String Instructions →

Direction Flag (DF) → determine in which direction pointer moves over the string.

DF = 0 ⇒ SI, DI will proceed to increasing memory address ie Left to Right (for array)

DF = 1 ⇒ SI, DI will proceed to decreasing memory address i.e. Right to Left (for array)

```
CLD ;  // clears DF  i.e. performs DF = 0
STD ;  // Sets DF to 1. performs DF = 1

STR1   DB 'HELLO'
STR2   DB 5 Dup(?)   // we want to copy contents of STR1 to STR2

MOV SB ; // no argument command, copies the contents of DS:SI into ES:DI
         then it increment or decrements both SI & DI depending upon DF value. DS & ES just represent from where SI & DI taking addresses. only SI & DI are used.

LEA SI, STR1
LEA DI, STR2
CLD
MOV SB
```

→ Here lines are missing. Note we have not set DI & SI

```
⇒ MOV AX, @DATA
  MOV DS, AX
  MOV ES, AX
* LEA SI, STR1
  LEA DI, STR2
  CLD
  MOV SB
  MOV SB
  MOV SB
  MOV SB
  MOV SB
```

5 times

this can be done as.

```
MOV CX, 5
CLD
REP MOV SB
```

REP command // repeats the command CX times.

Say this time we want to copy STR1 in STR2 but in reverse order.

Chongly ass

```
LEA   SI, STR1+4
LEA   DI, STR2
STD                    // to move both SI & DI to left
                       but we want DI to move right
MOV CX, 5
M1: MOV SB
    ADD DI, 2          // Adding 2 sets DI to correct next position
    LOOP M1
```

* MOV SW // Similar to SB. only diff it copies a word in place of byte

REP MOV SW // is valid. Repeats CX number of times.

ARR DW 10, 20, 40, 50, 60, ?

now insert 30 in b/w 20 & 40.

Sir

```
STD
LEA SI, ARR+08H  → 8     By SI points to 60
LEA DI, ARR+0AH  → 10    // DI point to '?'
MOV CX, 3
REP MOV SW
                 // After SI point to 20 & DI points to 40
MOV WORD PTR [DI], 30
    ARR
```

```
Lea SI, ARR+04
Lea DI, ARR+06
```

STOSB

Moves contents of AL into the byte addressed by DI. ES: DI = AL
Then DI is incremented / decremented depending on DF.

Similarly STOSW ES: DI = Ax & then DI is incremented / decremented

eg:- write a procedure to Read characters in a string untill Carriage return is typed. If user makes a typing mistake & hits the backspace key, the perious (last) character is removed.

Sir

```
READ_STR  PROC
; DI = offset address of string
; Bx = no. of characters in final string

PUSH Ax
PUSH DI
CLD
XOR  Bx, Bx
Mov  AH, 1
INT  21H
W1:  CMP  AL, 0DH
     JE   E1
     Cmp  AL, 08H        // 08H → backspace
     JNE  E2
     DEC  DI
     DEC  Bx
     JMP  R1
E2:  STOSB
     INC  Bx
R1:  INT  21H
     JMP  W1
E1:  END the Progr after popping
     POP  DI
     POP  Ax
     RET
RED_STR ENDP
```

DB STR1 ?

```
Proc ( x ,
Mov AH, 1
Lea SI, STR1
L1: Int 21H
    CMP AL, 0DH
    JE E1, CMP AL, 'backspace'
    MOV [SI], AL      JE E2
    ~~Inc SI~~
    Inc SI
E2: DEC SI
    Jmp L1
```

* LODSB → Loads AL = DS: SI, then SI is incremented / decrem depending on DF.

Similarly LODSW → AX = DS: SI then SI is incr/decrem depends on DF

write a procedure to display the contents of a string →

```
DISP_STR  PROC
; SI = offset address of string
; Bx = no. of characters in string
PUSH Ax        PUSH Dx
PUSH Bx        PUSH SI
PUSH Cx
```

```
MOV  Cx, Bx        // Bx contains no. of characters in string.
JCXZ PE1
CLD
MOV AH, 2
T1:  LODSB
     MOV DL, AL
     Int 21H
     LOOP T1
PE1: POP SI
     POP Dx
     POP Cx
     POP Bx
     POP Ax
     RET
DISP_STR ENDP
```

Date
13/9/22

Scan string

SCASB ; // target byte to be scanned must be in AL
it subtracts ES:DI from AL, depending on results flags
DI is either incremented or decremented depending on DF value

SCASW // similarly to above only diff is Ax.

eg ①

```
STR1 DB 'ABC'
CLD
LEA DI, STR1
MOV AL, 'B'
SCASB .        // ZF = 0
SCASB          // ZF = 1
```

B - B

REPNE SCASB → repeats SCASB until it is not equal
it comes out of loop either Cx = 0 or it becomes equal

eg. write a code where set of alphabets (all capital) is read & count the number of vowels and consonants in the string.

Sol Say instead of all capitals we can have any characters. Then we make two arrays one for 5 vowels & other contains all constants. Then for each character, we scan it vowel & consonant arrays using SCASB and increase counters correspondingly.

```
.DATA
STRI    DB  80 dup(0)
VL      DB  'AEIOU'          → 5
CL      DB  'BCD - - - Z'    → 21

OUT1    DB  0DH, 0AH, 'vowels = $'
OUT2    DB  'consonants = $'
  VC  DW 0
  CC  DW 0

.CODE                        // DF should also be DF=0 here (confirm)
MAIN  PROC
    Mov Ax, @Data
    Mov DS, Ax
    Mov ES, Ax
    LEA DI, STR1
    CALL READ_STR
    Mov SI, DI
R1:
    LODSB          // now the first character is in AL
    LEA DI, VL
    Mov Cx, 5
    REPNE SCASB
    JNE R2      → Checks if ZF is 0 or 1.
     INC VC     → if equal then VC++
     JMP R3

R2: LEA DI, CL
    Mov Cx, 21
    REPNE SCASB
    JNE R3
     INC CC
```

R3: DEC BX // BX contains no. of characters in STR1
JNE R1

* Next of implementing linkedlist

CMPSB : It performs ES:DI - DS:SI
↑ Subtracts

and sets the flags like ZF depending of value & then SI & DI are incremented / decremented depending on D;

Similarly CMPSW

eg ① STR1 DB 'ACB'
STR1 DB 'ABC'
Lea SI, STR1
Lea DI, STR2
CMPSB // ZF=1
CMPSB // ZF=0

REPE CMPSB or CMPSW // repeats it for Cx no. of times or till equal.

eg ②
STR1 & STR2 of length 10
We want to put AX = 0 if STR1 = STR2
= 1 if STR1 < STR2
= 2 if STR1 > STR2

Sol

```
MOV CX, 10
LEA SI, STR1
LEA DI, STR2
CLD
REPE CMPSB
JL str1_F
JG str2_F
MOV AX, 0
JMP E1
str1_F:
str2_F:
E1:
```

```
Lea DI STR1
Lea SI STR2
R1: CMPSB
JZ R1
```

Date 10/4/12

REPE / CMPSB

REPZ

Execute till match is found or cx becomes zero.

6) write a code to check if one string is a substring of the other or not.

```
.MODEL SMALL
.STACK 100H
.DATA
MSG1    DB 'Enter substring', 0DH, 0AH, '$'
MSG2    DB 'Enter Main string', 0DH, 0AH, '$'
MAINST  DB 80 Dup(0)
SUBST   DB 80 Dup(0)
STOP    Dw ?                    // stoping ptr
START   Dw ?
SUB_LEN Dw ?                    // length of substring
YESMSG  DB 0DH, 0AH, SUBST, 'is substring $'
NOMSG   DB 0DH, 0AH, 'SUBST is not substring $'
MAIN
    Mov Ax, @Data
    Mov DS, Ax
    Lea SI, MAINST
    Lea DI, SUBST
    Mov Cx, SUB_LEN
M:  CMP [SI], [DI]
    JNE L1
    INC SI
    INC DI
    JMP M  ---> CMP [DI], STOP
                JE E1
L1: Lea DI, SUBST
    JMP M
E1:                             ; Substring Found
```

```
.CODE
MAIN PROC
    Mov AX, @DATA
    Mov DS, AX
    Mov ES, AX
    Mov AH, 9
    LEA DI, MSG1
    INT 21H
    LEA DI, SUBST
    CALL READ_STR          // see in notes the function
    Mov SUB_LEN, BX
    LEA DX, MSG2
    INT 21H
    LEA DI, MAINST
    CALL READ_STR
    OR BX, BX              // to check if length of substr is zero or not
    JE N1
    CMP SUB_LEN, 0 JE N1
    CMP SUB_LEN, BX        // if substring len > main str then also not possible
    JG N1
    LEA SI, SUBST
    LEA DI, MAINST
    CLD
    Mov STOP, DI
    ADD STOP, BX           // now stop points at the last of main string
    Mov CX, SUB_LEN
    SUB STOP, CX           // sub sub_len now, after stop len of main str from stop to end is less than SUB_LEN so, no need to check for them.
    Mov START, DI
R1:
    Mov CX, SUB_LEN
    Mov DI, START
    LEA SI, SUBST
    REPE CMPSB
    JE Y1                  //
    INC START
    Mov AX, START
    CMP AX, STOP
    JNLE N1                // not a substring
```

```
    JMP  R1

Y1: LEA  DX, YESMSG
    JMP D1

N1: LEA DX, NOMSG

D1: MOV AH, 9
    INT 21H
```

MACRO – Block of statements are given a name. That block is called Macro.

Difference b/w Macro and PROC. is that In Macro calling statement is replaced by the body of the macro during assembly itself. In PROC, the control goes to begining of PROC statement.

