⇒ Lecture Notes Contd. (Dated :- 19-1-22)

⇒ System Software

- User convenience (UC)
- Efficient use (EU)
- Non-interference (NI)

⇒ Answer to which system is better depends on many factors

① Program development & Production environment

Program development: you are developing a SW & you are trying to fix its errors

Production environment: you have developed the SW. you're using it in diff. scenarios without modification

Compilers	Interpreter
- Translates HLL → ML (Ready for execution)	- does not generate ML program
- Analyses each statement in HLL & accordingly generates ML code	- Analyses the program P & directly carries out the desired computation.
- Phase 1: program is compiled Phase 2: ML instructions are generated Code generation	- Keeps track of sequence of 'P'
- Loop:- Analyses statement inside loop only once	- Loop:- analyses the statement everytime ↳ larger overhead ⇒ less efficient
	- During program development → better to use interpreter than in production environment

⇒ Debugger ⇒ system software program.

- Stepwise
- Interactive debugging ⇒ you can set (break points) in your program ☆ Helps

② Making software portable

⇒ Portable :- for every computing environment, instead of making it again & again, portability is desirable.

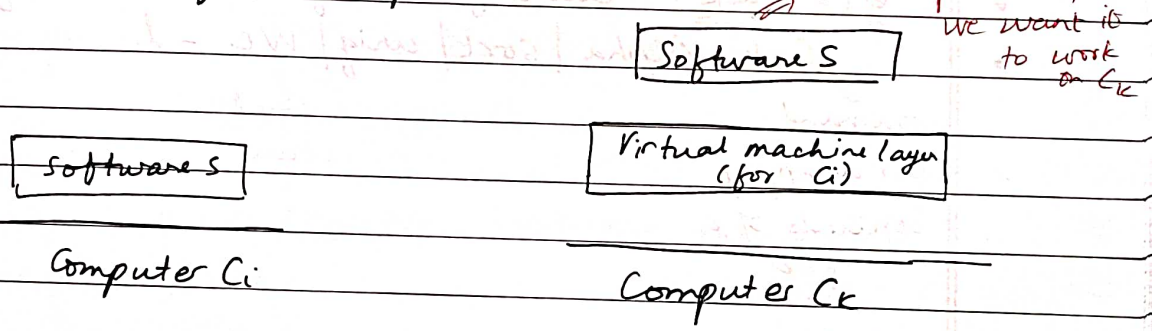
⇒ If program makes use of special features provided by OS or computer ⇒ difficult to make it portable.

⇒ Nowadays, HLLs are almost free of OS.

⇒ Virtual machine concept

- Convenient
- VM ⇒ Abstract computer that has all the desired features of a computer.

developed on C_i, we want it to work on C_k



- Software S now sees it as C_i only.
- Software layer as shown ⇒ Portability using VM but causes overhead

- Pascal programming :- 1970s ⇒ for systematic programming
- VM was designed for Pascal specifically.
- Pascal compiler ⇒ generates code for Pascal VM

P-code

- Java programming :- JVM ⇒ Compiled = Java byte code

Portable, that's why Java is so popular.

③ Realizing benefits of the internet

- Programs located on remote computers & integrate the result

- Download any unknown program but in doing so ⇒ danger of interference.
- Web server ⇒ dynamic data

④ Treating programs as components

⇒ To provide the facility of reuse.

- No one does programming from scratch ⇒ eg. deep learning ↓ taking code from GitHub.
- Provided by scripting languages. → Unix shell script
- eg:- Task: Counting unique names in file 'Alpha'
- (Cat) Alpha | sort | uniq | wc -l

(Cat): command for displaying contents of a file

wc: word count.

→ PERL

→ PYTHON

→ TCL/TK

→ Visual Basic

- Class file (OOP class)

⑤ Embedded system environment

⇒ Modern Computers

⇒ Important requirement in embedded system ⇒ should have quick response ↳ real-time response requirement.

