating.
:- modifies the object program so that it can be loaded at an address different ~~from~~ from location originally specified.

→ Linking.
:- Combines two or more separate object program.

## Basic loader function

① bringing an object program into memory.
② Starting its execution.//

## Absolute Loader

→ An object program is loaded at the address specified on the START directive.

→ No relation or linking is needed.

→ All functions are accomplished in a single pass as follows

ⓐ The header record of object program is checked to verify that correct program has been presented for loading.

ⓑ As each text record is read, the object code it contains is moved to begin execution of loaded program.

ⓒ When the end record is encountered, loader jumps to the specified address to begin execution of loaded program.

→ Fig ① shows the representation of program (SIC simple) after loading.

| Memory address | Contents | | | |
|---|---|---|---|---|
| 0000 | xxxxxxxx | xxxxxxxx | xxxxxxxx | xxxxxxxx |
| 0010 | xxxxxxxx | xxxxxxxx | xxxxxxxx | xxxxxxxx |
| ⋮ | | | | |
| 1000 | 14103348 | 20390010 | 36281030 | 30101548 |
| 1010 | 20613c10 | 0300102A | 0C103900 | 10200c10 |
| 1020 | 36482061 | 08103348 | 00004549 | 46000003 |
| 1030 | 000000xx | xxxxxxxx | xxxxxxxx | xxxxxxxx |
| ⋮ | | | | |
| 2030 | xxxxxxxx | xxxxxxxx | xx041030 | 001030E0 |
| 2040 | | | | |
| 2050 | | | | |
| 2060 | | | | |
| 2070 | 2C103638 | 20644c00 | 0005xxxx | xxxxxxxx |

Date
Page

## Algorithm for Absolute loader.

```
begin
    read Header Record
    verify program name and length.
    read first text record
    while record type ≠ E do
      begin
        {
        if object codes in characters ~~form~~ form, convert
        into internal representation.
        move object code to specified location in memory
        read next object program record.
        end
        jump to address specified in End record.
    end.
```

## A ~~Bootstrap~~ simple Bootstrap loader.

→ Special type of absolute loader.
→ loads the first program to be run by computer usually OS
→ bootstrap begins at address 0 in memory of machine.
→ load OS at address 80.
→ each byte of object code to be loaded, is represented on device f3. as two hexadecimal digit just as in text record of SIC object program.
→ the object code from device f3 is always loaded into consecutive bytes of memory starting at address 80.
→ The main loop of bootstrap keeps the address of next memory location to be loaded in register X.
→ After all of the object code from device F3 has been loaded, the bootstrap jumps to address 80, which begins execution of program that was loaded.

Date
Page

→ Much of work is performed by subroutine GETC,
→ GETC is used to read and convert a pair of characters from F1 representing 1 byte of object code to be loaded.
→ The resulting byte is stored at address currently in register X, using STCH instructions that refers to location 0 using index addressing.
→ TIXR is used to add 1 to value in X.

## Machine dependent loader feature.

→ One of the disadvantage of absolute loader is need for programmer to specify the actual address at which it will be loaded into memory.
→ On a large and more advanced machine, several independent programs are run together, sharing memory between them.
→ We do not know in advance where a program will loaded. Hence we write relocatable program instead of absolute ones.
→ Loaders that allow for program relocation are called relocating or relative orders.

## Relocation :-

- Use of two methods for specifying relocation as part of object program.

### First method

- a modification is used to describe each part of object code that must be changed when the program is relocation.

- Consider example 2 (SIC/XE)
- most of instruction used relative or immediate addressing.
- The only portion of assembled program that contains actual address are the extended format instruction on line 15, 35, 65.
- These are only items whose value is affected by relation.
- refer the corresponding object program.
- Here, each modification record specifies starting address and length of field whose value to be changed.
- it then describes modification to be performed.
- In this, all modification add the value of symbol COPY, which represents starting address of program.
- Although modification record scheme is convinent mean for specifying program relocation, it is not well suited for use with machine architectures.
- Consider the relocatable program written for SIC simple.

→ - here it does not use relative addressing
- So, the address of all instruction except RSUB must be modified when program is relocated.
- this would require 3 modification record which results in an object program more than twice as large as previous one.

Second method.

→ There are no modification records.

→ The text records are the same as before except that there is relocation bit associated with each word of object code.

→ Since all the SIC instruction occupy one ~~record~~ word, this means that there is one relocation bit for each possible instruction.

→ The relocation bit are gathered together in a bit ~~must~~ mask following length indicator in each text record.

→ This mask is represented as three hexadecimal digits.

→ If the relocation bit corresponding to a word of object code is set to 1, the programs starting address is to be added to this word when the program is relocated.

→ a bit value of 0 indicates no modification is necessary.

→ if a text record contains fewer than 12 words of object code, the bits corresponding to unused words are set 0 but mask = 12 bits

→ Eg:- bit mask FFC (1111 1111 1100) in first text record specifies that all 10 words of object code are to be modified during relocation.

Object program with ~~the~~ relocation bit mask.

H^COPY^000000^001071

T^000000^1E^FFC^140033^---^00002D

T^00001E^15^E00^0C0036^---^000000

T^001039^1E^FFC^4F0000^---^381032

T^001039^0A^800^100036^---^001000

T^001061^19^FE0^040030^---^05

E^000000

⇒ The LDX instruction line 210 begins a new text record.
- if it were placed in the preceding text record, it would not be properly aligned to correspond to a relocation bit because of 1byte data value generated from line 185.

Date
Page

SIC Simple (Relative)

| line | LOC | SYMBOL | OPCODE | OPERAND | object code. |
|---|---|---|---|---|---|
| 5 | 0000 | | | 0 | 140033 |
| | 0000 | | | | 481039 |

Program Linking (from slide)

# Machine Independent Loader Features

Date
Page

## Automatic Library Search

→ many linking loaders can automatically incorporate routines from a subprogram library into program being loaded.
→ in linking loaders the support for automatic library search must keep track of external symbols that are referred to but not defined in the primary input to the loader.
→ at the end of pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
→ the loader seaches the library or libraries specified for routines that contain the definitions of these symbol and process the subroutine found by this search exactly if they had been part of primary input stream.
→ the subroutine fetched from a library in this way may themselves contain external references it is therefore necessary to repeat the library search process until all references are resolved.
→ if unresolved external references remain after the library search is completed, these must be treated as errors.

## Loader Options.

→ Many loaders allow the user to specify options that modify standard processing.
→ Typical loader option 1: allows the selection of alternative source of input.

eg:- INCLUDE programme (library name)
might direct the loader to read the designated object program from library and treat it as if it were part of the primary loader input.

→ loader option 2: allow the user to delete external symbol or entire control sections.

eg:- DELETE CSECT – name might instruct the loader to delete the named control section from set of programs being loaded.

- CHANGE name 1, name 2 might cause the external symbol name 1 to be changed to name 2 wherever it appears in the object program.

- loader option 3:- involves automatic inclusion of library routines to satisfy external references.
eg: LIBRARY MYLIB.
- Such user specified libraries are normally searched before standard system libraries.

Loader design Options

→ Linking loaders perform all linking and relocation at load time.

→ There are two alternatives

ⓐ Linkage editors:- which perform linking prior to load time.

ⓑ Dynamic linking:- in which the linking function is performed at execution time.

Difference between linking editor and linking loades.

→ A linking loader performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution.

→ a linkage editor procedures a linked version of the program which is normally written to a file for later execution.

Linkage Editor:

→ When the user is ready to run the linked program a simple relocating loader can be used to load the program into memory.

→ the only object code modification necessary is the addition of actual address to relative values ~~with~~ ~~be~~ with in program.

→ ~~all items that need to~~ the linkage editor performs relocation of all control sections relative to start of linked program.

→ the means that the loading can be accomplished in one pass with no external symbol table required.

→ all items that need to be modified at load time have values that are relative to the start of the linked program.

→ Thus, if a program is to be executed many times without being reassemble the use of linkage editor can substantially reduces the overhead required.

