System software

↓ Computer (Hardware)

↓ Variety of Programs

SS ↓ act as interface btw system and user

Types of SS

1. OS
2. Translators
   - ↳ compiler
   - ↳ Interpreter
   - ↳ Assembler

High level language → Machine level language

Assembly language → asembler → Machine language

- * whole
- * Faster
- * Intermediate field

line by line

Easy detect errors

S S	Application S/w
* set of Variety Programs for computer Hardware	* set of variety programs for specific task

SIC (simplified Instruction set computer)

MRDA I³

- M → memory
- R → Register
- D → Data format
- A → Addressing modes
- I → Instruction format, Instruction sets, I/O instructions & operations

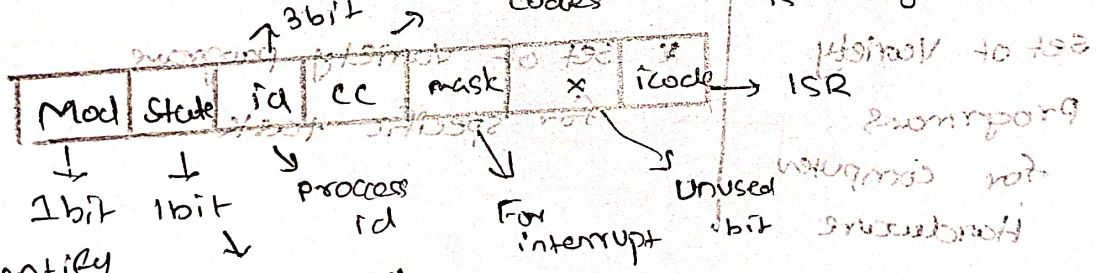
Memory

* 2^{15} bytes
* byte addressable memory
* 1 byte = 8 bits
* 1 word = 3 bytes = 24 bits

Registers (all are 24 bits)

id	Name	Purpose
0	A	Accumulator — Arithmetic calculation
1	X	Index — Address calculation
2	L	Linkage — For subroutine Jumps
8	PC	program counter — Addres of next Instruction
9	SW	Status word

Status word



Data Format

Integers → 24 bit

-ve nos → 2's complement

char → 8 bit ASCII

No Floating point

Instruction Format

8 bit	1 bit	15 bit
opcode	x	Address

Instructions (Operations) [What we have to perform]

eg: MOV, Add

Addressing modes

x => 0 direct Addressing mode

x => 1 indexed

TA = Address + [x]

target address

SIC / XE

XE => extra equipment

↘ Upward compatible

Memory

* $M = 2^{20}$ bytes
  = 1MB
* 20 bit address

Registers

* 9 registers
  5 of SIC and extra 4

id	Name	
3	B	Base register
4	S	general purpose
5	T	general purpose
6	F	Floating point accumulator

8 registers => 24 bits

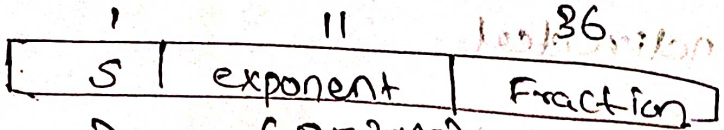
Floating point => 48 bits

Data Format

Integer → 24 bit
char → 8 bit

For negative => 2's complement form

Float → 48 bit



(0,1) (0-2047) (Bits 0-1)

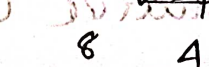
S = 0 → +ve
S = 1 → -ve

Instruction Format

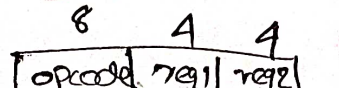
4 types

type-1 → 1 byte (= 8 bit)
type 2 → 2 byte
type 3 → 3 byte
type 4 → 4 byte

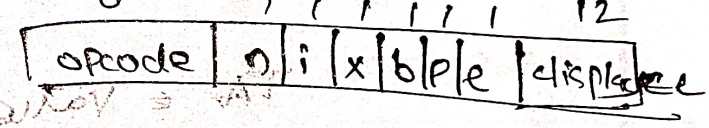
8 bit

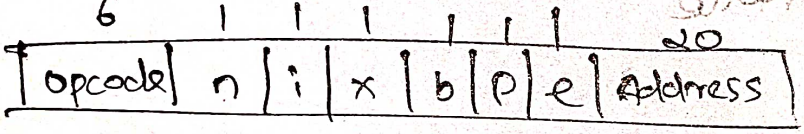


eg: SIO
TIO



source destination





~~Addressing modes~~

~~eg) Indexed~~

n → Indirect
i → Immediate
X → Index
b → base relative
p → pc bit
e → extended bit

bit	use
x=1	indexed addressing mode
e=0	type 3
e=1	type 4
b=0 p=1	Pc relative addressing mode
b=1 p=0	base relative addressing mode
b=0 p=0 b=1 p=1	direct ~~relative~~ address mode

n=0 i=1 immediate addressing modes

n=1 i=0 indirected

n=0 i=0 } directed
n=1 i=1 }

Addressing modes

target address Calculation

1) Base relative (b=1 p=0)