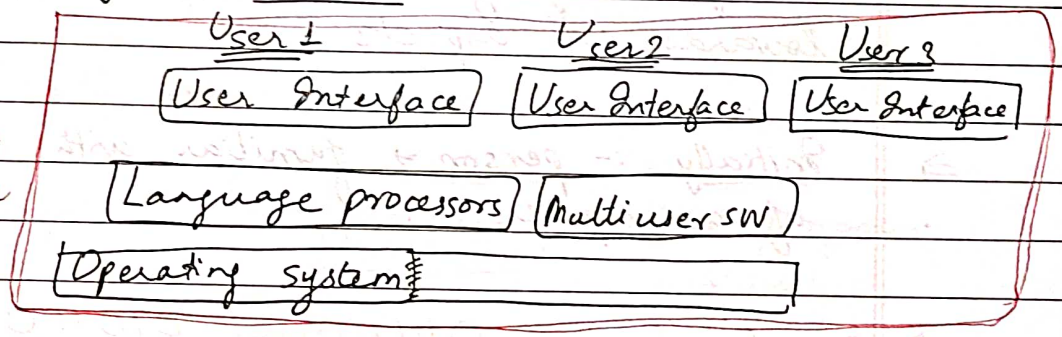
⇒ Application ⇒ using cross-compiler ↓ special compiler that runs in computer rich in resources.

aka cross platform software development

⑥ Dynamic specification, flexibility & adaptive software
↳ difficult to handle as it requires special handling

⇒ Lecture Notes Contd. (Dated :- 20-1-22)

⇒ Views of System software.



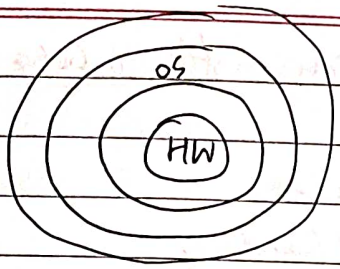
⇒ Computational needs.

- HLL , Assembly language
  ↳ Compiler ↳ Assembler
  Loader , Linkers
  Debugger
  Interpreter

⇒ System centric view

- Efficient usage of computer system
- Effective utilization
- Aiming for user convenience
- Also ensures non-interference.

PTO ⇒



⇒ Layered view ⇒ helps in abstracting more & more inner details (HW) ↳ outer layer may not know what is in the innermost layer

⇒ Moving away from hard aspects of computing system towards soft aspects

⇒ Initially :- person ⇒ familiar with hardware aspects of a computer.

⇒ • 0110 (Operation Code (Opcode)) 011100 (Operand 1) 010110 (Operand 2) ⇒ ML
Data

⇓ lesser details?

• Next level ↓ ML code to Mnemonic code

opcode	operand 1	operand 2
ADD	A	B
SUB	C	D
MUL	E	F

↳ aka mnemonic code

⇓ ⇓

• highest level :- C = A + B;
X = Y + Z - P;
⇓
Statement

So we're going to see assembly language code in this course.

