System Software

Introduction to Machine Architecture

Video-01 (Overview)

Softwares are categorized into two types:-

1) System Software
2) Application Software

System softwares are used for functioning of the computer

Application softwares are written to solve particular problem.

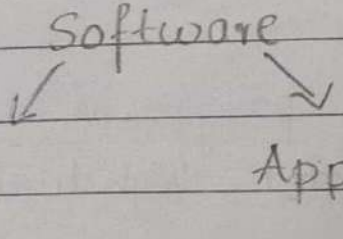
Computer consists of:

Hardware - CPU's, memory unit, I/O devices

All these are computer resource managed by operating system (which is a system software)

These are used for functioning of system.

Ex: OS, Loaders, Linkers, Text Editors, compilers, assemblers, debuggers



Ex for application softwares:

* Banking software
* Insurance software

Application software can be run on any machine and building a application software need not bothes about the machine architecture.

Internal design of microprocessor is called as machine architecture.

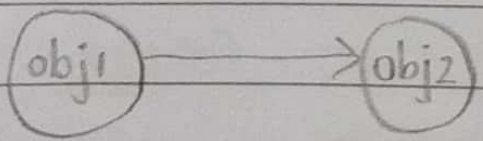
System varies from one Uprocessor to another System software depends on these processors or machine architecture.

Some microprocessors are:

8085, 8051, 8086, Pentium.

Loader → It is used to load to copy program into memory for execution after compilation

Linker → Combines two or more object feles



Text Editors → Edit programs

Compilers → Translates high level language to

machine level language
Assembler - translates assembly level language to machine level language
Debugger - used by developers for debugging

Chapter 01 - Machine Architecture

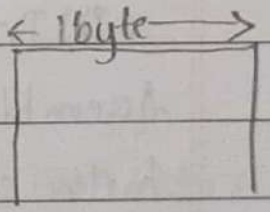
Video - 02

SIC - Simplified Instructional Computer

SIC is 8-Up (8-bit Up).
SIC XE (XE - Extra Equiment)
SIC - hypothetical / non-existing machine

Video - 03 (SIC Machine Architecture)

Characteristics:

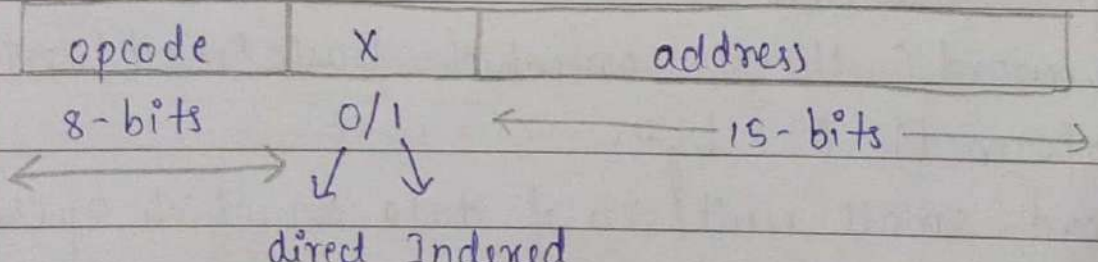


* SIC supports 15-address bits
* Address space - 2^{15} address
* Byte addressable
* One word (the size on which basic read / write is performed) is 3-bytes.
  word - small unit / size of data on which operations are performed.
* One word is stored in 3-consecutive memory locations as used as each address stores a byte i.e byte-addressable

* It supports 5 registers of 24-bit each (one word)
A - Accumulator (must be used as one operand)
X - Index register (for indexing array elements)
L - Linkage register (for subroutine call & return address)
PC - Program counter (points to next instruction to be executed)
SW - Status word (stores flags information)

* Data formats
→ Supports integer and characters
Integer - 24 bits
character - 8 bits

* Instruction format (2 types)
i) Direct mode
ii) Indexed mode.
Assembler converts instructions to 24-bits & flag decides
Instruction is stored in 24-bit format mode



Address calculation depends on X:
If X=0 address is fetched from address data directly that becomes effective addres

If X=1 the address is added to X (similar to arrays) that becomes effective address

Video-04 (SIC: Addressing modes, Instructions, I/O)

⟹ Addressing modes specify how operands are accessed.