In Macro we can pass parameters which is not possible in PROC.

Syntax

```
m_name  MACRO  d1, d2, d3 ___ dn
        Statements
        ENDM          // di are dangling parameters.
```

eg:

```
MOVW  MACRO  W1, W2        // normal Mov W1, W2 is not valid
      PUSH W2
      POP  W1
      ENDM
```

To call a macro

→

```
m_name  a1, a2, a3, ____ an   // ai are the actual parameters.
MOVW A, B   } → it will be replaced with these statements during assembling time
            PUSH B
            POP  A
```

If we open the .LST file after assembling, we can see the macro call replaced by macro block statements.

eg ②

```
EXCH   MACRO  W1, W2
       PUSH AX
       MOV  AX, W1
       XCHG AX, W2
       MOV  W1, AX
       POP  AX
       ENDM
```

eg ③

```
       TITLE  PG1 :
       .MODEL SMALL
MOVW   MACRO  W1, W2
       PUSH W2
       POP  W1
       ENDM
.STACK 100H
.DATA
A DW 1, 2
B DW 3
.CODE
MAIN PROC
MOV AX, @Data
MOV DS, AX
MOVW A, DX
MOV W A+2, BX
:
:
```

eg ④ write a macro which takes two word variables & larger of them is placed in AX.

```
FINDBIG MACRO  W1, W2
        MOV AX, W1
        CMP AX, W2
        JG  E1
        MOV AX, W2
E1:     ENDM
```

```
AX = W1
CMP AX, W2
JGE L
MOV AX, W2
L:
ENDM
```

If we call the macro 2 times, it gives an error message while assembling.

```
FINDBIG A1, A2
FINDBIG B1, B2
```

now, body is replaced with macro statements. so code has two labels with same name E1, it throws an error

so, while declaring a label inside a macro, we declare them as LOCAL.

eg:

```
FINDBIG  MACRO  W1, W2
         LOCAL  E1
         MOV AX, W1
         CMP AX, W2
         JG  E1
         MOV AX, W2
E1 :     ENDM
```

// it means each time it is called the assembles gives it a unique number

eg (5)

```
SAVE_REGS  MACRO  R1, R2, R3
           PUSH R1
           PUSH R2
           PUSH R3
           ENDM

COPY_M  MACRO  SO, DES, LN
        SAVE_REGS CX, SI, DI     // e.g. we can call a macro inside a macro as well
        LEA SI, SO
        LEA DI, DES
        CLD
        MOV CX, LN
        REP MOV SB
→       RESTORE_REGS DI, SI, CX  // we can call a macro before defining it.
        ENDM
```

eg (6)

```
DOS_RTN   MACRO
          MOV AH, 4CH
          INT 21H
          ENDM

NEW_LINE  MACRO
          MOV AH, 2
          MOV DL, 0DH
          INT 21H
          MOV DL, 0AH
          INT 21H
          ENDM

DISP_STR  MACRO  S1
          LOCAL S2, MSG
          PUSH AX
          PUSH DX  → PUSH DS
          JMP S2
MSG       DB  S1, '$'
S2:       MOV  AX, CS
          MOV  DS, AX
          LEA  DX, MSG
          INT 21H
          POP DS
          POP DX
          POP AX
          ENDM
```

// every variable or label that we define inside a macro, has to be a LOCAL

REPT

```
REPT  expression
      statements
      ENDM
```

used to repeat a certain block of statements 'expression' number of times.

eg ①

```
BLOCK   MACRO   N
        K = 1
        REPT N
        DW  K
        K = K+1
        ENDM
ENDM
```

```
A DW 1
  DW 2
  DW 3
  DW 100
```

```
A   LABEL WORD
    BLOCK 100
```

IRP

```
IRP   d , <a1, a2, ---- , an>
statements
ENDM
```

d → a_i

// statements are executed N number of times and at each execution d is replaced with a_i

eg ②

```
SAVE_REGS   MACRO   REGS
            IRP  D, <REGS>
            PUSH D
            ENDM
ENDM
```

while calling the macro

```
SAVE_REGS  <Ax, Bx>
SAVE_REGS  <Ax, Bx, Cx>
```

Date 11/10/22

Out hexadecimal in Macro form

```
.MODEL SMALL
SAVE_REGS   MACRO   REGS
    IRP   D, <REGS>
     PUSH D
    ENDM
  ENDM
RESTORE_REGS  MACRO  REGS
      IRP  D, <REGS>
       POP D
      ENDM
    ENDM
```

```
CON_TO_CHAR  MACRO  BYT
    LOCAL E1, E2
    CMP BYT, 9
    JG E1
      OR BYT, 30H
      JMP E2
E1: ADD BYT, 37H
E2: ENDM

DISP_CHAR  MACRO  BYT
    PUSH Ax
    Mov AH, 2
    Mov DL, BYT
    INT 21H
    POP AX
    ENDM

HEX_OUT  MACRO  WRD
    SAVE_REGS  < Bx, Cx, Dx >
    Mov Bx, WRD
    mov CL, 4
    REPT 4                  // assuming at max 4 char in hexadecimal.
    Mov DL, BH
    SHR DL, CL
    CON_TO_CHAR DL
    DISP_CHAR DL
    ROL Bx, CL
    ENDM
  RESTORE_REGS < Dx, Cx, Bx >
  ENDM

. STACK  100H
. CODE
```

repeating them 4 times

```
MAIN PROC
    MOV AX, 1AF4H
    HEX_OUT AX
```

conditional Pseudo-operations -

IF exp (exp ≠ 0) condition is true if exp ≠ 0

IF
ELSE
ENDIF

IFE exp (exp = 0)

IFB <arg> (argument is missing) true condition (-)

IFNB <arg> (argument is not missing)

IFDEF sym (Symbol is defined in code)

IFNDEF sym (Symbol is not defined)

IFIDN <str1>,<str2> (string1 = string2)

IFDIF <str1>, <str2> (str1 ≠ str2)

eg

```
BLOCK MACRO N, K
    I = 1
    REPT N
    IF K+1 - I
    DW I
    I = I+1
    ELSE
    DW 0
    END IF
    END M
ENDM
```

.ERR is a directive. When assembler encounters .ERR, the code execution is terminated there only.

what if we donot pass parameter to macro even though it requires one in definition. We get syntax error.
To stop this happening this is done.

```
IFNB    <BYT>
=
ELSE
.ERR
ENDIF
```

#-

EXTRN namelist → given as comma-separated
we want to use some proc or variables or symbols are not defined in the code but defined somewhere else.

PUBLIC namelist → the place they are defined, we set them public so that they can be accessed from outside / externally by other codes.

eg

PG1.asm

```
EXTRN   CONVERT
:
MAIN PROC
:
CALL CONVERT
:
MAIN ENDP
```

PG1A.asm

```
PUBLIC  CONVERT
:
CONVERT PROC
:
CONVERT ENDP
```

we are not writing any include statements, so, we need to compile/assemble both codes separately and then link them together by '+'. Then execute the main file which calls

C:> MASM PG1;

C:> MASM PG1A;

C:> LINK PG1 + PG1A;

C:> PG1

Segment directive

we can create our own segments and then treat in place of any other usual default segments (code, data, stack or extra segment)

Syntax

```
S-name  SEGMENT  allign  combine
        ; body
S-name  ENDS
```

combine → PUBLIC, COMMON, STACK

allign → PARA, BYTE, WORD, PAGE

way of adding / aligning two segments. If PARA two segments are combined after a gap of PARA.

```
ASSUME CS : C_SEG
       DS : D_SEG
```

this way we can tag our own segment with existing segments.

eg ①

```
D_SEG   SEGMENT
  MSG   DB 'Enter a lower case Letter $'
D_SEG   ENDS
C_SEG   SEGMENT
  :
  :
C_SEG   ENDS
```

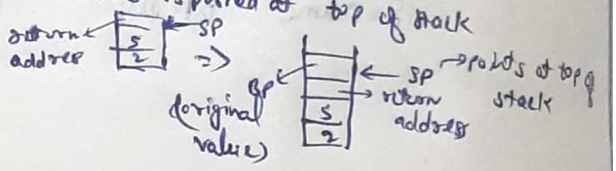
Application / uses of defining our segment

Date
17/10/22

Recursion :

```
.MODEL SMALL
.STACK 100H
.DATA
W1 DW 2
W2 DW 5
.CODE
MAIN PROC
MOV AX, @DATA
MOV DS, AX
PUSH W1
PUSH W2
CALL ADDWD  // when function is called return address is pushed at top of stack
MAIN ENDP
```

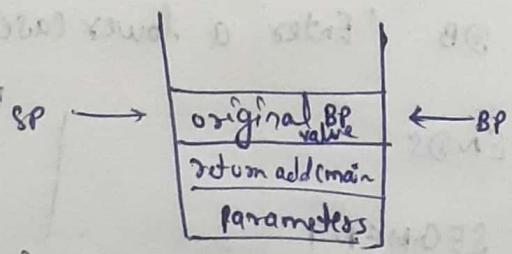
```
ADDWD PROC
PUSH BP
MOV BP, SP      // so that sp is never disturbed and top of stack is not disturbed
MOV AX, [BP+6]  // will give 3rd element as 32-bit else would have been access [BP+3] to
ADD AX, [BP+4]
POP BP
RET 4           // return address is poped off and addition 1 words also so stack is now empty
ADDWD ENDP
END MAIN
```