→ linkage editor can also be used to build packages of subroutines or other control sections that are generally used together.

→ Linkage editor often include variety of other options and commands like in linkage loaders. But in general

Date
Page

linkage editor tend to offer more control and flexibility

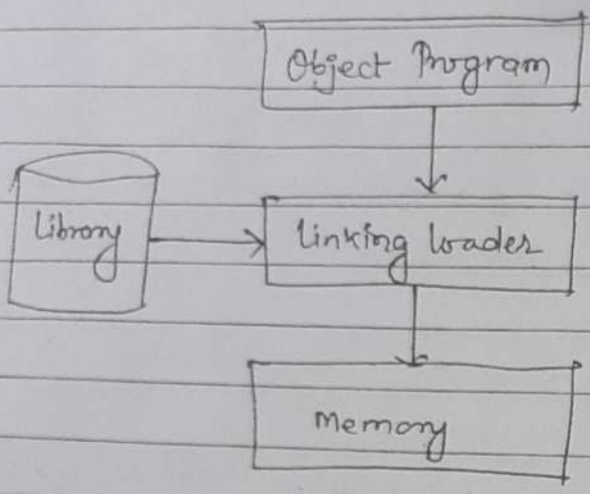



Fig:- Processing of object program in linking loader

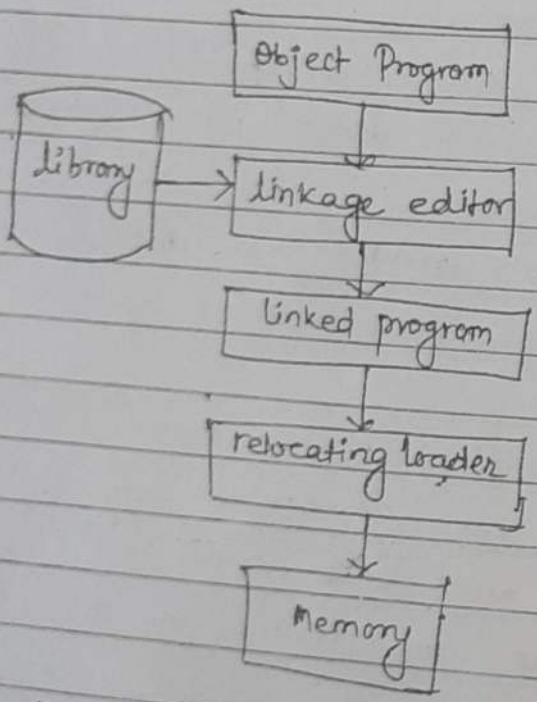



Fig:- Processing of object program in linkage editor.

Date
Page

## Dynamic linking

→ linkage editors perform linking before the program is loaded.

→ Linking loader perform linking at load time.

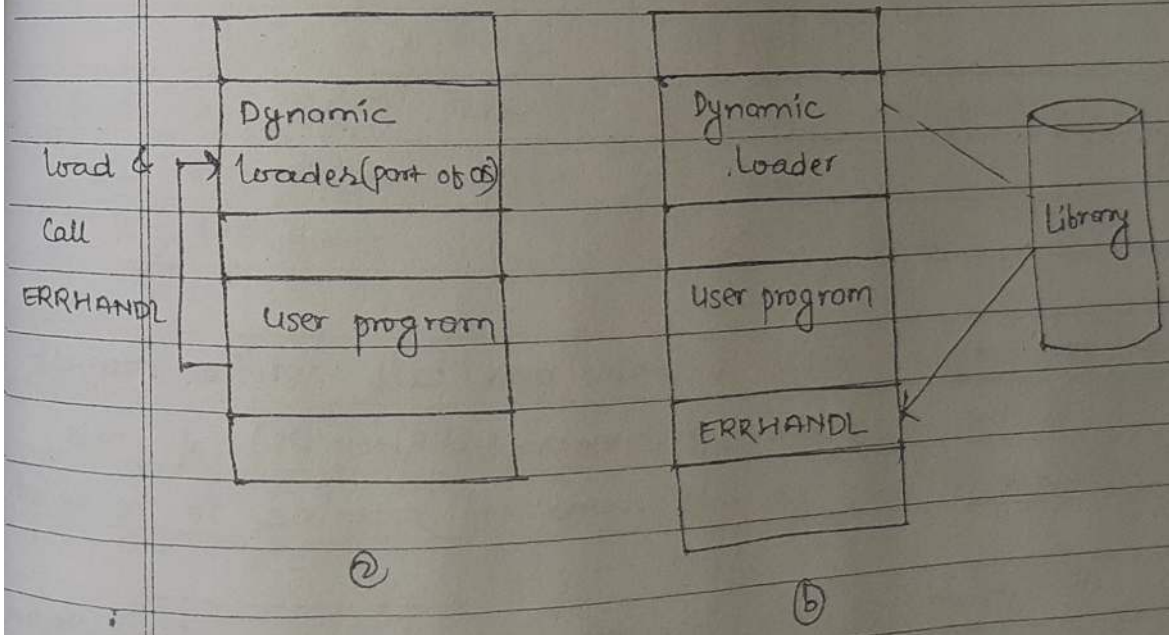
→ Dynamic linking postpones the linking function until execution time.

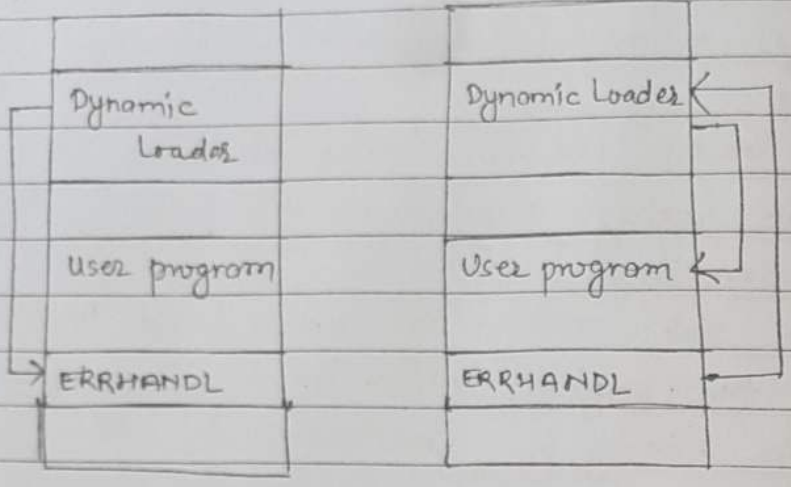
- a sub routine is loaded and linked to the rest of the program when it is first called.

→ Dynamic linking is often used to allow several executing programs to share one copy of subroutine or library.

→ eg:- a single copy of standard c library can be loaded into memory. All c programs currently in execution can be linked to this one copy instead of linking a separate copy into each object program.

→ Dynamic linking can avoid the necessity of loading entire library for each execution except those necessary subroutine.

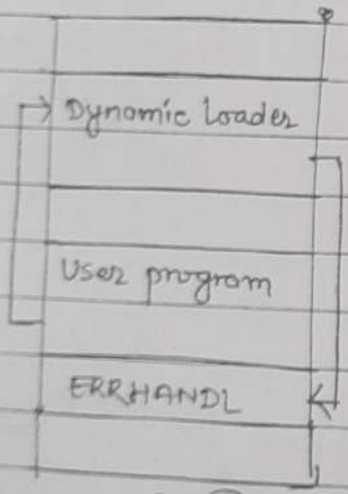





(c)




(d)




(e)

(a) → the program makes a load and call service request to OS the parameter argument (ERRHANDL) of this request in the symbolic name of routine to be called.

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

Date
Page

③→ Control is then passed from OS to routine being called.
④→ when the called subroutine completes its processing, it returns to its caller (i.e OS). OS then returns control to the program that issued the request.

