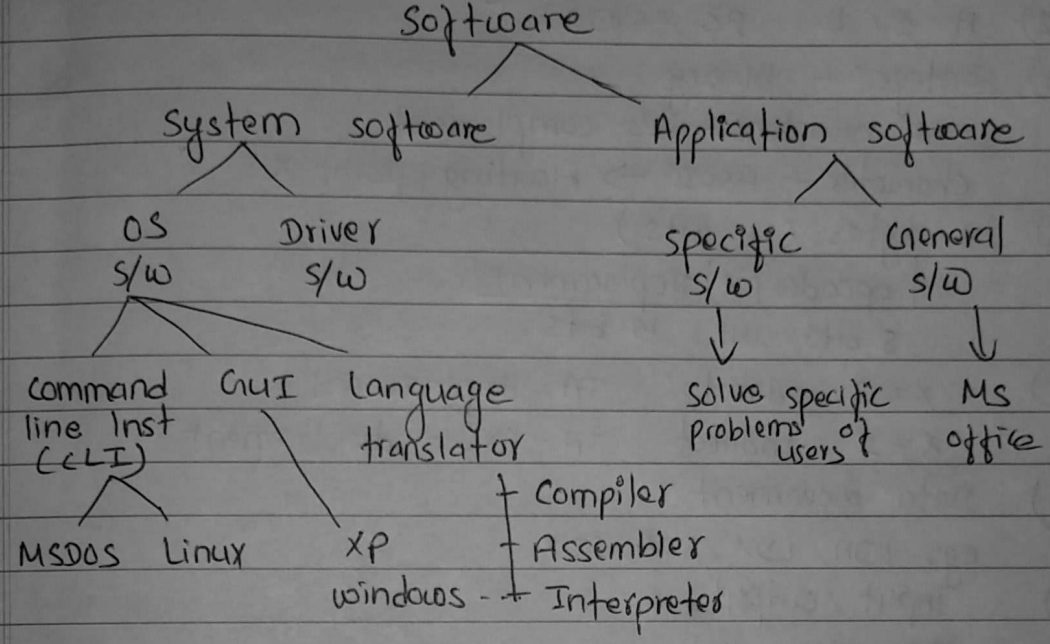
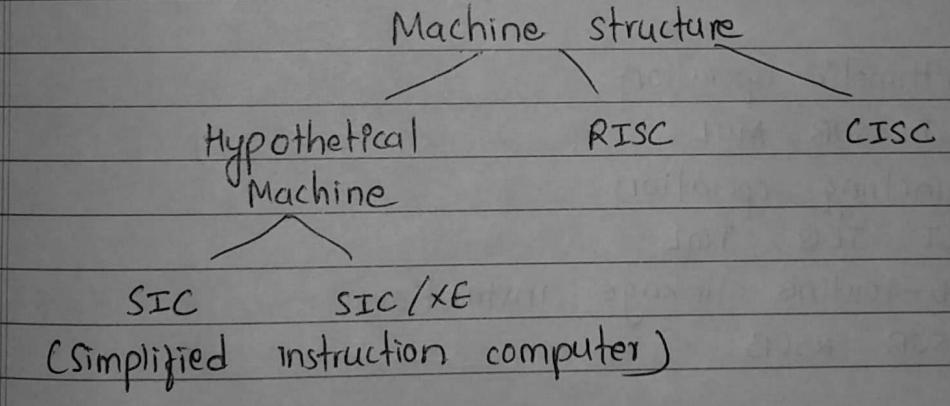
System Programming





SIC Machine structure:

1) Memory
2) Registers
3) Data formats
4) Instruction formats
5) Addressing modes
6) Instruction set
7) Input/output

Desc structures:

1) 8 bits = a byte, 3 byte = 1 word
2) A, X, L, PC, SW
3) Integer - binary
   -ve number - 2's complement
   character - ASCII -> Floating point X
4) 3 bytes (24 bits)

opcode	x	Displacement
8 bits	1bit	15 bits

5) X=0 direct TA = displacement
   X=1 indirect TA = (X) + displacement
6) Data movement
   eg: LDA, LDX, STA, STX
7) Input / output:
   TD RD WD

- Arithmetic operation
  ADD, SUB, MUL
- Branching operation
  JLT, JEQ, JGL
- Sub-routine linkage instruction
  JSUB, RSUB

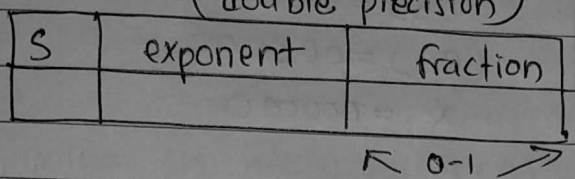
SIC/XE Architecture:

1) Memory: bit, 8 bits = byte 3 byte = word
   2^{20} memory (2 mb)
2) Registers: A, X, L, PC, SW, B, T, S, F
   → general purpose
3) Data format
4) Instruction format
5) Addressing modes

6) Instructions
7) Input and output (I/O)

3) Data format : integer → binary, -ve integer → 2s complement, character → ASCII

Floating point (single precision / double precision)



1 11 36 $= fx\ 2^{e-2024}$

4) Instruction format:

1 byte instruction (format 1) | opcode (8) |

eg: RSUB

2 byte instruction (format 2)

| opcode | operands |

8 r_1 r_2

4 bit 4 bit

comp A, S → converted to 4 bit hexa dec

format 3 : 24 bits

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

format 4 : 32 bits

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	disp

n i
0 1
1 0
1 1

e = 1 ⇒ format 4 is used

eg: +JSUB RDREC → ReadRecord

5) Addressing modes:

b=1 n=0 and i=1 → immediate addressing (no memory reference)

x=1 n=1 and i=0 → indirect addressing

PC=1 n=0 and i=0 → simple addressing TA is the location of the operand

e=1 n=1 and i=1 → simple addressing

3030	003600
3600	103000
6390	00C303
C303	003030

Memory

(B) = 006000

(PC) = 003000

X = 000090

n=0 and i=1 = TA = displacement = (B) + disp (12)

n=1 and i=0 ⇒ X b PC → search in ref of this bit

X=1 ⇒ TA = X + disp

PC=1 ⇒ TA = (PC) + disp (12)

e=1

↳ TA = disp (20)

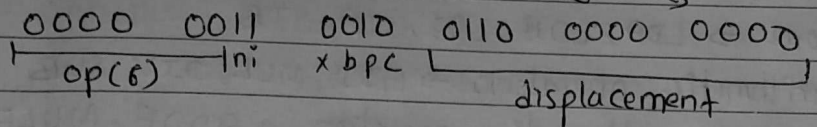
i)

X=1, P=1 then

TA = (X) + (B) + disp

Hex

Eg: 0 3 2 6 0 0



We have to calculate the final address

TA = (PC) + disp

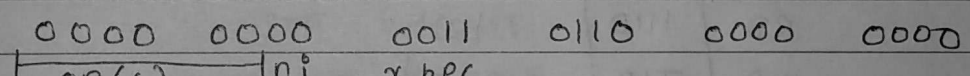
= 003000 + 600

= 003600

The value in accumulator (A) = 103000

003600 (Hex)

And final address (TA), Accumulator value (A) = ?