TA = [B] + displace

range of displace

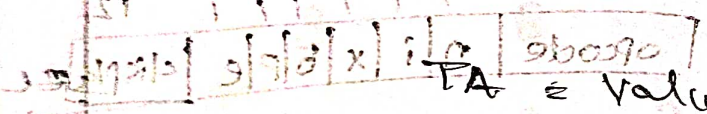
0 ≤ disp < 4095

2) Pc relative (b=0 p=1)

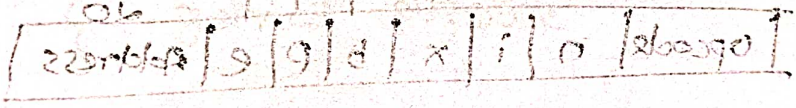
TA = [PC] + displace

-2048 ≤ disp ≤ 2047

3) n=0 , i=1



TA = value



4) n=1 i=0

TA = [operand]

Instruction sets

LDA B , STA B

ADDF, SUBF, MULF, DIVF ⇒ type-4

* It support register arithmetic ADDR, SUBR, MULR, DIVR ⇒ type-2

RMO → Register move

SVC → Supervisor code

RSUB, JSUB

I/O

Extra TIO, SIO, HIO } For type-I ⟹ TD, RD, WD ⊂ type-3

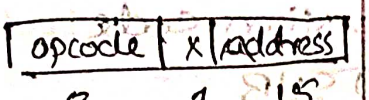
RSUB & JSUB } type-3 (normally) type-4

Q. perform a comparitive study of architech of SIC and SIC/XE

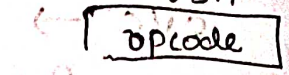
Component	SIC	SIC/XE
memory	2^{15} bytes	2^{20} bytes upward compatible
Register	5 registers of each 24 bits A, X, L, PC, SW	9 registers 8 → 24 bits 1 → 48 bits A, X, L, PC, SW B, S, T, F (48 bit)
Data Formats	Integers (24 bits) char (ASCII) -ve nos (2's complement) → No floating point	Integer 24 bits char (8 bit) Float 48 bit

Instruction format

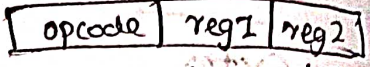
only I type



type 1

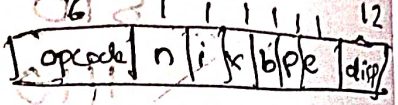


type 2



Source destination

type 3



Type 4

Addressing modes

x=0 → direct

x=1 → Indexed

TA = Addr + [x]

Base relative (b=1, P=0)

TA = [B] + disp

PC relative (b=0, P=1)

TA = [PC] + disp

n=0, i=1

TA = value

n=1 i=0

TA = [operand]

Instruction Set

1) LDA, STA, LDX, STX
2) ADD, SUB, DIV, MUX
3) COMP
4) JLT, JEQ, JGT
5) JSUB & RSUB

1) LDA, STA, LDX, STX
2) ADD, SUB, DIV, MUL
3) COMP
4) JLT, JEQ, JGT
5) JSUB & RSUB
6) LDAB, STAB
7) ADDF, SUBF, MULF, DIVF → type 4
8) ADDR, SUBR, MULR, DIVR → type 2

I/O	1) TD 2) RD 3) WD	1) TD 2) RD 3) WD 4) TIO 5) SIO 6) HIO } type-1 RSUB, JSUB } Type 3 (normal) Type -4

SIC programming

→ Data movement operations

Q. Integer → ALPHA ← 5
char → C1 ← 'Z'

```
1.  LDA    FIVE
2   STA    ALPHA
3.  LDCH   CHARZ
4.  STCH   C1
    .
    .
    .
ALPHA   RESW    1
C1      RESB    1
FIVE    WORD    5
CHARZ   BYTE    C'Z'
```