Address modes

↙ ↓

Direct (address)

Index (EA = X + address)

Data can be from memory, registers and immediate

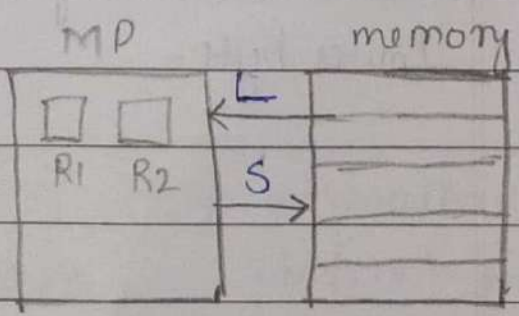
⟹ Instruction

It supports LDA (loading data) and SDA (storing data)

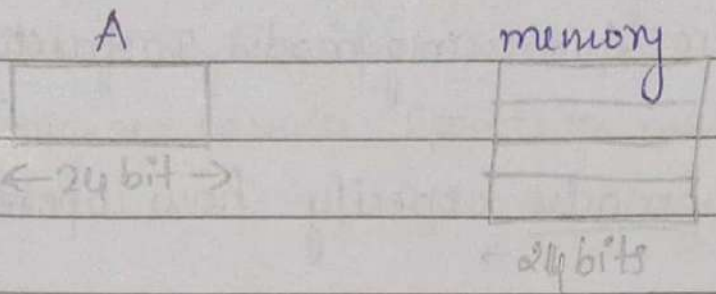
Arithmatic op'ns: ADD, MUL, DIV, SUB

LDA → Memory to µp

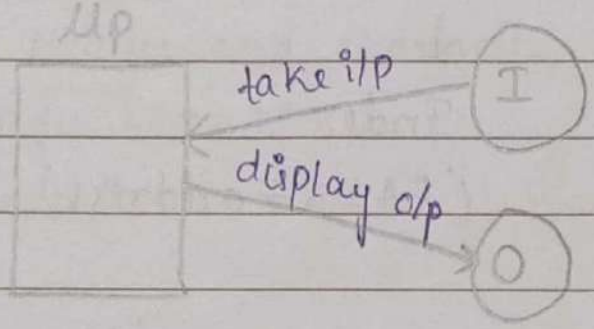
STA → µp to Memory



Operations are always performed using accumulator



⇒ I/O



TD - Test device (check I/O device is ready/not)
RD - Read from input device
WD - Write data to output device

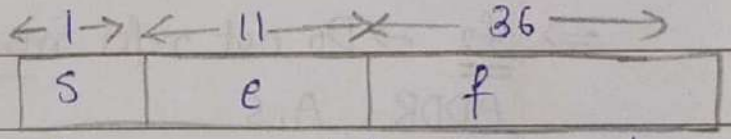
Only one byte of data can be read or written to o/p
The data is sent to A but it's 24-bits so sent to lower byte and passed to o/p from lower byte.

Video-

Architecture of SIC/XE

* Address bits is 20 = 1 Mbyte.
* Address space is 2^{20} address
* Since memory space is higher than SIC m/c more no. of instructions and instruction formats are possible.
* More no. of addressing mode.
* One word = 24-bits
* Various registers support 9 registers all of 24-bit except f:
  A - accumulator, B - Base Register (24-bit)
  X - indexing, S - } General purpose working registers
  L - linkage, T - }
  PC - programcounter, F → 48-bit floating point registers.
  SW - Status word

* Data formats.
  Integer - 24-bits
  floating points - 48-bits
  character - 8 bits



sign flag — exponential (+ve/-ve) — fractional

$f * 2^{(e-1024)}$

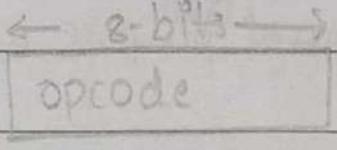
⇒ $e = 11 = 2^{11} = \frac{2048}{2}$ (+ve/-ve) → -1024 to +1023

⇒ f - normalized form
. before higher value
.002 → 0.2

Instruction formats

(how instruction is presented in memory)

* f1 - 1 byte
* f2 - 2 byte
* f3 - 3 byte
* f4 - 4 byte.



