4/8/17
Fri

Module 1

Software

s/w

System s/w
- General purpose s/w. (meets general purpose).
- To coordinate and control the operations of a computer system.
- manage s/m resources.
- support functionality for application running in the system.

Application s/w
- s/w which implements user functionalities.
- special purpose s/w. (execute specific purpose of user)
- Eg: Browser, VLC, iTune, MS Word, MS office suit

Eg: OS, DBMS, Translators, Text editor, Macro preprocessor.

Linker (to link object modules produced by compiler and produce single executable file) (Also links shared libraries to app pgms)

Loader (loads executable file from sec storage to main mem)

App° s/w
↓
s/m s/w
↓
h/w

Text editor:
Create program files, config files

Translator:
HLL to machine lang { compiler, Interpreter
machine lang { Assembler

System software	V/S	Application Softwares
→ General purpose software (meets s/m needs).		→ Special purpose software (meets user needs).
→ ~~Not mandatory~~ Essential software.		→ Not mandatory
→ Executed all the time.		→ Executed as and when needed
→ Provides environment for application software.		→ Creates its own environment or work in the environment provided by s/m s/w.
→ System software has direct access to the computer hardware.		→ Control hardware with the help of system software.
→ Less number of s/m s/w		→ Number of application s/w is more.

9/8/17
Wed

S/m s/w

- S/m support s/w
  - * Provide support for the application
  - * Eg: Translators, debuggers, text editors
    - → Compiler
    - → Interpreter
    - → Assembler
- S/m control s/w
  - * control & coordinate s/m
  - * Eg: OS, DBMS

Types of s/m s/w

1. DBMS
2. OS
3. Text editor
4. Macroprocessor.

5. Translators
6. Debuggers
7. Device drivers
8. Linker
9. Loader.

> Text Editor.

Sw that helps to create
* program file
* configuration file
* doc file.

Editing
* cut
* copy
* paste
* undo/redo
* syntax highlighting.

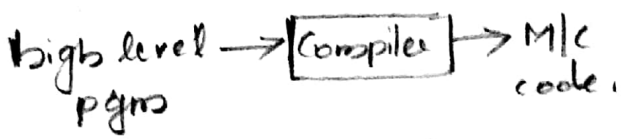
Eg: VI, Gedit, Notepad, Textpad.

> Macro processor

* macro substitution: substituting the macro with actual valu
* macroprocess does the macrosubstitution

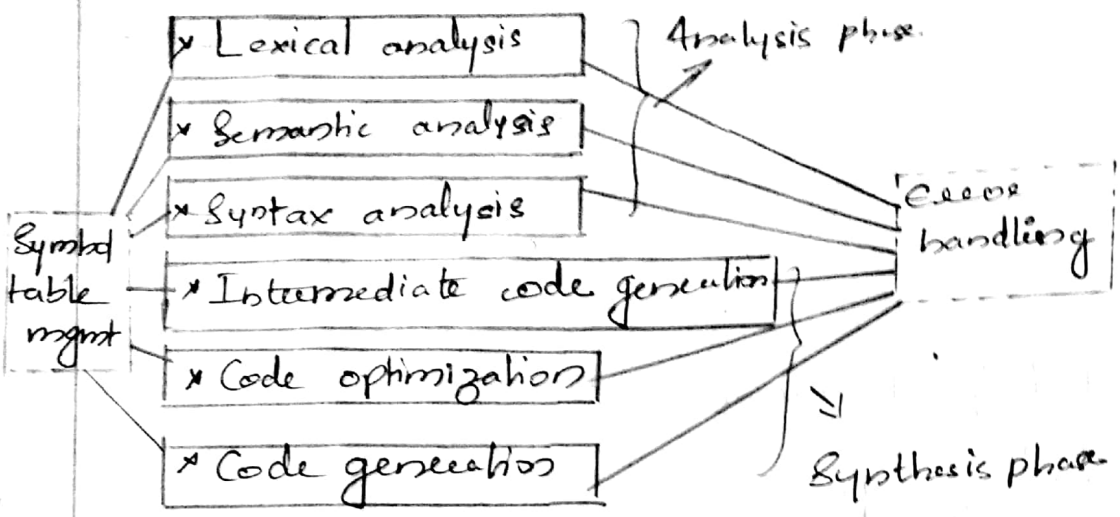
> Translators. Expanded code is given to translators.

* Compiler:
translates high level program to machine code.



Compilation. Two phases:
1. Analysis phase → o/p is intermediate code. (quadruple three add.
2. Synthesis phase → generate object code.

Analysis phase



→ Lexical analysis phase

Converts the instruction to sequence of tokens.

insln → seq of tokens

c = a + b * 5

↓

id = id + id * 5

→ Semantic analysis

checks any mismatches, checks meaning of instruction

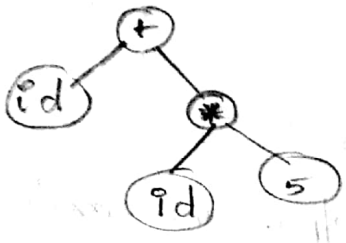
→ Syntax analysis

checks the syntax.

o/p: parse tree sep^n (Syntactical structure)

internal node = operators.

leaf node = identifiers / constants.



→ Code optimization

* Dead code elimination
* Move loop invariant

Output is → optimised code with improved CPU utilisation & less storage use.

→ Code generation - generates machine code.

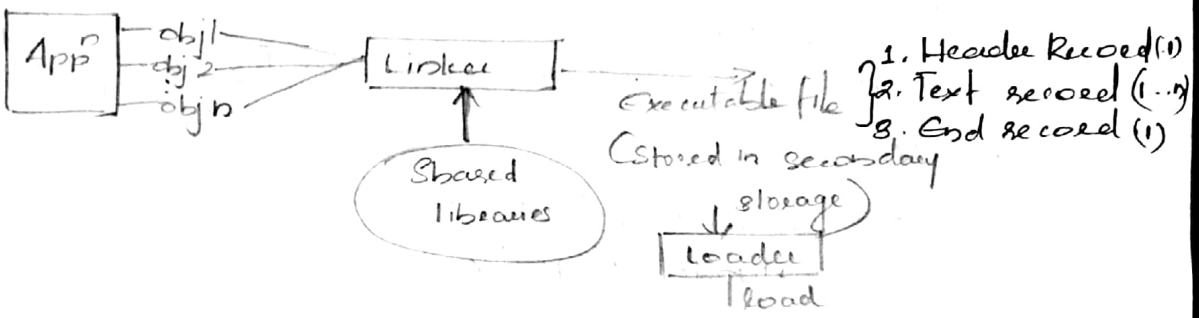
> Interpreter

line by line execution.

10/8/17 Thu

> Linker:

To link different object modules of same application
Links shared libraries



> Loader

It is a software that loads the executable file from the secondary storage to the main memory for execution.

→ Header record stores program name & size of the program.
→ Stores the object code - Text record.
→ End record store the starting address of execution.

Loader analyses these records of exec file.

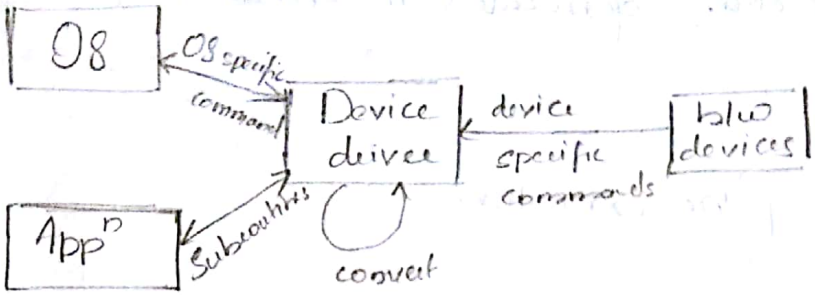
Header * Validate the program name
* Allocate size for the pgm.

Text * Loader loads object code from text sec & loads to m/m

End * jumps to the starting addrs for execution.

→ Device driver

* S/w interface to the h/w



11/8/17 Fri.

SIC (Simplified Instruction Computer)

* hypothetical computer (model, simulator)
* simplified h/w features
* model of a real machine with simplified architecture
* 2 Versions.
  - → Standard (SIC)
  - → Extended (SIC/XE).

SIC Architecture

1. Memory:
   - → 1 byte - 8 bit.
   - → 1 word - 3 bytes consecutively
   - → Memory address - 15 bits.
   - → Memory can address upto 2^{15} locations - 32768 bytes of data.

2. Registers:
   - → 5 registers.
   - → Each register 24 bit length.
   - → Two rep^n of registers.
     1. Character rep^n
     2. Numeric rep^n.

Character rep^n	Numeric rep^n	
1) A – Accumulator	(0) →	Perform Arithmetic op^n.
2) X – Index register	(1) →	Calculates address of operands. Stores offset.
3) L – Linkage register	(2) →	It store the return address of subroutine.
4) P.C – Program counter	(8) →	Address of next instr. to be executed.
5) S.W – Status word	(9) →	Stores the carry flag, overflow flag

3. Data formats:

> Integer – represented as 24 bit binary number.

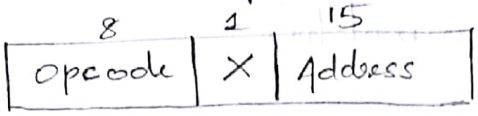
-ve number : 2's complement rep^n.

> Character – 8 bit ASCII code.

> Does not support floating point (SIC).
but supported in SIC XE

29/08/17 Thu

4. Instruction format.



Instr :

Label mnemonics operands.

label opcode operands.

Eg: LOOP1 MOV A,B

→ In SIC, we make use of two addressing modes:

* Direct addressing mode (X=0)
* Indexed addressing mode. (X=1)
(offset is specified)

↓
The way in which operand is specified in instruction

Target Address = Base Add + (X)

↓
offset stored at index reg.

Types of instructions in SIC: / Instruction set

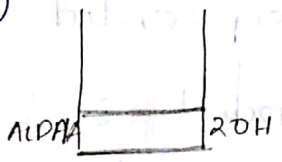
1> Load/Store.

* Loads the content from m/m to accumulator
* Store the content from accumulator to m/m.

→ LDA (Load Accumulator).

* (Acc ← M/m) (A ← M)
* Eg: LDA ALPHA.

A ← ALPHA



→ LDX (Load Index reg)

* (X ← (M))
* Eg: LDX ALPHA.

X ← ALPHA

→ STA (Store from the Accumulator)

* (M) ← A
* Eg: STA BETA.

BETA ← A

→ STX (Store from index reg)

* (M) ← X

→ LDCH (Load Character)

* (A) ← (M)
* LDCH BETA.

(A) ← BETA

→ STCH (Store Character from Acc to m/m)

* (M) ← (A)

2> Integer Arithmetic

The content of accumulator is manipulated with content in a m/m location & the result is stored in accumulator.

→ ADD.

* ~~Eg:~~ ADD ((A) ← (A)+(M))
* ~~Eg:~~ ADD ALPHA.

A ← A+ALPHA

→ MUL

* MUL ((A) ← (A) * (M))
* Eg MUL ALPHA

→ SUB

* SUB ((A) ← (A) - (M))
* Eg: SUB ALPHA.

→ DIV

* DIV ((A) ← (A) / (M))
* Eg: DIV ALPHA.

3> COMP (Compare values stored in m/m & accumulator).

*Eg: COMP ALPHA

- if (M) > (A) ⇒ CC = '<' (Conditional code) (present in status word)
- if (M) < (A) ⇒ CC = '>'
- if (A) = (M) ⇒ CC = '='

4> Conditional Jump Instruction.

→ JLT (Jump if less than)

- checks the cc.

→ JGT

→ JEQ.

5> Subroutine linkage.