SIC/XE

```
LDA   #5
STA   ALPHA
LDA   #90
STCH  C1

ALPHA   RESW   1
C1      RESB   1
```

// Immediate addressing mode

Arithmetic operations

Q ALPHA + INCR - 1 Assign to BETA

GAMMA + INCR - 1 " to DELTA

SIL

```
LDA   ALPHA
ADD   INCR
SUB   ONE
STA   BETA

LDA   GAMMA
ADD   INCR
SUB   ONE
STA   DELTA
 .
 .
 .
ALPHA   RESW   1
INCR    RESW   1
BETA    RESW   1
```

(ALPHA+INCR)

```
GAMMA   RESW  1
DELTA   RESW  1

ONE     WORD  1
```

SIC/XE

```
        LDS   INCR
        LDA   ALPHA
ADDR    ADDR  S,A
        SUB   #1
        STA   BETA

        LDA   GAMMA
        ADDR  S,A
        SUB   #1
        STA   DELTA
        :
ALPHA   RESW  1
INCR    RESW  1
BETA    RESW  1
GAMMA   RESW  1
DELTA   RESW  1
```

String copy

write a SIC

Q STR1 - 'TESTSTRING' copy to STR2

SIC

```
        LDX   ZERO
MOVECH  LDCH  STR1,X      // STR1[0]
        STCH  STR2,X
        TIX   ELEVEN      // compare and Increment
        JLT   MOVECH          with label
        .
        .
        .
STR1    BYTE  C'TEST STRING'
STR2    RESB  11
ZERO    WORD  0
ELEVEN  WORD  11
```

Write a SIC/XE

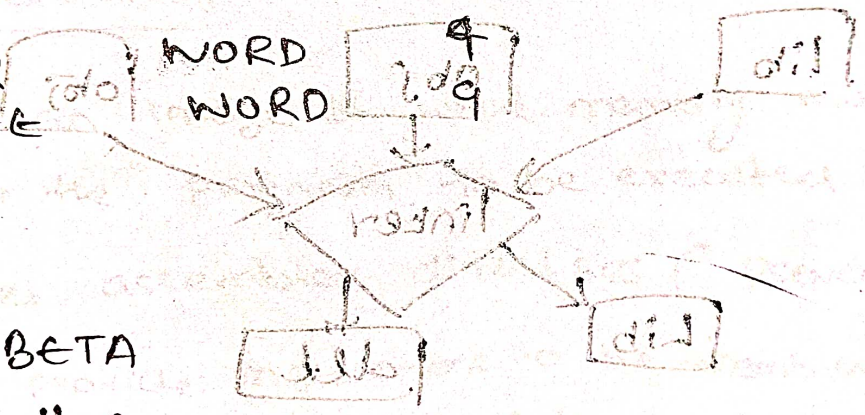
```
        LDX   #0
        LDT   #11
MOVECH  LDCH  STR1,X
        STCH  STR2,X
        TIXR  T
        JLT   MOVECH
        .
        .
        .
STR1    BYTE  C'TEST STRING'
STR2    RESB  11
```

Q write a sequence of instructions for both SIC and SIC/XE to set

ALPHA = 4 * BETA - 9

SIC

```
     LDA   BETA
     MUL   FOUR
     SUB   NINE
     STA   ALPHA
     :
ALPHA  RESW  1
BETA   RESW  1
     :
FOUR   WORD  4
NINE   WORD  9
```



SIC/XE

```
     LDA   BETA
     MUL   #4
     SUB   #9
     STA   ALPHA
     :
     :
ALPHA  RESW  1
BETA   RESW  1
```

Assembler

Assembly language to machine language

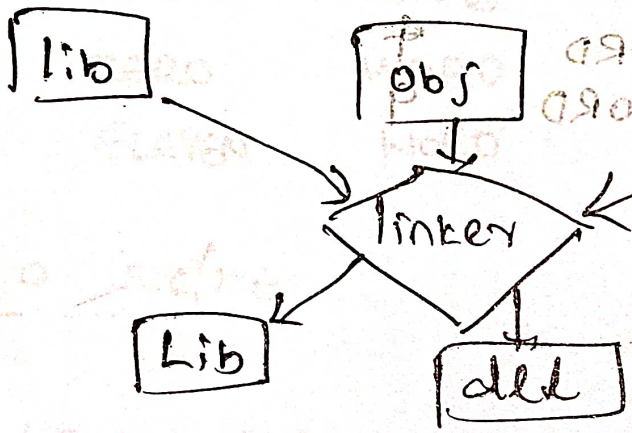
∞ object code

Loader

A program load from secondary memory to main memory

Linker

a program that takes one or more objects generated by compilers & assembler them into a single executable program