TA = (PC) + disp(20)

= 003000 + 00600

= 003600

∴ TA = 003600

A = 103000

SIC/XE

- Instruction sets:
  - Load/store → LDA, LDB, LDT, STA, STB
  - Integer arithmetic operation → ADD, MUL, DIV, SUB
  - Floating point arithmetic operation → ADDF, MULF
  - Compare → COMP A,S
  - conditional jump → JLT, JEQ, JGT
  - sub-routine linkage → JSUB, RSUB
  - Register manipulation, operands - from - registers, register to register arithmetics

Input/output (I/O)

- 2^8 I/O devices
- 1 byte will be transfer

TD	SIO	start
RD	HIO	Halt
WD	TIO	Test

Programming examples:

SIC	SIC/XE
Instruction variable: resource NUM1 RESB 1 DEVICE BYTE = X'F1' Hex const value String BYTE = C'TESTING' NUM2 RESW 1 Four word 4 variable datatype	

int C
Byte num1

Write a program to add two numbers.

```
            START   1000
ADDNUM      LDA     FOUR
            ADD     FIVE        A = 4
            STA     RESLT       A = A + FIVE = 9
                                RESULT = 9
FOUR        WORD    4
FIVE        WORD    5
RESLT       RESW    1
            END     ADDNUM
```

In SIC/XE:

```
ADDNUM      START   1000
            LDA     #4
            ADD     #5
            STA     RESULT
RESLT       RESW 1
            END ADDNUM
```

Assignment-1

Q) Write a SIC and SIC/XE program to find the sum of array having 5 different numbers.

-> SIC assembly language program:

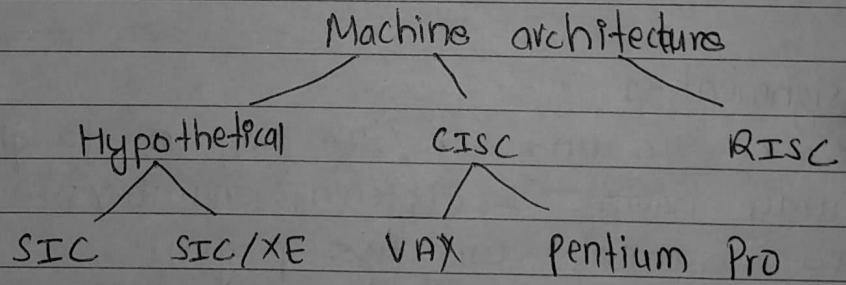
```
            START   1000
            LDA     zero
            LDX     zero
Loop        ADD     Address, X
            TIX     Count
            JLT     Loop
            STA     RESULT
```

```
Zero      WORD   0
Address   RESW   5
Count     WORD   5
Result    RESW   1
```

⇒ SIC/XE Assembly language program:

```
          LDT   #5
          LDX   #0
          LDA   #0
Loop      ADD   Address, X
          TIXR  T
          JLT   Loop
          STA   Result

Address   WORD  1000
Result    RESW  1
```



VAX architecture:

1) Memory:

8-bit = bytes, byte addressing, 2 bytes - word, four bytes - longword, 8 bytes form quadword
- operates in virtual address space (2^{32} bits)
  - → system space
  - → process space

2) Registers:
16 GPRs - 32 bit each
PC (R_{15}), SP (R_{14}), FP (R_{13}), AP (R_{12})
- PSL for flag.

3) Data format:
integer - binary in byte, word, longword, octaword
-ve integer - 2's complement
character - 8 bit ASCII
4 different floating point.

4) Instruction formats:
variable length instruction format 1 to 6 operand.

5) Addressing modes:
Register mode, register differ mode, autoincrement, autodecrement, base relative, PC relative, indexed, indirect and immediate.

6) Instruction sets:
Instruction are symmetric w.r.t datatype. The instruction mnemonics are formed by
- a prefix that specifies the ~~datatype~~ of the operands.
- suffix that specifies the datatype of the operand.
- a modifier that gives the number of operands involve.
eg: ADDW2, MULL3

7) Input / output:
use I/O device controller
- device control registers are mapped to separate I/O space.
- software and MMR are used for I/O operation.

Pentium Pro Architecture :

- 1995
- Intel X86 family
- Vast amt of software.
- different implementation details.

Memory :

physical level : byte, 2byte-word, long word
logical level : segments + offset

Registers : 32 bit 8 GPRS, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

Flags : Flag bit register CS, SS, DS, ES, FS, GS

Data formats :

integer - 8, 16, 32 binary nos
⊖ve - 2's complements
BCD - packed and unpacked

character(s) - ASCII code

Instruction formats : instructions uses prefixes to specify repetition count, segment registers, following prefix (if present) an opcode (1 or 2 bytes) and no of bytes to specify operands, addressing modes.

- varies in length from 1 to 10 byte or more.

Addressing modes : large no. of addressing modes
Immediate mode, register mode, direct, relative base, index with displacement.

TA = (b) + (index) * scalefactor + disp

Instruction set: large and complex, $\leq$ <100, may have 1 to 3 operand R-R, R-M, M-M, string manipulation, immediate value.

Input/output: input is form an I/O port into Register EAX output EAX to I/O port.

BCD: eg 487

packed: 0100 1000 0111 0000

1 byte 1 byte

unpacked: 0000 0100 0000 1000 0000 0111

4 8 7

Fp → single 32 = 24 + 7 + 1

64 = 53 + 10 + 1

80 = 64 + 15 + 1

logical level:

segment + page (no) + offset

base + (index) X scale factor + disp

CS (2, 4, 8)

Assignment 2:

RISC Architecture:

ultra sparc → presentation

1) UltraSPARC Architecture

→ Memory : 264 bytes in virtual address space
- consists of 8-bit bytes: halfword : 2 bytes
  word : 4 bytes
  doubleword : 8 bytes
- can be divided into pages
- virtual address is automatically translated into a physical address by the ultra SPARC Memory Management unit.

→ A large register file (>100 general-purpose registers)
- 32 bits for original SPARC, 64 bits for UltraSPARC
- Each procedure can access only 32 registers
  : 8 global registers and 24 registers in overlapped window
- Floating point computations are performed using a special floating-point unit.
- Program counter PC

→ Data formats:
- Characters : 8-bit ASCII codes
- Integers: 1, 2, 4, 8-byte binary numbers
  2's complement for negative values
- Floating-point numbers: single-precision : 32 bits
  Double-precision: 64 bits
  Quad-precision : 80 bits

→ Instruction formats:
- Fix-length instruction format (32 bits long)
- 3 basic instruction formats : call, branch, load
- Each instruction consists of : first 2 bits → identify formats
  op code
  operands

⇒ Addressing modes: Immediate mode
Register direct mode
PC relative mode only for branch instruction
Register indirect with displacement
Register indirect indexed

⇒ Instruction set
- less than 100 instructions
- Instruction execution is pipelined.
- Branch instructions are delayed branches
  - Instructions immediately following the branch instruction is actually executed before the branch is taken.