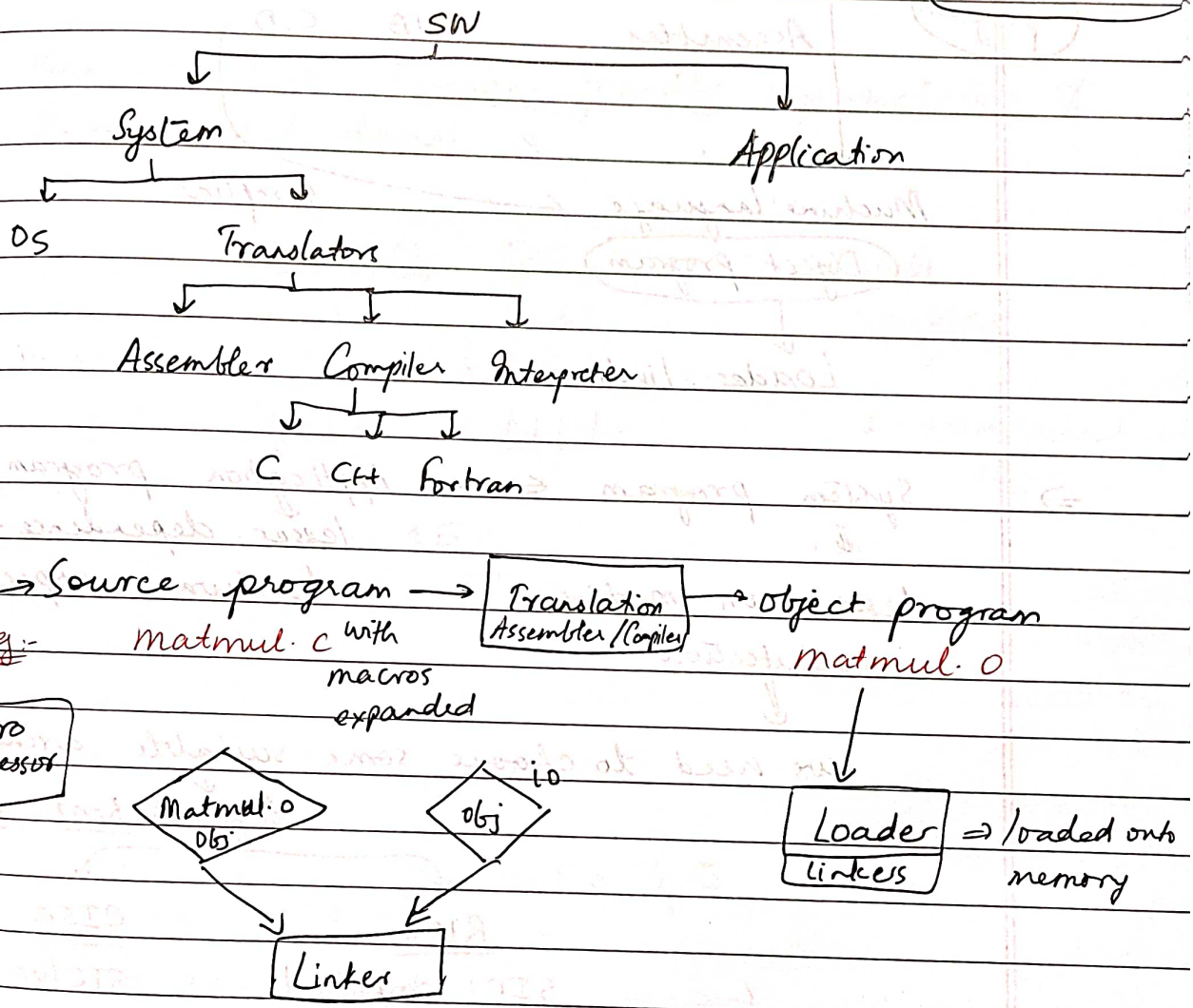
• harder

M/c code
↑ Translation assembler
Mnemonic code
↑ Translation compiler
Statement → HLL

↓ softer

⇒ H/W (1) + Translators (2) + Library programs + Utility programs I/O (3)

⇒



⇒ Source program → Translation (Assembler/Compiler) → Object program

eg:- matmul.c with macros expanded → matmul.o

⇒ Lecture Notes Contd. (Dated :- 21-1-22)

⇒ CS251 ⇒ things to be covered :-

- Assemblers
- Loaders
- Linkers
- Macroprocessor
- Utility programs

⇒ Assembly language ⇒ mnemonic codes

PTO ⇒

⇒ Assembly language ⇒ mnemonic codes

ADD A, B
SUB C, D

Source program → Assembler → Machine language ← Compiler

Machine language → Object program → Loader, Linkers

⇒ System program ⟷ Application program

System program ⇓ depend on machine architecture ⇓ we need to choose some suitable architecture ↓ hypothetical machines

Application program ⇓ lesser dependence on hardware aspects

- RISC
- CISC

SIC: simplified instructional computer

SIC/XE: Extra Equipment

- Aspects we'll be focusing on for these :-
  - Memory
  - Registers
  - Data formats
  - Addressing modes
  - Instruction set
  - I/O

⇒ Hexadecimal representation

- base -16 system
- positional numeral system

- 16 distinct symbols

0 1 2 3 4 5 6 7 8 9 A B C D E F

- Here we adapt human-friendly representation of binary coded decimal.

0000 0000 → 4-bits → nibble

0 0

1111 1111 → 1-hexadecimal digit

f f

15

0001 0101

⇒ SIC

⇒ Memory

- 8-bit bytes
- Word ⇒ 3- consecutive bytes
  - ↳ 24-bit
- Byte addressable
- Address of a word is given by lowest numbered byte
- Total of 32768 bytes ⇒ really small memory.
  - ↳ 2^{15}
  - ↳ 15-bit addresses

⇒ Registers

- 24-bit length
- 5 registers
- each register has a mnemonic symbol.

