SURYA Gold

Assignment-1

Date 8/02/18 Page 01

1a) Distinguish between System software and application software.

Ans: System Software mainly differs from application software is machine dependency.

→ An application program is concerned with solution of some problem, using computer as tool.

→ The focus of application software is mainly on the application but not on the computing system

→ whereas, System software are intended to support the operation and use of computer rather than any particular operation (or) application.

→ System Software is dependent on the architecture of the machine on which they are run.

example of system software: assemblers translate mnemonic instructions into machine code

example of application software: word processor, spreadsheet.

8b Briefly explain the SIC assembler directives with examples.

SIC Assembler directives:

i) START :- Specify name and starting address for the program

ii) END :- Indicate the end of the source program and specify the first executable instruction in the program.

iii) BYTE :- Generate character or hexadecimal

Assumed binary point is immediately before the high-order bit where for normalized floating point numbers the high-order bit of the fraction must be 1.

→ exponent is an unsigned binary number between 0 and 2047.

→ The absolute value of the number represented is $f * 2^{(e-1024)}$

f → fraction

e → exponent.

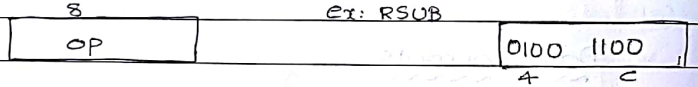
→ Here s is sign. where 0→positive & 1→negative.

→ Zero is represented by setting all bits to zero.

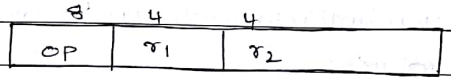
Instruction formats :-

There are 4 instruction formals in SIC/XE due to increase in memory

Format 1 (1 byte):

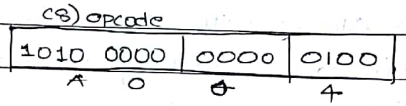


Format 2 (2 byte):

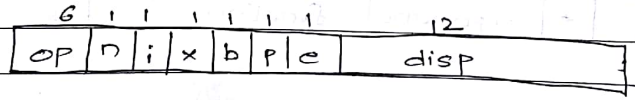


where r1 & r2 are registers

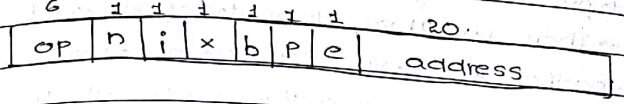
Ex: COMR A,S



Format 3 (3 bytes)



Format 4 (4 bytes):



Addressing Modes

Two adressing modes are available.

Mode	Indication	Target Address Calculation
Base relative	b=1, p=0	TA=(B)+disp (0≤disp<4095)
Program-counter relative	b=0, p=1	TA=(PC)+disp (-2048≤disp≤2047)

The above is for Format 3.

For Format 4, b=0 p=0

For Format 3, e=0 and Format 4, e=1

For immediate addressing mode, n=0, i=1.

For indirect addressing mode, n=1, i=0

If it is neither of them n=1, i=1.

Instruction Set

→ It have load & store register instructions (LDA, LDX, STA, STX etc.,) as well as for new registers (LDB, STB)

→ To perform floating-point arithmetic operations (ADDF, SUBF, MULF, DIVF)

→ Besides RMO instruction, these include register to register arithmetic operations (ADDR, SUBR, MULR, DIVR)

→ SVC (supervisor call) instruction is provided which is used for communication with operating system

Input and Output:

→ There are three IIO instructions, each of which specifies the device code as an operand

* Test Device instruction tests whether the addressed device is ready to send or receive a byte of data
* Read Device
* Write Device

→ Along with them IIO channels are used to perform input & output while the CPU is executing other instructions

→ They allow overlapping of computing & IIO resulting more efficient system operation

→ The instructions SIO, TIO and HIO are used to start, test and halt the operation of IIO channels

2b. Explain the SIC architecture.

Ans: SIC stands for Simplified Instructional Computer

Memory

→ Memory consists of 8-bit bytes

→ Any 3 consecutive bytes form a word (24 bits)

→ All addresses on SIC are byte addresses

→ Words are addressed by the location of their lowest numbered byte.

→ There are a total of 32,768 (2^{15}) bytes in memory

Registers

→ Each register is 24 bits in length

Mnemonic	Number	Special Use
A	0	Accumulator: used for arithmetic operations
X	1	Index register: used for addressing
L	2	Linkage register: the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter: contains the address of the next instruction to be fetched for execution
SW	9	Status word: contains a variety of information; including a condition code (CC)

Data Formats

→ Integers are stored as 24-bit binary numbers

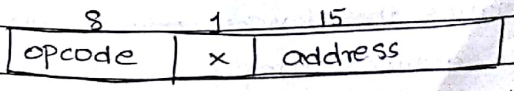
→ 2's complement representation is used for negative values