→ JSUB (Jump to subroutine).

* Jumps to the subroutine and also store the return address in linkage reg.

Eg: subroutine label.
LOOP1 LDA ALPH
STA BETA
RSUB
LDA ALPHA
JSUB LOOP1
ABC → LDX BETA

→ RSUB

* fetches address from linkage reg and returns to the instruction.

6> Input / Output.

→ TD (Test Device).
if Device ready CC = '<'
Device not ready CC = '='
→ RD (Read from Device).
→ WD (Write to the device)

Eg: LOOP1 TD INDEV (store the id of i/p dev)
JEQ LOOP1
RD INDEV
WD OUTDEV (stores the id of o/p dev)

05/8/17 FRI

Assembler Directives

* Gives direction to assembler.
* Not translated to object code

1> START

Eg: COPY START 1000
program name. starting address.

2> END

```
COPY  START  1000
FIRST  LDA   ALPH
       =
       END FIRST
```

gives Assembler instruction to end the execution

→ indicates end of program
→ gives address of first executable instruction

3> BYTE:

indicates char value is following

Eg: ABC BYTE C 'EOF'

ABC	EOF	20011

XYZ BYTE X 'F1'

(hexadecimal)

→ generates char & hexadecimal constants

4> WORD:
used to generate integer & hexadecimal constants

Eg: THREE WORD 3., THREE WORD 4

eg: LDA THREE
Load the value 3.

5> RESB (Reserved Byte).

Eg: RESB 4096.

(gives die to assemble to reserve 4096 bytes of m/m)

6> RESW (Reserve word).

Eg: RESW 2.
// 2 words reserved.

→ SIC program using TIX (for comparison)

```
        LDX   ZERO
MOVECH  LDCH  STR1,X
        STCH  STR2,X
        TIX   ELEVEN
        JLT   MOVECH
STR1    BYTE  C 'TEST_STRING'
STR2    RESB  11
ZERO    WORD  0
ELEVEN  WORD  11
```

[X] + 1 → operand.

Two pass Assembler processes the pgm two time.
1. Process assembler directive
2. Pgm code then generate object code.

1. Write a sequence of instruction for SIC to set ALPHA = Product of BETA and GAMMA.

$\alpha = \beta \times \gamma$.

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

1 word 3 byte - 24 bit - int

2. Write an SIC program to swap two values a=2 b=5

```
        LDA   A
        STA   TEMP
        LDA   B
        STA   A
        LDA   TEMP
        STA   B
```

```
A     WORD  2
B     WORD  5
TEMP  RESW  1.
```

SIC/XE:

26/8/17 Sat

→ **Memory.**

Address length = 20 bit

Max m/m capacity = 2^{20}.

→ **Registers.**

There are additional 4 registers in SIC/XE

Numeric rep"

3 - 1. B - Base register. (stores base address)

4 - 2. S > general purpose (to store 5 - 3. T > values).

6 - 4. F - Floating point (to store floating point value)

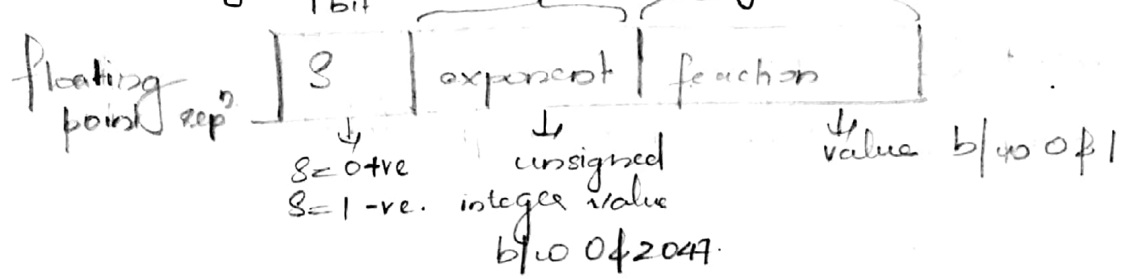
→ **Data Formats:**

Integer - 24 bit

Character - 8 bit ASCII

Negative numbers - 2's complement rep".

Floating point - 48 bit binary. 36 bits.

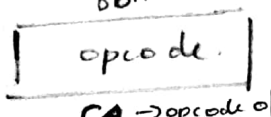


-X-

→ **Instruction Format in SIC/XE** (Instruction Set Appendix A (470 ... SIC/XE)

1) **Format 1** (Instruction only has opcode)

length - (1 byte)



CA → opcode of FIX

Eg: FIX (A) ← (F) convert

↓

floating to integer conversion of result stored in accumulator

opcode of CA = Object code

1100 0100

2) Format 2 (1 instr. length = 2 bytes).

- opcode followed by 2 registers
- Eg: COMPR A,S

opcode.
FIX = C4
COMPR = A0
LDA = 00
JSUB = 48

opcode → 8 bit
r1 → 4 bit
r2 → 4 bit

| opcode | r1 | r2 |

COMPR → opcode = A0

| 1010 0000 | 0000 | 0100 |

To generate object code.
↓
split into 4 bits.

1010 0000 0000 0100

A 0 0 4 → object code of COMPR A,S

3. Format 3. (3 bytes).

- opcode is followed by mem label.

| opcode (6 bit) | n (1) | i (1) | x (1) | b (1) | p (1) | e (1) | displacement (12 bit) |

flag bits.

* Flag bits represent addressing modes.

⇒ n → n=1 - Indirect addressing mode

Direct.
Eg: LDA ALPHA.

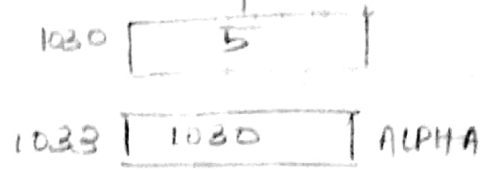
1033 | 5 | ALPHA
↓
operand value

1033 ALPHA RESW 1.

address where ALPHA occurs i.e. 1033 will be having the operand value.

30/8/17
TW

Eg: LDA @ALPHA



1033 ALPHA Result

⇒ i - immediate addressing mode.

Directly operand value is specified

Eg: LDA #3

i=1 - immediate adder mode.

term 14 → if preceding the opcode.

format 4 flag 'e' is set to 1.

⇒ x - indexed addressing mode:

Eg: LDA ALPHA,X

Target address = ALPHA(X)

X value in index reg

⇒ b, p - relative addressing mode.

- Base relative Addressing mode (b) ⇒ b=1
- Peogeam counter relative addressing mode ⇒ p=1

TA = Base address + offset

TA = PC + offset

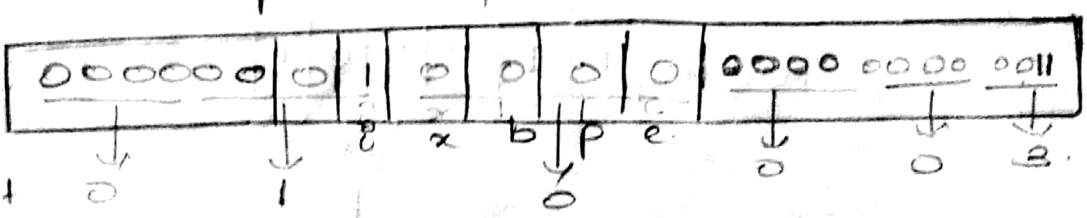
If displacement value is b/w 0 to 4095 - Base relative

" - 2048 to 2047 - Peogeam counter relative

Eg:

LDA #3. opcode LDA = 00

30/8/17 Tuesday



1). 1000 LDA ALPHA

1003 CLOOP RESB 1

:

1009 ALPHA RESW 1.

for pc & Base selective add mode n,i = 1.

Generate object code of LDA ALPHA ?.

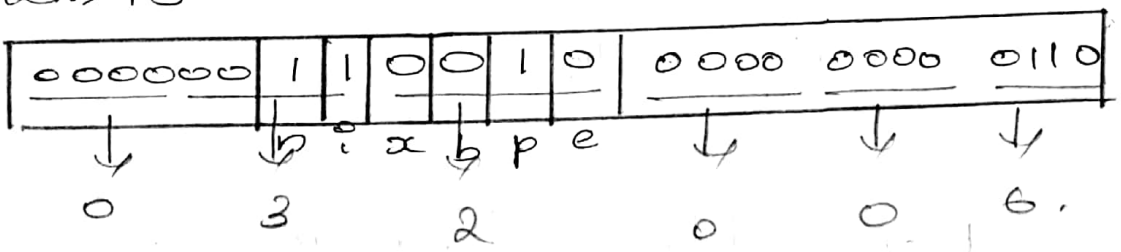
Ans. TA = Pc + displacement.

displacement = (TA) − (Pc)

= 1009 − 1003

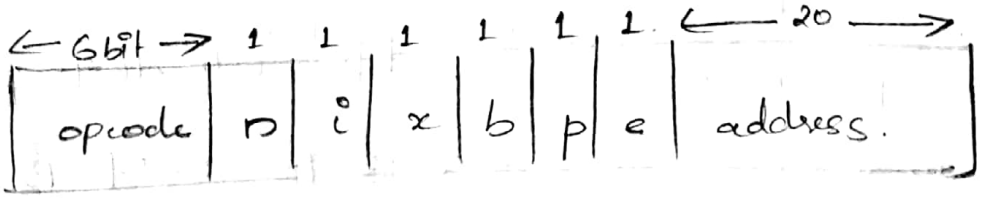
= 6.

format 3.



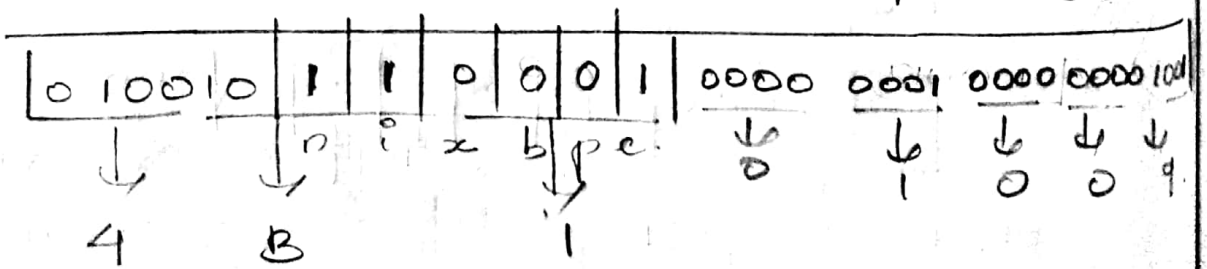
object code (032006)

4. Format 4. (4 byte instruction length)



Eg. +JSUB ALPHA.

0100 1000.
4 8
opcode = 48 (JSUB).



Pg 42
ex
sec 1.3
9

Prog: Write an SIC/XE program to set ALPHA equal to the integer portion of ~~β and γ~~ BETA and GAMMA.

→
```
LDF  BETA
DIVF GAMMA.   F → F / γ )
FIX
STA ALPHA.
```

```
ALPHA RESW 1
BETA  RESW 1.
GAMMA WORD 1
```

} if β=7 γ=3.
→BETA WORD 7.
GAMMA WORD 3.

Prog. WAP to ~~replace~~ SIC/XE clear 20 byte string with all blanks.

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

11/9/2019 Mon

MODULE 2.

ASSEMBLERS.

Functions:

* Converts operation code to machine code

LDA ALPHA
↓
00-----m/c code

This conversion is done with the help of OPTAB (containing opcode & m/c code).

* converts symbolic operand to machine address
* generation of object code for each assembly instruction.

Assembler.

Single pass	Two pass.
* Object code is generated by reading the pgm once. * Drawback: forward reference Reference to a label defined later in the pgm	* There are two passes. 1st pass: All the symbols & their address & store in SYMTAB. During the second pass object code is generated.