PTO ⇒

- Registers :-

Mnemonic	Number	
A	0	Accumulator → used Arithmetic operations
X	1	Index register To manage sequence of memory → array
L	2	Linkage register Jump to subroutine (JSUB)
PC	8	Program counter
SW	9	Status Word ∥ contain CC Condition code

XXX0 _____
XXX1 _____
XXX2 _____
XXX3 _____
XXX4 JSUB SUM
XXX5 _____
_____
_____
SUM _____ : _____ SUM
XXX10 _____
_____
XXX1A _____ END

eg:- CMP P, Q
JLT

1	0	0	0	
LT	EQ	GT	Z	N

↳ if P < Q

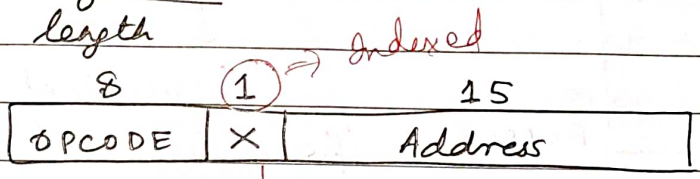
⇒ Data formats

(1) ≡ ~~Integers only supported~~
- 24-bits
- 2's complement

(2) ASCII codes
- 8-bit

⇒ Instruction formats

- 24-bit length

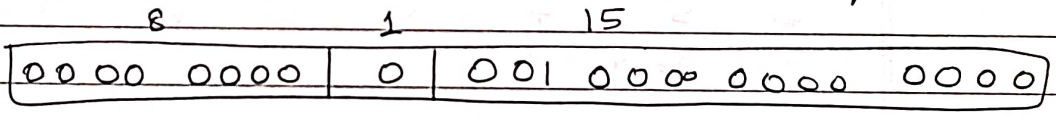


⇒ Addressing modes

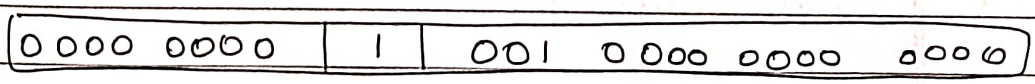
- 2 addressing modes
- X-flag shows which addressing mode is being used

①	Direct	X = 0	TA = address (Target address)
②	Indexed	X = 1	TA = Address + (X)

eg:- ① LDA TEN — Load from address, symbol TEN with



② LDA TEN, X — Indexed.



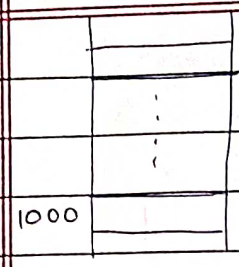
$$EA = 1000 + (X)$$

Lecture Notes Contd (Dated - 25-1-22)

⇒ System software depends on machine architecture

⇒ ③ STCH BUFFER, X ⇒ Indicates there's a need for indexed addressing ⇒ directive or info passed onto assembler from programmer to user

this is a mnemonic instruction



STCH BUFFER, X

⇓

hexadecimal opcode = 54

Assembler

OPCODE

0101	0100	1	001	0000	0000	0000
5	4		1	0	0	0

1 → via wire using indexed addressing

address of buffer = 1000

⇓

0101	0100	1001	0000	0000	0000
5	4	9	0	0	0

decoder sees nibble-by-nibble → & sees 549000

⇒

5	4	9	0	0	0
STCH		Address			

⇓

Indexed

- STCH = store the character ⇒ available in accumulator
- EA = 1000 + [X]
- If the x flag is not set :-

0101	0100	0	001	0000	0000	0000

⇓

[X] will not be added ⇒ it will be written again & again at 1000 only, since there's no offset.

⇒ Instruction set

- (Integer) arithmetic instructions
  - ADD
  - SUB
  - MUL
  - DIV

ADD ALPHA

⇓

(A) ← (A) + (ALPHA)

accumulator already has this

ALPHA

- DELTA = ALPHA + BETA

you would think ⇒ ADD DELTA, ALPHA, BETA ⇒ this is a complex instruction ⇒ not supported by SIC

↙ Solution

- Load & store
  - LDA , STA ⇒ Load & store accumulator
  - LDX , STX ⇒ Load & store index
  - LDCH , STCH

→ with,
- LDA BETA ⇒ Beta's value comes to accumulator
- ADD ALPHA ⇒ BETA ← ALPHA + BETA (in accumulator)
- STA DELTA ⇒ Content of accumulator goes to DELTA

- Comparison instruction
  - Accumulator will be one operand & operand in instruction is 2nd operand.
  - CMP ALPHA ⇒ if (ALPHA > A, < A, = A)
  - Result will be stored in Condition codes CC

EQ	LT	GT	...