→ characters are stored using their 8-bit ASCII codes

→ It doesn't support floating point numbers

Instruction formats

→ All instruction formats are 24-bits formal



→ Here flag bit x is used to indicate indexed-addressing mode

Addressing Modes:-

Two addressing modes are available

Mode	Indication	Target Address calculation
Direct	x=0	TA = address
Indexed	x=1	TA = address + (x)

Instruction Set :-

→ Load and store register instructions
LDA, LDX, STA, STX etc.
→ Integer Arithmetic operations
ADD, SUB, MUL, DIV.
all of them involve register A and a word in memory
where register contains result
→ COMP that compares the value in register A with a word in memory
→ A conditional code cc to indicate the result
→ Conditional jump instructions
JLT, JEQ, JGT
→ Subroutine linkage:
JSUB, RSUB

Input and Output :-

→ On standard version of SIC, input and output are performed by transferring 1 byte at a time to or from rightmost 8 bits of register A
→ There are three I/O instructions each of which specifies the device code as an operand.
→ The three devices are:
i> Test Device : tests whether the device is

ready to send or receive a byte of data.
→ Read device : reads data.
→ Write device : writes data.

3a) What is upward compatible? How it is ensured between SIC and SIC/XE?

Ans: Upward compatible means an object program for the standard SIC machine will also execute properly on a SIC/XE system.

→ To ensure upward compatability SIC/XE machines have a special hardware feature
→ If bits b and i are both 0, then bits b, p and e are considered to be part of the address field of the instruction (rather than flags indicating addressing modes)
→ This makes Instruction Formats identical to the format used on the standard version of SIC, providing the desired compatibility.

3. Write and explain the algorithm for pass-1 of two-pass assembler.

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



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

Explaination:

Pass 1 (define symbols):

1. Assign addresses 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. (This includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW, etc)

In Pass 1 we use three datastructures

i) OPTAB: To look up mnemonic operation codes and translate them into machine code equivalents

ii) SYMTAB: Used to store values assigned to labels

iii) LOCCTR: It is used to help in the assigment of addresses.

In Pass1, We use OPTAB to find the instruction length for incrementing LOCCTR and validate OPCODE.
→ Labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses.
→ Pass1 writes an intermediate file that contains each source statement together with its assigned address, error indicators etc..

Algorithm
if assembler find START in label it store the address #[OPERAND] in LOCCTR.
and after START other mnemonic codes are read and according to that SYMTAB & LOCCTR are incremented
→ Until OPCODE is not equal to END.
Assembler searches symbol in the LABEL field
if symbol is present in SYMTAB set a flag or else add the symbol(LABEL) and LOCCTR into SYMTAB
→To increment the LOCCTR, check OPCODE.
If OPCODE is
WORD ⇒ add 3 to LOCCTR
RESW ⇒ add 3* (#OPERAND) to LOCCTR
RESB ⇒ add # OPERAND to LOCCTR.
Where # OPERAND is a character string that represents a number.
BYTE ⇒ We find constant length in bytes and add length to LOCCTR.
And all of them are written into an intermediate file which is provided as input to pass2 algorithm.

4. Differentiate between literal and immediate operand with example.

Ans: A literal is identified with the prefix =, which followed by a specification of the literal value.

ex: 45 001A ENDFIL LDA =C'EOF' 032010
specifies a 3byte operand with value 'EOF'

Difference between a literal and immediate operand.

Immediate Addressing
→ The operand value is assembled as part of the machine instruction.

Literal :-
→ The assembler generates the specified values as a constant at some other memory location
→ The address of this generated constant is used as the target address for the machine instruction
→ The effect of using a literal is exactly the same as if the programming had defined the constant explicitly and used the label assigned to constant as the instruction operand.

Ex: Consider the object code generation of literal and immediate operand.

Line	LOC	Source statement	Objcode
45	001A	ENDFIL LDA =C'EOF'	032010
50	001D	STA BUFFER	0F2016
55	0020	LDA #3	010003

```
60      0023    STA   LENGTH      0F2000
65      0026    +JSUB WRREC       4B10105D
70      0028    J     @RETADR     3E2003
93              LTORG
.       002D    *     =C'EOF'     454F46
```

Literal:-

LDA =C'EOF'

n	i	x	b	p	e
000000	1	1	0	0	1	0	0010

disp = TA - PC TA → Generated constant address
= 002D - 001D
= 010

objcode = 032010

Immediate operand

LDA #3

n	i	x	b	p	e
000000	0	1	0	0	0	0	003

displacement is part of machine instruction

5. What are control sections ? How are they processed ?

Ans: A control section is a part of the program that maintains it's identity after assembly
→ Each control section can be loaded and relocated independently of the others.