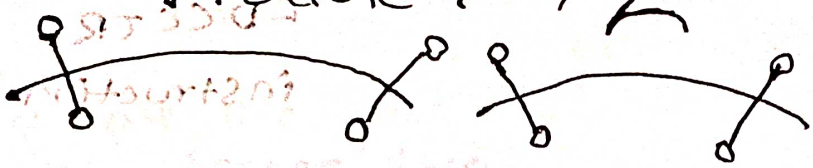
types

a) Linking Loader ⇒ linking & relocation operations all directly in to main memory for execution

b) Linkage editor ⇒

c) Dynamic Linker ⇒

Module → 2



Fundamental Functions

* Generate the machine language
  → Translate mnemonic operation code to machine code
* Assign addresses to symbolic labels used by the programmer

Additional Functions

* Generate an image of what memory must look like for the program to be executed
* Interpret assembler directives (pseudo-instructions)
  → provide instructions to the assembler
  → do not translate into machine code
  → might affect of the object code

  eg: START, END, RESW, RESB

- RDREC — read
- WRREC

Input ⇒ Assembly code (soure program)

output ⇒ Assembly Listing
object code

Intermediate files ⇒ Assembly listing

LDCCTR

instruction Length

Forward Reference problem

Types of assembler

1 pass assembler ⇒ It reads & scans source code once and produce object code

2 pass assembler ⇒ It scans twice source code and produce object code

(line by line translate)

10 1000 FIRST STL RETADR 141033

95 1033 RETABR RESW 1

Q Pseudo code for Pass 1

Pseudo code for 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
        initialise 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 mode)

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}

Object program format

3 type of records

→ Header record ⇒ starting address & Length

→ Text record ⇒ translated instructions & data of the program together with an indication of the addresses where these are to be loaded

→ End record

End of record

Format

Header

col. 1 H

col 2-7 program name

col 8-13 Starting address (hex)

col 14-19 Length of object program in bytes (hex)

Text

col-1 T

col 2-7 Starting address in this record (hex)

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

col 10-69 object code represented in hexa decimal

End

col.1 E

col 2-7 Address of first executable instruction (hex)

(~~END~~ END Program_name)

Pass 1 (define symbols)

* Assign addresses to all statements in the program
* Save the values (address) assigned to all labels for use in pass 2
* Perform some processing of assembler directives
* Including those for address assignment, such as BYTE & RESW etc.

Pass 2 (assemble instructions & generate object program)

* Assemble instructions
* Generate data values defined by BYTE, WORD etc
* perform processing of assembler directives not done by Pass 1
* write object program & assembly listing

Data structures of assembler

major 2 internal data structures

1) operation code Table (OPTAB)
2) Symbol table (SYMTAB)

Location counter (LOCCTR) → Variable

OPTAB

Content

* mnemonic, machine code (instruction format, length) etc

Characteristic

* Static table

Implementation

* hash table, easy for search
* Mnemonic operation code as key,

→ In more complex assemblers, contains information like instruction formats & length.

SYMTAB

content

* name, value (for each label)
* Flag ⇒ Indicate error condition.
* Type or length

characteristic

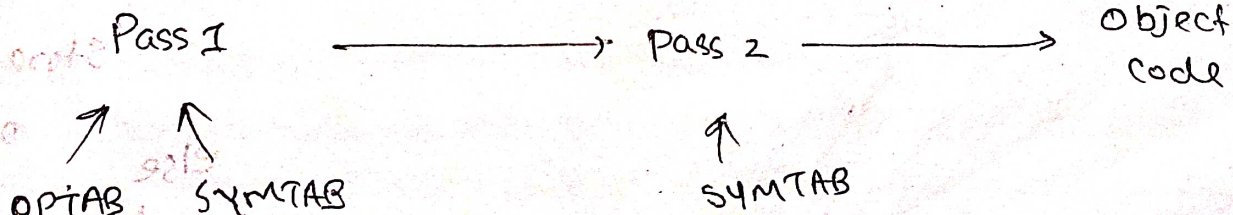
* Dynamic table (insert, delete, search)

Implementation

* hash table, non-random keys, hashing function.