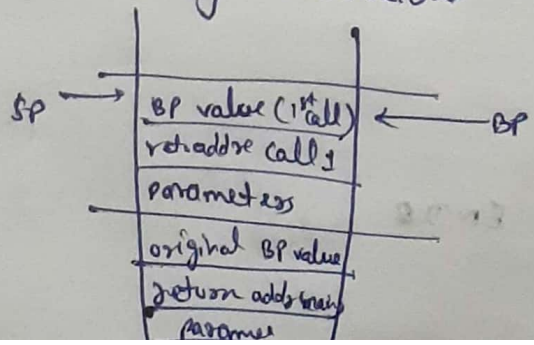
note

RET 4 → means it poped off 4 more additional byte (16 bit) off the stack so total 2 32-bit words. so stack becomes empty

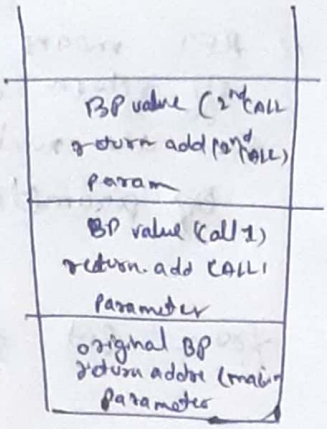
Now During recursion ~~during~~ After the first call from main stack looks like



when called for 2nd time through recursion



After 3rd call



while returning we must ~~take~~ make sure that while returning from 3rd CALL, stack transfers to that of 2nd CALL and so on. So, we need to pop parameters also.

```
FACT  (input : N,  output : result)
    If N = 1
      then
        result = 1
    else
        CALL  FACT (input : N-1 , output : result)
        Result = N * result
    end if
    Return
```

eg

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AX, 3
    PUSH AX
    CALL FACT
MAIN ENDP
```

```
FACT  PROC
  PUSH  BP
  MOV BP, SP
  CMP WORD PTR  [BP+4], 1
  JG E1
  MOV AX, 1
  JMP R1
E1:
  MOV CX, [BP+4]
  DEC CX
  PUSH CX
  CALL FACT
  MUL WORD PTR  [BP+4]
R1: RET 2
FACT ENDP
```

// Also push Cx before using using it. Just take care while returning

```
R1:  POP  BP
     RET  2
     FACT ENDP
     END MAIN
```

// RET means visualize as return add from stack. then the number 2 is for popping of parameters.

eg ✓ Find largest element from set of numbers using recursion.

mine

```
MAX PROC
    PUSH BP
    MOV BP, SP
    CMP WORD PTR [BP+4], 1
    JEQ E1
    MOV AX, [SI]
    JMP R1
E1: MOV CX, [BP+4]
    DEC CX
    PUSH CX
    CALL MAX
    CMP [SI+BP+4-1], AX
    JLE R1
    MOV AX [SI+BP+4-1]
R1: POP BP
    RET 2
    END MAX
```

Lea SI, array

A[BP+4]

A[BP

sir

```
.MODEL SMALL
.STACK 100H
.DATA
A DW 10, 50, 20, 4
.CODE
MAIN PROC
MOV AX, @DATA
MOV DS, AX
MOV AX, 4
PUSH AX
FIND_M MAINENDP
```

$^nC_k = {}^{n-1}C_{k-1} + {}^{n-1}C_k$

```
FIND_M  PROC
        PUSH BP
        MOV  BP, SP
        CMP  WORD PTR [BP+4], 1
        JGE  E1
        MOV  AX, A[0]   or [A]
        JMP  E2
E1:     MOV  CX, [BP+4]
        DEC  CX
        PUSH CX
        CALL FIND_M
        MOV  BX, [BP+4]
        SHL  BX, 1          // BX = N
        SUB  BX, 2          // BX = 2N
                            // BX = 2N-2 = 2*(N-1)
        CMP  A[BX], AX
        JLE  E2
        MOV  AX, A[BX]
E2:     POP  BP
        RET  2
FIND_M  ENDP
END MAIN
```

eg: find $^{n.m}$

$^nC_k = {}^{n-1}C_k + {}^{n-1}C_{k-1}$ $\quad n>k>0$

$C(n,k) = C(n-1,k) + C(n-1,k-1)$ $\quad n>k>0$

Base case $C(n,n) = C(n,0) = 1$

[Comm. checking]

Advanced Arithmatic

1) ADC destination, source

Performs

dest = dest + source + CF

2) SBB dest, src

dest = dest - src - CF

3) A+2 : A negate it

```
NOT  A+2    // 1's complement of A+2
NOT  A      // 1's complement of A
INC  A
ADC  A+2, 0 // if A has carry that goes to A+2.
```

4) perform (A+2 : A) - (B+2 : B)

```
Mov  Ax, A
Mov  Dx, A+2
SUB  B, Ax
SBB  B+2, Dx
```

5) A+2 : A

```
SHL A, 1
RCL A+2, 1
```

} Shifts A+2 : A towards left

Binary Coded decimal

0000 to 1001 1010 - 1111 invalid

eg: 1001 0001 0011

9 1 3 = 913

Packed BCD format — A Byte stores 2 decimal digits

In Unpacked format — one Byte stores only 1 digit.

eg 59 = 3B H

Binary = 0011 1011

Packed BCD 59 : 0101 1001

Unpacked BCD 59 : 00000101 00001001

Advantage of Packed over unpacked, it allows arithmatic of higher number 18-digits in 8087.

AAA ; AL → operand

AAS ; for subtraction

AAM ; for multiplication { MUL BL, AAM

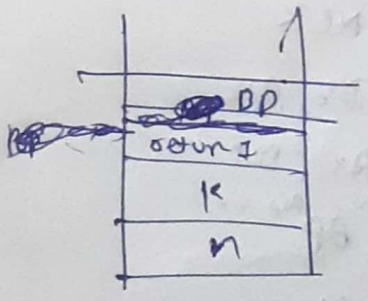
AAD ; for division

{ AAD, DIV BL, AAM

H.W

C(n,k) = C(n-01, k) + C(n-1, k-1)

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
Mov AX, 5      ; n
Mov Bx, 3      ; k
PUSH Ax
PUSH Dx
CALL NCK
MAIN ENDP
```



Date 18/10/22

4.30 to 6.30 tommorow show code

8087 processors

Data types

Word integer (16 bits) 2bytes
short " (32 bits) 4bytes
Long " (64 bits) 8bytes
10 byte packed BCD format → 1 byte for sign
remaining 9 bytes can store total 18 digits.

short real 8E (exponent), 24M (mantissa) (4 Data bytes)
Long real 11E, 53M (8 Data bytes)
Temporary real 15E, 64M (10 Data bytes)

8087 → has Eight 80 bit data registers along with those in 8086.

we can access those 8 registers like a stack.

ST(0), ST(1), ... ST(i) → to access (i+1)th register.

Load & Store

Load → loads src at the top of the stack.

FLD source → loads real
FILD " → loads integer
FBLD " → loads packed BCD

} use depending on the data type to be loaded.

Once, some data type element is loaded to stack, it can be assigned back to any memory location (even if they do not have same data type). We can achieve data type conversion

FST destination → stores real no.

FIST " → stor integers

FSTP " → store real no. and also pop it off

FISTP " → stores integes & pop off

FBSTP " → stores BCD & pops it off

eg: FILD NUM // stored as integes

FSTP NUM // restoring back as real number

now

FADD → Adding real numbers

FIADD → Adding integer numbers

FSUB → Subtract real no.

FISUB → " integer no.

FMUL → mul real no.

FIMUL → " integer

FDIV → div real no.

FIDIV → " integer no.

They may have 0,1,2 operands. If no operands, it takes at 2 elements from 8087 stack & perform addition. If 1 operand then it takes 1 from stacks & adds them.

For 2 it simply adds them.

eg:

FLD NUM1

FADD NUM2 // it adds num1 & num2 and stores the result at the top of 8087 stack.

FSTP NUM3

↳ now the sum is stored (which was at top of stack) & stored in NUM3 as real number.

eg Read a multi-precision real number. Assume input no. is at most 18 digits.

```
READ_INTEGER  PROC
; input : Bx = address of 10 byte buffer of 0's (initially)
        XOR  BP, BP        // BP=0, to count the no. of character already read.
        MOV  SI, BX
        MOV  AH, 1
        INT 21H
        CMP  AL, '-'       // if negative input is given, then store 80H at the MSB.
        JNE  R1
        MOV  BYTE PTR [Bx+9], 80H
        INT21H             // read next char   // [Bx+9] address of MSB
R1:     CMP  AL, 0DH       // if Carriage return
        JE  R2
        AND  AL, 0FH       // converting ascii to digit (binary value)
        INC  BP            // since 1 digit is read
        PUSH  AX           // 8086 stack pushing, necessary as we are reading in left to right but BCD is given right to left digit conversion.
        MOV  AH, 1
        INT  21H
        JMP  R1
R2:     MOV  CL, 4
R3:     POP  AX
        MOV  [Bx], AL      → [Bx] = 00000101 , Bx points to LSB
        DEC  BP                                 01010000
        JE  R4
        POP  AX
        SHL  AL, CL        → AL = 4 = 00000100
        OR  [Bx], AL       → AL = 01000000
        DEC BP             → 01000101
        JG R3
INC Bx ←
```

eg. say input is −12345 → dry run explained above. for first 2 & 4

```
R4:     FBLD  TBYTE PTR [SI]   → for 10 bytes → loads those entire 10 byte data to stack.
        FSTP  TBYTE PTR [SI]   ; store packed BCD at top of 8087 stack.
        RET
READ_INTEGER  ENDP
```