⇒ Input and output
- A range of memory locations is logically replaced by device registers.
- Device driver read/write values into those registers.
- software routines read/write values in this area using standard instructions.

2) PowerPc Architecture

- Developed by IBM in 1990.
- Power → Performance Optimization with Enhaned RISC.
- Memory : 8 bit bytes
  - 2 bytes - half word
  - 4 bytes - word
  - 8 bytes - double word
  - 16 bytes - quad word

Programs can be written using virtual memory address of 2^{64} bytes.

- Register: 32 general purpose
  - 32 bit
  - full power PC - 64 bit
  - 32 floating point register - 32 bit
  - control register
  - Link register
  - status register

- Data format : Integer : 8, 16, 32, 64 bit binary numbers.
  - -ve numbers → 2's complement.
  - floating point numbers
  - characters - 8 bit ASCII code

- Instruction format:
  - ⇒ 7 Instruction format

- Addressing modes:
  - Immediate
  - Register direct

3)

- Register indirect
- Register Indirect with index
- Indirect with immediate index
- Absolute
- Relative

- Instruction sets:
  - Approximately 200
  - floating point, multiply and add instructions

- I/O:
  - Direct store segments.
  - Normal virtual memory address.

3) CRAY T3E Arch
- announced by Cray Research, Inc (1995)
- Massively parallel processing system (MPP)
- Scientific computing
- T3E
  - 16 to 2048 processing element (PE)
  - three - dimensional network
  - each PE consists of a DEC Alpha EV5 RISC microprocessor, local memory and performance accelerating control logic.

Memory:
- 64 MB to 2GB
- physically distributed, logically shared memory.
- byte, longword, quadword
- 64 bit virtual addresses

Register
- 32 general purpose register designated R0 through R31; R31 always contains the value zero.
- 32 Floating point register F0 to F31. F31 always contains the value zero.
- program counter (PC)
- other status and control register.

Data formats
- two different types of floating point data formats.
- one for compatibility with VAX.
- the other is for IEEE standard formats.
- characters are stored using 8-bit ASCII codes.
- since there are no byte or store operations, characters that are to be manipulated seperately are usually stores one per longword.

Instructions format
- Five Basic Instruction formats
- 32 bit long
- The first 6 bit identity specify the opcode.
- Some instructions have an additional function field.

Addressing modes
- immediate mode, register direct mode.
- memory addressing

Mode	Target address calculation
PC-relative	TA = (PC) + displacement {23 bits, signed}

Register indirect with displacement	TA = (register) + displacement {16 ~~big~~ bits, signed}

- register indirect with displacement mode is used for load and store operations and for subroutine jumps.
- PC relative mode is used for conditional and unconditional branches.

Instruction set
- 130 machine instructions, no byte or word load and store In instruction.