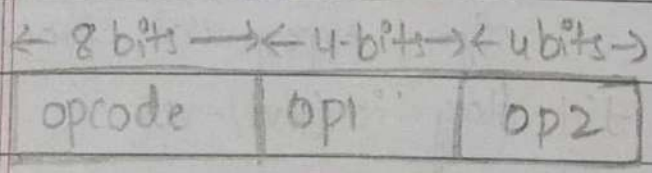
⇒ f1

Ex: RSUB → return from subroutine

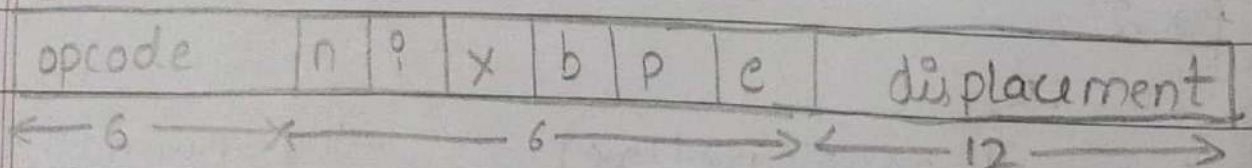
The instructions where no data is referred or only when opcode is used. (↳ memory/ register)

⇒ f2 → Do not refer memory, only refer registers

ADDR A,S



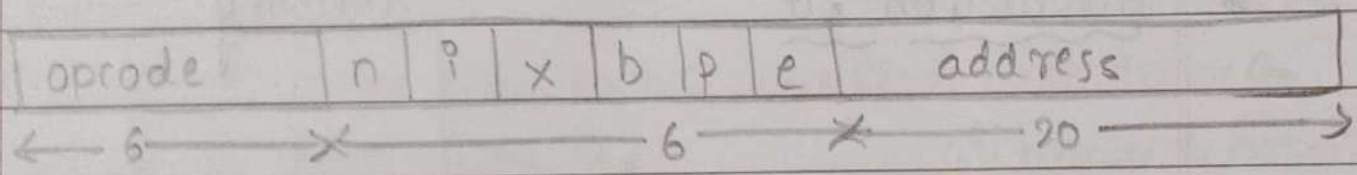
⇒ f3



n - indirect addressing mode Displacement
i - immediate " " ⇓
x - indexed " " Relative Address
b - base relative " "
p - program relative " "
e - distinguish f3 & f4

f3 → e = 0
f4 → e = 1

=> f4



Video-
Addressing modes in SIC/XE

n=0, i=1 (Immidiate addressing)
n=1, i=0 (Indirect addressing)
n=1, i=1 } No immediate/indirect
n=0, i=0 }

p=1 → program counter relative AM EA = [PC] + disp
x=1 → then EA = [x] + address /disp
b=1 → base relative addressing mode EA = [B] + disp

Displacement → 0-4095
In base disp is +ve
But in PC relative as PC can be -ve/+ve.
disp -2048 --- +2047.