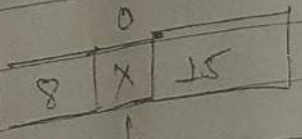
⑤→ If a subroutine is still in memory, a second call to it may not require another load operation control may simply be passed from dynamic loader to the called routine.

| Line | loc | symbol | opcode | Exp | Object Code |
|---|---|---|---|---|---|
| 10 | 5020 | STRCPY | START | 5000 | |
| 20 | 5000 | FIRST | LDX | Zero | 04501E |
| 30 | 5003 | MOVECH | LDCH | STR1,X | 50500F |
| 40 | 5006 | | STCH | STR2,X | 540013 |
| 50 | 5009 | | TIX | ELEVEN | 2C5021 |
| 60 | 500C | | JLT | MOVECH | 385003 |
| 70 | 500F | STR1 | BYTE | C'ABCD' | 41424344 |
| 80 | 5013 | STR2 | RESB | 11 | |
| | 90 | | WORD (3) | 0 | 000000 |
| 501E | 100 | ZERO | WORD (3) | 11 | 000000B |
| 5021 | 110 | ELEVEN | END | FIRST | |
| 5024 | | | | | |

LDX → 04
LDCH → 50
STCH → 54
JLT → 38
TIX → 2C

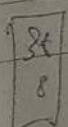
50 500F
8

0101 0000 0101 0000 0000
1
D




1

12/2 = 6
06




38/16

इन सभी absolute

```
5000    H^STRCOPY^005000^000024
5010    T^005000^13^04501E^---^41424344
5020    T^00501E^06^000000^000011
50      E^005000
000     xxxxxxxx    xxxxxxxx
```

| | | | | |
|---|---|---|---|---|
| 5000 | 04501E50 | D00F54D0 | 132C5021 | 38500341 |
| 5010 | 42434400 | xxxx xxxx | xxxx xxxx | xxxx 00 00 00 |
| 5020 | 00000800 | xx xx xxxx | xxxxxxx | xxxxxxxx |

Assignment

1. How forward reference is handled in one pass assembler?
2. What is relocation? How relocation is carried out in a loader?
3. What is advantage of relative addressing mode over absolute addressing mode?
4. Write about program blocks and control section?
5. What are main features of machine dependent loader when logically related parts of programming are linked then what is generated and why it is important?

Macro Processors.

Date
Page

→ A macro instruction is simply a notational convenience for programmer.

→ Macro represents commonly used group of statements in the source program

→ the macro processor replace each macro instruction with corresponding group of source statements.
- this operation is called 'expanding the macro'.

→ using macros allows programmer to write a shorthand version of a program.

→ eg:- before calling a subroutine, the contents of all registers may need to be stored this routine work can be done using a macro.

Macroprocessors.

→ its function essentially involves the substitution of one group of lines or another.

→ it doesn't analyze the text it handles.

→ meaning of the statements are of no concern during macro expansion.
Hence, design of macro processor generally is machine independent.

→ Macro mostly are used on assembly language programming, also be used in high level programming language such as C, C++.

Macro definition

→ Two directives MACRO and MEND are used in macro definition.

→ Macro's name appear before the MACRO directive.

→ Macro's parameters appear after the MACRO directive.

→ each parameter begins with '&'
→ between MACRO and MEND is the body of macro. These are the statements that will be generated as the expassion of the macro definition.

## Macro Invocation

→ Macro invocation statement (macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.

## Macro Expansion

→ each ~~more~~ macro invocation statement will be expanded into the statements that form the body of the macro.
→ arguments from the macro invocation are substituted for the parameters in macro prototype.
→ the arguments and parameters are associated with are another according to their positions
- the first argument in the macro invocation corresponds to the first parameter in macro prototype.
→ Comment lines within the macro body have been deleted but comments on individual statements have been retained.
→ Macro invocation, statement itself has been included as a comment line.

Example:-

```
COPY    START   0
RDBUFF  MACRO   &INDEV, &BUFADR, &RECLTH
.
.       macro to read record into buffer
.
        CLEAR   X
        CLEAR   A
        CLEAR   S
        +LDT    #4096
        TD      =X'&INDEV'
        JEQ     *-3
        RD      =X'&INDEV'
        COMPR   A,S
        JEQ     *+11
        STCH    &BUFADR,X
        TIXR    T
        JLT     *-19
        STX     &RECLTH
        MEND
WRBUFF  MACRO   &OUTDEV, &BUFADR, &RECLTH
        .
        .
        MEND
        .
        .
        main program
        .
        .
FIRST   STL     RETADR
CLOOP   RDBUFF  F1, BUFFER, LENGTH
```

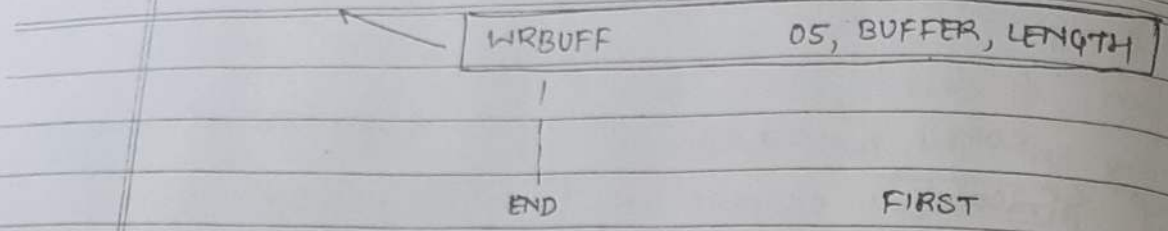
name of macro

macro def.

macro definition

macro invocation

Date
Page



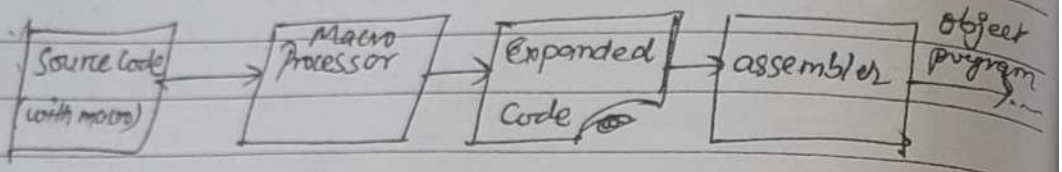
```
COPY    START   0
FIRST   STL     RETADR
CLOOP   RDBUFF  F1,BUFFER,LENGTH

CLOOP   CLEAR   X
        CLEAR   A
        CLEAR   S
        +LDT    #4096
        TD      =X'F1'
        JEQ     *-3
        RD      =X'F1'
        COMPR   A,S
        JEQ     *+11
        STCH    BUFFER,X
        TIXR    T
        JLT     *-19
        STX     LENGTH
```

Fig:- Macro Expression Example.

→ after macro processing the Expanded file can be use as input to the assembler.

→ the statements generated from the macro expansions will be assembled exactly as they had been written directly by the programmer




Difference between macro and subroutine

→ Statements that form the body of macro are generated each time a macro is expanded.

→ Statements in subroutine appear only once, regardless of how many times times the subroutine is called.

Macro processor algorithm and data structures.

→ Two pass macro processor can be changed where
- all macro definition are processed during first pass.
- all macro invocations statements are expanded during second pass.
- two pass macro processor wouldn't allow the body of one macro instruction to contain definition of other ~~words~~ macros.
- because all macros would have to be defined during the first pass ~~define~~ before any macro invocation were expanded.