- Conditional jump instructions (branch)
  - JLT, JGT, JEQ
  - J ⇒ no condition.

while ⇒ unconditional jump
do while ⇒ conditional jump

- Subroutine
  - JSUB ⇒ Jump to subroutine
  - RSUB ⇒ Return from subroutine

- Input & Output
  - will be done when the device is ready
  - its going to poll continuously
    - ⇒ TD ⇒ test device
      WD/RD ⇒ write to device / read to device
  - eg:- DEV1 = device's code
    - ⇒ Read TD DEV1
      JEQ Read ⇒ if device is not ready, jump back to read
      RD DEV1 goes to accumulator

      Write TD DEV2
      JEQ WRITE
      WD DEV2

⇒ SIC/XE - Machine Architecture

⇒ Memory
- 24 bit word
- 2^{20} bytes ⇒ 1MB

⇒ Registers

A	0
X	1
L	2
PC	8
SW	9

+ 4 more registers

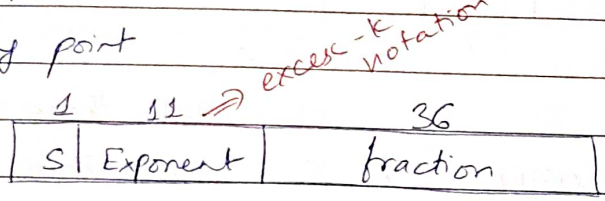
B	3	Relative Base register } Addressing
S	4	} General working register
T	5	} can be used by programmer
F	6	Floating point accumulator
48 bit		

⇒ Data formats

① Integers ⇒ 24-bit

② Character

③ Floating point



⇒ 48-bit rep

⇒ Instruction formats

- 4 types of instructions formats

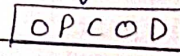
⇒ Lecture Notes Contd. (Dated :- 27-1-22)

⇒ Instruction formats

- Different formats ⇒ cuz its close to CISC.
- Supports relative addressing.
- 4 formats :-

① Format 1

- length = 1 byte
- consists of only 8-bit OPCODE



- eg :- ① FIX ⇒ load accumulator with contents of F ⇒ (A) ← (F)

integer accumulator

floating point accumulator

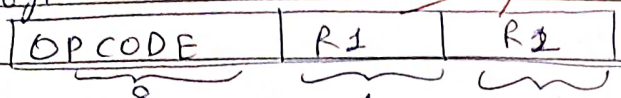
floating point no converted to nearest integer

- eg ② :- FLOAT ⇒ (F) ← (A)

- eg ③ :- HIO ⇒ halt the I/O lines

② Format-2

- Length = 2 bytes = 16 bits



Register operands

↳ not there in SIC.

- eg :- COMPR A, S ⇒ ~~COM~~ CMP was used with accumulator

A 0 0 4

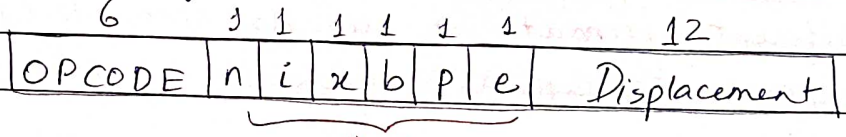
opcode of compr ⇒ A 0 0 4

- Register based instructions

⭐ V. imp

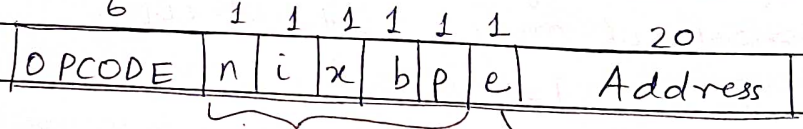
③ Format-3

- Length = 3 bytes = 24 bits



④ Format-4

- Length = 4 bytes = 32 bits



indicate addressing modes

0 ⇒ Format 3

1 ⇒ Format 4 ⇒ indicates next 20 bits are address of the operand

⇒ Addressing modes

- Relative addressing ⇒ used in format-3 instruction
  ↓ used in file explorer on our PC.