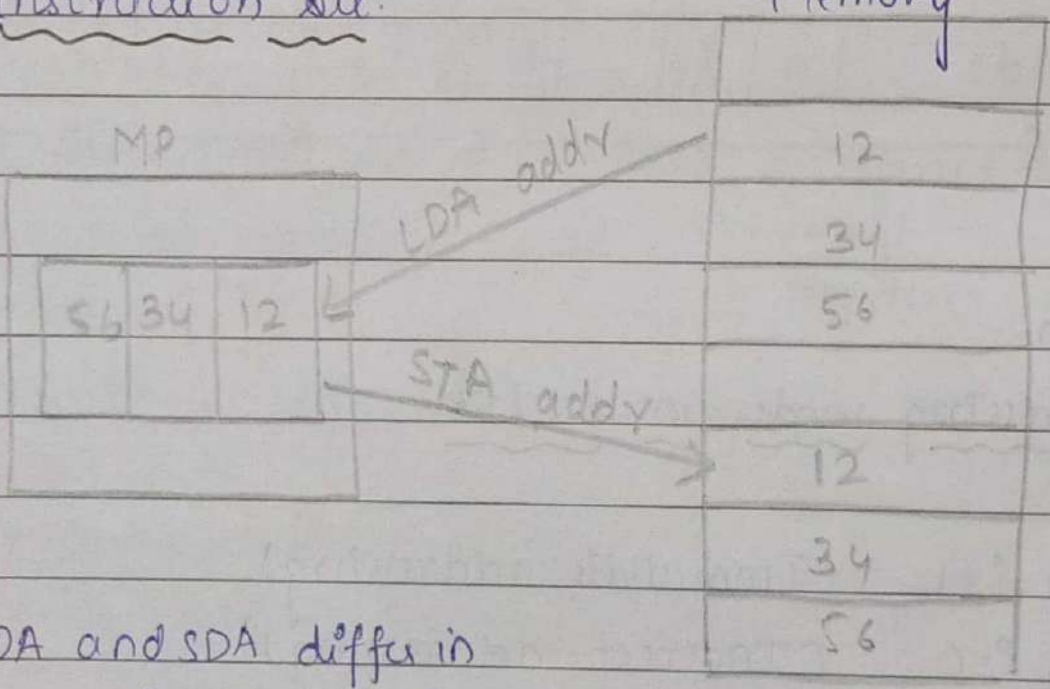
* Base
disp → 0-4095

* PC Relative
disp → -2048 to 2047

* JMP back
JMP next

* Instruction Set.

⇒



LDA and SDA differs in directions of data flow

⇒ In arithmatic one operand must be A and another memory address/register and result is stored back in A

* I/O

RD - Read 1byte from i/p

WD - Write 1byte to o/p

Addition

r1 ← r1 + r2

ADDR S, A (2 byte)

(A) ← (A) + (S)

RMO src, dest

(2 byte)

Difference b/w instruction and pseudoinstructions.

Instructions	PseudoInstructions
→ Operations	→ Assembler directives
→ Translated to machine code	→ Gives instructions to assembler & not translated to m/c code
⇒ executable statements	→ not executable

To define variables we use RESB, RESW

To define constants we use WORD, BYTE

START addr.

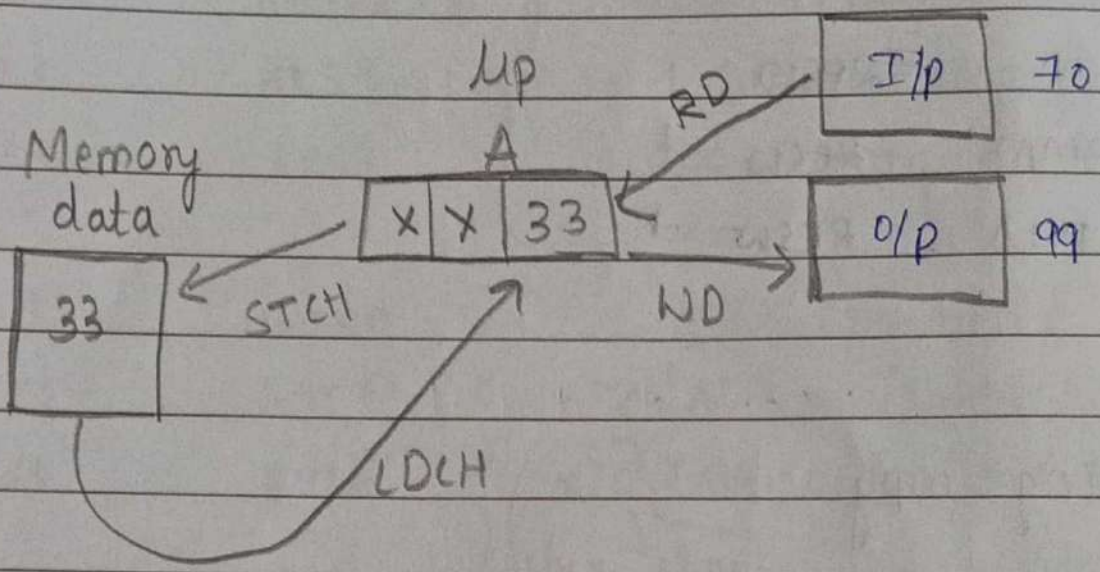
END.

Ex: START, END, WORD, BYTE,

RESB - Reserved byte

RESW - Reserved word.