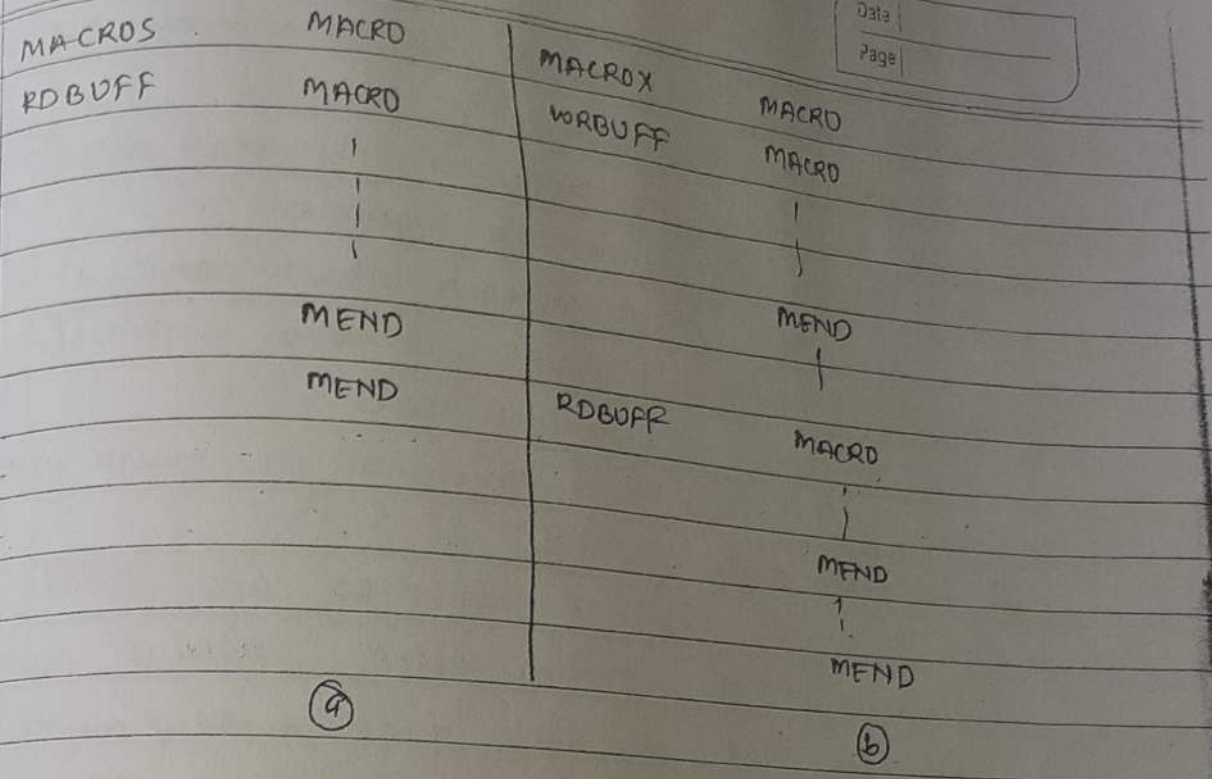



Fig- Examples of difinition of macros
writhin macros body.

(c)

- Defining MACROS and MACROX does not define RDBUFF and other macro instructions.
- these definitions are processed only when an invocations of MACROS and MACROX is expanded.
- a one pass macro processor that can alternate between macro definition and macro expanssion is able to handle macro like these
- There are 3 main data structures in macro processor

(a) Definition Table (DEFTAB)

- The macro definition themselves are stored in definition table (DETTAB) which contains macro prototype and statements that makeup macro body.
- Comment lines from macro definition are not entered into DETTAB ∴ they will not be part of macro expansion.

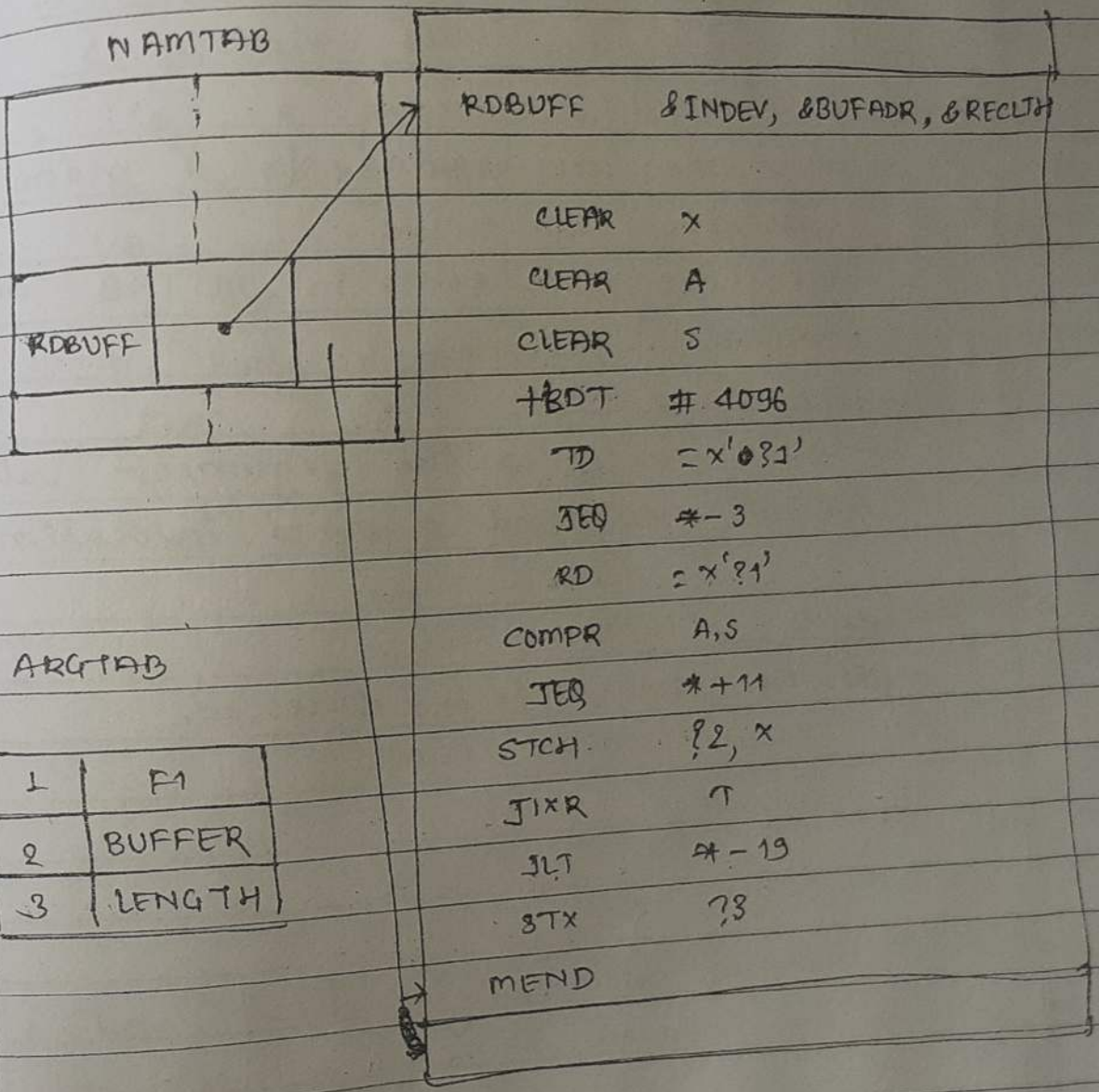
(b) Name Table (NAMTAB)

- references to macro instructions parameters are corrected to a positional entered into NAMTAB, which serves index to DEFTAB.
- For each macro instruction defined, NAMTAB ~~Conta~~ contains pointers to begining and end of definition in DEFTAB.

(c) Argument table (ARGTAB)

- is used during expanssion of macro invocation.
- when macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.

As the macro is expanded, arguments from ARGTAB are substitute for the corresponding parameters in macro body.



Date
Page

→ when the ?n notation is ~~reg~~ recognised in a line ~~form~~ from DEFTAB, a simple indexing operation supplies the property argument from ARGTAB.

Algorithm

1) Procedure DEFINE
   - Called when the begining of a macro definition is recognized.
   - make appropriate enters in DEFTAB and NAMTAB.

2) Procedure Expand EXPAND
   - Called to set up the argument values in ARGTAB and expand a macro invocation statement

3) Procedure GETLINE
   - get the nextline to be processed.

Date
Page

# Machine Independent Macroprocessor Features.

## Concatenation of Macro parameters.

→ Most macro processors allow parameters to be concatenated with other character string.