Date 25/10/22

1-35 at 3:30 mid sem paper tommorow

Arithmatic operation Addition/ sub/ multiplication/ Div.

```
TITLE   PG1 :
.MODEL SMALL
.8087
                        // works as header file for 8087
.STACK 100H
.DATA
  NUM1  DT  0
  NUM2  DT  0           // DT = 10 bytes
  SUM   DT  ?
  DIFF  DT  ?
  PROD  DT  ?
  QUO   DT  ?
  CR  EQU  0DH
  LF  EQU  0AH
NEW_LINE MACRO
   Mov DL, CR
   Mov AH, 2
   Int 21H
   Mov DL, LF
   INT 21H
END M

DISPLAY  MACRO  x
  Mov  DL, x
  Mov  AH, 2
  INT  21H
END M
```

prev code will be included in it

```
.CODE
INCLUDE PGM1A.asm      ;; it contains Proc to read the multi-prec
INCLUDE PGM1B.asm      ;; it contains Proc to print the multi-precision

MAIN PROC
Mov Ax, @Data
Mov DS, Ax
Mov ES, Ax
DISPLAY , '?'
Lea Bx, NUM1
CALL READ_INTEGER
NEW_LINE
DISPLAY '?'
Lea Bx, NUM2
CALL READ_INTEGER
NEW_LINE
FLD NUM1      } 2 no. are at top of 8087 stack
FLD NUM2      }
FADD          -> no parameter so it adds the two values at top of 8087 stack, stores back at top of stack
FSTP SUM      -> FWAIT   ;; to add synchronization. the result
Lea Bx, SUM               control waits till 8087 complete its all execution & then control goes to 8086. Add it always. if we need to shift control from 8087 to 8086.
CALL PRINT_BCD
```

Procedure to read a Real number / float number

```
READ_FLOAT  PROC
; input : Bx = add. of 16 byte buffer
XOR DX, DX      ; DH = 1 for decimal point
XOR BP, BP      ; counting total no. of digits.
MOV SI, Bx
R1:
MOV AH, 1
INT 21H
CMP AL, '-'
```

```
        JNE   R2
        Mov   Byte Ptr  [Bx+9], 80H      // storing 80H to MSB if negative
        Jmp   R1
R2:     Cmp   AL, '.'                    // checking for decimal point
        JNE   R3
        INC   DH                         // to indicate we have encountered a decimal.
        Jmp   R1
R3:     Cmp   AL, 0DH                    // checking for carriage return
        JE    R4
        AND   AL, 0FH                    // converting to ascii value
        INC   BP                         // BP counts no. of digits.
        PUSH  Ax
        Cmp   DH, 0
        JE    R1
        INC   DL                         // DL for counting no. of digits after decimal.
        Jmp   R1
R4:     Mov   CL, 4
        Pop   Ax.
        Mov   [Bx], AL
        DEC   BP
        JE    R5
        Pop   Ax
        SHL   AL, CL
        OR    [Bx], AL
        INC   Bx
        DEC   Bp
        JG    R4
R5:     FBLD  TBYTE   PTR [SI]
        FWAIT
        Cmp   DL, 0
        JE    R6
        XOR   Cx, Cx
        Mov   CL, DL
        Mov   Ax, 1
```

```
R7 :  IMUL   BX
      LOOP   R7          // say DL = 5 then AX will have 105
      MOV [SI], AX
      FIDI   WORD PTR [SI]

R6 :
      FSTP   TBYTE  PTR [SI]
      FWAIT
      RET

READ_FLOAT  ENDP
```

n.w write the ~~print~~ for real number.

```
PRINT_FLOAT  PROC
      ; input  Bx = address of buffer

      MOV  WORD PTR[BX], 10000
      FIMUL  WORD PTR [BX]        ; scale up by 10000
      FBSTP  TBYTE PTR [BX]       ; store as BCD
      FWAIT
      TEST  BYTE PTR [BX+9], 80H
       JE   R1
       MOV  DL, '-'
       MOV  AH, 2
        INT 21H

R1 :  ADD   BX, 8
      MOV   CH, 7
      MOV   CL, 4
      MOV   DH, 2

R2 :
      MOV   DL, [BX]
      SHR   DL, CL
      OR    DL, 30H
      INT 21H
      MOV   DL, [BX]
       AND  DL, 0FH
      OR    DL, 30H
```

```
INT 21H
DEC BX
DEC CH
JG  R2
DEC DH
JE  R3
DISPLAY '.'
MOV CH,2
JMP R2
R3 : RET
PRINT_FLOAT ENDP
```

Date
31/10/22

Intel Instruction Format : (Machine coding)

Prefix : 0-4 bytes
opcode : 1-2 bytes → mandatory all others are optional
Mod R/M : 1 bytes
SIB : 1 bytes
Displacement : 1 byte or word
Immediate : 1 byte or word

Prefix : it may modify instruction behaviour. It can

Opcode : tells the processor the instruction to be executed. It searches the opcode in an table (hash table) then match the syntax to find which instruction it matches with.

eg:

NOT → 1111101 w
OR → 000010dw

w = 1 if word operation
w = 0 for byte operation

d → determines the src & destination among given parameters.

Date
31/10/22

Intel Instruction Format : (Machine coding)

Prefix : 0-4 bytes
opcode : 1-2 bytes → mandatory all others are optional
Mod R/M : 1 bytes
SIB : 1 bytes
Displacement : 1 byte or word
Immediate : 1 byte or word

Prefix : it may modify instruction behaviour. It can

Opcode : tells the processor the instruction to be executed. It searches the opcode in an table (hash table) then match the syntax to find which instruction it matches with.

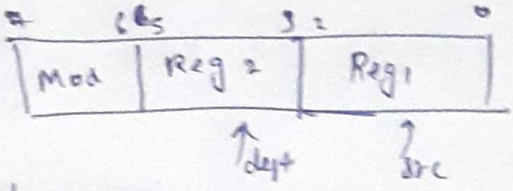
eg:

NOT → 1111101 w
OR → 000010dw

w = 1 if word operation
w = 0 for byte operation

d → determines the src & destination among given parameters.

Mod R/M will talk about the operands on which instruction will be performed.



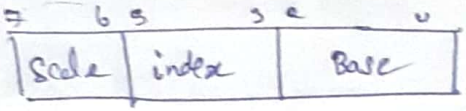
Each register has a code & Reg1, Reg2 are those codes.

eg: Mod

00 → operand's memory address in Reg1

SIB when Mod contains correct Mod value

[Scale * Index + Base]



SIB = index $\times 2^{scale}$ + base

Displacement :

mod → 01 or 10 can be for a byte or a word.

when mod is 01, displacement comes into picture

Immediate → whenever we want to use some constants, they are treated as immediate.

Assembly Process

Assembly source code → Assembler → Machine code

① assigns addresses

② generate machine code for the instructions.

eg: Jmp L1 — it will not be able to generate machine code of this line as without knowing the address of L1, we can not know where to jump & start executing.

L1:

so, this assembling process can not be done in a single pass.
So, After pass one, A table will be created and each variable or label will be assigned an address along with converting to machine code of those instructions if possible.
In pass 2, the remaining instructions are converted to machine code.

eg: ADD Ax, 24

for Add

add reg, r/m 03/r

add r/m, reg. 01/r

add r/m, immediate 81/0 id.

so) ADD Ax, 24 → 81 / 0 id → constant / immediate

81 → operation code; 0 → mode value

7 6	5 4 3	2 1 0
mod	Reg / opcode	r/m
11	0 1 0	0 0 0

81C0 18000000

18000000 → 24 immediate

directive command

eg:

```
.data
a1  dd 4
a2  dd ?
a3  dd  5 dup(0)
.code
main :
    mov  cx, 20
    mov  ax, 15
    mov dx, 0
    jz  target_label
Loop1:
    add dx, ax
```

dd → define 4 bytes

```
imul   [BP+8]
dec CX
jg Loop1
target_label:
end
```

There will be 2 things – memory map of code segment & memory map of data segment.

Symbol table

Symbol	address
a_1	0040 0000
a_2	0040 0004
a_3	0040 0008
	0040 000C
	0040 0010
	0040 0014
	0040 0018
main	0000 0000 → main
Loop1	0000 0018 → Loop
target label	0000 0024 → target label

(a_3 array)

main is also a symbol.

memory map of data segment

	address	value / content
a_1 →	0040 0000	0000 0004 (4 is stored)
a_2 →	0040 0004	0000 0000 (if nothing is initialised it assigns zero)
a_3 →	0040 0008	0000 0000
	0040 000C	0000 0000
	0040 0010	0000 0000
	0040 0014	0000 0000
	0040 0018	0000 0000

now for the instructions.

b8+rd

i) mov cx, 20
mov reg, immediate
→ machine code: b9 14 000000
20

ii) mov ax, 15
b8 15 00 00 00

iii) mov dx, 0
ba 00 00 00 00

iv) JZ target_label
of 84 "cd" → 32 bit offset label

v) add dx, ax
03/r → 11010 000 → 03d0

vi) imul [bp+8] → f7/5 → f7ad 08000000

Memory map for code segment

address	Content (→ 5 byte data)
0000 0000	b9 14 00 0000
0000 0005	b8 ff0000 000
0000 000a	ba 0000 000
0000 000f	0f 84 ?? ?? ?? ??
0000 0015	03 d0
0000 0017	f7 ad 08 000 000
0000 001d	49
0000 001e	0f 8f 0000 0015
0000 0024	

vi) Dec cx → 49

vii) 0f 8f "cd" → address of Loop1 from symbol table

Pass 1 is over

now job of pass is to resolve the issues which could not be resolved after pass 1.

eg. replacing "??" to next address of 0000 0024.

Job of assembler is done.

Both passes are executed always.

If we have external variables / procedures to link the address of those variables with this file, we need linker.

All these are relative addresses. So, the loader upon getting starting address during execution, fixes all these addresses for execution.