* I/o code



↗ hex

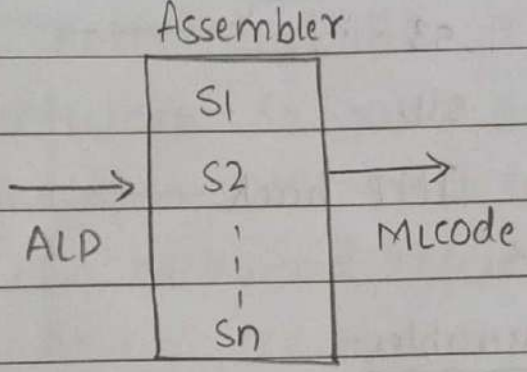
INDEV	BYTE	X'70'
OUTDEV	BYTE	X'99'
DATA	RESB	1

```
back1   TD  INDEV        (if equal set then device
        JEQ back1           not ready & polling happens)
        RD  INDEV
        STCH DATA

back2   TD  OUTDEV
        JEQ back2
        LDCH DATA
        WD  OUTDEV
```

Assemblers - 01

→ It is a system software that translates ALP to ML code.



* The features that change from one machine to another are machine dependent feature.
* Some assembler features are m/c independent
* Types of assembler:
  1) One-pass " (pass refers to no. of times a
  2) Two-pass " assembler goes through the
  3) multi-pass " program)

→ translation from ALP to ML code by scanning ALP once

→ ALP is scanned for complete translation

Functions of Assembler - 02

What is the need for two passes?

Consider the program below

```
        S1          → ✓
back:   S2          → ✓
        JMP next    → ✗   ← forward ref
        S3
next:   S4
        JMP back    → backward reference
```

functions of assembler:

→ Understanding symbols.

There are two types of symbols:

1) Variables (ex: SUM, SUM1)
2) labels to (ex: next, back) statements.

→ During first pass assembler assigns address for each statement and updates the symbol table with symbol name & address value. It processes assemble directives like RESW. Validates instructions using optable. Symbol table.

SUM -1012
SUM1 -1015
next -1009

→ During pass2 some operations are performed. statements are converted m/c code.

// Need?

In the example program consider assembler starts it function and starts converting each statement to machine level code. So S1 in m/c form, S2 in m/c form, JMP next this is an example of forward reference. So next is not yet assigned with address so JMP next can't be translated to m/c code. In backward reference like JMP back there will be no issues as back is already assigned with address.

Due to presence of forward references we need two-pass assembler.

Consider the ALP to be converted.

Pass I

			m/c equivalent	
1000		JMP next (3)	70	1012
1003		ADD SUM (3)	99	1015
1006		SUB SUM1 (3)	77	1018
1009	next:	MUL XYZ (3)		

1012	SUM	RESW	1	(3)
1015	SUM1	RESW	1	(3)
1018	XYZ	RESW	1	(3)

Each statement is assigned with addresses.
The opcodes are replaced

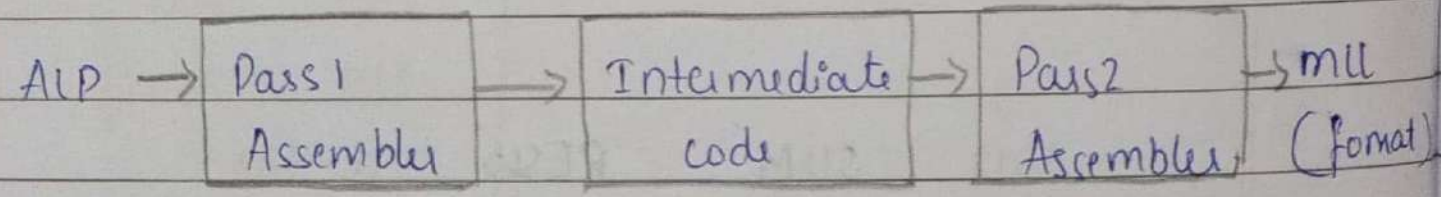
→ During pass 2 m/c equivalent is found.
→ During pass 2 it uses op tables it considers the opcodes for instruction record in op table op table has operation mnemonic & opcode for it

OPTABE

ADD : 70
SUB : 99
MUL : 77

→ Some processes some assemble directives (EOF → 4748 49)
→ If some constant EOF is then its internal representation is ASCII is found.
→ Constants to binary representation.

Assemble data structure-03.



address, symbols, assemble directive

translate assembler directive

Datastructures used:

1) OPTAB (instruction and m/c equivalent
2) SYMTAB.

Variable: LOCCTR - Keep track of statements.