→ Say, a program contains one series of variables named by the symbols XA1, XA2, XA3, ——. another series named by XB1, XB2, XB3, —— etc.

→ the body of macro definition might contain statements like

```
SUM     MACRO   &ID
        LDA     X&ID1
        ADD     X&ID2
        ADD     X&ID3
        STA     X&ID5
        |
        |
        MEND
```

→ here, the begining of macro parameter is identified by starting symbol & however the end of parameter is not marked.

→ The problem is that the end of the parameter is not marked. So, X&ID1 may mean 'X'+&ID+1 or 'X'+&ID1

→ To avoid this ambiguity, a special concentration operator '→' is used.

→ Now, new form becomes X&ID → 1

→ here, '→' will not appear in macro expanssion.

Date
Page

eg:-

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

Now,

| SUM A | SUM BETA |
|---|---|
| ⇓ | ⇓ |
| LDA XA1 | LDA XBETA1 |
| ADD XA2 | ADD XBETA2 |
| ADD XA3 | ADD XBETA3 |
| STA XA5 | STA XBETA4 |

* Generation of unique labels.

→ if a macro is invocated and expanded multiple times, labels in the macro body may cause duplicate labels.

→ to generate unique labels for each macro invocation, we must begin a label with $ while ~~writing macro invocation, we must begin a label with $ while~~ with $ while writing macro definition.

→ during macro expansion, the $ will be replaced with $xx where xx is a two character alphanumeric counter of the number of macro instruction expanded.

- xx will start from AA, AB, AC, ----

→ Consider definition of WRBUFF

```
5       COPY    START   0
        :
135     TD              =X'&OUTDEV'
        |
140     JEQ             *-3
        |
155     JLT             *-14
        :
255     END             FIRST.
```

- here, if a label was placed on TD instruction on line 135, this label would be defined twice, once for each invocation of WRBUFF.

→ This duplicate definition would prevent correct assembly of the resulting expanded program.

→ the relative addressing on line 140 & 155, may be acceptable for short jumps.

→ for longer jumps spanning several instructions, such notation is very inconvenient, error prone and difficult to read.

→ these problems are avoided using special types of labels within macro instructions.

eg:-

RDBUFF Definition.

```
RDBUFF    MACRO    &INDEV, &BUFADR, &RECLTH
          CLEAR    X
          CLEAR    A
          CLEAR    S
          +LDT     #4096
$LOOP     TD       =X'&INDEV'
          JEQ      $LOOP
          RD       =X'&INDEV'
          COMPR    A,S
          JEQ      $Exit
          STCH     &BUFADR,X
          TIXR     T
          JLT      $LOOP
$Exit     STX      &RECLTH
          MEND
```

⇒ Expansion,

Macro Expansion,

```
RDBUFF      F1, BUFFER, LENGTH
              ⇓
            CLEAR   X
            CLEAR   A
            CLEAR   S
            +LDT    #4096
$AALOOP     TD      =X'F1'
            JEQ     $AALOOP
            RD      =X'F1'
            COMPR   A,S
            JEQ     $AALOOPExit
            STCH    BUFFER,X
            TIXR    T
            JLT     $AALOOP
$AAExit     STX     LENGTH,
```

## Conditional Macro Expansion.

→ So far, when a macro instruction is invoked, the same sequence of statements are used to expand macro.

→ here, depending on the arguments supplied in the macro invocation, the sequence of statements generated for macro expansion can be modified.

→ ~~for this~~ this adds greatly to power and flexibility of macro language.

→ macro time variable.

- is a variable that begins with '&' and that is not a macro instruction parameter.
- can be initialized to a value of 0.
- Can be set by macro processor directive SET.
- Can be used to
  - (i) store working values during expansion.
  - (ii) store the evaluation result of Boolean Expansion.
  - (iii) Control macro time conditional structures.

→ Macro time Conditional Structure.

i) IF - ELSE - END IF

ii) WHILE - ENDW

## Implementation of Conditional Macro Expansion.

(i) IF - ELSE - ENDIF

- Firstly a symbol table is maintained by macro processor that contains the values of all macro time variables uped
- entries in this table are made or modified when SET statements are processed.
- this table is used to look up the current value of

Date
Page

macro time variable whenever it is required.
the testing of condition and looping are done while macro is being expanded.
when an IF statement is encountered during the expanssion of a macro, the specified boolean expression is evaluated.
- if value is TRUE
  - the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
  - if ELSE is encountered, then skips to ENDIF.
- if value is FALSE
  - the macroprocessor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement.

(ii) WHILE - ENDW

- When a WHILE statement is encountered during the expanssion of a macro, the specified Boolean expression is evaluated.
- if value is true
  - the macroprocessor continues to process lines from DEFTAB until it encounters next ENDW statement.
  - when ENDW is encountered, the macro processor returns to the preceding WHILE, re evaluates boolean expression and taken action again.
- if value is false.
  - the macro processor skips a head in DEFTAB until it finds the next ENDW statement and then resumes normal macro expression.

* Keyword Macro Parameters.

1. Positional Parameter

→ Parameters and arguments are associated according to their positions in the macro prototype and invocation.

→ programmer must specify the arguments in proper order

→ If an argument is to be omitted, a null argument should be used to maintain proper order in macro invocation statement.

→ For eg:- suppose a macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only the 3rd and 9th parameters are to be specified.

Then statement is

GENER ,,DIRECT,,,,,3,

→ It is not suitable if a macro has a large no. of parameters and only a few of these are given values in a typical invocation.

2. Keyword Parameter.

→ Each argument value is written with a keyword that names the corresponding parameter.

→ arguments may appear in any order.

→ null arguments no longer need to be used.

→ if the 3rd parameter is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation would be

GENER TYPE = DIRECT, CHANNEL = 3

→ it is easier to read and much less error prone than positional method.

Macroprocessor design option

→ Recursive macro invocation and expansion can't be handled by previous macro processor design.

→ Reasons!-

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

2) the boolean variable EXPANDING would be set to FALSE when the inner macro expansion is finished, that is, the macro processor would forget that it had been in the middle of expanding 'outer' macro.

3) A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.

→ Solutions!-

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

2) Use a stack to take care of pushing and popping local variables and return address.

(a) Two pass macro processor.

→ Pass 1

- process macro definition.

→ Pass 2

- expand all macro invocation statements.

→ Problems

- this kind of macro processor can not allow resursive macro definition; i.e. the body of a macro contains definitions of other macros.

Date
Page

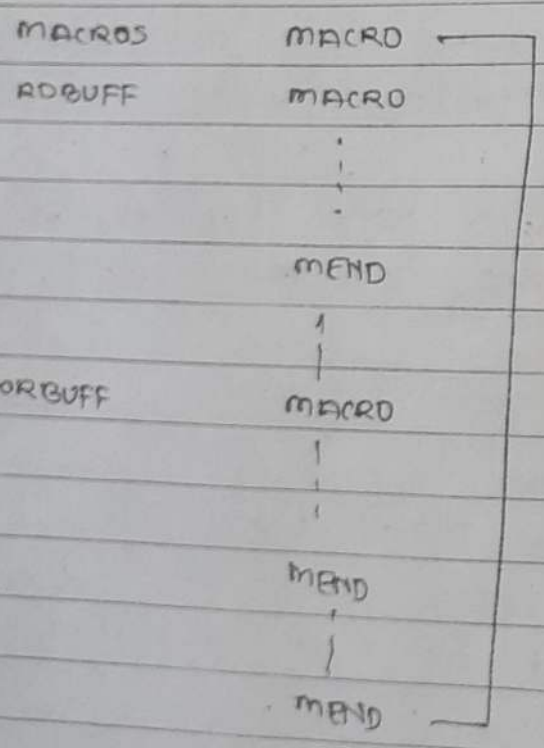
example of recursive macro definition.

→ Main MACROS (for SIC)
- Contains the definition of RDBUFF and WRBUFF written in SIC instruction.