Date 1/11/22

Data Structures

i) Location counter (LC): points to the next location where the next code is present.

ii) OP. code translation table → contains instruction code,

iii) Symbol table (ST) → ~~string~~

iv) String storage Buffer (SSB) → Contains ascii characters of each string (string name).

v) Forward Reference table (FRT) → contains pointer to the string in SSB and offset address where the value will be inserted in the object code

~~Ex~~ This is some other Assembly code not (8086)

eg

```
        START
        LDA #0          0000    01
                        0101    00
                        0102    00
                        0103    05
        LDX #0          0104    00
                        0105    00
                        0106    18
Loop :  ADD LIST, X     0107    01   ] pass2
                        0108    19   ]
                        0109    2C
        TIX COUNT       010A    01   ] pass2
                        010B    15   ]
                        010C    38
        JLT Loop        010D    01   ] pass1
```

R SUB	010E	06] RSUB
	010F	4C
	0110	00
	0111	00
LIST : WORD 200	0112	00
	0113	02
	0114	00
	0115	00
COUNT : WORD 6	0116	00
	0117	06

END

Symbol Table

symbol	address
i) LOOP	0106
ii) LIST	0112
iii) COUNT	0115

SSB

DC00	40H	ASCII code of L, I, S, T
DC01	49H	
DC02	53H	
DC03	54H	
Dc 04	55H	Seperation, character

FRT

Offset	SSB pointer
0107	DC00
010A	DC05

when loaded in main memory, we will have to change all these addresses depending upon the starting address. All these are relative to starting address 0000 00. So, another table is required called direct address table.

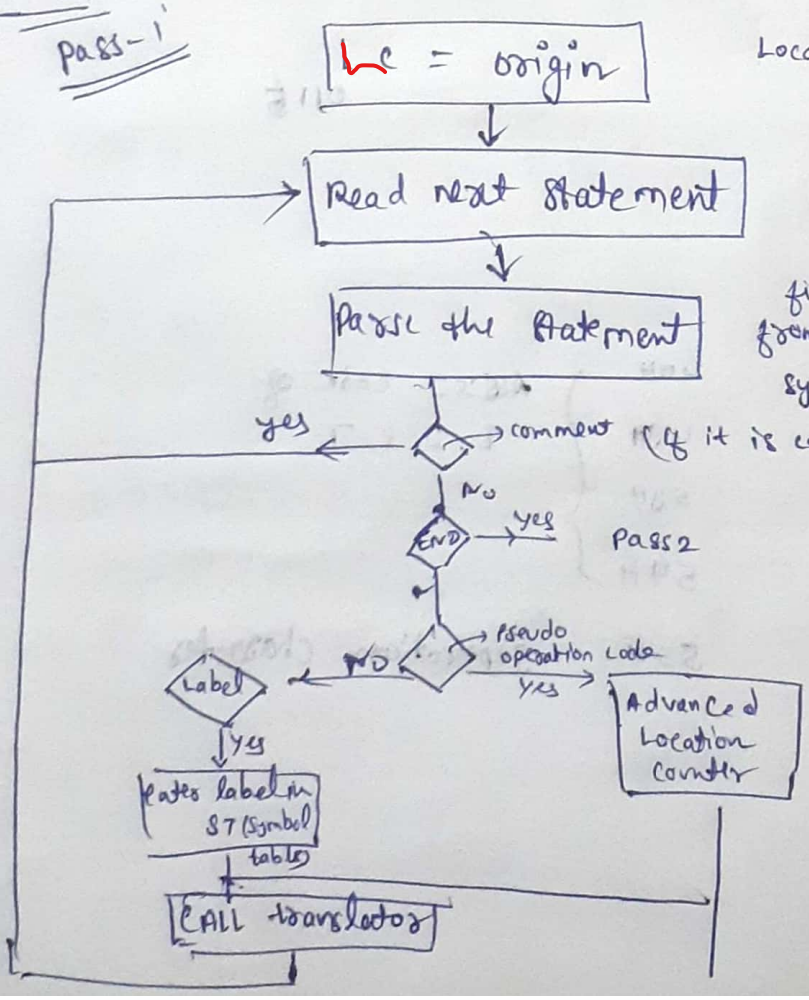
DAT

0107
010A
010D

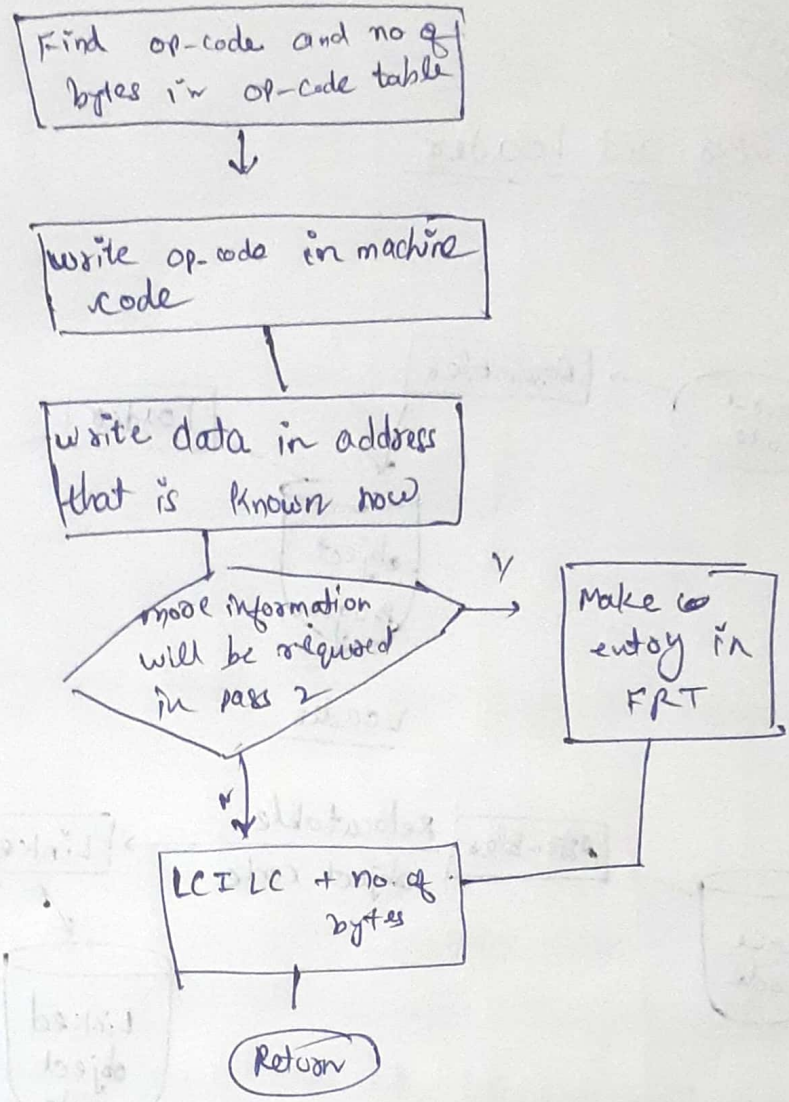
contains addresses which needs to be relocated. Created at the end of pass 2.