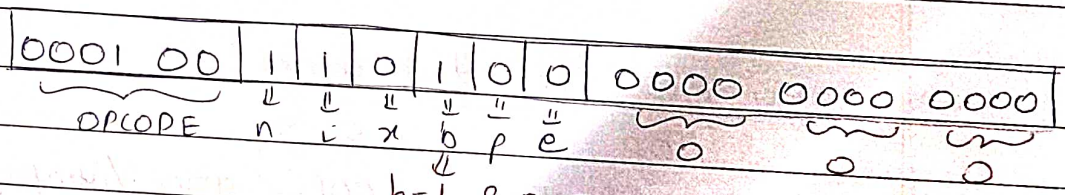
① Base relative addressing

↳ Table

	Type	Indication	Target address
①	Base Relative	b=1, p=0	TA = [B] + Disp ⇒ ~~0 ≤ disp ≤ 12~~ ↳ $0 \le disp \le 4095$ ($2^{12}-1$)
②	Program Counter (PC) Relative	b=0, p=1	TA = [PC] + disp = $-2048 \le disp \le 2047$
	Neither base relative nor PC relative	b=0, p=0 or b=1, p=1	

- eg for base relative addressing :-

STX LENGTH



b=1 & p=0

⇒ base relative

- [B] ⇐ 0033

base register

- x = 0 ⇒ direct addressing

e = 0 ⇒ format - 3

n = 1, i = 1 ⇒ simple addressing

b = 1, p = 0 ⇒ base relative addressing.

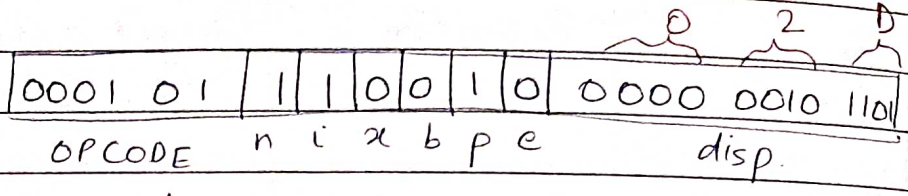
- TA = [B] + Disp = 0033 + 0000

= 0033

Command stores contents of instruction at 0033

- eg for PC relative ⇒ 90% of addressing in SICXE

STL RETADR



- x=0 ⇒ direct addressing
  e=0 ⇒ format-3
  n=1, i=1 ⇒ simple addressing
  b=0, p=1 ⇒ PC relative addressing

- TA = [PC] + Disp

- [PC] ⇔ 0003
  Disp = 0 2D ⇒ 0003
  + 002D
  0030 ⇒ TA

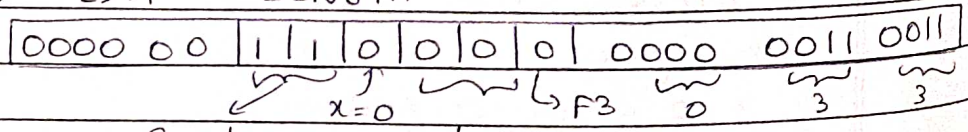
- When does -ve disp comes?

```
LOOP  TD   05
      JEQ  LOOP
```

going/jumping backward
↳ -ve displacement

eg for others →

eg:- ① LDA LENGTH



not relative addressing
⇒ address is directly given in disp field

contents @ 0033 ⇒ directly loaded onto accumulator

(2) STCH BUFFER, X

0101 01	1	1	1	1	0	0	0000 0000 0011

↓

x=1

⇓

indexed addressing

- b=1, p=0 ⇒ base relative add.

Disp = 0 0 3

- [B] ⇔ 0033

[X] ⇒ 0

☆ TA = Disp + [B] + [X] = 0033 + 0003 + 0 = 0036

character stored ∈ 00 3 6

in accumulator is stored in loc 0036.

☆ ∴ Indexing & relative addressing can occur together

⇒ LDA LENGTH → Assembly language

0000 00	1	1	0000	0000 0011 0011

↳ machine language ← Assembler

⇒ In this course, we're concerned about the stuff above the other way round :-

Disp = TA - [PC]

Disp = TA - [B]

Lecture Notes Contd (Dated :- 28-1-22)

⇒ n & i flags

① $i=1, n=0$:- Immediate addressing

- Operand is directly given in instruction itself.
- eg :- LDA #9
  - 0000 0000 ignored
  - ↳ indication of immediate add.