Assignment:
- flowchart pass1, pass2 assembler
- SIC/XE object code

Pass1 and Pass2 Flowchart

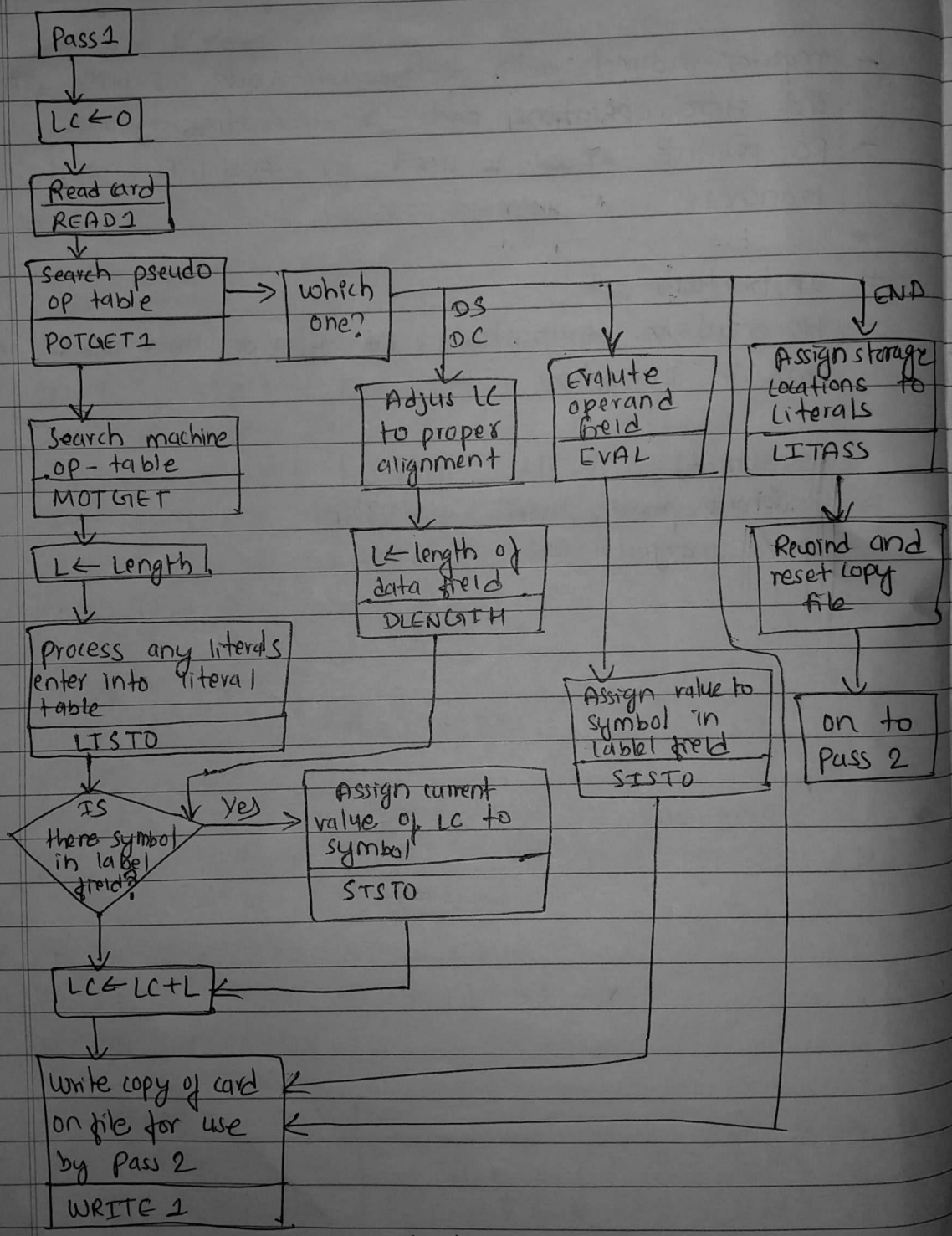
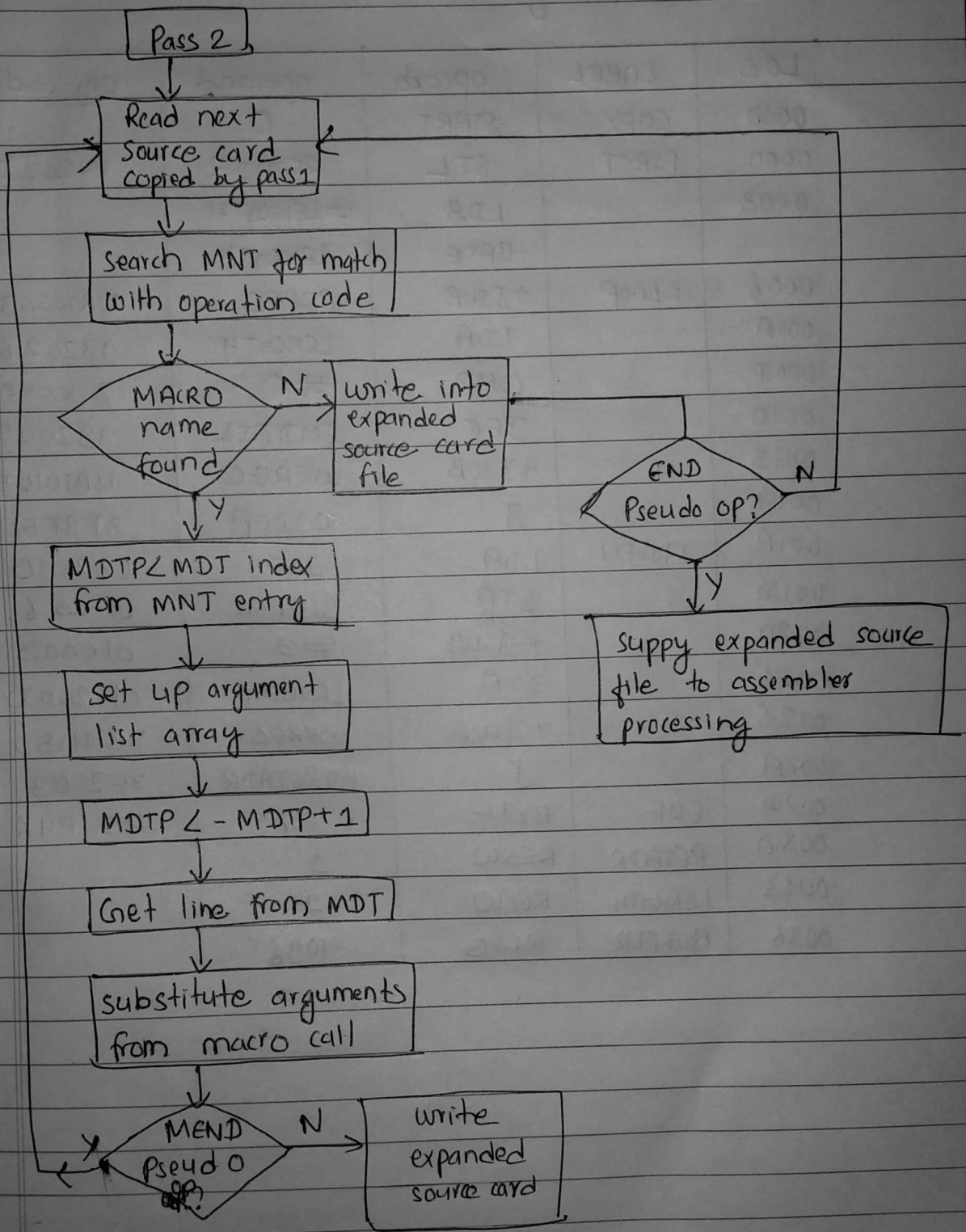


fig: Detailed pass1 flowchart


Pass 2
Read next source card copied by pass1
Search MNT for match with operation code
MACRO name found
N
Y
write into expanded source card file
MDTP<MDT index from MNT entry
Set up argument list array
MDTP<-MDTP+1
Get line from MDT
substitute arguments from macro call
MEND Pseudo OP?
N
Y
write expanded source card
END Pseudo op?
N
Y
suppy expanded source file to assembler processing

SIC/XE assembly with object code

LOC	LABEL	opcode	operand	obj. code
0000	COPY	START	0	
0000	FIRST	STL	RETADR	17202D
0003		LDB	#LENGTH	
		BASE	LENGTH	
0006	CLOOP	+JSUB	RDREC	4B101036
000A		LDA	LENGTH	032026
000D		COMP	#0	290000
0010		JEQ	ENDFIL	332007
0013		+JSUB	WRREC	4B10105D
0017		J	CLOOP	3F2FEC
001A	ENDFIL	LDA	EOF	032010
001D		STA	BUFFER	0F2016
0020		LDA ~~+JSUB~~	#3	010003
0023		STA	LENGTH	0F200D
0026		+JSUB	WRREC	4B10105D
002A		J	@RETADR	3E2003
002D	EOF	BYTE	C'EOF'	454F46
0030	RETADR	RESW	1	
0033	LENGTH	RESW	1	
0036	BUFFER	RESB	4096	

Machine dependent Assembler features

a) Instruction formats and addressing modes
b) Program relocation

a) Instruction formats and addressing modes

1) PC - relative or base relative	op m
2) Indirect addressing	op @m
3) Immediate addressing	op #c
4) Extended format	+op m
5) Register to Register	op Reg1, Reg2
6) index addressing	op m

SIC Machine

```
SUM     START   4000
FIRST   LDX     ZERO
        LDA     ZERO
LOOP    ADD     TABLE,X
        TIX     COUNT
        JLT     LOOP
        STA     TOTAL
        RSUB
TABLE   RESW    2000
COUNT   RESW    1
ZERO    WORD    1
TOTAL   RESW    1
        END     FIRST
```

SIC/XE Machine

```
SUM     START   4000
FIRST   LDX     #0
        LDA     #0
LOOP    ADD     TABLE,X
        TIX     COUNT
        JLT     LOOP
        STA     TOTAL
        RSUB
TABLE   RESW    2000
COUNT   RESW    1
TOTAL   RESW    1
        END     FIRST
```

XE

opcode	n	i	x	b	p	e	disp
6	1	1	1	1	1	1	12

FIRST LDX #0 050000
→ 04 ⇒ 0000 0101 0000 0000 0000 0000
0 5 0 0 0 0

PC relative :

TA = (PC) + disp(12)