ex:

```
LOC   Line  Source Statement                          Objcode
0000  5     COPY  START  0
      6           EXTDEF  BUFFER, BUFFEND, LENGTH
      7           EXTREF  RDREC, WRREC
0000  10    FIRST   STL   RETADR                     172027
0003  15    CLOOP   +JSUB RDREC                      4B1000
                                                     000
      :
0033  105   BUFFER  RESB  4096
1033  106   BUFFEND EQU   *
1000  107   MAXLEN  EQU   BUFFEND-BUFFER
                                  ↑ 1 control section
      109   RDREC   CSECT
                  SUBROUTINE TO READ RECORD INTO BUFFER
      122         EXTREF  BUFFER, LENGTH, BUFFEND
0000  125         CLEAR   X                          B410
      :
0020  175   EXIT    +STX  LENGTH                     13100000
      :
0028  190   MAXLEN  WORD  BUFFEND-BUFFER            000000
                                  ← 2 control section
0000  193   WRREC   CSECT
                  SUBROUTINE TO WRITE RECORD INTO BUFFER
                  EXTREF  LENGTH, BUFFER
0000  210         CLEAR   X                          B410
      :
000C  225         +LDCH   BUFFER, X                  53900000
      :
      255   END     FIRST
                                  ↑ 3 control section.
```

In the example program, there are 3 control sections

First → main control section (COPY).

Second → BDREC

Third → WRREC

Each Control Section is identified by CSECT

→ Control Sections differ from program blocks in that they are handled seperately.

→ Symbols that are defined in one control section may not be used directly by another control section

→ They must be identified as external references for the loader to handle.

In control sections, we use 2 assembler directives to identify such references.

→ EXTDEF (external definition).

→ It defines symbols called external symbols that are defined in a control section and may be used by other control sections.

→ Control Section names do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols.

→ EXTREF (external reference)

→ It names symbols that are used in this control section and are defined elsewhere.

ex: The symbols BUFFER, BUFFEND & LENGTH are defined in control section named COPY and made available to other sections by the EXTDEF statement.

→ The control section (WRREC) uses length, buffer are specified in its EXTREF statement.

→ The external references are handled in the following way

consider

15 0003 CLOOP +JSUB BDREC 4B100000

BDREC operand is in EXTREF statement for control section, so this is an external reference.

→ The assembler has no idea where the control section containing BDREC will be loaded, so it cannot assemble the address for this instruction

→ Instead, the assembler inserts an address of zero and passes the info to the loader, which will cause the proper address to be inserted at load time.

→ Consider another statement

190 0028 MAXLEN WORD BUFFEND-BUFFER 000000

As before, the assembler stores this values as zero.

→ When program is loaded, the loader will add to this data area the address of BUFFEND and subtract from it the address of BUFFER, which results in the desired value.

Hence:

→ The assembler must remember in which control section a symbol is defined.

→ Any attempt to refer to a symbol in another control section & symbol must be flagged as an error unless the symbol is identified as an external reference.

6. Generate the complete object code for the following assembly level program and give the reason if the code is not possible for any instruction.

Loc	Length				
0000	3	SUM	START	0	
0000	3	FIRST	LDX	#0	050000
0003	3		LDA	#0	010000
0006	4		+LDB	#TABLE2.	6910234B
000A	3	LOOP	ADD	TABLE,X	1BA016
000D	3		ADD	TABLE2,X	1BC000
0010	3		TIX	COUNT	2F200D
0013	3		JLT	@LOOP	3B2FF4
0016	4		+STA	TOTAL	0f104673
001A	3		STA	@TOTAL	error.
001D	3		RSUB		4F0000
0020	3	COUNT	RESW	1	
0023	2328	TABLE	RESW	3000.	
234B	2328	TABLE2	RESW	3000.	
4673	3	TOTAL	RESW	1	
4676			END	FIRST	

LDX #0

n i x b p e

0000 01	0	1	0	0	0	0	0000 0000 0000

0 5 0 0 0 0

⇒ 050000.

LDA #0

0000 00	0	1	0	0	0	0	0000 0000 0000

n i x b p e

010000.

+LDB #TABLE2.

0000 10	0	1	0	0	0	1	0 2 3 4 B

n i x b p e

⇒ 6910234B

ADD TABLE,X

0001 10	1	1	1	0	1	0	0000 0001 0110

n i x b p e

disp = TA - (PC)

= 0023 - 000D ⇒ 1BA016

= 0016 (HD)

= 22 (D) → Range −2048 < pc < 2047 ✓

ADD TABLE2,X

0001 10	1	1	1	1	0	0	0000 0000 0000

n i x b p e

disp(PC) = TA - PC

= 234B - 0010

= 233B (HD)

= 9019 (D) Not in Range

disp = TA - (base)

= 234B - 234B

= 0

⇒ 1BC000.

TIX COUNT

0010 11	1	1	0	0	1	0	0000 0000 1101

n i x b p e 0 0D

disp = TA - PC

= 0020 - 0013

= 000D (HD)

= 13 (D) −2048 < pc < 2047 ✓

2F200D.