LOCCTR

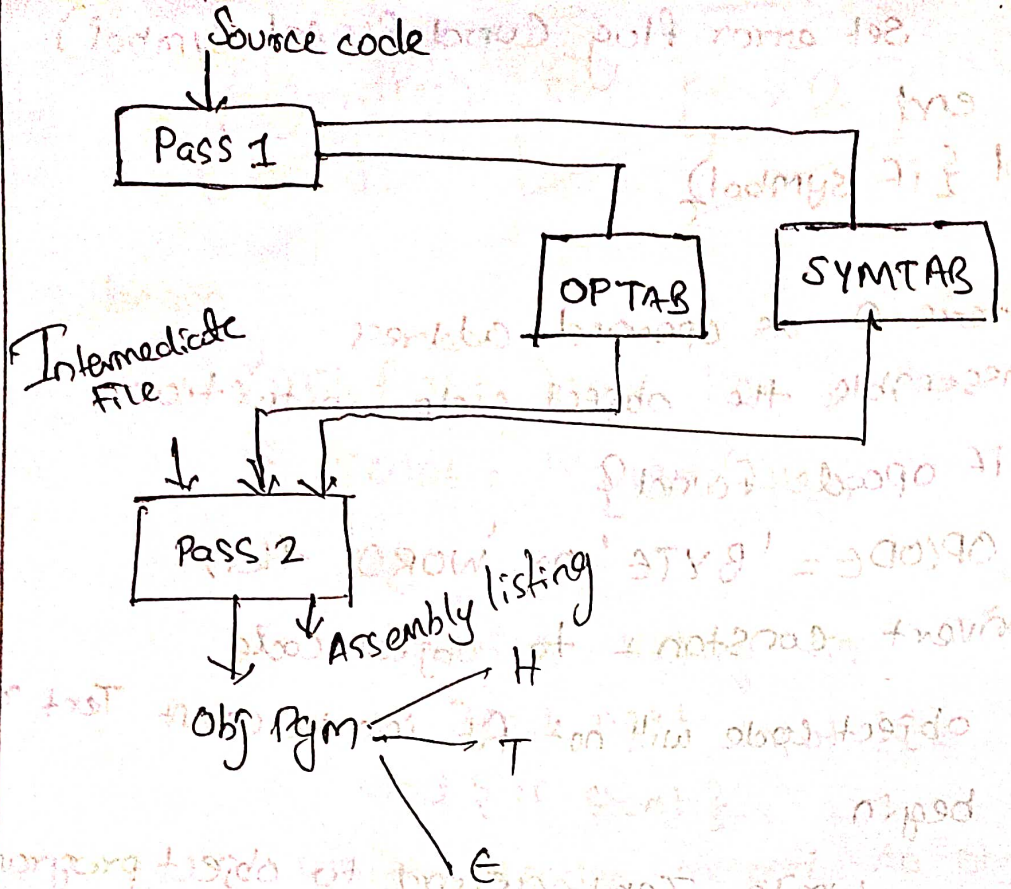
* used to help in the assignment of address
* initialized starting addresses of the program
  then after processed the length of the assembled instruction is added to the LOCCTR (for next instruction)



Algorithm for pass 2

```
begin
    read first input line { from intermediate file}
    if OPCODE = 'START' then
        begin
            write listing line
            read next input line
        end {if start}
    write Header record to object program
    initialize first Text record
    While OPCODE ≠ 'END' do
        begin
            if this is not a comment line then
                begin
                    search OPTAB for OPCODE
                    if found then
                        begin
                            if there is a symbol in OPERAND
                                field then
                            begin
                                search SYMTAB for OPERAND
                                if found then
                                    store symbol value as
                                        operand addre
                                else
                                    begin
                                    store 0 as opera d
                                        address
```

```
                Set error flag (undefined symbol)
            end
        end {if symbol}
    else
        store 0 as operand address
        assemble the object code instruction
    end {if opcode found}
else if OPCODE = 'BYTE' or 'WORD' then
    convert constant to object code
    if object code will not fit into the current Text record
                                                    then
        begin
            write Text record to object program
            initialize new Text record
        end
        add object code to Text record
    end {if not comment}
    write listing line
    read next input line
end {while not END}
write last Text record to object program
write End record to object program
write last listing line
end {pass 2}
```



Q. write down the relevance of SymTab and OPTAB in Pass 1 and Pass 2 in Two pass assembler.

~~In Pass~~

$1E \Rightarrow 1 \times 16^1 + 14 \times 16^0 = 30$

In pass 1

~~dupl~~ set ~~error flag~~ duplicate symbol

Symtab ⇒ Check found ⇒

not found ⇒ insert symbol to SYMTAB

Ans)

* In pass 1, the OPTAB is used to look up and validate the operation code in the source program
* In pass 2, it is used to translate the operation code to machine code.