→ Assembler: Data Structures:

1> OPTAB - Operation Code Table
2> SYMTAB - Symbol Table
3> LOCCTR - Location Counter.

1> OPTAB

opcode	machine code	format	length
LDA	00	3	3

* In two pass assembler, the OPTAB is constructed during pass 1.

2> SYMTAB.

SYMBOL	TYPE	ADDRESS
RETADR	R	1033
CLOOP	A	2033

Type R - Relative addressing (operand addr depends on start address)
Type A - Absolute Addressing. (independent of starting address)
Fixed address program

* SYMTAB construction is during pass 1 of the two pass assembler.

3> LOCCTR.

* Store the address of instruction.
* Tracks each instruction.
* Initial value - starting address.

19/2017 word.

1. Write the opcode for the following SIC/XE program.

LDX=04, LDA=00, LDB=68, ADD=18, STA=0C,

JLT=38, TIX=2C, RSUB=4C

13/9/19 Wed.

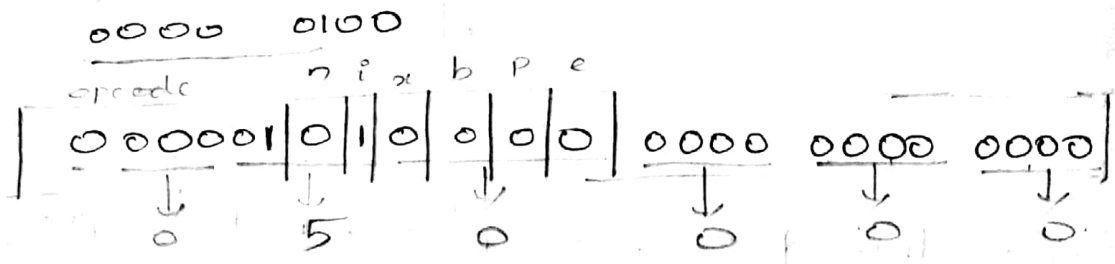
LOC	LENGTH	LABEL	MNEMONIC	OPERAND	OBJECT COD
0000		SUM	START	0	
0000	3		LDX	#0	050000
0003	3		LDA	#0	010000
0006	4		+LDB	#TABLE2	69100000
			BASE	TABLE2	
000A	3	LOOP	ADD	TABLE,X	1BA013
000D	3		ADD	TABLE2,X	1BC000
0010	3		TIX	COUNT	2F200A
0013	3		JLT	LOOP	2B2FFA
0017	4		+STA	TOTAL	0F102F00
001A	3		RSUB.		4C0000
001D	3	COUNT	RESW	1	
0020	1770	TABLE	RESW	2000	
1790	1770	TABLE2	RESW	2000	
2F00	3	TOTAL	RESW	1	
2F03			END	FIRST	

Case I LDX #0 (format 3)

↓ immediate addr (displacement is 0 for immed)

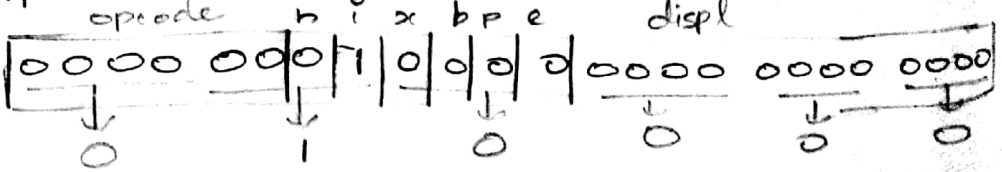
flag bit e=1.

LDX=04.



Case II LDA #0.

LDA=00



Case III +LDB #TABLE2. → format 4.

LDB=68

0110 1000

displ - add of operand = 1790

opcode n i x b p e

| 0 1 1 0 1 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0000 | 0000 | 0000 | 0000 |

6 9 1 0 0 0 0

Case IV ADD TABLE,X. (format 3).

Assume that addr mode is Program counter relative

ADD = 18

0001 1000

TA ~~disp~~ = PC + disp

disp = TA − PC.

= 0020 − 0000

= 13 (hex) → dec (10).

Index

disp <= 2047

Prog relative

0 0 0 1 1 0 1 1 1 0 1 0 0000 0000

displacement

opcode n i x b p e

→ | 0 0 0 1 1 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0000 | 0001 | 0011 |

1 B A 0 1 3

Case V ADD TABLE2,X (format 3)

Assume addr mode is program counter relative

ADD = 18

0001 1000

> 2047 Base rel.

19 < 2047.

∴ program counter relative

0 0 0 1 1 0 1 1 1 1 0 0 0000 0000 0000

disp = TA − PC.

1790 − 0010.

1 B C 0 0 0

1790 − 0010 = 1780

Case VI TIX COUNT. format 3.

TIX = 2C

0010 1100 n=1 i=1.

6016 > 2047 base relative

Base relative

disp = operand − base (TA)

1790 − 1790 = 0000

Assume PC relative addr mode

displ = TA − PC

= 001D − 0013 = A 10 < 2047.

opcode n i x b p e. P=1 displ.

0 0 1 0 1 1 | 1 | 0 0 1 0 | 0000 | 0000 | 1010

2 F 2 0 0 A

Case VII JLT LOOP (format 3).

JLT = 38

0011 1000

n=1 i=1

disp = TA - PC

= 000A - 0017

= FFFFFFF3 -13 < 2047

p=1.

n	i	x	b	p	e	disp		
001110	1	1	0	0	1	0	1111	1111	0011
↓ 2	↓ B			↓ 2			↓ F	↓ F	↓ 3

Case VIII +STA TOTAL. (format 4).

STA = 0C

0000 1100

n=1 i=1 e=1.

14/9/17 Thu

ASSEMBLER OUTPUT FORMAT:

- Format of object program

Object Program

1> Header (1)

2> Text record (more than one)

3> End record (1)

1> Header Record.

col 1 - H

col 2-7 - Program name.

col 8-13 - Starting address. (hex)

col 14-19 length of object program

Pgm: (Assembly)

```
1000  COPY  START  1000
1000  FIRST  STL  RETADR  141033.
1003  CLOOP  JSUB  RDREC  482039.
                .
                .
                .
              LDA  THREE  001020
101E  STA  LENGTH  001036.
1021  JSUB  WRREC  482061
                .
                .
                .
2079  END  FIRST.
```

H^COPY^001000^001079.

- COPY: ↑ pgm name
- 001000: ↓ starting addr
- 001079: ↓ Length of a pgm.

2> Text Record.

col 1 - T

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

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

col 10-69 - object code (in hex).

T^001000^1E^141033^482039^001036^281030^301015^
482061^3C1003^00102A^0C1039^00102D

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

⋮

3> End Record.

col 1 E

col 2-7 Address of 1st executable instruction.

E^001000

1 object code - 3 bytes
1 digit - 4 bit.
6 dig - 6x4-24 = 3 bytes

Object Program:

H^COPY^001000^001079
T^001000^1E^141033^482039..^00102D

T^001016^15^0C1036^482061...

⋮

E^001000

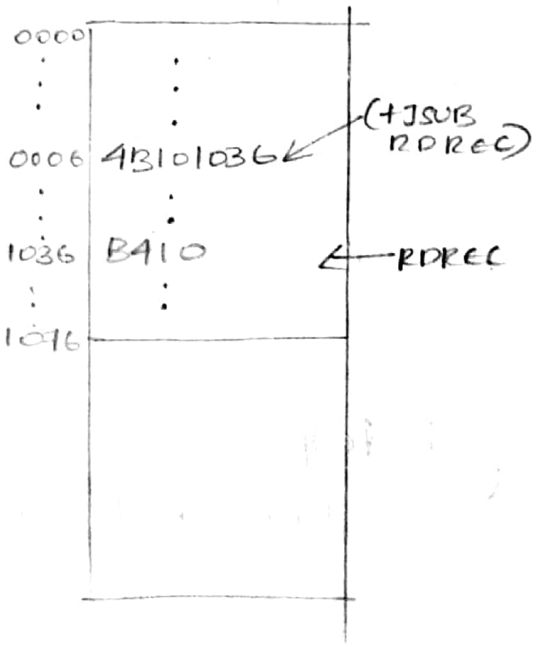
15/9/17 Fri

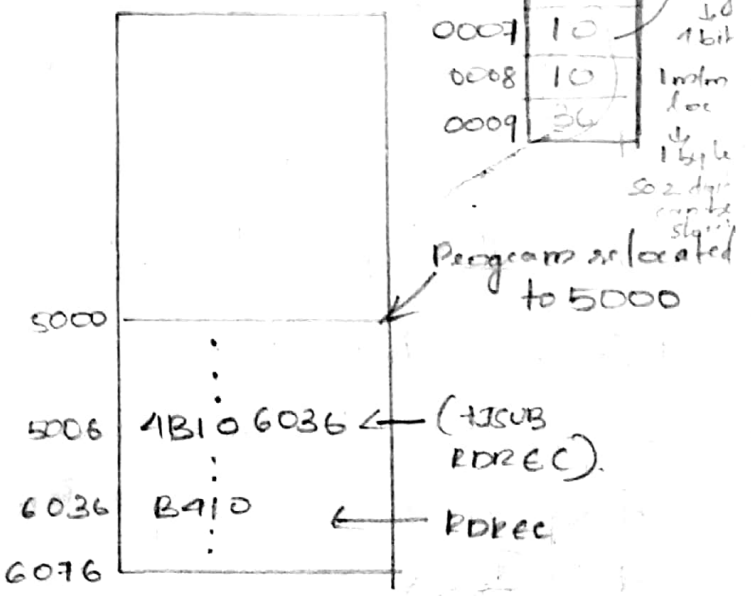
Machine Dependent Assembler Features.

- Instruction format
- Addressing mode
- Program Relocation.
  (Relocating object pgm to a new location as the specified location is already occupied)

Sample program.

LOC	LABEL	MNEMONIC	OPERAND	OBJECT CODE.
0000	COPY	START	0	
0000	FIRST	STL	RETADR	17202D
0003		LDB	#LENGTH	69202D
0006	CLOOP	+JSUB	RDREC	4B101036
		:		
0013		+JSUB	WRREC	4B10105D.
		:		
1076		END	FIRST.	





Modification Record

Col 1: M

Col 2-7: Starting location of address field to be modified

col 8-9: length (half byte)

col 10: flag (+/-)

col 11-17: Segment name

H^COPY^000000^01076

T^000000^1D^17202D^----

:

M^000007^05^+COPY

M^000014^05^+COPY

E^000000

→ Two Pass Assembler Algorithm.

Pass I (Define symbols).

1) Assign address to all statements in the program

2) Save the values assigned to all labels for use in pass 2

3) Perform some processing of assembler directives

Pass II (Assemble instructions & generate object program)

1) Assemble instructions

2) Generate data values defined by byte & word.

3) Perform processing of assembler directives not done during pass I.

4) Write the object program (to hard disk)

Pass I

```
begin
  read first input line
  if opcode = 'START' then
  begin
    save #[operand] as starting address
    initialize LOCCTR to starting address
    write line to intermediate file.
    Read next input line
  end {if start}
  else
    initialize LOCCTR to 0
  WHILE opcode ≠ 'END' do
  begin
    if this is not a comment line then
    begin
      search SYMTAB for LABEL.
      If found then
        set error flag (duplicate symbol)
      else
        insert (LABEL, LOCCTR) into SYMTAB
      end {if symbol}
    Search of OPTAB for OPCODE
      If found then
        add 3 {instruction length} to LOCCTR
      else if OPCODE = 'WORD' then
        add 3 to LOCCTR
      else if OPCODE = 'RESW' then
        add 3 * #[OPERAND] to LOCCTR
```

else if OPCODE = 'RESB' then
add #[OPERAND] to LOCCTR
else if OPCODE = BYTE then
begin
find length of constant in bytes
add length to LOCCTR
end {if BYTE}
else
set error flag (invalid operation code)
end {if not a comment}

Checking validity of OPCODE & incrementing LOCCTR

Write line to intermediate file read next input line
end {while NOT END}

Writing last record

Write last line to intermediate file.
save (LOCCTR - starting address) as program length
end {PASS 1}.