$\hookrightarrow$ -2048 + 2047

BASE Relative TA = (B) + disp(12) } LDB and NO BASE

value = 0 $\curvearrowright$ 4095

1) PC-relative

eg: (a) 10 000 FIRST STL RETADR 17202D

(b) 40 0017 J CLOOP 3F2FEC

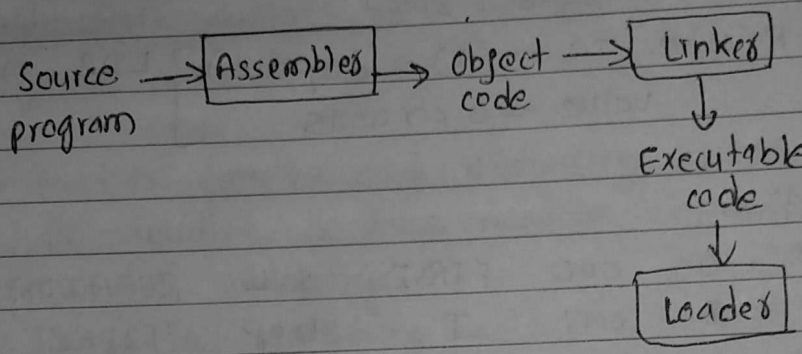
2) indirect addressing

002A J @RETADR 3E2003

Register to Register, COMPR A, X

Assignment:

Chapter-3
Loaders and linkers



Basic loader functions
- A loader brings an object program into memory and starting its execution.
- A linker performs the linking operations and a separate loader to handle relocation and loading.

Machine dependent Loaders features:

- Relocation
- Programming linking
- Algorithm and Data structure for a linking loader.

Machine Independent Loader feature:

a) Automatic library search
b) Loader options

a) Automatic library search

b) loader options

- INCLUDE prog_name (library name)
- DELETE (sect-name)
- CHANGE Name1, Name2

Include READ (UTLIB)
Include WRITE (UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE

LIBRARY MYLIB

NOCALL STDDEV, PLOT

Macro processors:

Macros

Macroprocessors

Basic Macro processor function
a) Macro definition and expansion
b) Macro processor algorithm and data structures

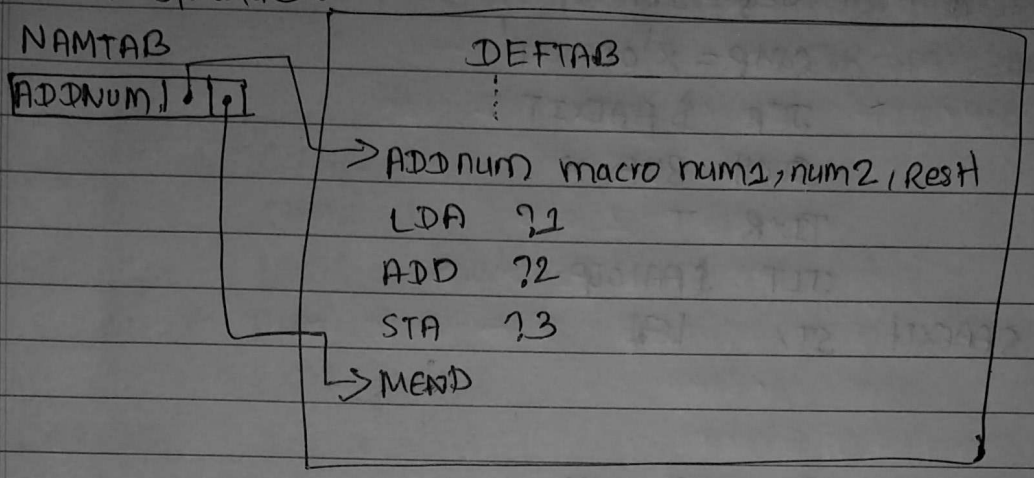
SIC

```
Name  Macro  Parameters
        :
      body
      MEND
```

```
ADDNum   Macro   num1, num2, RESLT
         LDA   Num1
         ADD   Num2
         STA   RESLT
         MEND
```

```
ADDNUM   A, B, R
ADDNUM   A1, B1, R
  LDA A1
  ADD B1
  STA R
ADDNUM   C, D, DF
```

Data structure:



Avg_tab

1	A1
2	B1
3	R

Functions:
GetLine
Processline
Define
expand

Macro time variable:
RDBUFF

```
RDBUFF   F1, BUF, LEN, (00,03,04)
         CLEAR  X
         CLEAR  A
         +LDT   #4096
$AALOOP  TD = x'F1'
         JEQ  $AALOOP
         RD = x'F1'
         COMP = x'000000'
         JEQ  $AAEXit
         COMP = x'000003
```

```
        JEQ  $AAEXIT
        COMP = X'000004'
        JER  $AAEXIT
        STCH BUF,X
        TIXR T
        JLT  $AALOOP
$AAEXIT STX  LEN
```

```
#  RDBUFF F1,BUF,LEN
           CLEAR X
           CLEAR A
           +LDT #4096
$ABLOOP    TD =X'F1'
           JEQ $ABLOOP
           RD =X'F1'
           STCH BUF,X
           TIXR T
           JLT $ABLOOP
$ABEXIT    STX LEN
```

Keyword Macro Example

```
RDBUFF  BUFADR=BUFFER, RECLH = LENGTH,
                        EOR=04, ~~MAXLEN=4096~~

        CLEAR X
        CLEAR A
        LDCH =x'04'
        RMO  A,S
        +LDT #4096
$AALOOP TD  =x'F1'
        JEQ  $AALOOP
        RD  = x'F1'
        COMPR A,S
        JEQ  $AAEXIT
        STCH BUFFER,X
        TIXR  T
        JLT   $AALOOP
$AAEXIT SIX   LENGTH
```

Macro processor Design option

1) Resursive macro Expansion
2) General purpose macro processors
3) Macro processing within language translator

1)

```
ABC  Macro  A1, A2
      :
XYZ  Macro  A4, A5
      :
     MEND
     MEND
```

Implementation Examples

1) Masm macro processor

```
ABSDIF  Macro  op1, op2, size
        LOCAL  Exit
        IFNB   <size>
        IFDIF  <Size>,<E>
          ;Error---- size must be E or blank
          .ERR
        Exitm
        END IF
        END IF
        MOV    size & AX, op1
        SUB    size & AX, op2
        JNS    Exit
        NEG    size & Ax
Exit : ENDM
```

```
* ABSDIF  J, k
     ↓
  MOV   AX, J
  SUB   AX, k
  JNS   2000000
  NEG   AX
2000000:
```

~~ABSDIF M,N,E~~

```
* ABSDIF  M, N, E

    MOV  EAX, M
    SUB  EAX, N
    JNS  ?000001
    NEG  EAX
?000001:
```

```
* ABSDIF M,N,B
⇒ Error-size must be E or
  blank
```

2) AnSI C macro language

```
#define EQ ==

while (I EQ 0)

while (I == 0)
```

define ABSDIF (X,Y) (X)>(Y)? (X)-(Y): (Y)-(X)