JLT LOOP

0011	10	1	1	0	0	1	0	F F 4
		n	i	x	b	p	e	

disp = 000A - 0016 ⇒ 3B2FF4

= FF4 (HD)

= -12 (D) $-2048 \leq pc \leq 2047$

+STA TOTAL

0000	11	1	1	0	0	0	1	0	4673

⇒ 0F104673

STA @ TOTAL

0000	11	1	0	0			0	
		n	i	x	b	p	e	

disp = 4673 - 001D (TA - PC) disp = 4673 - 234B

= 4656 (HD) = 2328 (HD)

= 18006 (D) x = 9000 (D) x

RSUB

0100	11	1	1	0	0	0	0	000
		n	i	x	b	p	e	

⇒ 4F0000

The code is not possible for STA @ TOTAL because the displacement are not in the range of pc relative and not in the range of base relative.

7. Explain in detail Program Blocks.

Ans. Program Blocks refer to segments of code that are rearranged within a single object program units.

ex:

Line	Loc	Block	Source State	ObjCode
5	0000	0	COPY START 0	
10	0000	0	FIRST STL RETADR	172063
⋮				
70	0024	0	J @ RETADR	3E203F
92	0000	1	USE CDATA	
95	0000	1	RETADR RESW 1	
100	0003	1	LENGTH RESW 1	
103	0000	2	USE CBLKS	
105	0000	2	BUFFER RESB 4096	
106	1000	2	BUFEND EQU *	
107	1000		MALEN EQU BUFEND-BUFFER	
			SUBROUTINE TO READ RECORD INTO BUFFER	
123	0027	0	USE	
⋮				
180	004A	0	RSUB	4F0000
183	0006	1	INPUT BYTE X'F1'	+1
			SUBROUTINE TO WRITE RECORD FROM BUFFER	
208	004D	0	USE	
⋮				
245	0063	0	RSUB	
252	0007	1	USE CDATA	

The above program have multiple Program blocks

1) first block - executable code
2) CDATA Block - data areas of short length

3> CBLKS block - data areas of large length

USE directive:

→ USE <name> begins a new block <name> or resume a previous block <name>

→ Use resume the default block

In above example, line 92 signals that beginning of the block named CDATA

→ A block can contain seperate segments of code in source program.

→ Assembler will logically re-arrange them together

→ The assembler accomplishes this logical rearrangement of code by maintaing, during Pass1, a seperate location counter for each program block.

In Pass 1

i> location counter is initialized to 0 when the block begins

ii> Save the location counter value when switching to another block and restore the location value when resuming a previous block

iii> Each label is assigned the address which is relative to the start of its block. The blocks name or number the label belongs to is also stored with its relative address

iv) At the end of Pass1, the location counter of each block indicates the length of the block.

v) Finally each block is assigned a start address which is relative to the beginning of the entire program.

In Pass 2,

→ Compute the address for each symbol, which is relative to the start of entire object program. i.e., relative starting address of the block + relative address of the symbol

At the end of pass1, the assembler constructs a table that contains the starting Addresses and lengths for all blocks

for above example:

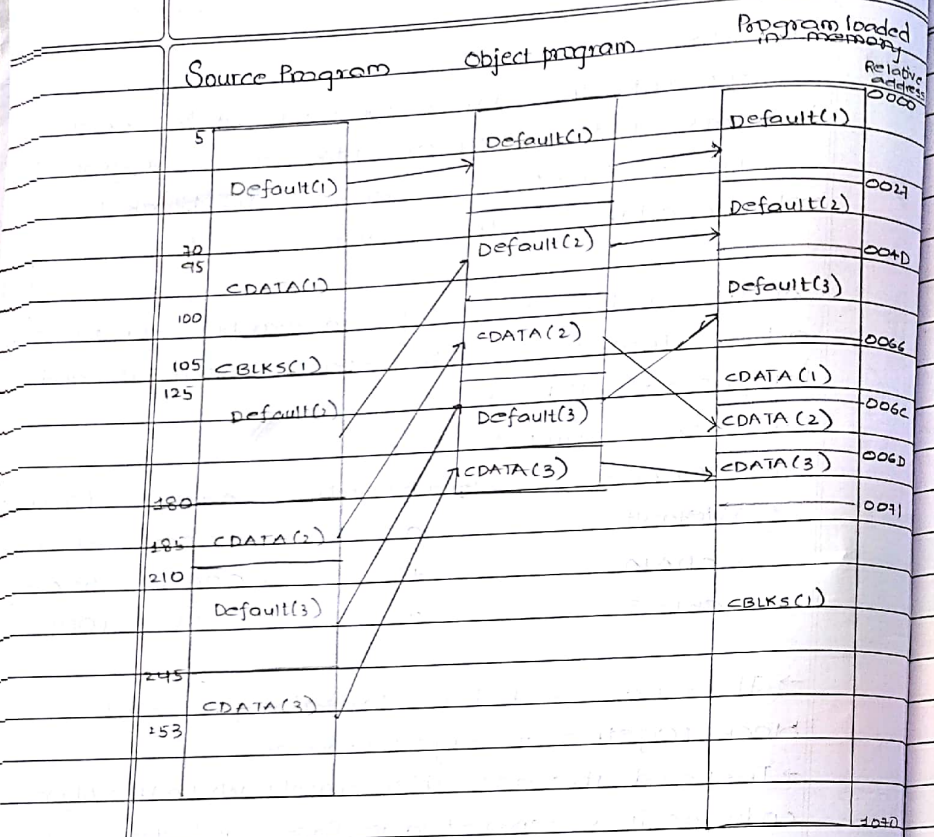
Block Name	Block Number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