→ MACROX (for SIC/XE)
- Contains definitions of RDBUFF and WRBUFF written in SIC/XE instructions.

→ A program that is to be run on SIC system could invoke MACROS whereas a program that is to be run on SIC/XE system could invoke MACROX.

→ Defining MACROS or MACROX doesnot define RDBUFF and WRBUFF. These definitions are processed only when an invocation of MACROS or MACROX is expanded.



Date
Page

(1) one pass assembler.

→ one pass macroprocessor alternates between macro definition and macro expansion in a recursive way, to handle recursive macro definition.

→ Because of ~~one~~ one pass structure, the definition of a macro must appear in source program before any statements that invoke that macro.

Handling recursive macro definition.

→ in DEFINE procedure

- when a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached.
- this wouldnot work for recursive macros definition because the first MEND encountered in the inner macro will terminate whole macro definition process.
- to solve this problem, a counter LEVEL is used to keep track of level of macro definition.
- increase level by 1 each time a MACRO directive is read.
- decrease level by 1 each time a MEND directive is read.
- A MEND can terminate the whole macro definition process only when LEVEL reaches 0.

(2) General purpose macro processor

Goal:-

→ macro processors that do not depend on any particular programming language, but can be used

with a variety of different languages.

advantages:-
→ programmers do not need to learn many macro languages.
→ although its ~~dependent~~ development costs are somewhat greater than those for a language specific macro processor, this expense does not need to repeated for each language, thus save overall cost.

disadvantages.
→ large number of details must be dealt with real programming language.
→ simulation in which normal macro parameter substitution should not occur.
→ Syntax.

(d) Macro processing within language translations.
→ Macro processor can be
(a) Preprocessors
→ process macro definition
→ expand macro invocation
→ produce an expanded version of source program which is then used as input to an assembler or compiler.

(b) line by line macro processor.
→ used as a sort of input routine for the assembler or compiler.
→ read source program.

→ process macro definition and expand macro invocations
→ pass output lines to the assembler or compiler.
→ it avoids making an extra pass over the source program.
→ Data structure required by the macro processor and the language translator can be combined.
→ utility subroutines can be used by both processor and the language translator.
  - Scanning input lines
  - Searching table.
  - data format conversion

② Integrated macroprocessor.
→ an integrated macro processor can potentially make use of any ~~instruction~~ information about the source program that is extracted by language translator.
→ it can support macro instructions that depend upon the context in which they occur.

* Drawback of line by line or integrated macro processor.
→ they must be specially designed and written to work with a particular implementation of assemble or compiler.
→ the cost of macro processor development is added to the costs of language translator, which results expensive development.
→ the assembler or compiler will be large and complex.

M.T.V. = macro time variable

Date
Page

# Conditional macro expanssion (example)

```
RDBUFF    MACRO    &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLEN
          IF       (&EOR NE '')
&EORCK    SET      1
          ENDIF
          CLEAR    X
          CLEAR    A
          IF       (&EORCK EQ 1)
          LDCH     =X'&EOR'
          RMO      A,S
          ENDIF
          IF       (&MAXLEN .EQ '')
          +LDT     #4096
          ELSE
          +LDT     #&MAXLEN
          ENDIF
          .
          .
          MEND
```

* condition  RDBUFF  F2, BUFFER, LENGTH, 3, ,
1 2 3 4 5

1. checks If (&EOR NE '') → true
2. checks If (&EORCK EQ 1) → true
3. checks If (&MAXLEN EQ '') → true

```
          CLEAR    X
          CLEAR    A
          LDCH     =X'3'
          RMO      A,S
          +LDT     #4096
```

Assignment

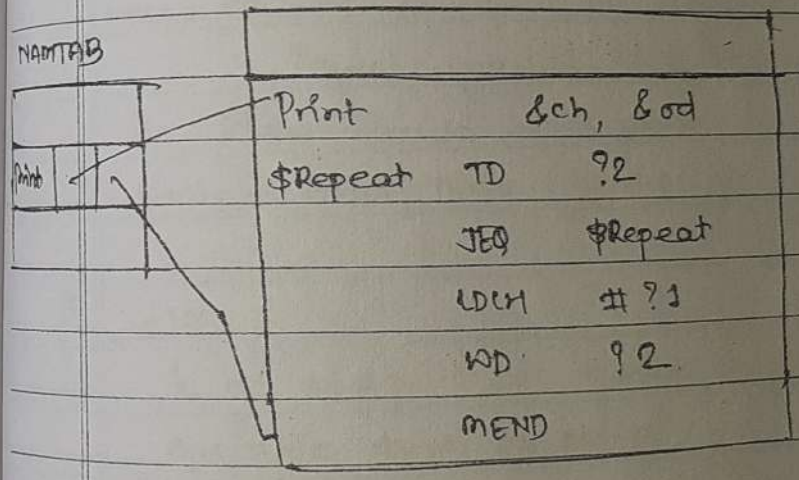
Date | Page |

1. Consider the macro definition given below and show macro expansion for the macro call statement "print 64, F1". Show all data structures used by macro processor clearly.

```
print   MACRO   &ch, &od
$Repeat TD      &od
        JEQ     $Repeat
        LDCH    #&ch
        WD      &od
        MEND
```

Ans// Data structure

DEFTAB




ARGTAB

| | |
|---|---|
| 1 | 64 |
| 2 | F1 |

Date
Page

Pg 2

Expansion

```
• print      64, F1
$AAARepeat   TD    F2
             JEQ   $AAARepeat
             LDCH  #64
             WD    F1
```

# chapter 5:- Object oriented System Design.

Date
Page

→ Focus on the objects handled by the system, rather than algorithms.

→ programs are designed and implemented as collection of objects not as collection of procedures.

## principles of object programming.

### ⓐ object

- is a basic unit of oop.
- is a component of a program that knows how to perform certain ~~contain~~ actions and how to ~~perform certain actions~~ interact with other elements of the program.
- Contains some data and defines a set of operations on that data that can be invoked by other parts of program.

eg:- Consider symbol-table as an object used by assembler. Here, set of operation or methods are like invert_symbol and lookup_symbol its data would be contents of hash table used to store symbols and their addresses.

### ⓑ class

- is a blueprint or template or set of instructions to build a specific type of object.
- defines the instance variables and methods of an object.
- an instance is a specific object from specific class.
- many objects can be created from same class.

eg:- for an assembler to translate programs for different versions of machine, class could be opcode_table. from this class, object could be created to define instruction set for machine.

- means that the interval representation of an object is generally hidden from view outside of objects definition.
- is the hiding of data implementation by restricting access to accessors and mutators.

③ Abstraction

- is a model, a view or some other focused representation for an actual item.
- is the implementation of an object that contains some essential properties and actions we can find in the original object we are representing.

④ Inheritance

- is a way to reuse code existing objects or to establish a subtype from an existing object.
- the relationship of classes through inheritance gives rise to a hierarchy.

Subclass :- is a modular, derivative class that inherits one or more properties from another class.

Superclass :- establishes a common interface and foundation functionality, which specialized subclass can inherit, modify and supplement.

⑤ Polymorphism

- means one name, many forms
- manifests itself having multiple methods all with some name, but slightly different functionality.

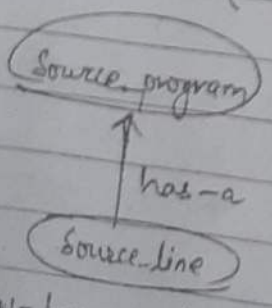



fig1- has-a relationship

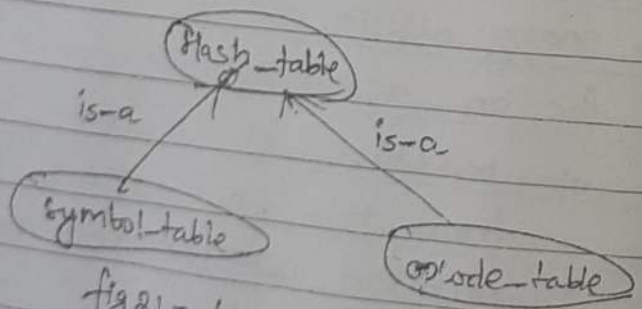



fig2 - is-a relationship or inheritance.