* During pass 1 : labels are entered into the symbol table along with their assigned addresses (LOCCTR) as they are encountered

* All the symbols address value should get resolved at the pass 1

* During pass 2 : symbols used as operands are looked up the symbol table to obtain the address value

Machine - Dependent Features of Assembler

1. Instruction format & Address modes
2. Program relocation

4 type

Format -1

opcode

* 1 byte = 8 bit

eg : T10, H10, S10

Format -2

opcode	reg1	reg2
8	4	4

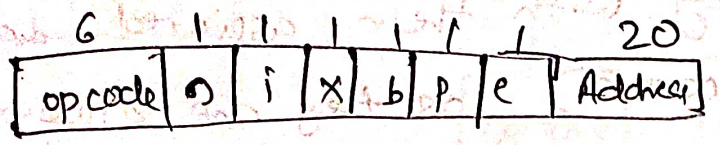
eg: ADDR S,T

Format -3

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

eg: RSUB
JSUB

Format-4



e = 0 Format 3
e = 1 Format 4

n → Indirect
i → Immediate
x → Indexed
b → base Relative
p → PC relative
e → extended bit

extra

+ ⇒ Format 4

#~~#~~ ⇒ immediate

@ ⇒ indirect

X ⇒ indexed

PC relative

0017	J	CLOOP
Loc	opcode	operand

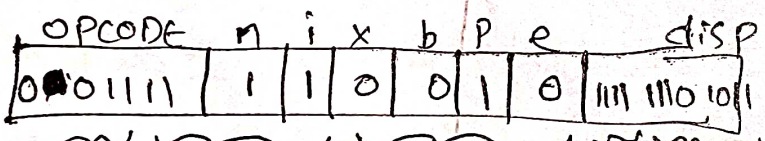
OPTAB

J	3C
STCH	54
LDT	74

SYMTAB

CLOOP	0006
BUFFER	0036
LENGTH	0033

→ Format 3 (NO + symbol)