2/9/17 Wed

Pass II

begin.
read first input line {from intermediate file}.
if OPCODE = 'START' then
begin
write listing line
read next input line
end {if START}

```
Write Header record to object program
Initialize First text Record.
while OPCODE ≠ 'END' do
  begin
    if this is not a comment line then
    begin
      search OPTAB for OPCODE
      if found then
        begin
          if there is a symbol in OPERAND field then
          begin
            search SYMTAB for OPERAND.
            if found then
              store symbol value as operand address
            else
              begin
                store 0 as operand address
                set error flag (undefined symbol)
              end
            end {if symbol}
        else
          store 0 as operand address.
Assemble the object code instruction
{ end ~~if OPCODE found~~}
      end {if OPCODE found}.
    else if OPCODE = 'BYTE' or 'WORD' then convert
constant to object code.
    if the object code will not fit into the current
text record then
```

```
    begin
      Write text record to object program
      initialize new text record.
    end
  add object code to text record
end {if not comment}.
  Write listing line
  read next input line.
end {while not END}
  Write last text record to object program
  Write END record to object program
  Write last listing line
end {Pass 2}
```

⇒ **Single Pass Assembler Algorithm**

Load & go
After assembled it is loaded in directly to memory (Assembler loader)

Normal
the object program is stored in secondary and loader loads from there

```
begin
  read first input line.
  if opcode = 'START' then
  begin
    save #[operand] as starting address
    initialize LOCCTR as starting address,
    read next input line.
  end [if START]
  else
    initialize LOCCTR to 0
  Write header record, to the object program.
  while opcode ≠ 'END' do
      begin
        if there is not a comment line then
```

```
begin
 if there is a symbol in the LABEL field then
 begin
    search SYMTAB for LABEL
    if found then
 begin
      if symbol value is null
      set symbol value as LOCCTR and search the
    linked list. with the corresponding operand.
    move to the operand pointer address & change the
 operand value as LOCCTR value.
    PTR addresses and generate operand addresses
    as corresponding symbol values.
 Set symbol value as LOCCTR in symbol table and
 delete the linked list.
 end
 else
     insert (LABEL, LOCCTR) into SYMTAB
 end
     search OPTAB for OPCODE
     if found then
        begin
           search SYMTAB for OPERAND address
     if found then
          if symbol value not equal to null then
    store symbol value as OPERAND address
    else
          insert at the end of the linked list with
    a node with address as LOCCTR
    else
      insert (symbol name, null)
    add 3 to LOCCTR
    end
    else if OPCODE = 'WORD' then
        add 3 to LOCCTR & convert constant to
    object code.
```

```
else if opcode = 'RESW' then
    add 3 * [OPERAND] to LOCCTR
else if opcode = 'RESB' then
    add #[OPERAND] to LOCCTR
else if opcode = 'BYTE' then
  begin
    find length of constant in bytes
    add length to LOCCTR
    convert constant to object code
  end
  if object code will not fit into current text
  record then
  begin
    write text record to object program
    initialize new text record.
  end
  add object code to Text record.
end
write listing line
read next input line
end
  write last Text record to object program
  write End record to object program
  write last listing line
end (Pass 1).
```

9/10/2017 Wed

Module 3

Assembler Design Options:

→ Machine independent features of assembler.

- Program block
- Control section

* Rearrange the object code within the object program

fragment of code in the source program that can be rearranged in the ~~so~~ object program is called program block.

Eg: Program : Program block.

```
0000   COPY   START  0
0000   FIRST  STL    RETADR
0003   CLOOP  JSUB   RDREC
0006          LDA    LENGTH
0009          COMP   #0
                :
0000          USE    CDATA
0003   RETADR RESB   1
                :
               USE    CBLKS.— for higher bytes
       BUFFER RESB   4096
                :
               USE
       RDREC  CLEAR  X —> 0009
              CLEAR  A
                :
              USE CDATA
                :
```

Default

unnamed prgm block.

CDATA

CBLKS

default

```
USE
:
:
USE CDATA
:
:
END
```

default

CDATA

Program Block Data Structure.

Block name	Block Number	Address	Length
Default	0	0000	0066
CDATA	1	0066 (0000 + 0066)	000B
CBLK	2	0071 (0066 + 000B)	1000.

Order in the input

Assembler directive

USE → defines the named pgm block.

Eg: USE CDATA → pgm block name.

The statements following this name belong to pgm block CDATA until next USE

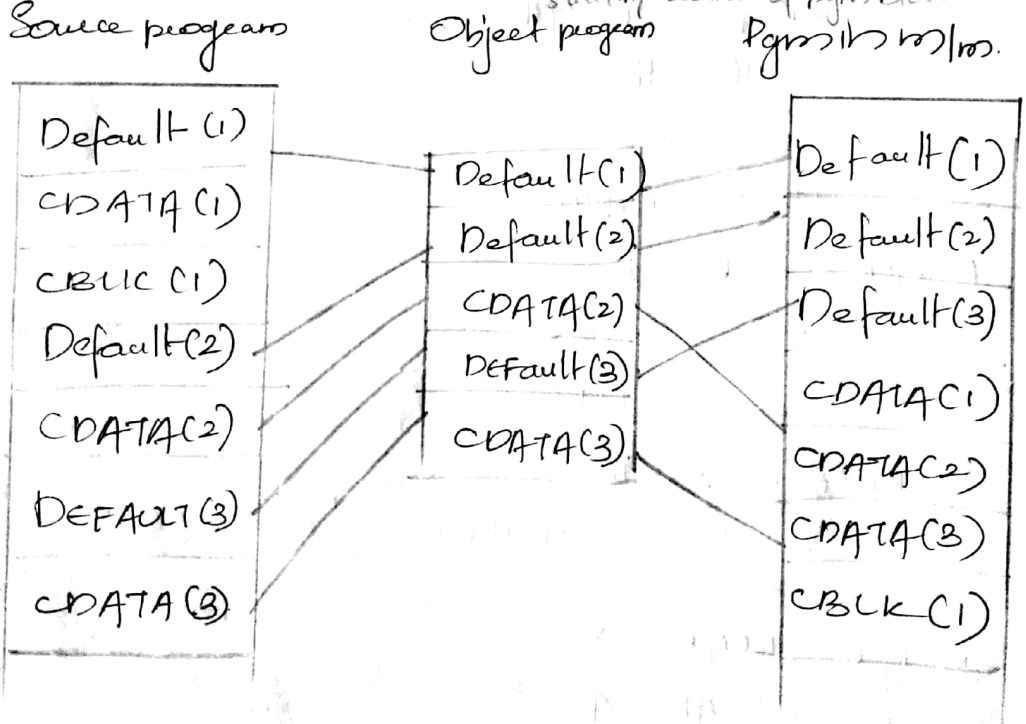
If we write USE alone → first following belongs to default pgm block.

→ Each program block has separate LOCCTR

→ modification in SYMTAB: done using pgm block SYMTAB.

LABEL	VALUE	BLOCK NUMBER
LENGTH	0003	1

Relative value of label with respect to starting addr of pgm block



displ. Address calculation

LOA LENGTH: TA − PC

= 0069 − 0009

= 0060 //

→ Control Sections:

The output of assembler will be n object modules if we use control sections

Separately assembled, loaded, relocated & modified.
Each control section has separate text header, text & end records

Eg: Program:

```
0000  COPY   START  0
             EXTDEF  BUFFER,BUFEND,LENGTH
             EXTREF  RDREC,WRREC
0000  FIRST  STL    RETADR  172027
0003  CLOOP  +JSUB  RDREC   4B100000
              :
0023         +JSUB  WRREC   4B100000
              :
0020  LENGTH RESW   1
              :
0033  BUFFER RESB   4096
1033  BUFFEND EQU   *
              :
0000  RDREC  CSECT          → defines control section.
             EXTREF  BUFFER, LENGTH, BUFEND
0000         CLEAR  X   B410
0002         CLEAR  A   B400
0004         CLEAR  S   B4400
              :
0020  EXIT   +STX   LENGTH  13100000
              :
0028  MAXLEN WORD   BUFEND-BUFFER  000000
0000  WREC   CSECT
```

5/9/2019
Thu

```
EXTREF  LENGTH,BUFFER

0000 CLEAR X
0002 +LDT LENGTH
       :
       :
     END FIRST.
001B  =X'05'
```

EXTREF

EXTDEF:

Object program

```
H^COPY^000000^001033
D^BUFFER^000033^BUFFEND^001033^LENGTH^00002D
R^RDREC^WRREC
T^000000^1D^172027^4B100000^...
       :
       :
M^000004^05^+RDREC
M^000011^05^+WRREC
       :
E^000000
       .
       .
       .
       .
H^WRREC^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^7710000^...
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

→ Define Record:

col 1 D

col 2-7 Name of external symbol defined in this control section

col 8-13 Relative address of symbol within this control section (hexadecimal).

col 14-72 Repeat info in col 2-13 for other external symbols

→ Refer record:

col 1 R

col 2-7 Name of external symbol referred to in this control section

col 8-73 Names of other external reference symbols

→ Modification Record:

col 1 M

col 2-7 starting address of the field to be modified, relative to the beginning of the control section. (hexadecimal)

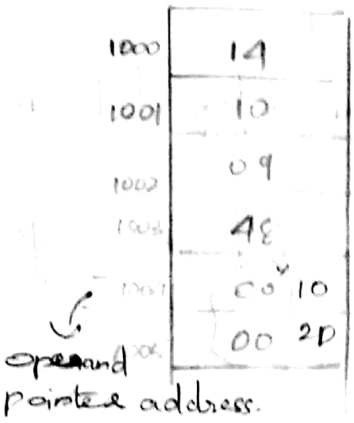
col 8-9 length of field to be modified, in half bytes

→ Single pass Assembler Algorithm

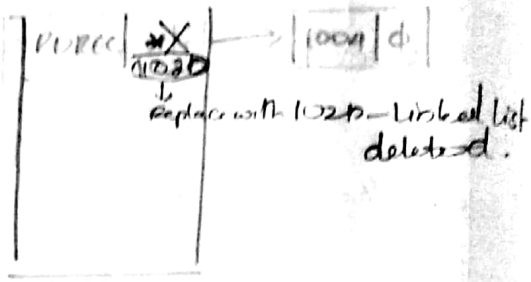
1) Load & go:

Eg