→ It is not needed to place the pieces of each block together in object program

→ Instead, the assembler simply write the object code as it is generated in Pass2, and insert the proper load address in each Text record.

→ It doesn't matter that the text records of the object program are not in sequence by addresses, the loader can just load the records into the indicated addresses.

Program blocks traced through the assembly & loading processes.



→ In the above program, the pieces of blocks are in different places of object program
→ Some data blocks may not appear in object program e.g CDATA(1), CBLKS(1)
→ After loading, the pieces of a block will be placed together and
→ The memory will be automatically reserved for data blocks, even they donot appear in object program.

Q. Explain how multi pass assembler handles the forward reference.

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

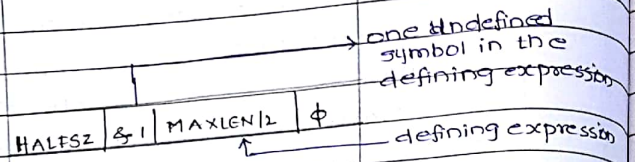
Assume that, when assembler goes to line 4, location counter contains 1034 (Hex)

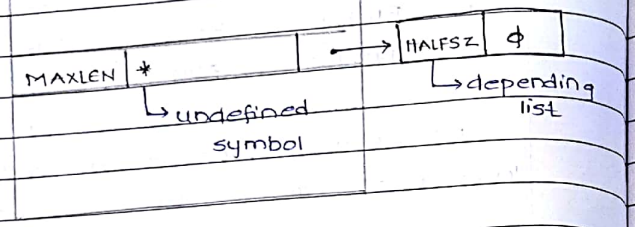
Ans. To handle forward references, MultiPass assemblers uses a symbol table
→ to store symbol definitions that involve forward references
→ to indicate which symbols are dependent on the value of others
→ to facilitate symbol evaluation.

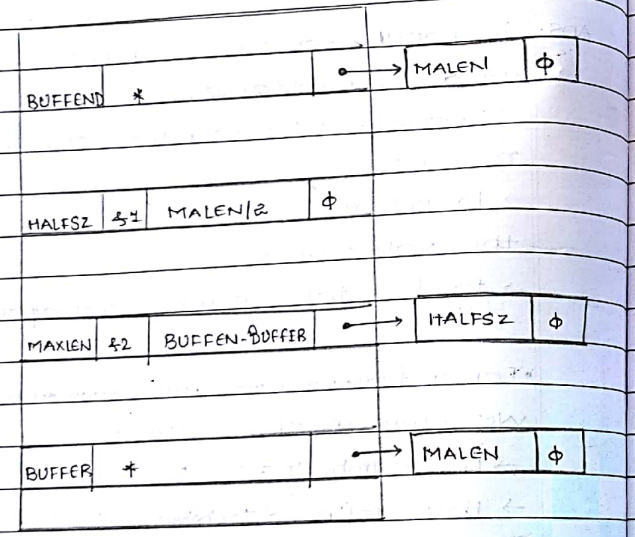
• For a forward reference in symbol definition, we store in the SYMTAB:
→ the symbol name
→ the defining expression
→ the number of undefined symbols in the defining expression
→ the undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.

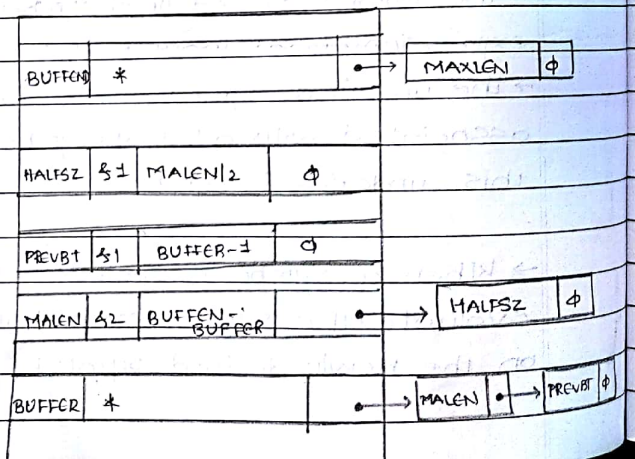
→ When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol

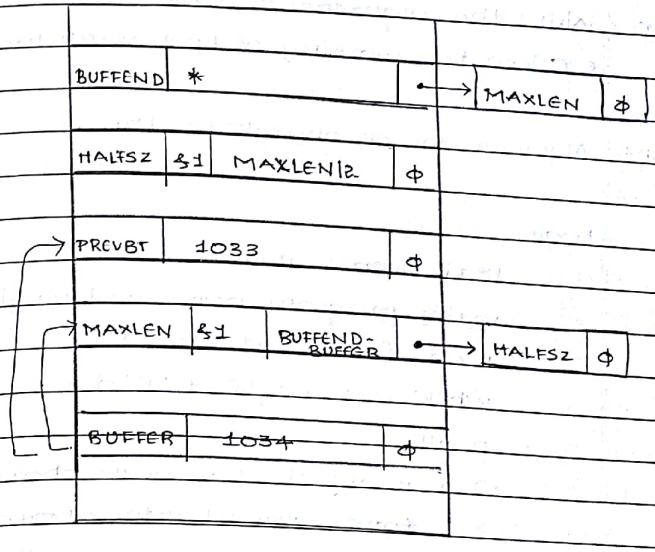
a)

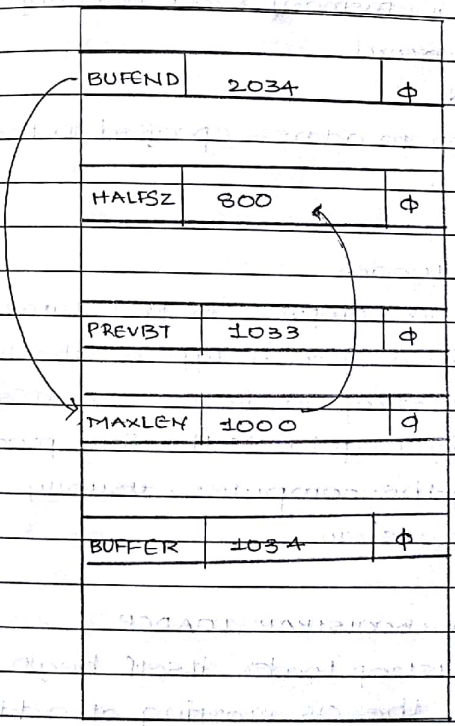

one undefined symbol in the defining expression
HALFSZ
&1
MAXLEN/2
ϕ
defining expression


MAXLEN
*
HALFSZ
ϕ
undefined symbol
depending list


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


BUFFEND
*
MAXLEN
ϕ
HALFSZ
&1
MALEN/2
ϕ
PREVBT
&1
BUFFER-1
ϕ
MALEN
&2
BUFFEN-BUFFER
HALFSZ
ϕ
BUFFER
*
MALEN
PREVBT
ϕ


BUFFEND
*
MAXLEN
ϕ
HALFSZ
&1
MAXLEN/2
ϕ
PREVBT
1033
ϕ
MAXLEN
&1
BUFFEND-BUFFER
HALFSZ
ϕ
BUFFER
1034
ϕ


BUFEND
2034
ϕ
HALFSZ
800
ϕ
PREVBT
1033
ϕ
MAXLEN
1000
ϕ
BUFFER
1034
ϕ

```
GETC    TD      INPUT       TEST INPUT Device
        JEQ     GETC        Loop until Ready
        RD      INPUT       READ CHARACTER
        COMP    #4          If character is Hex 4
                            (EOF)
        JEQ     80          JUMP to Start of
                            program just loaded
        COMP    #48         Compare to hex 30
                            (character '0')
        JLT     GETC        SKIP characters
                            less than '0'
        SUB     #48         Subtract hex 30
                            from ASCII code
        COMP    #10         If result is less than
        JLT     RETURN      10, conversion is
        SUB     #7          complete. Otherwise,
                            subtract 7 more
                            (For Hex digits 'A'
                             Through 'F')
RETURN  RSUB                RETURN to caller
INPUT   BYTE    X'F1'       CODE FOR Input
                            device
        END     Loop
```

In this the subroutine performs majority work.
→ It reads character from device F1 & convert it from ASCII code to hexadecimal digit represented by character
ex: ASCII code 0 to numeric value 0

In SIC/XE bootstrap Loader
→ main loop of the bootstrap loader keeps the address of next memory location

to be loaded in register x.
→ GETC is used to read & convert a pair of characters from device F1
→ These two hexadecimal digits values are combined into a single byte by shifting the first one left 4 bit positions and adding the second to it
→ This is done by using STCH & TIXR instruction.