3 F 2 F E B

3C

0011 1100

TA = [PC] + disp

0006 = 001A + disp

0017 + 3

001A

disp = 0006 − 001A

= −0014 12 bit

0006
001A

= 0000 0001 0100

= 1's compl

001A
0006
14

= 1111 1110 1011

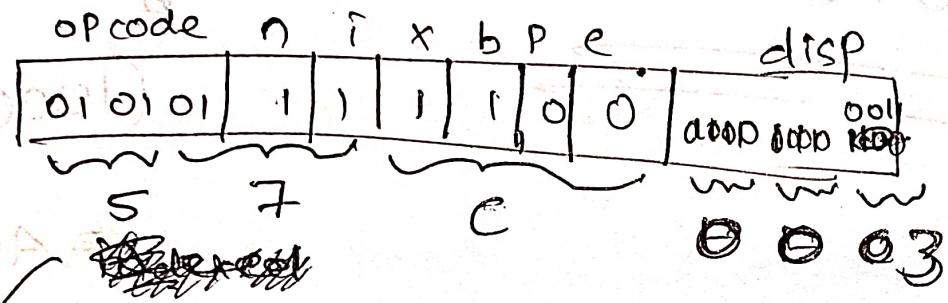
Base relative Addressing Mode

STCH BUFFER, X

Base address = Length

→ Format 3

4C 1012



54

0101 01

TA = [B] + disp

0036 = 0033 + disp

disp = 0003

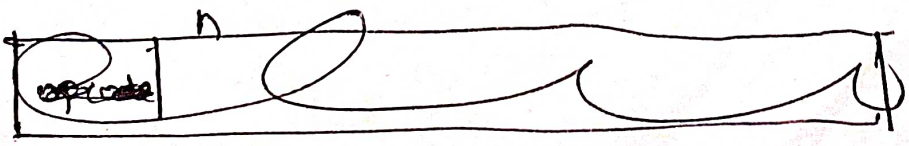
57C003

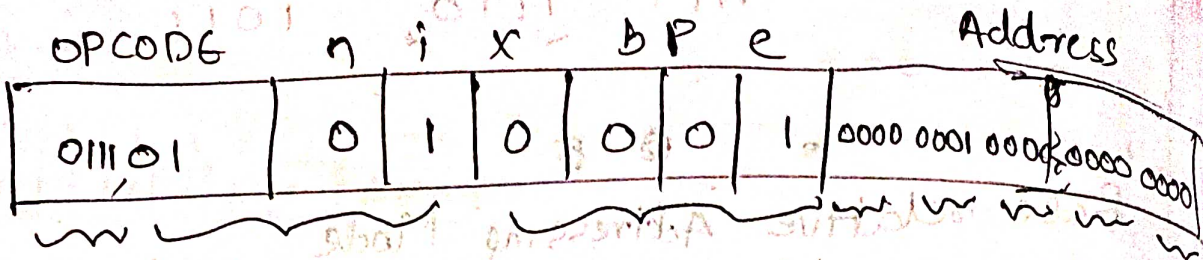
0000 0000 0011

Immediate addressing mode

Format 4

103C + LDT #4096





7 5 1 0 1 0 0 0

011101

01

75101000

(1000)$_{16}$

= 4096

0000 0001 0000 0000

HW

002A ↗ @ RETADR

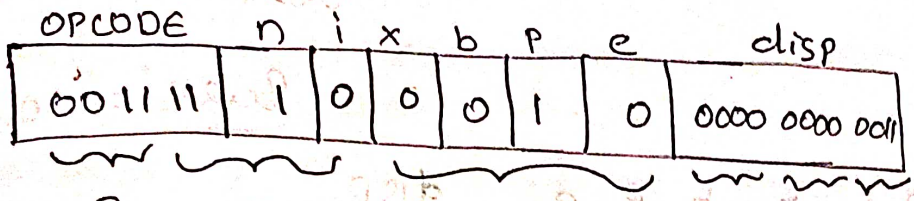
J = 3C

RETADR = 0030

indirect ✓

PC relative ✗

→ Format 3



3 E 2 0 0 3

OPTAB

J	3C

SYMTAB

RETADR	0030

3C

= 0011 1100

TA = [PC] + disp

0030 = 002D + disp

002A +
3

002D

disp = 0030 − 002D

= +003

0000 0000 0011

0030
002D

0003

3 E 2003

Indexed addressing mode

LDCH BUFFER X

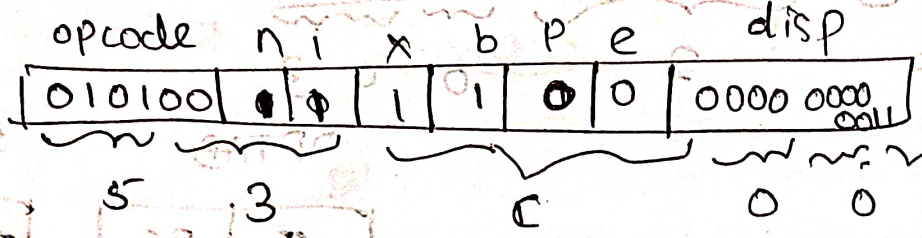
LOCH = 50

BUFFER = 0036

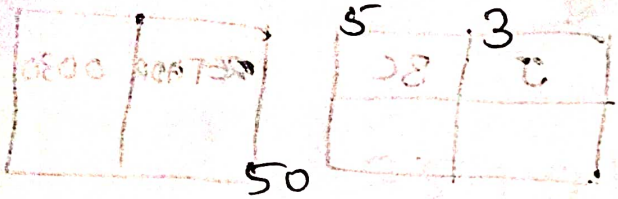
Base = 0033

Base relative

→ Format 3



5 3 C 0 0 3



TA = [B] + disp

= 0101 0000

0036 = 0033 + disp

disp = 003

5 3 C 0 0 3

Program Relocation

→ Relocatable code

→ Need → memory too small → Swap in & swap -o

→ Examples

→ Modification Record

Modification Record (M)

X ——— X ——— X ——— X

Col 1 — M

Col 2 – 7 — Starting location of the address relative field to be modified to the beginning of the program (Hexadecimal)

6 bit

Col 8 – 9 — Length of the address to be modified in half bytes

eg:- 0006 +JSUB RDREC

M^000007^05

Length of address = 20/4 = 5

Q TIXR T

TIXR → B8

Format 2

OPCODE	R_1	R_2
10111000	0000	0101

B405

Q perform the hand assembly of the following program

	STRCP2	START	1000
1000	FIRST	LDT	#11
1003		LDX	#0
1006	MOVECH	LDCH	STR_1,X
1009		STCH	STR_2,X
100C		TIXR	T
100E		JLT	MOVECH
1011	STR_1	BYTE	C'TEST STRING'
101C	STR_2	RESB	11
1027		END	FIRST

101D - 1000 / 001D

Slno	opcode	n	i	x	b	p	e	displabel	object code
2	0111 01	0	1	0	0	0	0	0000 0000 1011	75000B
3	0000 01	0	1	0	0	0	0	0000 0000 0000	050000
4	0101 00	1	1	1	0	1	0	0000 0000 1000	53A008
5	0101 01	1	1	1	0	1	0	0000 0001 0000	57A010
6	1011 1000	0000	0101						B805
		R_1	R_2						
7	0011 11	1	1	0	0	1	0	1111 1111 0100	3B2FF4

Tutorial

Label	Mnemonic	Operand
WREC	START	1050
	LDX	ZERO
	LDT	LENGTH
WLOOP	TD	OUTPUT
	JEQ	WLOOP
	LDCH	BUFFER, X
	WD	OUTPUT
	TIX	LENGTH
	JLT	WLOOP
	RSUB	
OUTPUT	BYTE	X 'OS'
ZERO	WORD	0
BUFFER	RESWB	400
LENGTH	RESWB	2
	END	WREC

generate machine language equivalence of each instruction.