Overview

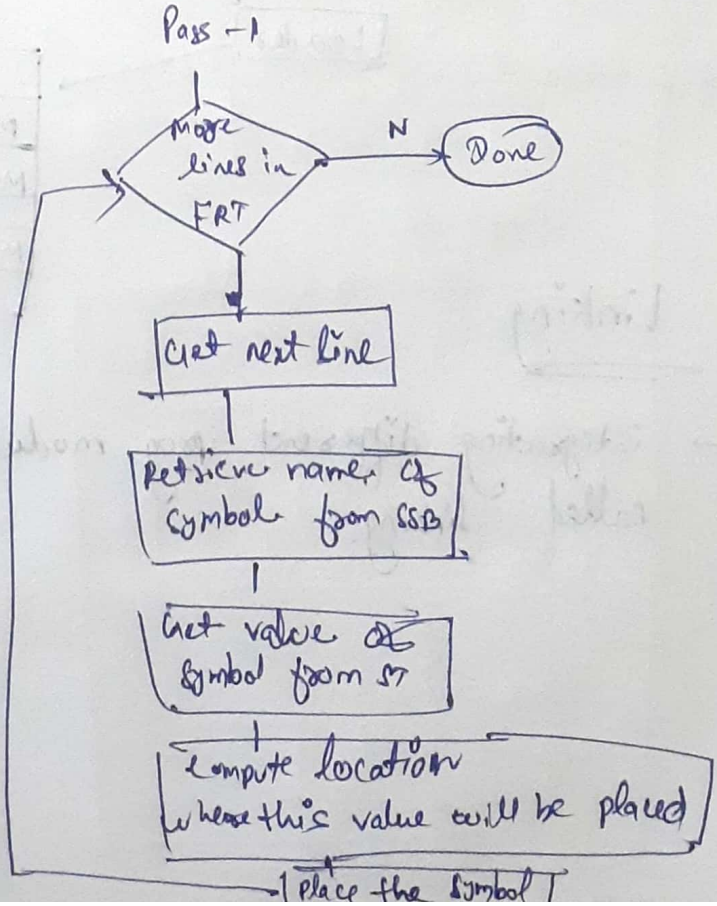
Pass-1



Translator

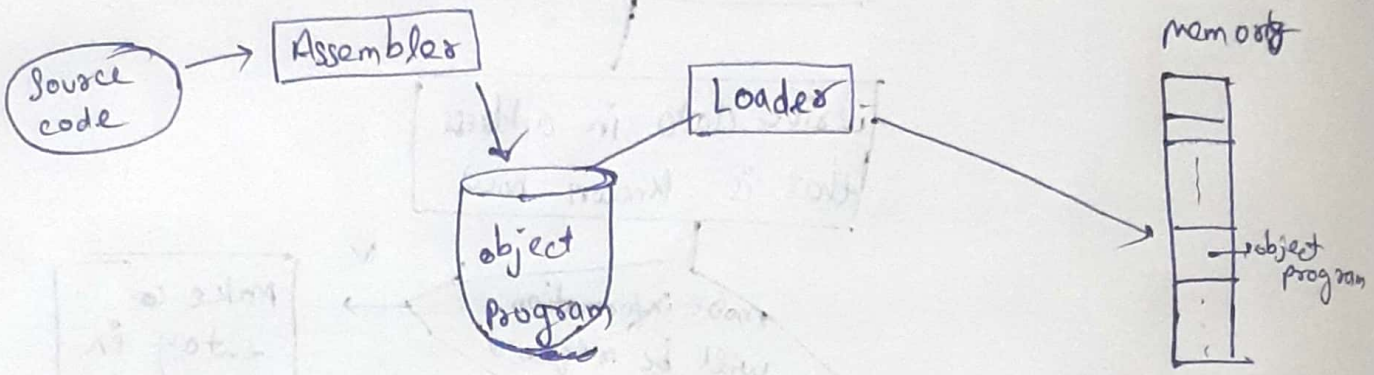


Pass-2

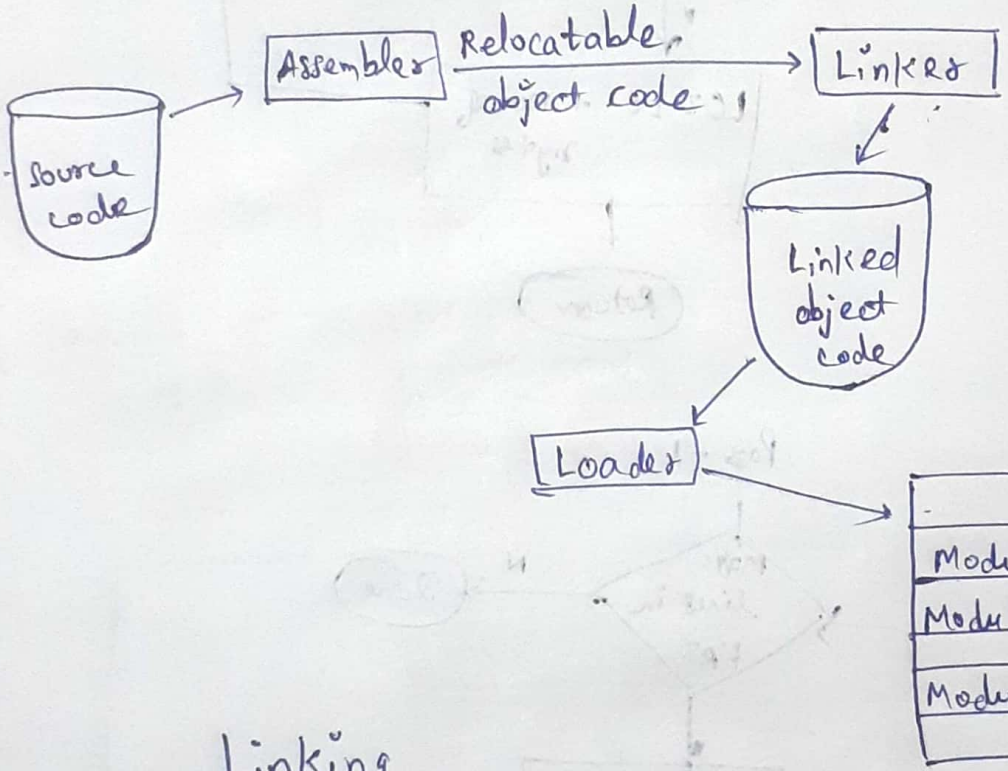


Date
7/11/22

Linker and Loader

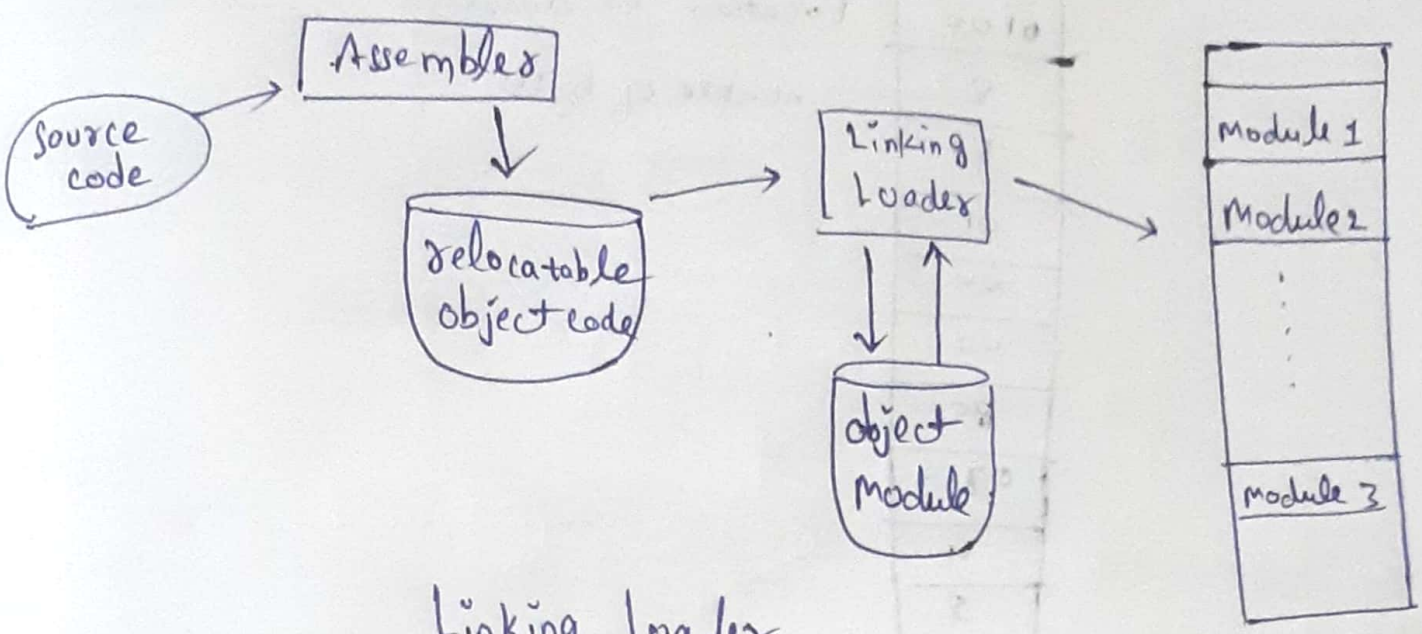


Loader



Linking

Linking → integrating different prog modules to execute is called linking.



Linking Loader

Here, the job of linking and loading by same software Linking loader.

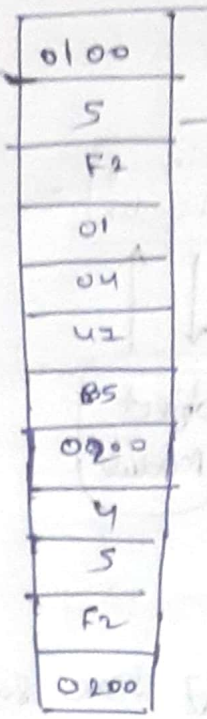
3 types of loaders

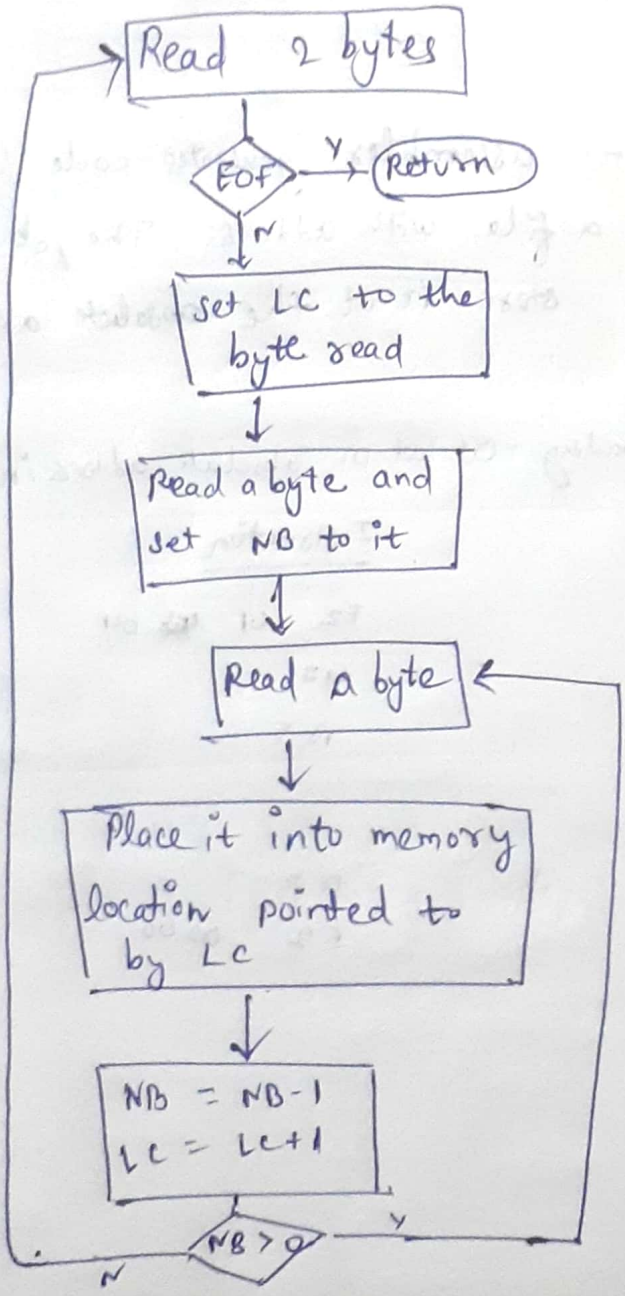
1) Absolute loaders → assembler generates code & store the instruction in a file with address. The job of loader is to read the file and store it at the absolute address given in the file.

eg: ① Booting → loading OS at an absolute address in the memory.