```
1000   FIRST   STL   RETADR   141009
1003           JSUB  RDREC    481000
                 :
                 :
                 :
102D   RDREC   RESB  1036.
```



SYMTAB.



2) Normal Assembler

The object codes are present in sec storage. The loader has to load this into memory for modifying.

Additional info needs to be provided by assembler to loader to make the changes.

T^001004^02^102D

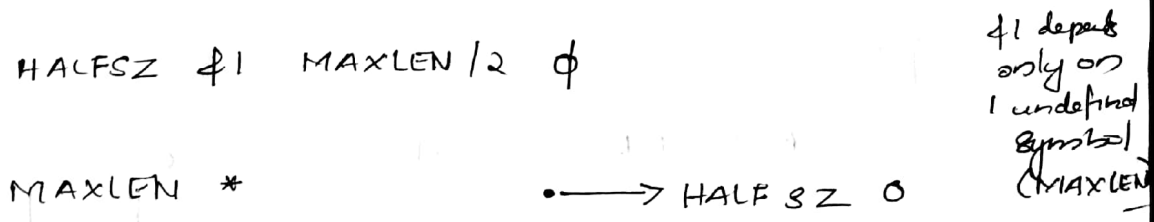
↓ location ↓ bytes ↓ operand add.

7/10/24

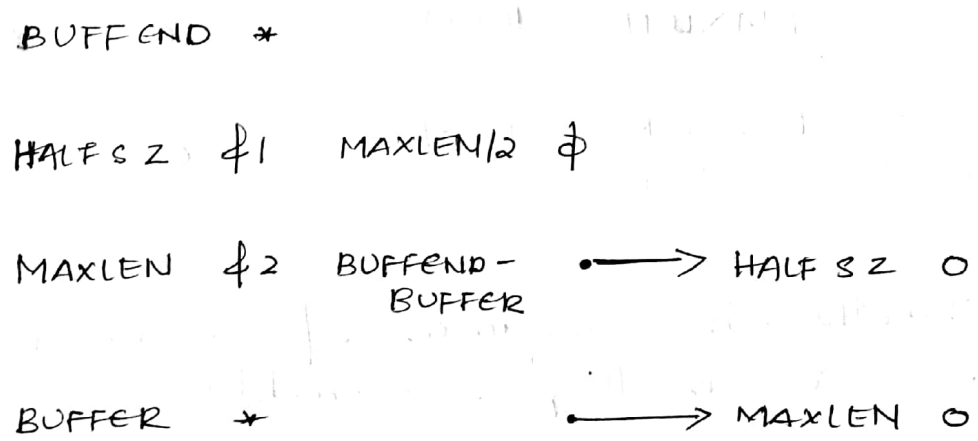
Multipass Assembler

```
1. HALFSZ   EQU   MAXLEN/2
2. MAXLEN   EQU   BUFFEND-BUFFER
3. PREVBT   EQU   BUFFER-1
                  :
1034 4. BUFFER   RESB  4096
2034 5. BUFFEND  EQU   *
```

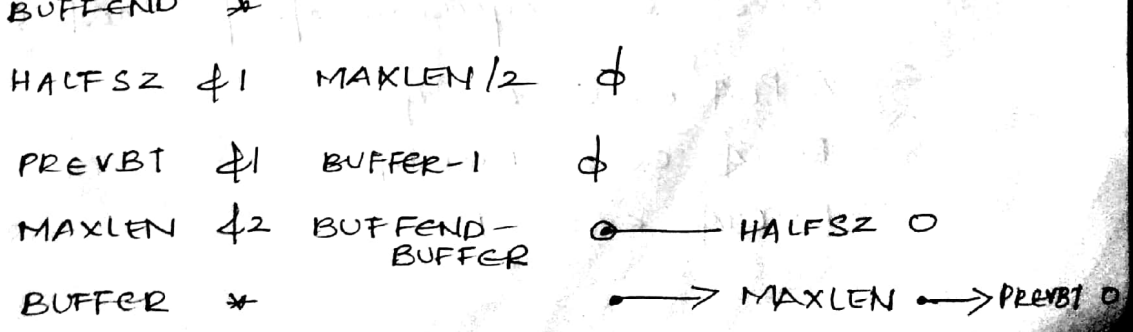
fig(1).



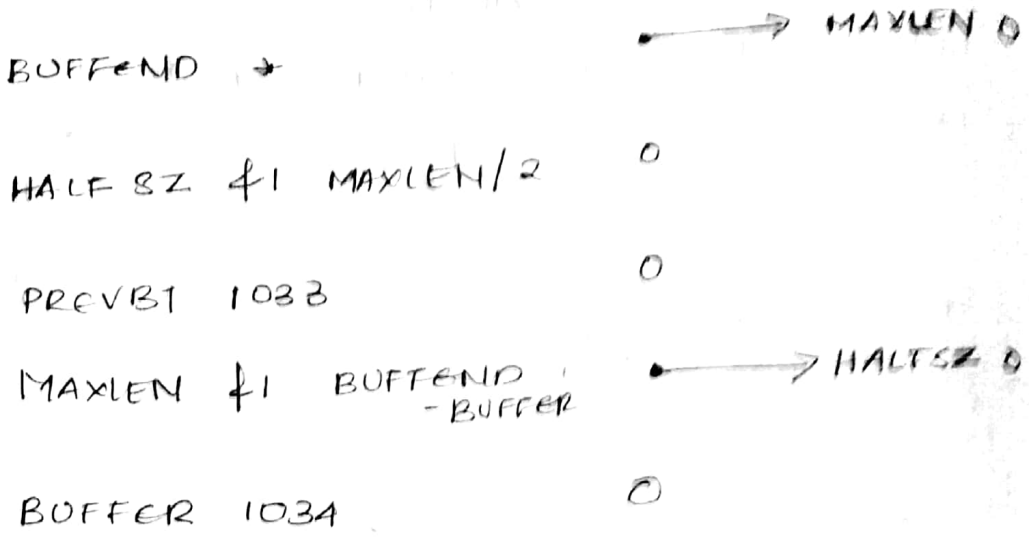
fig(2)



fig(3).



fig(4)



fig(5).

BUFFEND	2034	0
HALFSZ	500	0
PREVBF	1023	0
MAXLEN	1000	0
BUFFER	1034	0

When there is a sequence of forward reference, we use multi pass assembler

Multi pass done only on the fragment of code that involves a sequence of forward references

Eg: ALPHA EQU BETA
BETA EQU DELTA
DELTA EQU 1

9/10/2024 Mon

Module 5

MACRO PREPROCESSOR.

Macro:

It is an abbreviated name given to a fragment of code

3 components:

1> Macro call

2> Macro definition

3> Macro expansion

→ **Macro definition. (in SIC/XE)**

3 parts:

1> Macro prototype statement (header stmt)

gives info on macro name, formal arguments used and its type

General Syntax:

<Macro name> MACRO <formal argument specification>

↓ keyword ↓ &<arg name> <kind>

Eg: STRG MACRO &a1, &a2

macro name — formal argument

2> 1 or more model statements.
normal assembly stmts.

3> End statement (End stmts).

MEND. → to denote end.

Eg: STRG1 MACRO &a1, &a2
(&a1 → prototype)

model stmt
{ STA &a1
  STL &a2

MEND
(→ end stmt)

→ Macro Call.

Syntax:
<Macro name> <actual arguments>

Eg: STRG1 a,b
(STRG1 → macro name; a,b → actual arg.)

→ Macro expansion

Whenever macro call occurs macro code is substituted in it.

Formal arguments substituted with actual arg.

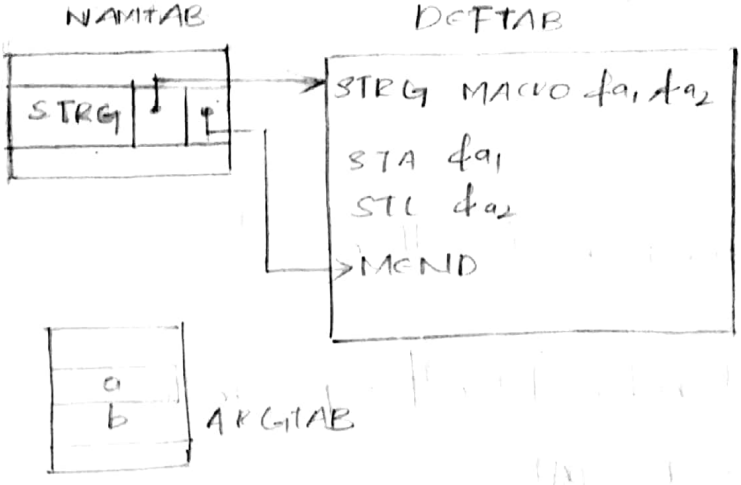
Eg:
```
COPY   START  1000
STRG1  MACRO  &a1, &a2   //macro def.
STA    a1
STL    a2
MEND
  :
STRG1  a,b    //macro call
END    FIRST.
```

(macro def. → given → macro processor)

Expanded pgm.
```
COPY  START  1000
        :
STA   a      } macro expand
STL   b      }
        :
```

→ Data Structures:

1> DEFTAB → Stores macro definition
2> NAMTAB → Stores macro name + 2 pointers
   ├ 1. start of definition in DEFTAB
   └ 2. END "
3> ARGTAB → Stores actual arg.



* Algorithm for one pass macro-processor.

```
begin {macro processor}
    EXPANDING = FALSE
  while OPCODE ≠ 'END' do
  begin
      GETLINE
      PROCESS LINE
  end {while}
end {macro processor}.
procedure PROCESSLINE
  begin
     Search NAMTAB for opcode.
     If found then
     EXPAND
     else if opcode = 'MACRO' then
     DEFINE
     else write source line to Expanded file.
      end {PROCESS LINE}
```

```
COPY START 1000
STRG MACRO fa1, fa2
     LDA a1
     LDX a2
MEND
  .
  .
  .
STRG b1, b2
  .
  .
  .
END
```

```
procedure DEFINE
begin
  enter MACRO name into NAMTAB.
enter macro prototype into DEFTAB
LEVEL:=1
while LEVEL>0 do
 begin
   GETLINE
 If this is not a comment line then
 begin
   substitute positional notation for parameters
   enter line into DEFTAB.
   If OPCODE = 'MACRO' then
    LEVEL := LEVEL+1
   else if opcode = 'MEND' then
     LEVEL := LEVEL-1
   end{if not comment}
 end{while}
  store in NAMTAB pointers to begin & end of definition
 end{DEFINE}
 procedure EXPAND
 begin
    EXPANDING := TRUE
    get first line of macro definition {prototype} from DEFTAB
    set up arguments from macro invocation in ARGTAB
    write macro invocation to expanded file as comment
    while not end of macro definition do
```

```
begin
    GETLINE
    PROCESS LINE end{while} expanding := FALSE
end {EXPAND}

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

→ Machine independent Macroprocessor feature:

1> Concatenation of macro parameters
2> Generation of unique labels.
3> Conditional macro expansion.
4> Keyword macro parameter.

```
<1> STRG MACRO (&ID)
        LDA  X &ID→1
        ADD  X &ID→2
        ADD  X &ID→3
        STA  X &ID→4
        MEND.
        :
        STRG A.
```

```
STRGA → LDA XA1
        ADD XA2
        ADD XA3
        STA XA4.
```

A1, A2, A3, A4 are increment of variable A

TAB.

& and → are used for concatenation.

This saves storage and increases & efficiency.

An argument can be concatenated with string to end & beginning

We can generate series of variables/arguments.

<2> Generation of unique labels.

Eg:

```
START
STRG MACRO &a1
$LOOP LDA a1
      .
      .
      .
      JLT LOOP
      .
      .
      MEND
STRG a
STRG b
END
```

```
START
.
.
.
$AA LOOP LDA a
$AB LOOP LDA b
.
END
```

The labels should be unique. To avoid repetition the macroprocessor allows appending a '$' sign to the beginning of the label.

$ is replaced with alphanumeric code $xx

↓

AA
AB
AC
AD

<3> Conditional macro expansion.

(I) IF...ENDIF / IF...ELSE...ENDIF

Eg:1) MACROS MACRO &a1, &a2

```
IF (&a1.NE.' ')
LDT #4096.
END IF
LDX #400.
```

macro call: MACRO 3

a1 ≠ NULL F.

a1 ≠ NULL

LDT #4096

LDX #400

↓ Expanded code a1 ≠ NULL

LDX #400

Adv:

MEND.

Eg2> MACROS MACRO &a1

```
if (&a1 EQ 1)
LDT #200
else
LDT #300
endif
MEND.
```

Expanded code

MACRO 1,3
↓
LDT #200.