LDX	04
LDT	74
TD	E0
JEQ	30
LDCH	50
WD	DC
TIX	2C
JLT	38
RSUB	4C

Ans

	WREC	START	1050
1050	1050	LDX	ZERO
1060	1060	LDT	LENGTH
1063	1063	TD	OUTPUT
1066	1066	JEQ	WLOOP
1069	1069	LDCH	BUFFER, X
1072	106A	WD	OUTPUT
1075	106B	TIX	LENGTH
1078	107E	JLT	WLOOP
	1081	RSUB	
	1084	OUTPUT	BYTE X'05'
		ZERO	WORD 0
		BUFFER	RESB 400
		LENGTH	RESB 2
		END	WREC

Slno	opcode	n	i	x	b	p	e	displadd	objcode
2	000001	1	1	0	0	0	0	000000000000	070
3	011101	1	1	0	0	0	0		

Object program

H^STRCP2^001000^00002B 27 →

T^001000^1B^750008^050000^53A008

53A010^B805^3B2FF4

E^001000

5×3 = 15
7×2 = 2
17
(11)16

16|400
16|25
1

(190)16

1070
190
1260
11F6

Module-3

Machine Independent Assembler Features

1. Literal
2. Symbol defining statements
3. Expression
4. program blocks
5. control sections & program linking

1 Literal

* an operand is called a literal because the value is literally in the instruction
* Literal is identified with prefix '='.

eg: LDA =C'EOF' Directly assign

literal pool ⇒ *collection of literals in the program
*Normally placed End of the program

LTORG ⇒ directive used to create literal pool in between the program

* LITTAB ⇒ used to store literals and corresponding value

2 Symbol defining statements

Assemble directive used is EQU (for assign

Syntax ⇒ symbol EQU value

eg: MAXLEN EQU 4096

LDT #MAXLEN

* used to improve the program readability, avoid using magic numbers, make it easier to find & change constant values.
* It is enter to symtab

ORG (origin) directive

* used to indirectly assign values to symbols
* ORG value (format)
* ORG To change value of the location counter pointer register.

3 Expression

→ Absolute Expressions
→ Relative Expressions: eg: MAXLEN EQU BUFEND-BUFFER

Illegal expression ⇒ BUFFER + BUFEND

3 - BUFFER

3 * BUFEND

4. program block

USE ⇒ It means use a program block

// Dependent each other

CDKTA

5. control section

Same as program block but is independent of each other.

6. program linking

* Instructions in one control section may need to refer to instructions or data located in another control sections. (Like external variables used in C language)

* reference btw control sections => external reference

External references

Two assembler directives are used

EXTDEF (external definition) => For define

EXTREF (external references) => for linking

Define record

Col 1	D
Col 2-7	Name of external symbol defined in this control section.
Col 8-13	Relative address of symbol within this control section (hexadecimal)
Col 14-73	Repeat information in col 2-13 for other external symbols

Reference record :

col-1	R
col 2-7	Name of external symbol referenced to in this control section
col 8-73	Names of other external reference symbols.

Modification record (revised)

col 1	Same as previous Modification record (M)
col 2-7	
col 8-9	
col 10	Modification flag (+ or -)
col 11-16	external symbol whose value is to be added to or subtracted from the indicated field

Loaders & Linkers

Loading : brings the object program into memory for execution

Relocation : modifies the object program so that it can be loaded at an address different from the location originally specified

Linking : combines two or more seperate object programs and supplies the information needed to allow

references btw them.

1. loading
2. Relocation
3. Linking.

Types of Loaders

1. Absolute loader :

1 & 2 ⇒ Relocation / relative

1 + 2 + 3 ⇒ linking loader

Absolute loader

* All functions are accomplished in a single pass

* → check header record for verify that the correct program has been presented for loading

→ Each Tex record read, object code it content is moved to indicated address memory

→ End record is encountered, the loader jumps to the specified address to begin execution of the loaded program