ABSC 'D', 'A')
⇒ D>A? D-A : A-D

define ABSDIFF (x,y) x>y? x-y : y-x

ABSDIFF (3+1, 10-8)
⇒ 3+1>10-8 ? 3+1-10-8 : 10-8-3+1 = -14

define Display (EXPR) printF ("EXPR = %d \n", EXPR)

Display (I*J+1)
printf ("#EXPR = %d \n", I*J+1)

define X 1

if X == 1
printF (
END DIF

or

#ifndef X
define X 1024
endif

age

3) The ELENA macro processor
%1 = %2 + %3
invoke
ALPHA = BETA + GAMMA

%1 := ABSDIFF (%2, %3)

%1 = (%2) > (%3) ? (%2) − (%3) : (%3) − (%2)

z := ABSDIFF (x, y)

↓

```
    MOV  EAX, %2
    SUB  EAX, %3
    JNS  &STOR
    NEG  EAX
&STOR MOV  EAX, %1
```

z := ABSDIFF (x, y)

↓ z = (2) > (3) ? (x) − (y) : (y) − (x)

```
    MOV  EAX, X
    SUB  EAX, Y
    JNS  STOR 0001
    NEG  EAX
STOR 0001  MOV  EAX, Z
```

Time variable

- SET

while ....

ENDW

1.a)

→

→

i)

ii)

Year: 2012
Semester: Spring

1.a) Explain the main purpose of system software in computer?

→ System software is a type of computer program that is designed to run a computer's hardware and application programs. If we think of the computer system as a layered model, the system software is the interface betn the hardware and user applications.

Eg: The programs used to manage a computer, such as installing or removing software and the programs that create the desktop environment.

→ System software includes os such as device drivers, severs, windowing systems and utilities.

The main purpose of system software are:

i) Allocating system resources:

The system resources are time, memory input and output. The time in the CPU is divided into time slices. The time slices is measured in terms of milliseconds. Based on the priority of tasks the time slices are assigned. Memory is also managed by os. Disk space is the part of main memory. The data flow is controlled by os.

ii) Monitoring system activities:

The system security and system performance is monitored by system software. system performance includes response time and CPU utilization. system security is a part of os. Multiple users can't

access without the security code or password.

iii) File and disk Management:

The user needs to save, copy, delete, move and rename the files. The system software will handle those functions. Disk and file management is the technical task.

1.c) Differentiate the RISC and CISC machine with example.

=>

RISC	CISC
1) It stands for Reduced Instruction set computing.	1) It stands for complex Instruction set computing.
2) The microprocessor is designed using hardwired control.	2) The microprocessor is designed using code control.
3) It executes at least one instruction in a cycle.	3) It executes several cycles in one instruction.
4) It has several general purpose registers and large cache memory.	4) It has a small number of general purpose registers.
5) It has a single clock.	5) It has multi clock.
6) Pipelining is easy.	6) Pipelining is difficult.
7) Spends more transistors on memory registers.	7) Transistors used for storing complex instructions
8) LOAD and STORE register-to-register are independent.	8) LOAD and STORE memory-to-memory is induced in instructions.
9) Complexity in compiler.	9) Extensive use of microprogramming
10) Eg: PIC18, ARM	10) Eg. 8051, 8086

2a) Write the algorithm of PASS-1 in assembler.

→ Pass 1:

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

```
add 3 * # [OPERAND] to LOCCTR
else if OPCODE = 'RESB' then
    add # [OPERAND] to LOCCTR
else if OPCODE = 'BYTE' then
    begin
        find length of ~~str~~ constant in bytes
        add length to LOCCTR
        end {if BYTE}
else
    set error flag (invalid operation code)
end {if not a command}
write line to intermediate file
read next input line
end {while not END}
write last line to intermediate file
Save (LOCCTR - starting address) as program length
end {pass 1}
```

2.b) Explain how object codes are generated by assembler with example.

=> In computing object code, or sometimes an object module, is what a compiler produces. In a general sense object code is a sequence of statements or instructions in a computer language, usually a machine code language such as register transfer language.

=> Object code is a portion of machine code that has not yet been linked in a complete program. It is the machine code for one particular library or module that will make up the completed product. It may also contain placeholders or offsets, not found in the machine code of a completed program, that

the linker will use to connect everything together. Whereas machine code is binary code that can be executed directly by the CPU, object code has the jumps partially parameterized so that a linker can fill them in.

→ An assembler is used to convert assembly code into machine code (object code). A linker links several object files to generate an executable. Assemblers can also ~~also~~ assemble directly to machine code.

For example:

Line	LOC	Source	statement	object code
10	1000	FIRST	STL RETADR	141033

To translate the source code into object code, the assembler converts STL to machine equivalent value 14. The symbolic operand RETADR is translated to 1033.

In the example, the instruction contains a forward reference i.e reference to label is defined later in the program. It is impossible to process this line as the address that will be assigned to RETADR is not known. So, most assemblers make two passes over source program where the second pass does actual translation.

The assembler must process statements called assembler directives which are not translated into machine instructions but they provide instructions to assembler.

Eg: RESW and RESB instructs assembler to reserve memory locations.

3.a) Explain the use of loader and linker in system software.

→ Loader is program which accepts the object program and prepares it for executions (loads into memory).

Function / use of loader:

i) Allocation → Allocates space in memory for program
ii) Linking → Resolves symbolic references betn objects codes.
iii) Relocation → Adjust all the address dependent locations
iv) Loading → physically place M/c instructions and data into memory

→ Linker is the tool that merges the object files produced by separate compilation or assembly and creates an executable file.

Use of linker:

i) searches the program to find library routines used by programs, eg: printf(), math routines...
ii) Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references.
iii) Resolves references among files.

b) How different programs linking is done. Explain with the help of examples.

→ Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. Linking can be performed at compile time,

when the source code is translated into machine code, at load time, when the program is loaded into memory and executed by the loader and even at run time, by application programs. On early computer systems, linking was performed manually On modern systems, linking is performed automatically by programs called linkers.

=> Linkers play a crucial role in software development because they enable separate compilation. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files.

eg: Let us consider following examples:

LOC		statement
0000	PROGA	START 0
		EXTDEF LISTA, ENDA
		EXTDEF LISTB, ENDB, LISTC, ENDC
0020	REF1	LDA LIST A
0023	REF2	+LDT LIST B+4
0027	REF3	LDX #END A - LIST A
0040	LISTA	EQU *
0054	ENDA	EQU *

4.a) Explain the dynamic linking in macro.

⇒ The term 'dynamically linked' means that the program and the particular library it references are not combined together by the linker at link time.

⇒ Instead, the linker places information into the executable that tells the loader which shared object module code is in and which runtime linker should be used to find and bind the references.

⇒ This means that the binding betn the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.