→ Hash_table is base class
→ other two are subclasses.
→ if insert_item and search_item are methods of base class then other subclasses automatically contains definition of methods.

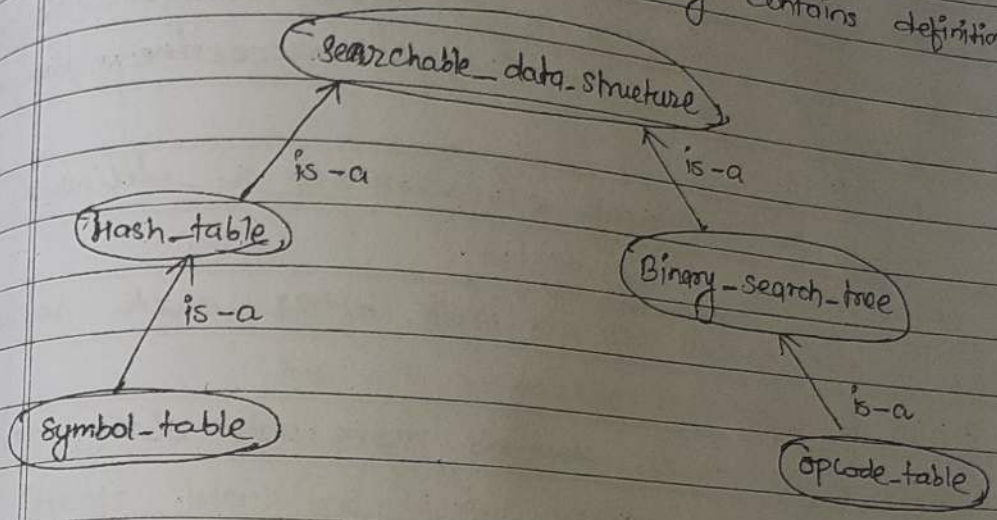



Fig 3 polymorphism.

→ here,
Superclass Searchable_data_structure defines two methods insert_item and search_for_item
→ Hash_table and Binary_search_tree are subclasses, so inherits ~~and~~ above methods,
→ implementation of the methods are different ~~development processes in micro to macro.~~
→ ~~Boochs macro process represents overall activities of development on a long~~
in these subclasses, But ~~name~~ names of methods and why of invokation are same
→ if Search_for_item method is invoked as instance of Symbol_table, it will result in retrival from hash table.
→ if same method is invoked on an instance of opcodetable it will result in binary_search_tree.
→ This shows polymorphism.

Object oriented design of an assembler

→ according to Booch, two different development processes (i) micro (ii) macro

Date
Page

→ Booch's macro process represents overall activities of development on a long range scale.

i) Establish the requirement for the sw (conceptualization)
ii) Develop an overall model of system behaviour (analysis)
iii) Create an architecture for the implementation (design)
iv) Develop the implementation through successive refinements. (evolution)
v) Manage the continued evolution of a delivered system (maintenance)

→ this macro process repeats itself after each release of s

→ Similar to waterfall model.

→ Booch's micro process ~~elements~~ represents daily activities of system developer.

i) Identify the class and objects of system.
ii) Establish the behaviour and other attributes of the classes and objects.
iii) analyze the relationship among the classes and obje
iv) Specify the implementation of classes and objects.

→ These activities may be repeated as needed wit increasing level of details.

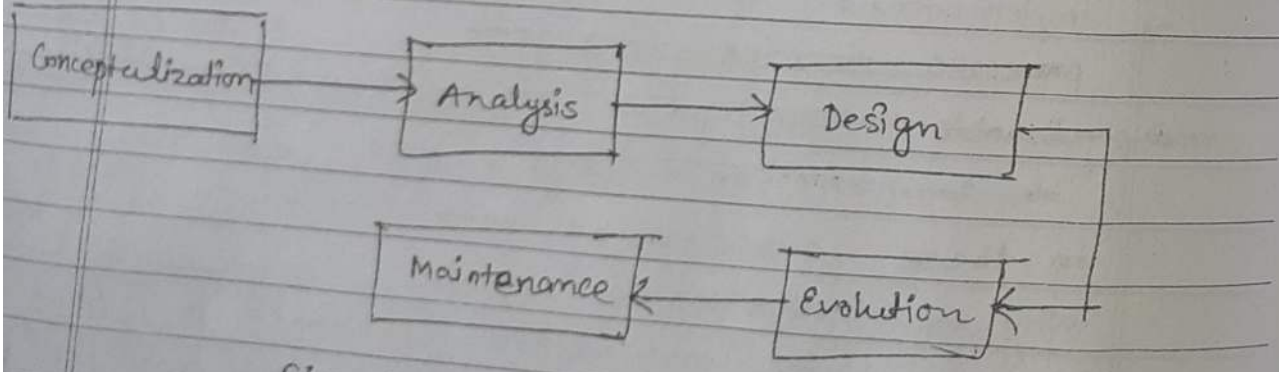
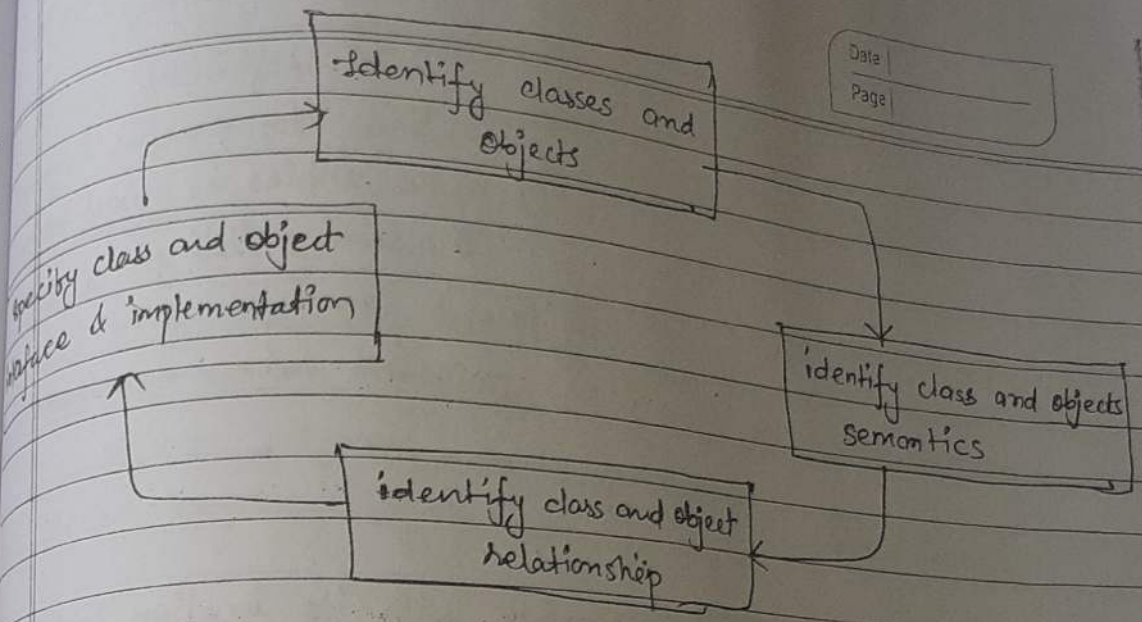



Fig:- Booch's macro process.




Fig! - Booch's micro process.

* Objects identified during design of assembler.

1) Source_program

i) Contents

- program
- current location counter values, errors.
- one object of class source_line for each line of program.

ii) methods

a) Assemble

- translate source program, produce an object program, and an assembly listing

2. Source_line

i) Contents.