MACRO 2,3
↓
LDT #300.

(II) While - Endw

Eg3>

```
MACROS MACRO &a1
while (&a1 LE 4)
    LDA a1
    LDX a2
    &a1 set &a1+1
endw
STLH &a2,X
    :
MEND
```

MACRO 2.

2 → LDA a1
LDX a2.

3 → LDA a1
LDX a2

4 → LDA a1
LDX a2

5 → STLH &a2,X

set ↓ a1 = a1+1

Eg4>

```
MACROS MACRO &a1
&cte set 1
while (&cte LE &a1)
    LDA &a1
    &cte set &cte+1
endw
MEND.
```

The statements are filtered based on conditions.

Any variable inside the macro definition which is preceded by '&' symbol is called macro time variable.

Adv: We can exclude some code during macro expansion.

<A> Keyword macro parameter.

⇓

default values.

Eg: MACROS MACRO &b1=, &b2=2, &b3=3, &b4=

:

MEND.

macro call

MACROS b1=1, b4=5

MACROS b1=1, b2=4, b4=6.

Each argument is identified with the name not by position

1> b1=1, b2=2, b3=3, b4=5

2> b1=1, b2=4, b3=3, b4=6

Two ways:

1. Positional parameter specification.
2. Keyword parameter "

Using positional parameter specification

⇓

Eg: MACROS MACRO &b1, &b2, &b3, &b4

:

MEND.

If b2 & b3 has to be assigned null values.

b1 & b4 has some value leave the space for null value

MACROS 1, , , 5

MACROS 1, 4, , 5

⇓

Even though no value is to be assigned the space has to be left

Not an efficient method.

⇓

Hence use keyword parameter specification

14/10/2011 Sat

Macroprocessor Design Options

- Recursive macro expansion
- General macroprocessor
- Macroprocessor with language translator.

1) Recursive macro expansion

Calling a macro inside the definition of another macro → nested macro call.

If we call the same macro within the macro definition → macro recursion.

```
Eg: STRG  MACRO  &a1 ,&a2
        :
    sample b1  //Nested macro call
        :
    MEND
    SAMPLE MACRO &b
        :
    MEND

START
STRG a,b  //outer macro call
    :
END.
```

* nested macro call is not supported in the macroprocessor.

When nested macro call occurs.

ARGTAB will get overwritten by the arguments of inner macro call.

2) General purpose macro processor

It process macro calls in any programming language. Development cost is very less. Saves learning cost maintenance effort is less.

→ Challenges faced by developer

i) Language specific command issues.
(eg. commands, grouping facility, tokens (identifiers, operators, keywords), syntax (change in syntax of macro invocation and macro expansion).

3) Macroprocessor with language translator.

→ i) Line by line macroprocessor

Line by line expand the & the expanded code is directly send to assembler. There is no need for expanded file.

Advantages:
→ Avoid extra passes
→ Some data structures like NAMTAB and OPTAB can be combined
→ Subroutines can be shared (utility fns like search..)
→ It can identify the errors and display at the right time. There is no delay as it is a line by line macroprocessor.

<2> Integeated macroprocessor

The macroprocesso r is apart of the compiler. There is a merging of assembler and translator.

Development cost is less.

Drawbacks.

Size of the system will be high. It requires more memory. Complexity of the s/w is high.

Time required to translate a line in the source pgm is more in case of integeated macroprocessor.

Assembler acts first then the macroprocessor. Ie, there is a mixing of functionality. (Assembler scans to get the variable names).

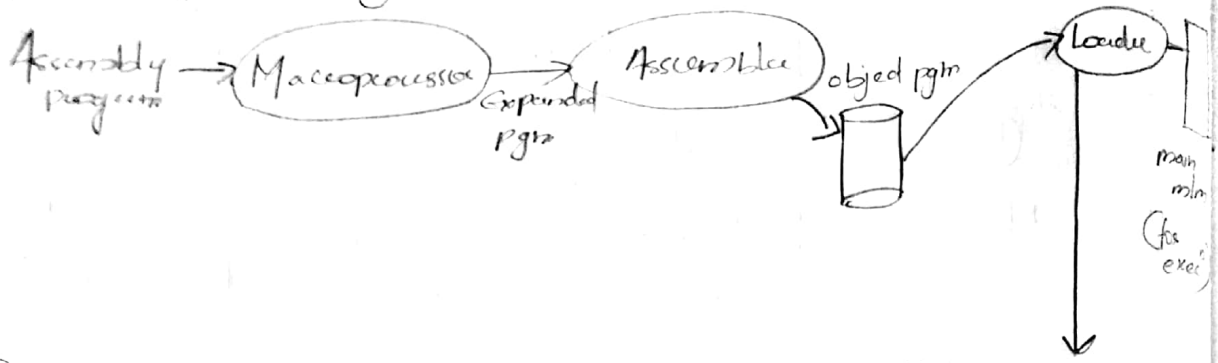
28/9/2014
Mon

Module 4

Linker and Loader

Loader:

It is a sys soo that loads the progeam from secondary storage to main memory.



Functions: 1> Loading
2> Relocation
3> Linking

Relocation:

Implemented by loader using modification record or using modifi bit.

Linking

Linking of n object modules (using control sections) for final execution of a single executable file

Types of loaders:

1> Assembler & go loader
2> Absolute loader
   - Bootstrap loader.
3> Relocating loader
4> Linking loader

1) Assemble & go loader.

The object pgrm is directly loaded to main mem.

Drawback

→ The progrm need to be assembled for each execution.
→ Parellely the program is assembled & loaded. So a part of the memory is occupied by assembler. So mem is limited to store object pgrm.
→ ~~three~~

2) Absolute loader.

Proogram is loaded to the absolute address (addr specified in progam). No space for relocation & linking. Absolute loader takes the progam from sec storage

Drawback

→ Progammer ~~need not be a~~ faces a tideous task.

Advantage

Entire portion of mem is available for storing object progams.

⇒ Bootstrap loader.

Stored in ROM. It loads the OS (loaded to 0x80)

Absolute Loader Algorithm

```
begin
  Read Header record
  Verify program name and length
  Read first text record
  while record type ≠ 'E' do
  begin
    If object code is in character form
    convert into internal representation
    Move object code to specified location in memory
    Read next object program record
  end
  jump to address specified in End record
end
```

Header record

Text record

Processing End record

Machine dependent Loader features:

1) Relocation
2) Linking

Program linking:

Control section1 PROGA

```
0000  PROGA   START  0
               EXTDEF LISTA, ENDA
               EXTREF LISTB, ENDB, LISTC, ENDC
0020  REF1    LDA    LISTA
0040  LISTA   EQU    *
0054  ENDA    EQU    *
```

```
0054   REF4   WORD   ENDA-LISTA+LISTC
                 :
                 :
                END
```

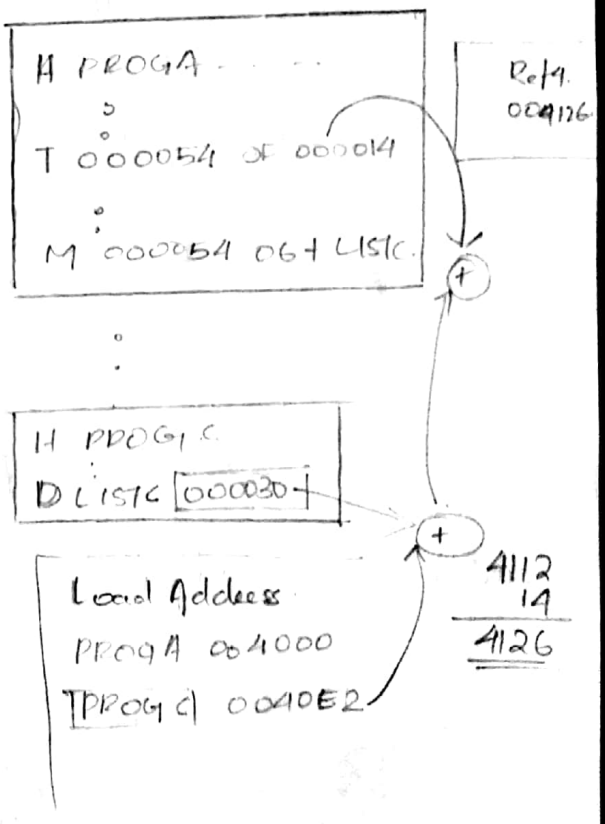
Control Section 2

```
0000   PROGC   START  0
               EXTDEF LISTC,ENDC
               EXTREF ...
                 :
                 :
0030   LISTC   EQU *
                 :
                 :
               END
```

Object Program

```
H^PROGA^000000^000063
D^LISTA^000040^...
R^....
  :
T^000054^0F^000014^----
M^000054^06^+LISTC
E^000020

H^PROGC^000000^00005
D^LISTC^000030^...
  :
E^---
```



→ Algorithm for Pass 1 of a linking loader

26/10/17 Thu

Pass 1:

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

27/10/17 F

end {for}
end {while ≠ 'E'}
add CSLTH to CSADDR (starting address for next control section)
end {while not EOT}
end {Pass 1}

27/10/17 Fri

Pass 2:

Pass 2
Process header rec
Text rec
modif^n rec

begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
begin
read next input ~~line~~ record {Header record}
set CSLTH to control section length
while record type ≠ 'E' do
begin
read next input record
if record type = 'T' then
begin
{if object code is in character form, convert into internal representation}
move object code from record to location {CSADDR + specified address}
end {if 'T'}
else if record type = 'M' then
begin
search ESTAB for modifying symbol name.
if found then

add or subtract symbol value at location
(csADDR + specified address)

else
set error flag (undefined external symbol)

end {if 'M'}

end {while ≠ 'E'}

if an address is specified {in End record} then

set EXECADDR to (CSADDR + specified address}

add CSLTH to CSADDR

end {while not EOF}

jump to location given by EXECADDR (to start execution of loaded pgm)

end {Pass 2}.

25/10/17 Program Relocation

1. Using modification record.
2. Using modification bit.

Using modification bit

Eg:

H^COPY^000000^001074

T^000000^1E^FFC^140033^...

Bitmask

:

E^000000

FFC

1111 1111 1100

F F C

12 bit-Bitm
— hex 44
if bit value
0
no modif
requ
if bit-1
modif
requ

Algorithm

```
begin
    get PROGADDR from operating system
    while not end of input do
    begin
        read next record
        while record type ≠ 'E' do
        while record type = 'T'
        begin
            get length = second data
            mask bits (m) as third data
            for (i = 0; i < length; i++)
            if Mi = 1 then
                add PROGADDR at the location PROGADDR
                                    + specified address.

            else
                move object code from record to location
                                    PROGADDR + specified addr
            read next record
        end
        end
    end
end {algorithm}
```

mask bit - 12 = no of bytes of code in the text rec.

8/9/2017

Machine independent loader features:

There are two types:

1> Automatic Library search

2> Loader options.

Automatic Library search

A loader feature for linking the subroutines that are predefined in the library object progra

Check ESTAB during pass 1

↓

Those which may not be found may be subroutines.