⇒ This type of program is called a partially bound executable, because it isn't fully resolved. The linker, at link time, didn't cause all the referenced symbols in the program to be associated with specified code from the library.

4b) Explain about macro expansion with an example.

→ A macro represents a commonly used group of statements in the source programming language.

→ Macro expansion means replacing each macro instruction with the corresponding group of source language statements.

For eg:

- On SIC/XE requires a sequence of seven instructions to save the contents of all registers.
  ↳ then write one statement like SAVEREGS.

→ A macro processor is not directly related to the architecture of the computer on which it is run.

→ Macro processors can also be used with high level programming languages, Os command languages.

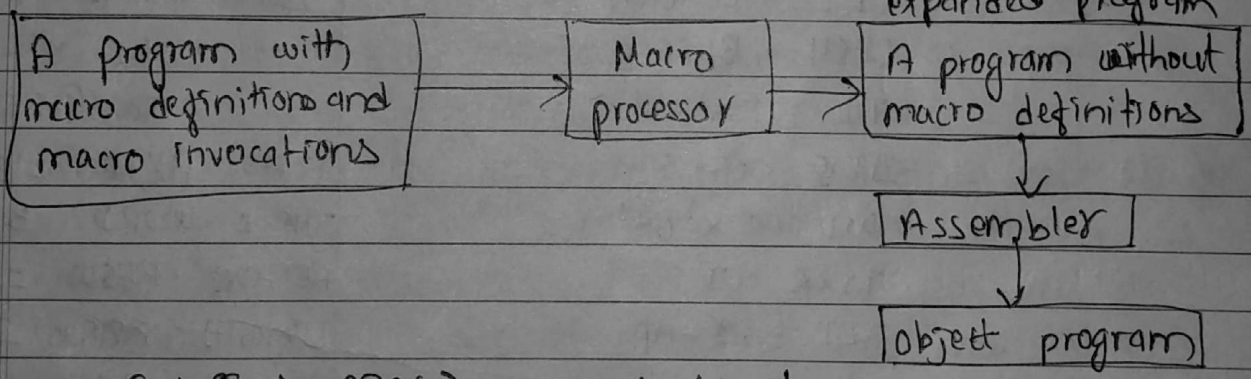


fig: Basic macro processor functions.

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

```
clear   S
+LDT    #4096
        TD      =X'F1'
        JEQ     *-3
        RD      =X'F1'
        COMPR   A,S
        JEQ     *+11
        STCH    BUFFER,X
        TIXR    T
        JLT     *-19
        STX     LENGTH
        LDA     LENGTH
        COMP    #0
        JEQ     ENDFIL
        WRBUFF  05,BUFFER,LENGTH
        CLEAR   X
        LDT     LENGTH
        LDCH    BUFFER,X
        TD      =X'05'
        JEQ     *-3
        WD      =X'05'
        TIXR    T
        JLT     *-14
        J       CLOOP
ENDFIL  WRBUFF  05,EOF,THREE
        CLEAR   X
        LDT     THREE
        LDCH    EOF,X
        TD      =X'05'
        JEQ     *-3
        WD      =X'05'
        TIXR    T
        JLT     *-14
        J       @RETADR
EOF     BYTE    C'EOF'
THREE   WORD    3
RETADR  RESW    1
LENGTH  RESW    1
BUFFER  RESB    4096
        END     FIRST
```

Fig: Program with macros expanded

Q.c Explain briefly about machine independent loader features.

⇒ The Machine Independent ~~tot~~ loader features are:

i) Automatic Library search
ii) Loader options

i) Automatic Library search

⇒ It is a standard system library and subprogram library.

⇒ The programmer does not need to take any action beyond mentioning the subroutine names as external references.

⇒ Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

⇒ Libraries ordinarily contain assembled or compiled versions of subroutines (object programs)

⇒ Library search by scanning the define records for all of the object programs on the library is quite inefficient.

↳ special file structure is used for the libraries and this structure contains a directory that gives the name of each routine and a pointer to its address within the file.

ii) Loader options

⇒ Many loaders have a special command language that is used to specify options as

- a separate input file to the loader that

contain control statements.
- control statements embedded in the primary input stream beth object programs.
- control statements are included in the source program, and the assembler or compiler retains these commands as a part of the object program.

Commands of loader options.

* INCLUDE program-name (Library-name)
It directs the loader to read the designed object program name specified as a part of input program.
* DELETE csect-name
It deletes the named control section(s) from the program loaded when not used.
* CHANGE name1, name2
It changes the external symbol name 1 to name2 appeared in the object program.

5.a) Define two different development processes that Booch suggested.

=> The Booch method is a method for object oriented software development. It is composed of an object modelling language, an iterative object oriented development process and set of recommended practices.

=> There are two different development processes, which he calls micro and macro.

i) Macro process

-> Booch's macro process represents the overall activities of the development team on a long-range scale.

Semester - ~~Fall~~ Spring
Year - 2012

1.a) ~~Explain the main purpose of system software in computer?~~

It includes the following activities:
- Establish the requirements for the software (conceptual-ization).
- Develop an overall model of the system's behaviour (analysis).
- Create an architecture for the implementation (design).
- Develop the implementation through successive refinements (evolution).
- Manage the continued evolution of a delivered system (maintenance).

This macro process repeat itself after each major release of a software product.

9i) Micro process

⇒ Booch's micro process essentially represents the daily activities of the system developers. It consists of the following activities:
- Identify the classes and objects of the system.
- Establish the behaviour and other attributes of the classes and objects. For eg: the methods that are applicable to each.
- Analyze the relationships among the classes and objects. For eg: the use of aggregation, inheritance and polymorphism.
- Specify the implementation of the classes and objects.

5.b) Explain the interaction diagram for assembler with example.