②

Address	Instruction
0100	F2 01 04
0103	47
0104	B5
⋮	⋮
0200	05
0201	F2 0200



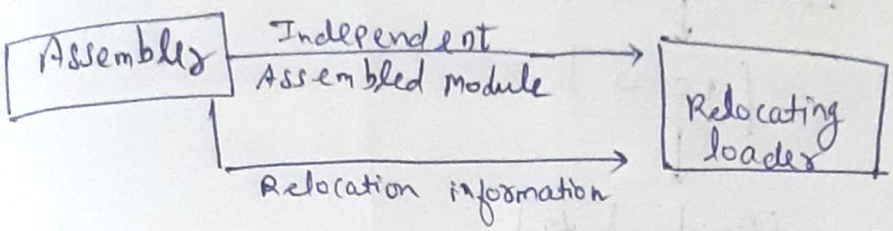


LC : location counter

NB : no. of bytes

Absolute loader

2) Relocating Loaders :



Adress	Instruction
0000	3A 00 , 04
0002	47
0004	B5
0000	05
0001	F2 0000

DAT → direct address table

DAT

0001

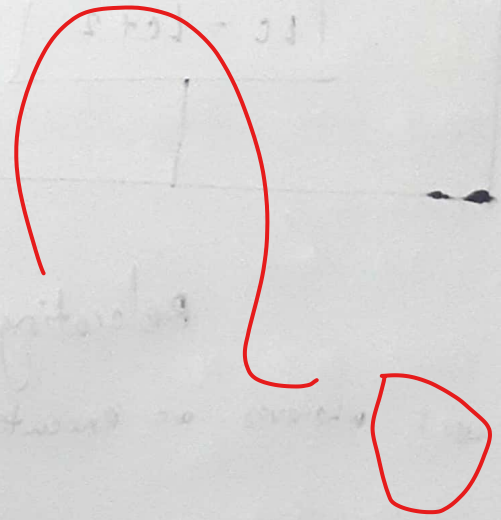
DAT

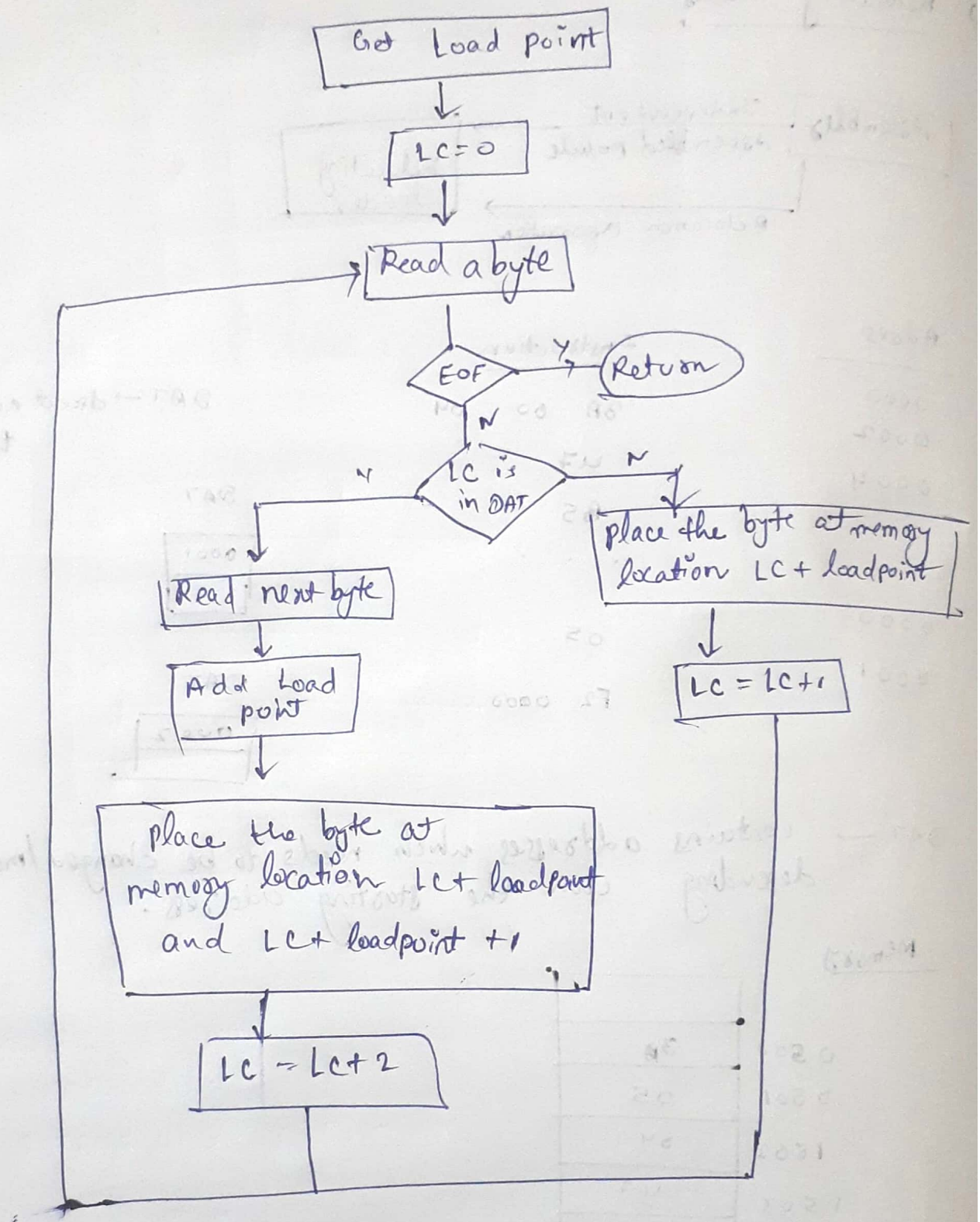
0002

DAT → contains addresses which needs to be changed/modified depending upon the starting address.

Memory

0500	3A
0501	05
1502	04
1503	47
1504	05
0700	05
0701	F2
0702	07
0703	00





Relocating loader

eg: it used whenever we execute our assembled code

(3) **Linking Loader** : used when compiling high level codes c++, java

It has 4 function

(a) Allocation → allocates space in main memory for execution

(b) linking → resolving external references b/w the modules like proc (external) or including other files.

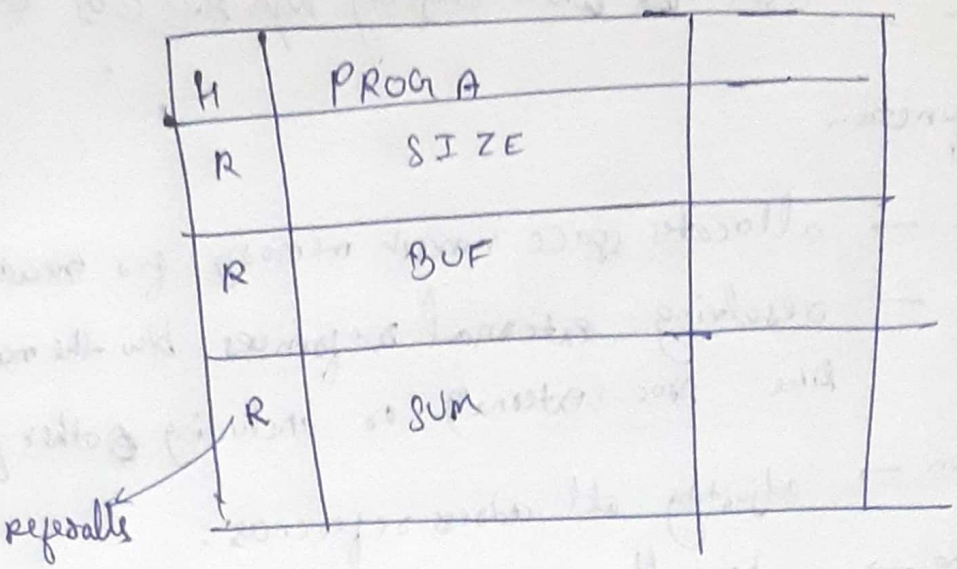
(c) Relocation → adjusting all adress references.

(d) Loading → physically putting the machine code in main memory for execution.

eg

```
PROG A : START
         EXTRN   SIZE, BUF, SUM
         LDA #128
         STA #SIZE
         LDA #1
         LDX #0
L1 :     STA BUF, X
         ADD #1
         TIX SIZE
         JEQ L2
         J L1
L2 :     JSUB SUM
         RSUB
         END
```

```
0000  29  01  28
0003  0C  -   -
0006  29  00  01
0009  05  00  00
000C  0F  -   -
000F  19  00  01
0012  2C
0015  30  00  0B
0018  3100  0C
001B  48  -   -
```



DAT	0016	
DAT	0019	
M	SIZE	0804
M	BUF	0000
M	SIZE	0013
M	SUM	001B

modification required

Dat defined for only external records which is M records

```
PROGB :  START
         LOCAL  SUM
         EXTRN  Size, BUF, TOT

SUM :    LDA  #0          0000   29 00 00
         LDX  #0

         ADD  BUF, X      0003   05 00 01
L3 :     TIX  SIZE        0006   1B — — —
         JEQ  L4          0009   2C — —
         J L3             000C   3C 0012
                          001F   3C 0006
```

L4: STA TOA 1012 0C - -

RSUB

END

H	PROGB	
D	SUM	0000
R	SIZE	
R	BUF	
R	TOT	

DAT	006D	
DAT	0010	
M	BUF	0009
M	SIZE	000A
M	TOT	0013

PROGC : START

LOCAL SIZE, BUF, TOT.