↓

unresolved external symbols in ESTAB

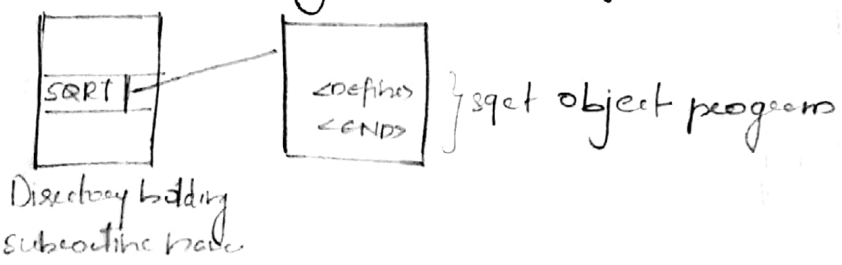
↓

Subroutine in library

↓

By scanning object programs (define records) of subroutine in library

Efficient method: Using a directory structure (indexing)



Loader options:

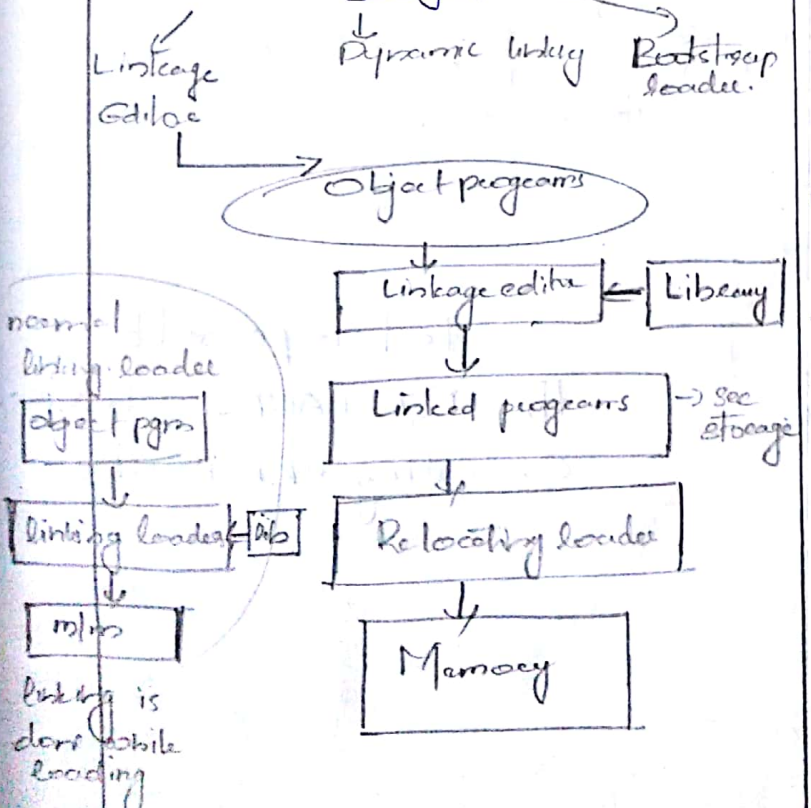
Eg: a> INCLUDE program-name (Library name)
b> DELETE CSECT-name // delete a named CSECT.
c> CHANGE name1, name2 // to specify a change in the name of external symbol
d> INCLUDE READ(UTLIB)
INCLUDE WRITE(UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ.
CHANGE WRREC, WRITE
e> LIBRARY MYLIB

Functionalities of linkage editor:
→ generating linked version of program for loading, relocation & execution
⇒ used to replace subroutine linked version.

How to define these options?

<1> By using command languages.
<2> Maintain a file which includes all commands.
<3> Specify all commands after an object program module
<4> Give commands in source program.

Loader Design Option.



Those programs which are executed frequently → linkage editor is useful.:

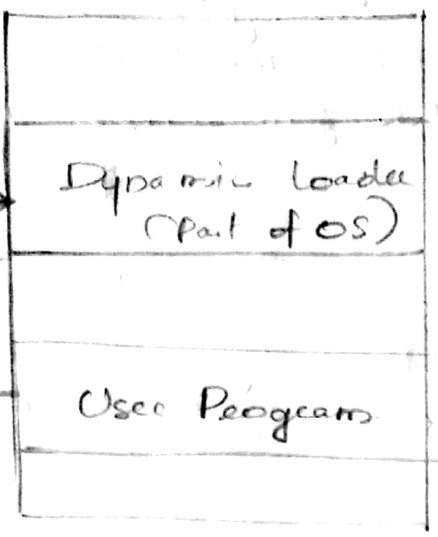
Linking loader outperforms linkage editor for those pgms. which are not frequently executed.

Linkage editor is useful when there are frequent changes in the pgm.

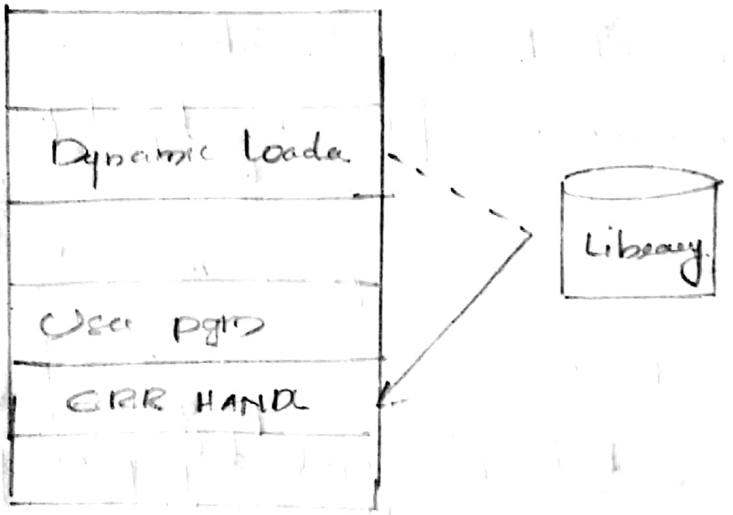
1/11/2018
Wed

Dynamic Linking

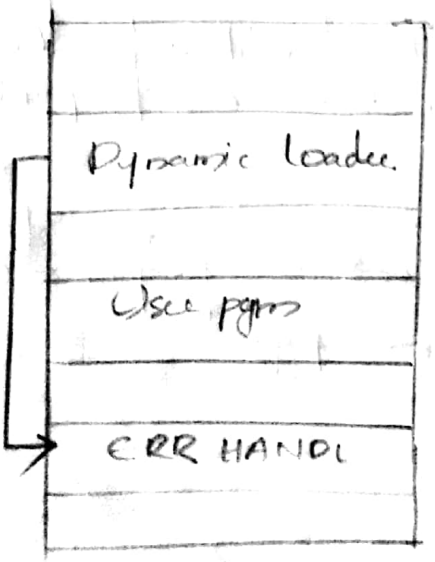
Linking is done at run-time



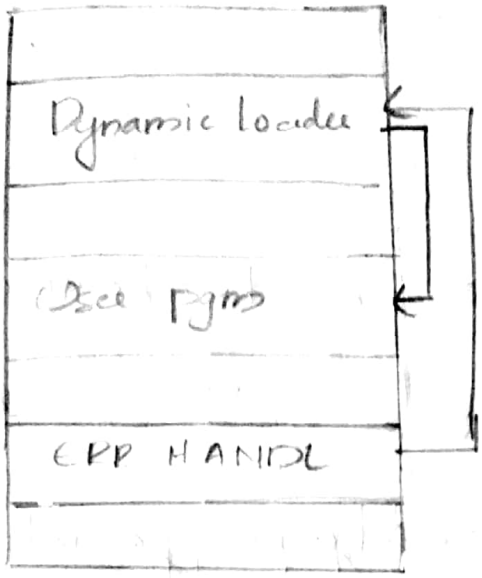
Call ERR HANDL



Dynamic loader searches library. Fetches the subroutine ERR HANDL from library & load to mem



Control is passed to the ERR HANDL subroutine Executing ERR HANDL



After execution control is returned back to dynamic loader & loader return control to user program.

→ Dynamic linking / dynamic loading / load on call.

→ Dynamic loader performs dynamic linking.

Advantages compared to linking loader.

1> No memory wastage.

As the subroutines which are necessary during execution time has to be linked.

This saves time and memory.

Module 6.

1/11/2017

→ Device drivers.

- Interface b/w OS and hardware devices.
- OS specific commands ⟺ Device specific commands
- 1 for module separately plugged into OS kernel

→ Types of device drivers:

1) Character device driver
2) Block device driver
3) Terminal device driver
4) Stream. "

Translator – s/m s/w
↓
OS specific call → Driver → Peripheral command

Appn → OS → Driver → Peripheral device
s/m call, s/m call, commands

* part of OS kernel.
* separate s/w module plugged into OS kernel

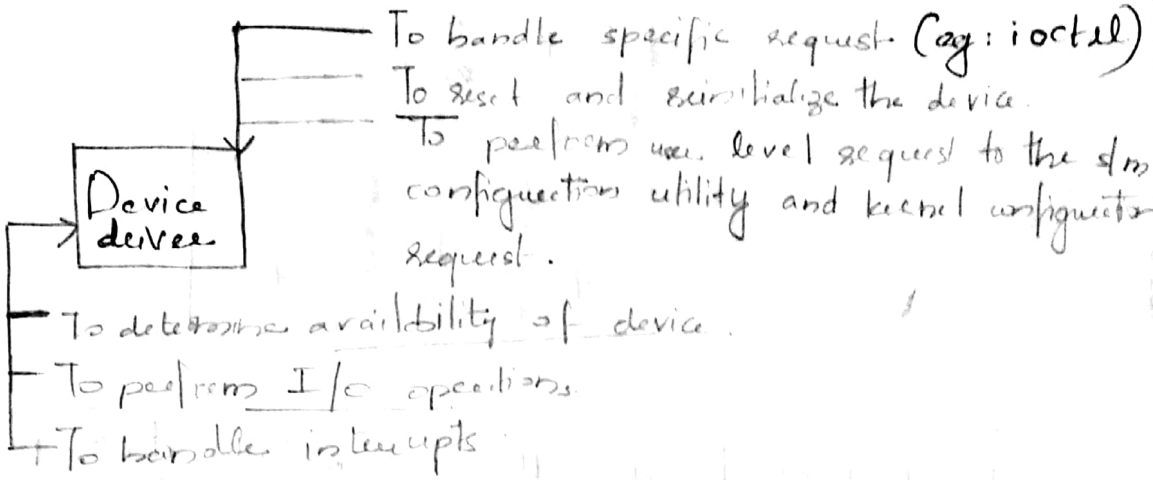
1) Block Device driver.

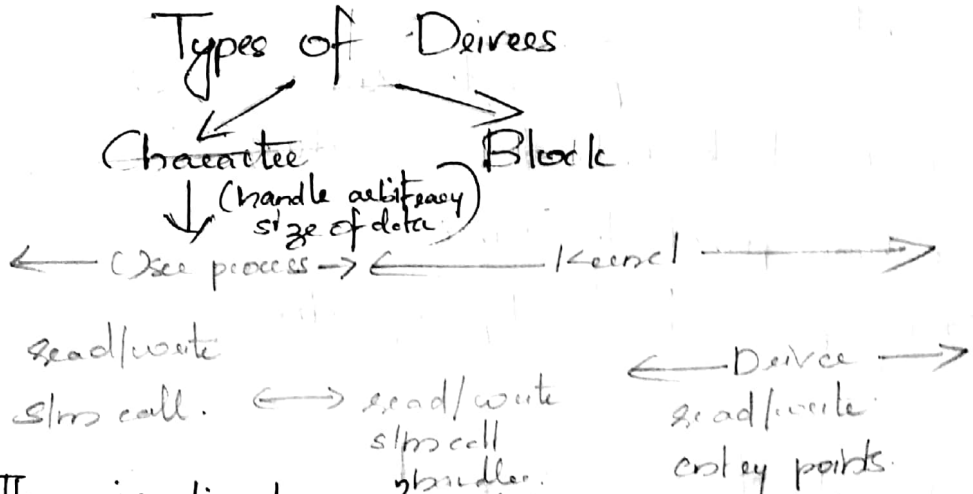
Data is read/written in the form of blocks. Peripheral devices which has file s/m communicate with OS using block device driver.

for connecting harddisk we need block device driver.