OPTAB is constant and used by both pass 1 & 2
SYMTAB is initially empty.
Pass 1 updates SYMTAB.

RSUB - 1 byte.

Format

Header Record
Text Record

Code:

. → comment

RSUB → format1 / format3

START → Specify name and starting address for program

END → End of source program and optionally specify the first executable instruction

BYTE → Generate character / hex constant, occupying no. of bytes as needed to represent the constant

WORD → Generate one word integer constant.

RESB → Reserve the indicated no. of bytes for a data area.

RESW → " " " " words " "

Code for i/p device:

```
INPUT   BYTE   X'F1'
```

→ For addressing var WORD Integer always takes 3 bytes.

→ Only if it is byte then check operand bytes

→ If RESB operand times *

→ If RESW operand * 3.

→ RSUB - 4C0000

OPTAB

Mnemonic	Opcode		
LDX	04	RD	D8
LDA	00	TIX	2C
ADD	18	STX	10
STA	0C	LDCH	50
RSUB	4C	WD	DC
TIX	2C	LDT	74
JLT	38		
STL	14		
JSUB	48		
COMP	28		
JEQ	30		
J	3C		
LDL	08		
TD	E0		

Opcodes:

WORD → dired integ constant in hex

BYTE → internal rep of characters in ASCII ('hex')
→ hex constant

ASCII ↔
- 0 - 9 (48 - 57)
- A → 65 Z - 90
- a - 97 z - 122

In hex ASCII

A - 41	a = 61	0 - 30
Z - 5A	z = 7A	9 - 39

→ For RESW no opcode.
→ For RESB no opcode.

→ For SIC !!! (Careful)

opcode (8)	x	address (15)

0/1
↳ when x is used

→ Object program will be loaded in memory for execution which is in the form of records.
→ 1) Header:
C1: H
C2-7: Program name.
C8-13: Starting address in hex.
C14-19: length of object program.
→ 2) Text:
Col 1: T
Col 2-7: Starting address for object code in this record (hex)
col 8-9 Object code length in bytes (hex)
col 10-69 " " in hex (2 colum / byte of object co

3) End Record

Col- E

col2-7 : Address

A - separate fields visually

Pass1: → Assign SA

→ Assign addresses to all statements in the program

→ Save the values (addresses) assigned to all the labels for use in Pass2

→ Preprocess the assembler directive.

→ Writh heading LOC, and at end write. program length = LOC - SA.

→ Show the addresses with codes as intermediate code.

Pass2

→ Write opcode.

→ Process assemble directives not done during pass1

→ write object program and the assembly listing

→ circular array implement

T 1, 6, 6, 6

H 1, 6, 2, - - -

E 1, 6

* Pass1 - optab for lookup & validation of opcodes in source program.
* Pass2 - translate opcodes to machine language

* In SIC, both can be done in pass1.
In SIC/XE in pass1 - optab to check the instruction length for incrementing LOCCTR.
In pass2 - to tell us which instruction format to use in assembling

* In SIC/XE registe to registe addressing is used
COMP ZERO → COMPR A,S.
SIC

STCH BUFFER,X
↳ indicate indexing takes 3 bytes
CLEAR X,S → 2 bytes.

Register addresses

Name	address
A	0
X	1
L	2
B	3
S	4
T	5
F	6
PC	8
SW	9

* For PC relative
  disp = TA - PC
* For base relative (if TA - PC > -2048 and +2047)
  disp = TA - B

* Possible
  → x + PC
  → x + Base
  → PC + immediate (when symbol is in statement

* An object program that contains the information necessary to perform this kind of modification is called relocatable program.

How to write M records in relocatable program?

* M - modification Record

Col 1 - M

col 2-7 Starting location of address field to be modified relative to beginning of program

col 8-9 Length of the address field to be modified (in half bytes)

Device - 8 bit

Register - 4 bit

ISA-01

1 a) Because of forward references, i.e a reference to a label that is defined later in the program. So we can't process such statements ∴ we don't know the address that will be assigned to label later in the program.

Datastructures:

In Pass I

→ OPTAB → for lookup and validation of operation codes in source program.