- line of source program
- location counter value; error

ii) methods

a) create ~~and~~

- Create and initialize new instance of source-line

b) Assign_location

- assign location counter value to line
- return updated location counter value.
- enter label on line (if any) in symbol table.

c) Translate

- translate the instruction or

data definition on the line into machin language.
- make entries in object program & assembly listing.

② Record_errors
- record error detected.

3. Symbol_table

(i) Contents
- labels defined in src program with its location center value.

(ii) Methods

ⓐ Error
- enter a label and location counter value into table.
- return error if label is already defined.

ⓑ Search
- Search table for specified label.
- return location counter value of label or error if label is not defined.

4. Opcode_table

(i) Contents
- mnemonic instruction
- includes machine instruction format and code.

(ii) Methods

ⓐ Search:-
- Search table for specified mnemonic instruction
- return information about instruction format and operand required.

return error if mnemonic instruction not defined.

object_program

Contents

- object program after assembly.
- includes machine language translation of instruction and data de from object program.
- includes program length.

Methods

enter_text

- enter machine language translation of an instruction or data def? into object program.

Complete.

- enter program length and complete generation of external obj program file.

Assembly_listing

Contents

- listing of lines of source program and corresponding machin language translation.
- includes errors for each line & summary of errors in progr

Methods

① Enter_line

- Enter source_line, the corresponding machine language translation and description of errors detected for the line into assembly listing.

Complete.

- Enter summary of errors detected and complete the generati

of external assembly listing file.

* Object program
- indicates the methods that are invoked by each object.
- eg:- source_program object invokes method create, Assign-loca and translate on source_line object.

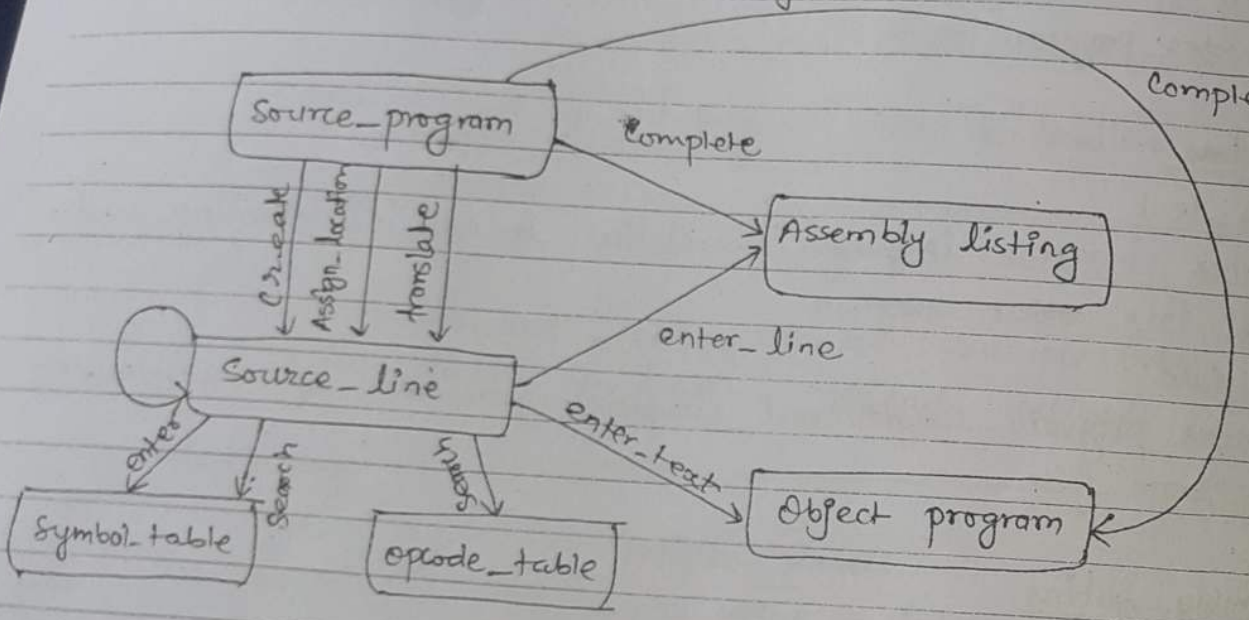



Fig:- Object diagram of assembler.

Object diagram ~~may~~ may also indicate the class of each object.

## Interaction Diagram.

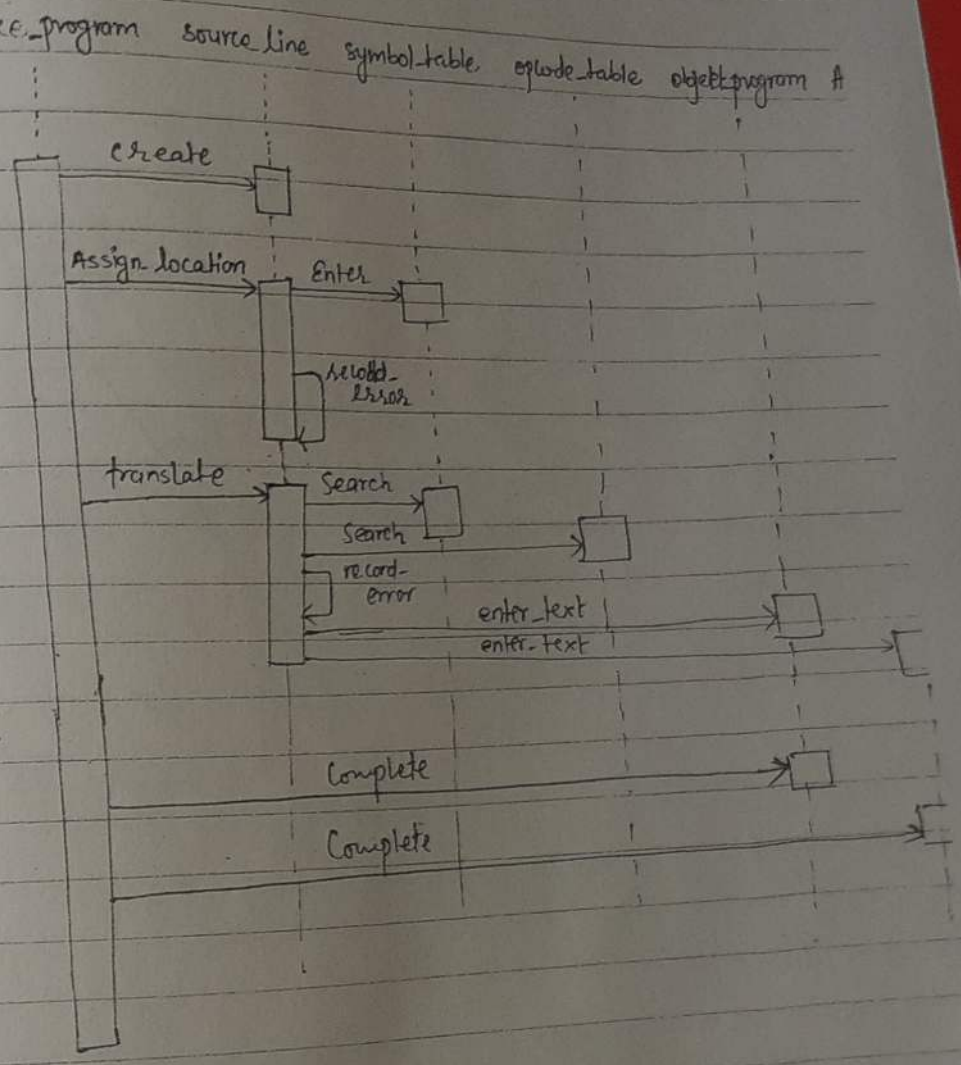



Fig:- Interaction Diagram.

→ interaction diagram makes easy to visualize the sequence of objec invocation and flow of control between objects.

→ each object is represented by dashed vertical line.

→ invocation of method is shown by horizontal line between objects.

→ the sequence is indicated by their vertical position in diagram.

→ a script is often written at L.H.S of diagram to describe

condition and iteration.

→ a narrow vertical box can be used to indicate the time the flow of control is focused in each object.