11. Explain the algorithm and data structures for linking loader.

Ans. The algorithm for a linking loader is more complicated than absolute loader algorithm.
→ The input to this algorithm (loader) are control sections, that are to be linked together.
→ Control section makes an external reference to a symbol whose definition doesn't appear until later in the i/p stream.
→ For linking, it involves 2 passes
Pass 1: Assigns addresses to all external symbols
Pass 2: Performs the actual loading, relocation and linking

Data Structures for linking loader:
→ Main DataStructure needed is an external symbol table ESTAB.
1. It is used to store the name and address of each external symbol in the set of control sections being loaded.

→ Two other important variables:
1. PROGADDR: It is the beginning address in memory where the linked program is to be

loaded, value is supplied by the operating system.

2. CSADDR: It contains the starting address assigned to the control section being scanned by the loader.
→ The value is added to all relative addresses within the control section to convert them to actual addresses

ESTAB Structure:

Control Section	Symbol Name	Address	Length
PROGA		4000	
	LISTA	4040	0063
	ENDA	4054	
PROGB		4063	
	LISTB	40C3	007F
	ENDB	40D3	
PROGC		40E2	
	LISTC	4112	0051
	ENDC	4124	

Pass1: (Algorithm)

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

→ In the first pass, the loader is concerned only with Header and Define record types in the control section
→ At the end of pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.

→ Pass 2 of loader performs the actual loading, relocation and linking of the program

```
Pass 2 :
    begin
    set CSADDR to PROGADDR
    set EXECADDR to PROGADDR
    while not end of input do
      begin
        read next input 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 program}
    end {Pass 2}
```

In pass 2, end record for each control section may contain the address of the first instruction in that control section to be executed. Loader takes this as the transfer point to begin execution.

→ This algorithm can be made more efficient if a slight change is made in the object program format

→ This modification involves assigning a reference number to each external symbol referred to in a control section.

→ This reference number is used in modification records.

→ reference number 01 is given to the control section name.

→ The other external reference symbols may be assigned numbers as part of the refer record for the control section.

ex: H^PROGA^000000^000063
D^LISTA^000040^ENDA^000054
R^02^LISTB^03^ENDB^04^LISTC^05^ENDC
:
T^000020^0A^03201D^771000^050014
:

T^000054^0F^000014^FFFFF6^000F3F^000014^FEFECD

M^000024^05^+02
M^000064^06^+04
M^000057^06^+05
M^000057^06^-04
M^00005A^06^+05
M^00005A^06^-04
M^00005A^06^+01
M^00005D^06^-03
M^00005D^06^+02
M^000060^06^+02
M^000060^06^-01
E^000020

REF 01 → program name PROGA

→ The main advantage of this reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section
→ An external reference symbol can be looked up in ESTAB once for each control section that uses it.
→ The values for code modification can then be obtained by simply indexing into an array of these values.

11.2. Discuss the detailed design of a linking and relocating loader.

Ans Loaders that allow for program relocation are called relocating loaders or relative loaders. These loaders follow two mechanisms / methods for specifying relocation as part of the object program

First Method:
→ A modification record is used to describe each part of the object code that must be changed when the program is relocated.
→ If we consider SIC/XE most use relative or immediate addressing and some contains actual addressing
ex: extended format instructions

```
0006   CLOOP   +JSUB   RDREC   4B101036
0013           +JSUB   WBREC   4B10105D
0026           +JSUB   WBREC   4B10105D
```

→ These are the ones which are affected by relocation.
→ There is one modification record for each value that must be changed during relocation
→ Each modification record specifies the starting address and length of the field whose value is to be altered and then describes the modication to be performed.
→ This Modification record scheme is a convenient means for specifying program relocation but not well suited for all machine architecture.

SIC/XE relocation loader algorithm

```
begin
    get PROGADDR from operating system
    while not end of input do
    begin
        read next record
        while record type ≠ 'E' do
        begin
            read next input record
            while record type = 'T' then
```

```
begin
    move object code from record to
location ADDR + specified address
end
while record type = 'M'  // for program relocation
add PROGADDR at the location PROGADDR +
    specified address
end
end
end
```

Consider SIC program:

```
5   0000   COPY    START   0
10  0000   FIRST   STL     RETADDR   140033
15  0003   CLOOP   JSUB    RDREC     481039
20  0006           LDA     LENGTH    000036
25  0009           COMP    ZERO      280030
30  000C           JEQ     ENDFIL    300015
35  000F           JSUB    WRREC     481061
40  0012           J       CLOOP     3C0003
45  0015   ENDFIL  LDA     EOF       000024
50  0018           STA     BUFFER    0C0039
55  001B           LDA     THREE     000027
60  001E           RSUB              4C0000
```

In SIC, there is no relative addressing. When we consider above program, except RSUB all should be relocated. This may involve many modification records, which results in an object program more than twice as large as the SIC/XE program.

→ So on a machine that primarily uses direct addressing & has a fixed instruction format

it is often more efficient to specify relocation using a different technique.