→ OPTAB is organized as hash table with mnemonic operation code as key

→ OPTAB is mostly static

→ LOCCTR - location counter variable that is used in assignment of addresses.
(Initialized with address in START statement/zero)

→ SYMTAB is generated during pass I. It has label name and address pairs for each label, as well as flags to indicate error condition
Organized as hash table with labels as keys

Pass2

→ OPTAB → to translate and get the machine equivalent of opcodes in source program

→ SYMTAB → to get address of labels

→ OPTAB → to tells which instruction format to use

Pass 2

→ Assemble / translate instructions

(translate opcodes to machine equivalend codes and labels to addresses)

→ Preprocess the assemble directive not done in pass1

→ Generate values defined by BYTE, WORD etc.

→ Write object program and assembly listing

(b)

2a) Pass 1

Intermediate file.

LOC				
0000	SWAP	START	0	—
0000		LDX	ZERO	074000
0003		+LDB	#ZERO	691 01F62
		BASE	ZERO	—
0007		LDT	#4000	7 50FA0
000A	BACK	LDA	ALPHA,X	03A01J
000D		STA	TEMP	0F4003
0010		LDA	BETA,X	03AFAF
0013		STA	ALPHA,X	0FA00C
0016		+LDA	TEMP	03101F65
001A		STA	BETA,X	0FAFA5
001D		TIXR	T	B805
001F		JLT	BACK	3B2FE8
0022	ALPA	RESB	4000	—
0FC2	BETA	RESB	4000	—
1F62	ZERO	WORD	0	000000
1F65	TEMP	RESW	1	—
1F68		END		—

3b) Assuming SIC as all are 3 bytes.

```
ADD     START   1000
        LDA     ZERO
        STA     INDEX
ADDLP   LDX     INDEX
        LDA     ALPHA,X
        ADD     BETA,X
        STA     GAMMA,X
        LDA     INDEX
        ADD     THREE
        STA     INDEX
        COMP    K300
        JLT     ADDLP
        END
```

How do I know addlp is the label JLT label check start address then append label.

3c) Program relocation according to new start address and modification of m/c codes only in format4 instruction

Ch-02

1) BASE label
NOBASE
START
END
BYTE , RESB
WORD , RESW

2)

3) Keep track of address while assigning them to statements.

4) SORT - try

5)

6) when SA is changed during execution LDA 100 where this 100 is absolute memory location we can move the contents to 100 to process space in execution

SIC max.

```
MAXIN START 0.
ZERO   WORD 0
INDEX  RESW 1
THREE  WORD 3
ALPHA  RESW 100
MAX    RESW 1
K100   WORD 100

   LDA ZERO
   STA INDEX
   LDX INDEX
   LDA ALPHA, X
   STA MAX
   LDA INDEX
   ADD THREE
   STA INDEX

L1: LDX INDEX
   LDA ALPHA, X
   COMP MAX
   JGT RELOC
L2: LDA INDEX
   ADD THREE
   STA INDEX
   COMP K100
   JLT L1
```

```
RELOC: STA MAX
       J L2
```

SIC/XE

```
MAX     RESW    1
        LDS     #3
        LDT     #100
        LDX     #0
ALPHA   RESW    100
        LDA     ALPHA,X
        STA     MAX
        ADDR    S,X

L1:     LDA     ALPHA,X
        COMP    MAX
        JGT     RELOC
L2:     ADDR    S,X
        COMPR   X,T
        JLT     L1

RELOC:  STA     MAX
        J       L2
```

LOC				OPCODE
0500	SUM	START	500H	-
0500	FIRST	LDX	#0	050000
0503		LDA	#0	010000
0506		LDS	#3	6D0003
0509		LDT	#1500	+505DC
050C		+LDB	#TABLE2,	655016BB
		BASE	TABLE2.	[illegible]
0510	LOOP	ADD	TABLE, X	1BA01H
0513		ADD	TABLE2, X.	1BC000
0516		ADDR	S, X	904 1
0518		COMPR	X, T	A015
051A		JLT	LOOP	3B2FF3
051D		+STA	TOTAL	0F5005D6
0521		RSUB.		4C0000
0524	COUNT	RESW	1	
0527	TABLE	RESW	1500	
16BB	TABLE2	RESW	500	
1C97	TOTAL	RESW	1	
1C9A		END	FIRST.	

Program length = 179A

SYMTAB

SUM	0500
FIRST	0500
LOOP	0510
COUNT	0524
TABLE	0527
TABLE2	16BB
TOTAL	1C97