2/11/2017 Thu

Functionalities of Drivers





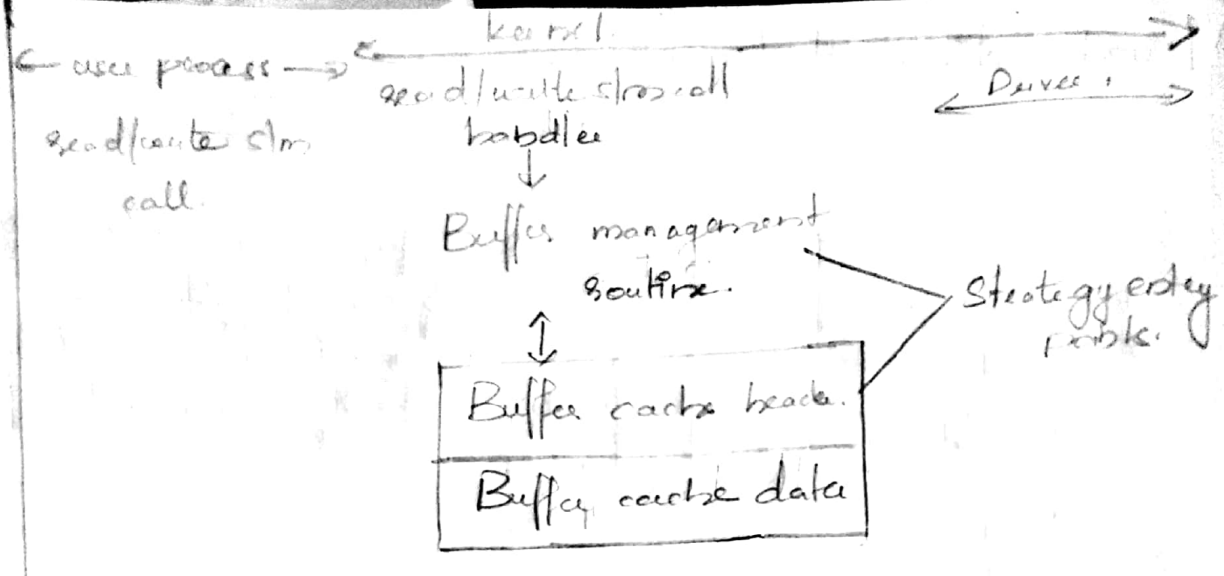
* There is direct comm b/w OS & device driver. & device & device driver.

Block device drivers: (fixed size data blocks)

Eg. driver for harddisk, ~~pendrive~~

* There is a cache s/m maintained to store the block of data.

* Device driver fetches the data blocks from cache s/m. Similarly data is put to cache for use by OS.



Anatomy of Device driver: (includes driver prologue & driver entry pts)

- Driver contains
  - Data structure private to driver.
  - Reference to kernel data structure
  - Routine private to driver.
  - Entry points to do I/O operation

- Initial part of driver is prologue
  - #include → Define various kernel data structure
  - #define → Define constants to define conditions.
  - Declarations

Eg:

#include <sys/type.h> — definition of data type
#include <sys/param.h> — " kernel parameter
#include <sys/dir.h> — " directory structure
#include <sys/errno.h> — " error code

Entry points <ptr to device entry point that handle I/O device.

a) init() - initialize device & b/w.

b) start() - Also does initialization

c) open (dev, flag, id) → called by kernel when user process performs open s/m call.

d) close (dev, flag, id) - called by kernel to close all file.

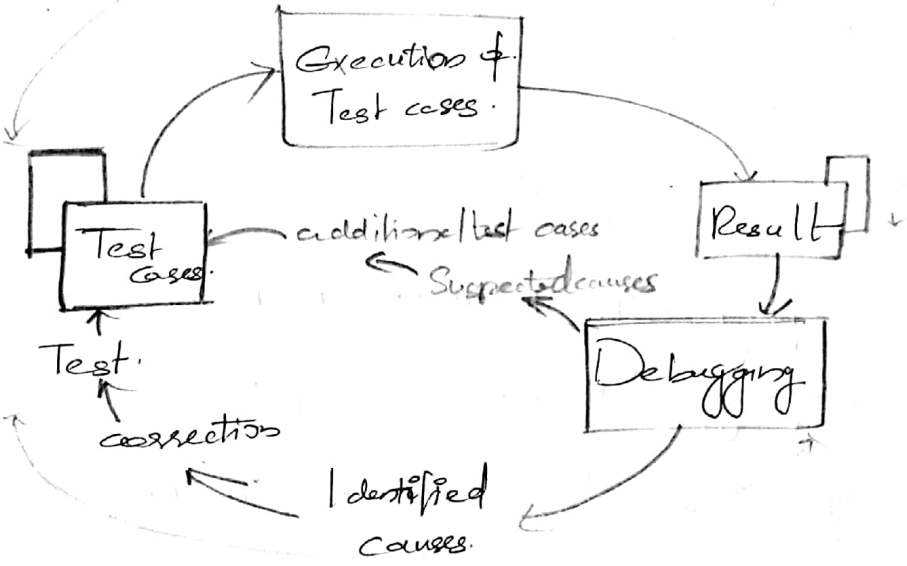
e) halt() - called by kernel just before shut down.

f) probe (vector)

g) read(dev).

3/11/2017 Fri

Debugging:



Testing: Locating the error (using test case expected o/p & observed o/p

debugging:

- followup process after testing.
- Two step process.
  1) Localising the error and identifying the nature.
  2) Correcting the error.

→ Debugging functionalities & Capabilities.

1> Execution sequencing (position to observe flow of execution of pgm).
- Halting: Suspend execution of pgm.
- Breakpoints: Suspend pgm execution at a particular pt.
- Conditional expression: If expr is T, pgm is suspended.

2> Tracing
Tracking the program control flow / logic flow.

3> Traceback
Tracking the path through which program has executed.

4> Program display capability.
- Display with statement number.
- Display with macro expansion.
- Able to modify and incrementally recompile
- Symbolically displaying memory constants

→ Need for single debugging system: - to handle multiple languages: should be able to switch context

→ Debugging sys should be able to deal with optimized code.

→ Relationship with other parts of the system

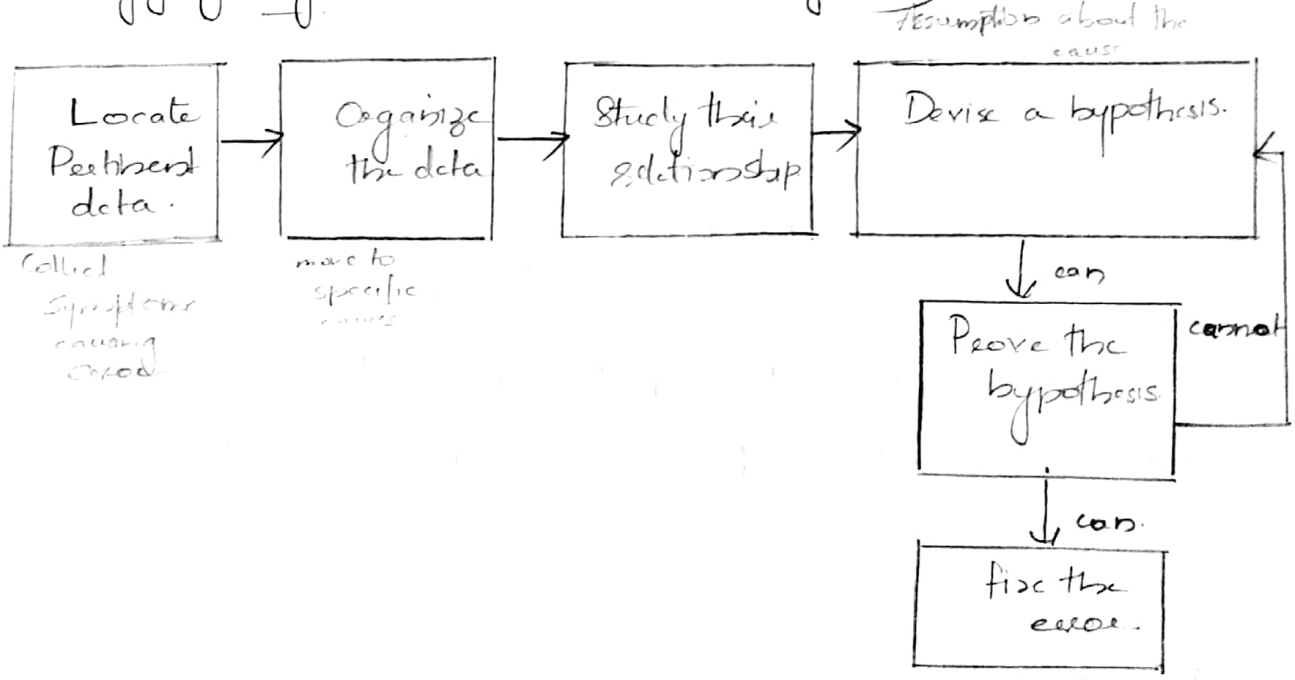
* should be an integral part of the system.
* communicate with OS components
* consistent with security & integrity component of s/m.
* coordinate activities with compiler / interpreter

→ * Debugger should provide authorization
* maintain audit trail (record all actions performed by user).

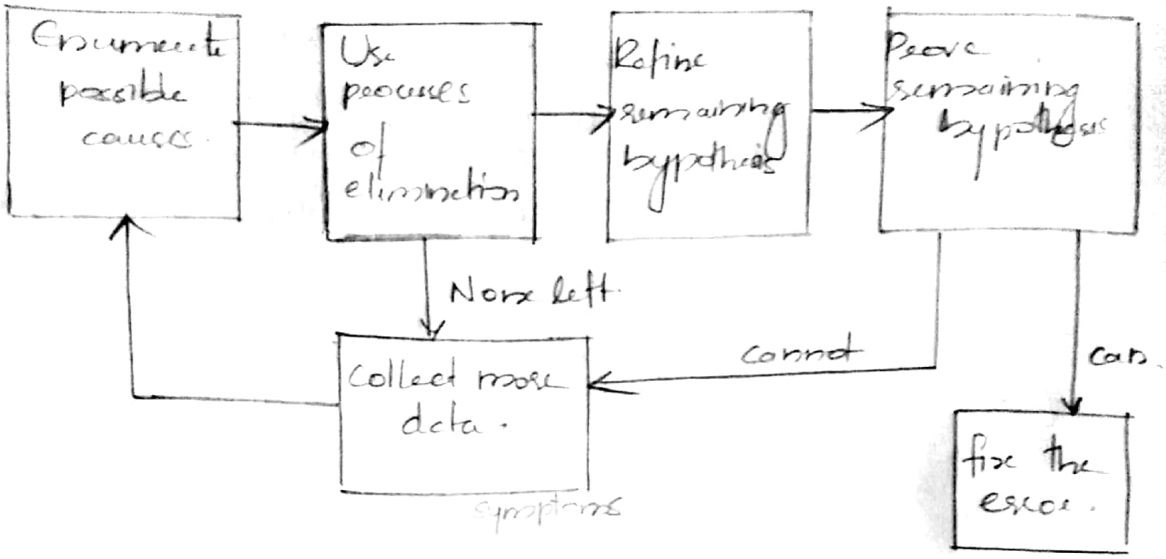
6/11/2017 Mon

Debugging methods

1) Debugging by Induction: (specific → general)



2> Debugging by deduction: (General to specific.)



Collect all possible causes of the error

Take all relevant causes.

3> Debugging by backtracking:

mental reversing of the program until we find the cause of the ~~prog~~ error. (point which caused the error).

8/11/2017 Wed

Text Editor:

* Text editor definition
* User interface
* Editor structure.