SIZE : RESW 1 0010

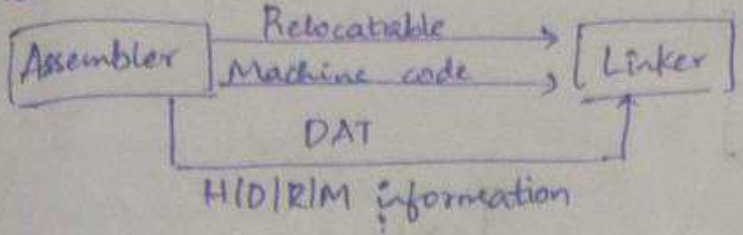
TOT : RESW 1 0013

BUF : RESW 100 0006

END

H	PROGC	
D	SIZE	0000
D	TOT	0006
D	BUF	0006

14/10/22



Pass 1:

* Get load point of all programs
* It will create table ESTAB D/R. Calculate absolute address as load point + relative address

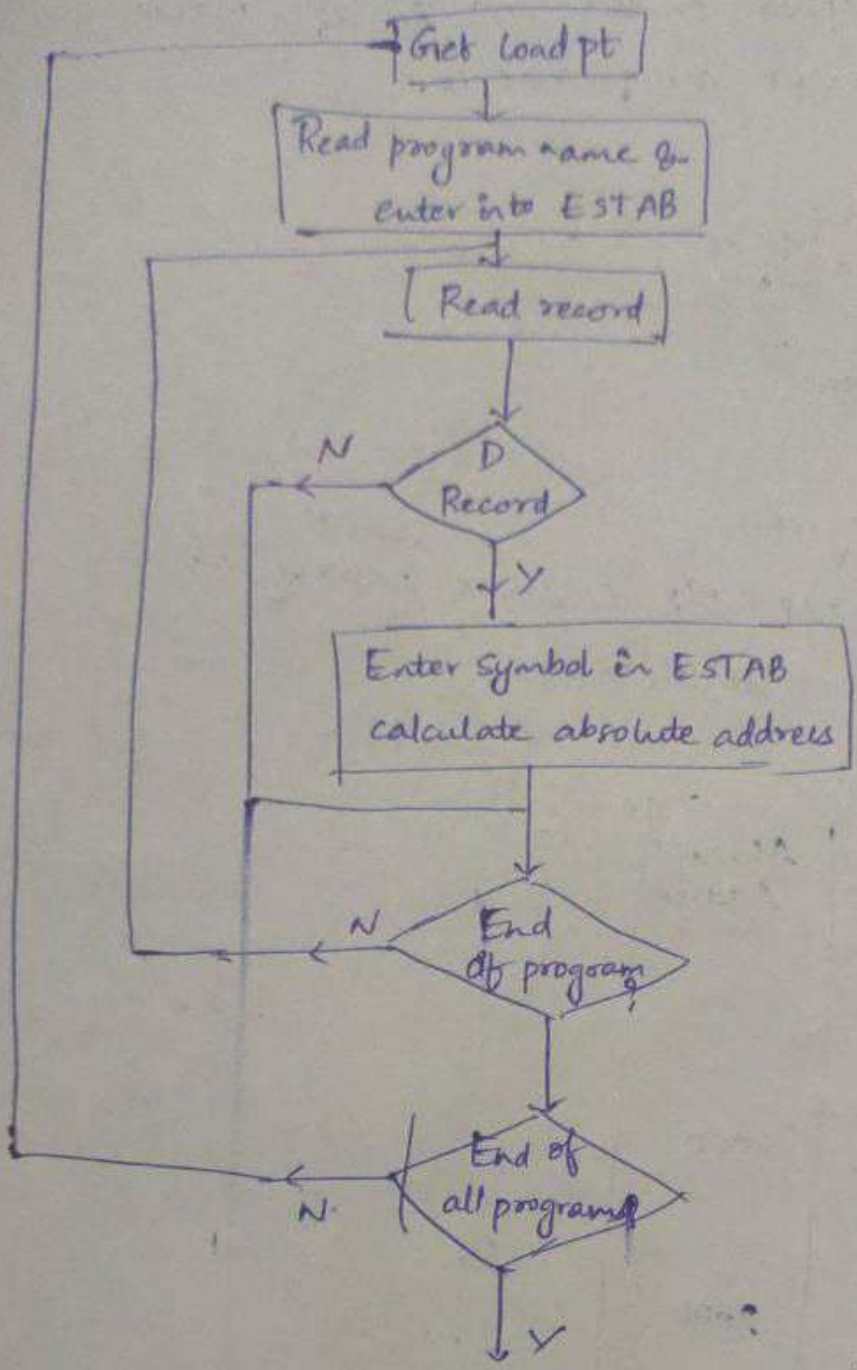
ESTAB

Program	Load Pt	Label	Absolute Address
PROGA	1000		
PROGB	2000		
		SUM	2000
PROGC	3000		
		SIZE	3000
		TOT	3003
		BUF	3006

Pass 2:

* M informations
* DAT information

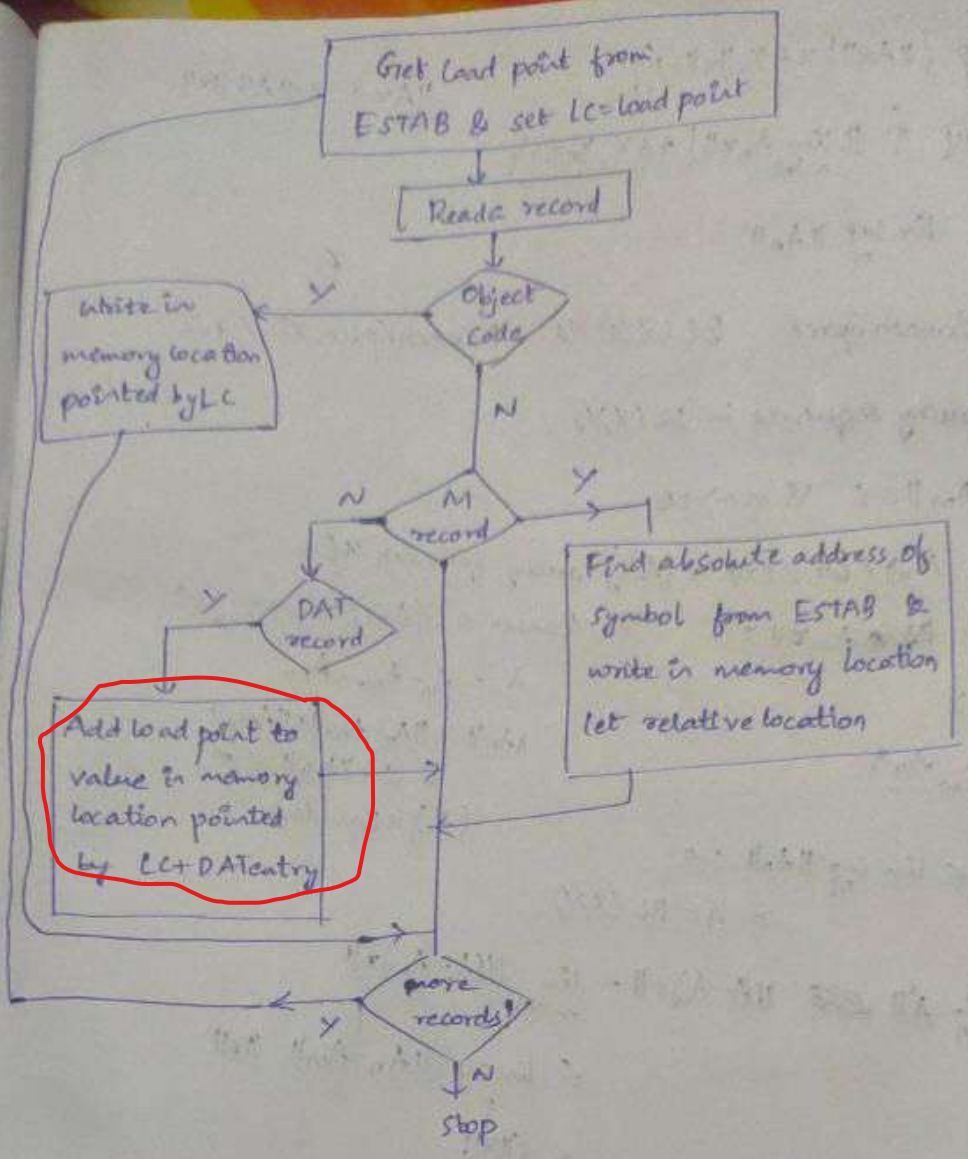
H: Header file (Program name)
D: definition (local vars)
M: modify (external vars in dat table)
R: referral (external vars)



Pass 1:

D record: local record (defined in this file to be used by others)

R record: extern record(defined in other file)



Dynamic linking:

In Dynamic linking a subroutine is loaded and linked to other programs when it is first called during execution.

Dynamic linking allows several executive programs to share the same copy of a subroutine. Dynamic linking provides the ability to load the routines only when they are needed. Dynamic linking also avoids the necessity of loading entire library for each execution. This is true for the case of Polymorphism.