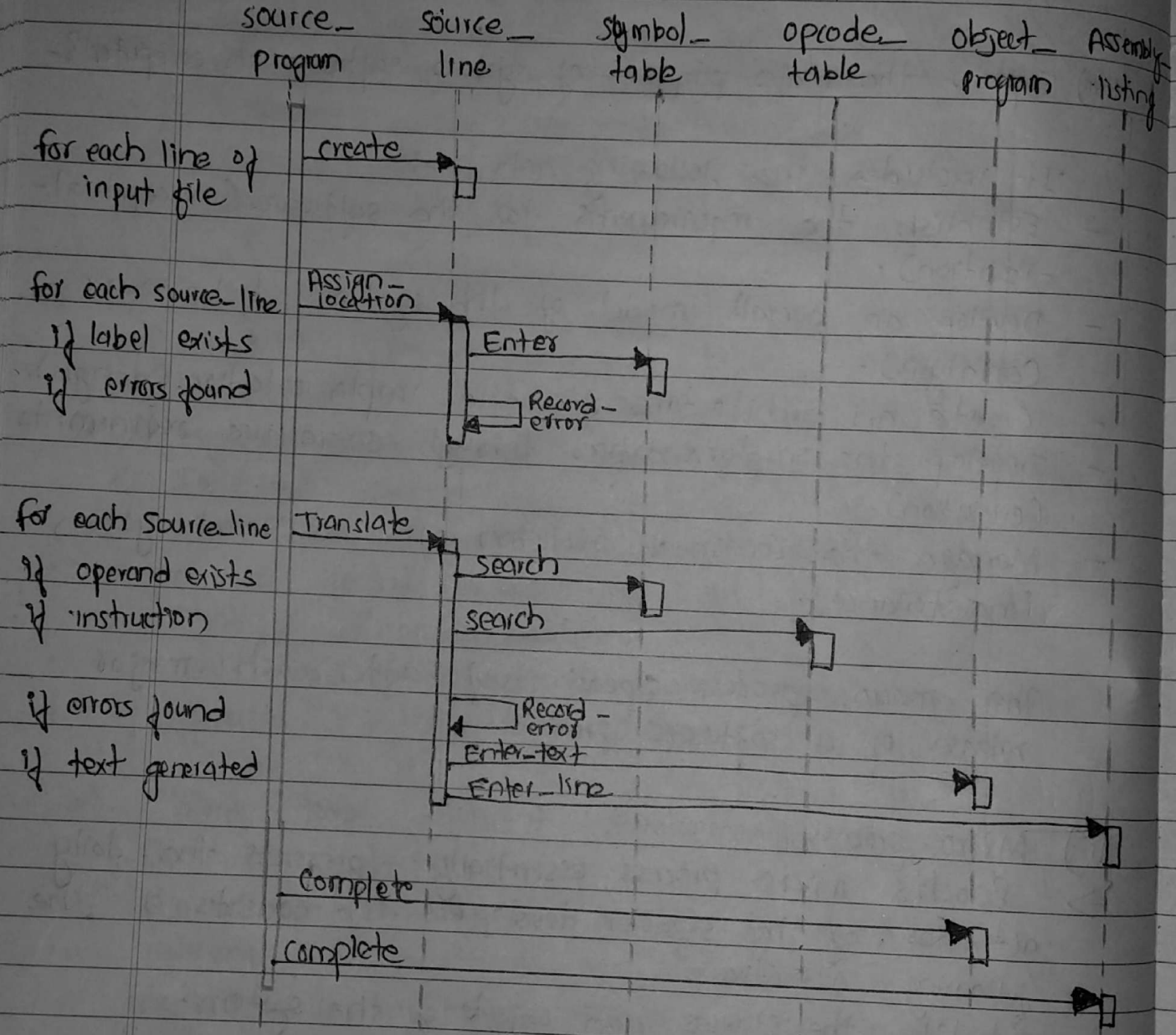


Fig: Interaction diagram for assembler

Interaction diagrams make it easier to visualize the sequence of object invocations, and the ~~follow~~ flow of control betn objects.

Consider, for example, the interactions diagram

for assembler in above figure. The primary focus of control is in the object source_program. The assemble method of source-program begins by creating a new instance of source_line for each line in the input file for the assembler. The method Assign_location is then invoked for each source-line object. During its period of control, source_line may invoke Enter to make an entry in the symbol table. If errors are detected, it may invoke the method Record-error on itself.

After the iteration of Assign_location is complete, a second iteration is used to translate each source line. During this process, source_line may invoke methods on several other objects, depending on the conditions encountered.

6.a) Explain the advantages and disadvantages of writing a loader using high level programming language.

6. b) Explain about ANSI Macro language.

=> In ANSI C language, definitions and invocations of macros are handled by a preprocessor. The preprocessor is generally not integrated with the rest of the compiler.

For eg: # define NULL 0

define EOF (-1)

These two ANSI C macro definitions means that the every occurrence of NULL will be replaced by 0 and EOF by (-1).

similarly, if the macro # define EQ == is defined then,

while (I EQ 0)

statement is converted into

while (I == 0)

by the macro processor.

ANSI C macros can be defined with parameters such as

define ABSDIF (X,Y) ((X)>(Y)?(X)-(Y):(Y)-(X))

Here, macro name is ABSDIF, parameters are X, Y. If (X) > (Y) condition is true, value of the expression is (X) - (Y) otherwise value is (Y) - (X).

In ANSI C, macro invocation can be done as ABSDIF (3+1, 10-8)
Then, it is expanded as
3+1 > 10-8 ? 3+1 -10 - 8 ; 10 - 8 - 3+1
which would produce output -14 instead of 2.

ANSI C provides stringizing operator # when name of macro parameter is preceded by #, argument is substituted.
For eg: # define DISPLAY (EXPR) printf (# EXPR "= %d \n", EXPR)

Then invocation;
DISPLAY (I * J + 1)
would be expanded as
printf ("I * J + 1" - "= %d \n", EXPR)

Moreover, ANSI C provides conditional compilation statements to be sure that a macro is defined at least once.

7) write short notes:
a) Multi - pass assembler
⇒ Multi-pass assembler completes it's task in more than one pass. In multipass assembler only parts of the program having forward references must be processed in multiple passes, but not the whole program need multiple passes. To implement multipass assembler, we use a symbol table to store the

symbols that are not defined. We store name and numbers on the table. When a symbol is defined, we use it's value to re-evaluate values of other symbols.

Thus, multipass assembler completes its task in two pass. In first pass, it scans code, validates it and creates symbol table. In second pass, it solves forward references and converts assembly code to machine code.

b) Algorithm and data structure for a linking loader.

→ The algorithm for linking loader is more complicated than absolute loader algorithm. A linking loader makes two passes over its input, which is similar to that of assembler. pass1 assigns address to all external symbols. Pass 2 performs actual loading, relocation and linking.

The main data structure needed linking loader is an external symbol table ESTAB. This table is analgous to SYMTAB in assembler algorithm. It is used to store name and address of each external symbol in the set of control sections being loaded. A hashed organization is used for this table. The other two variables are PROGADDR (program load address) and CSADDR (control section address). PROGADDR is beginning address in memory where linked program is to be loaded. It's value is supplied by OS to loader. CSADDR contains starting address assigned to control section currently being scanned by loader.

c) Program blocks in assembler

=> Program block allow the generated machine instructions and data to appear in object program in a different order from the corresponding source statements. Program blocks refers to the segments of code that are rearranged within a single object program unit. The assembler directive USE indicates which portion of source program belongs to various blocks. At begining, program statements are assumed to be of default block. If no USE statement is used then, entire program belongs to single block. Program block can contain several segments of source program. The assembler rearranges the segments to gather the pieces of each block together. These blocks will be assigned addresses in object program in the order in which they were first begun in source program.

eg:

```
COPY      START   0
FIRST     STL     RETADR
CLOOP     JSUB
          :
          JSUB    WRREC
          :
          USE
RDREC     CLEAR   X
          CLEAR   A
          :
          USE
WRREC     CLEAR   X
```

Here, there are different program blocks RDREC and WRREC to read and write record into buffer.