→ In this technique, there are no modification records

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

Obj. program

```
H^COPY^000000^00107^
T^000000^1E^FFC^140033^481039^000036^
    280030^300015^481061^3C0003^000002^^
0C0039^00002D.
```

→ The relocation bits are gathered together into a bit mask following the length indicator in each Text record.

→ This mask is represented as three hexadecimal digits.

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

→ A bit value of 0 indicates that no modification is necessary.

→ If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.

ex: Consider lines from 5 to 55 then

bit mask is 1 1 1 1 1 1 1 1 1 1 0 0

F F C

SIC relocation loader 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
            address
        read next record
    end
    end
end.
```

Program linking:

→ The control sections could be assembled together or they could be assembled independently of one another

→ The programmer thinks the program as a logical entity that combines all of the control sections

→ From the loader's point of view, no such thing as a program. there are only control sections

that are to be linked, relocated & loaded

→ The loader has no way of knowing which control sections were assembled at the same time.

ex: 1 control section (PROGA)

LOC	Source statement			objcode
0000	PROGA	START	0	
		EXTDEF	LISTA, ENDA	
		EXTREF	LISTB, ENDB, LISTC ENDC	
0020	REF1	LDA	LISTA	03201D
0023	REF2	+LDT	LISTB+4	77100004
0027	REF3	LDX	#ENDA-LISTA	050014
		:		
0040	LISTA	EQU	*	
		:		
0054	ENDA	EQU	*	
0054	REF4	WORD	ENDA-LISTA+LISTC	000014
0054	REF5	WORD	ENDC-LISTC-10	FFFFF6
0054	REF6	WORD	ENDC-LISTC+LISTA	00003F
0054	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	

The above is example of one control section (PROGA)

consider there are 2 other control section same as the above PROGB, PROGC

for each of them, there are external symbols: LISTA, LISTB, LISTC, ENDA, ENDB, ENDC.

→ In these control sections there are some references where

REF1 to REF3 : instruction operands

REF4 to REF8 : data words values.

Consider: REF1
For the first program (PROGA) REF1 is simply a reference to a label within the program
→ It is assembled in the same way as a program counter relative instruction.
→ No modification for relocation or linking is necessary.

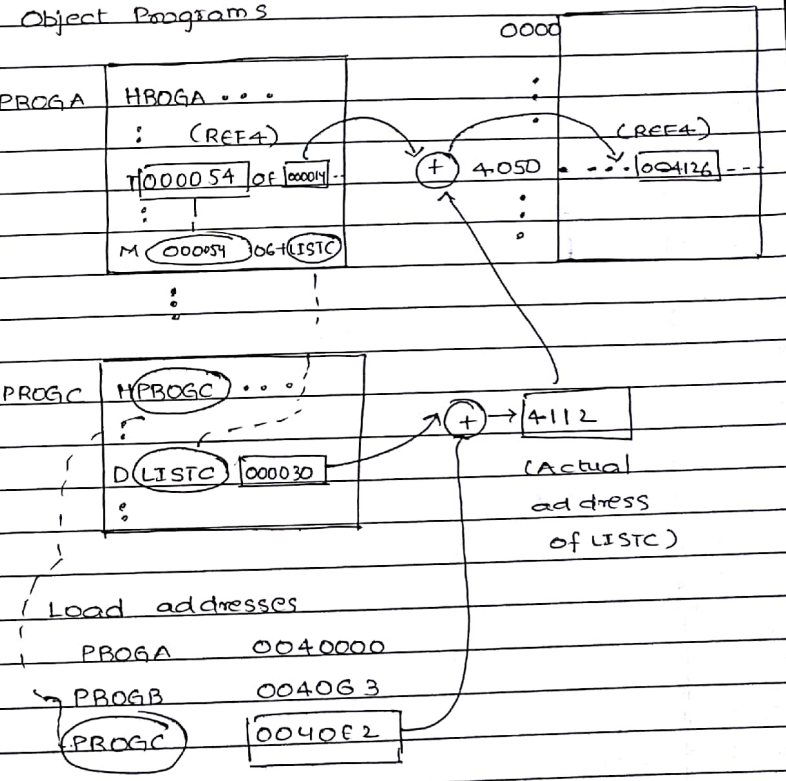
Whereas, when we consider REF2, for PROGA the operand expression consists of a external reference plus a constant.
→ The assembler stores the values of the constant in the address field of the instruction.
→ A modification record directs the loader to add to this field the value of LISTB.

obj program for PROGA
H^PROGA^000000^000063
D^LISTA^000040^ENDA^000054
R^LISTB^ENDB^LISTC^ENDC
:
T^000020^0A^032010^771000004^060014
:
M^000024^05+LISTB
M^000054^06+LISTC

Consider, REF4 : the assemble for PROGA evaluate all of the expression in REF4 except for the value of LISTC.

This results in an initial value of 0014 and one modificatio record

Relaxation & linking operations performed on REF4 from PROG A



1. What is System Software?

Ans: → It is a type of computer program that is designed to run a computer's hardware and applications programs.