0000 00	0	1	0	0	0	0	000 0000 1001

- When this is decoded ⇒ decoder notices $n=0, i=1$
  - ↳ A ← 9

② $n=1, i=0$ ⇒ Indirect addressing

- eg :- J @RETADR
  - ↳ indication of indirect addressing
  - implying $n=1, i=0$
- opcode of J = 3C ⇒ 0011 11 00 ignored?
  - yes

⇒

0011 11	1	0	0	0	1	0

✓ assuming PC relative

- assuming [PC] = 002D
  - RETADR = ~~0003~~ 0030
  - → disp = 0030 ⇒ 002D
    - 002D ⇒ FFD2

(X) ⇒ 0030
+ FFD2
0002
0030
002D
0003

0011 11	1	0	0	0	1	0	0000 0000 0011

effective address → Place where address to jump is stored

0030	0033

0033 → ACTUAL address to go to
↳ indirect add.

③ $n=0, i=0$ / $n=1, i=1$ ⇒ Simple addressing ⇒ target address is taken as the location of the operand.

- $n=0$ $i=1$ ⇒ Immediate } indexing cannot be
  $n=1$ $i=0$ ⇒ Indirect } used

- $b=0, p=0$ ⇒ direct addressing
  ↳ no relative addressing.

F3 → possible when operand address fits in 12-bits
⇓
displacement ⇒ taken as operand address/ TA

F4 → no need of relative addressing. ⇒ operand address fits in 20 bits.

- $b=1, p=0$ ⇒ base relative
  $b=0, p=1$ ⇒ PC relative

⇒ Fig. 1-1 Illustration from the textbook

① 032600 ⇒ 0000 00(1) 0010 0000 0000 0000
(expand nibble-by-nibble)

0000 → LDA
00(1) → simple addressing
0010 → n i x b p e
b p → PC relative
0000 0000 → disp
600

⇒ 3000

⇒ TA = [PC] + Disp

⇒ 003000
000600
003600

⇒ 103000 gets loaded onto accumulator

⇒ A ← 103000 //

[B] = 006000
[PC] = 003000
[X] = 000090

H.W

(2) 03 C300

(3) 02 2030

(A)(2) 03C300

0000 0011 1100 0011 0000 0000

LDA n i x b p e disp = 300

TA = 006000 + 003000 + 000

[TA] = B + PC + X

= 006390

∥

has 00C303

006000
003000
000090
006300

⇒ A ← 00C303

(3) 022030

0000 0010 0010 0011 0000

LDA n i x b p e 0 3 0

∥

[PC] register. TA: disp.

relative.

⇒ TA = [PC] + disp

= 003000
000030 → WOT

≡ Lecture Notes Contd. (Dated :- 1-2-22)

(2) 03C300

(A) 0000 0011 1100 0011 0000 0000
LDA | n i x b p e | disp = 300
Base relative

⇒ TA = [B] + 300 + [X]
= 006000
+ 000300
6300
+ 90
6390

⇒ EA ← 00C303 //

(3) 022030

(A) 0000 0010 0010 0000 0011 0000
LDA | n i x b p e | 0 3 0
Indirect PC relative

⇒ TA = Address stored at
[PC] + 030
= 003000
000030
003030 ⇒ Address at 3030

3600 ⇒ Val = 103000

⇒ A ← 103000 //

(4) 010030

(A) 0000 0001 0000 0000 0011 0000
LDA | n i x b p e | 0 3 0
Immediate Addressing

⇒ A ← 30 //

(5) 003600

(A) 0000 0000 0011 0110 0000 0000
ni xbpe
LDA ‖ 6 0 0
PC relative.

PC = 3000 Disp = 600
A ← 3600

(6) 03106303 ⇒ Straightforward
0000 0011 0001 0000 1100 0011 0000 0011

⇒ LDA LENGTH
032026 ⇒ In our assembler discussion, we're finding out how to generate this

⇒ Instruction set

- LDB, LDT, LDS, STB
- FP ⇒ arithmetic
- Register to register operations
  ef:- ADDR, SUBR, MULR, DIVR
  ADDR r1, r2
  r2 ← r1 + r2

⇒ Input Output ⇒ Same as SIC

⇒ Programming examples ⇒ SIC

⇒ Data Movement
- One memory location to another
- Directly ain't possible

- Movement happens by 1 word


```
LDA   FIVE
STA   ALPHA
```

} no MOV instruction directly. you have to use accumulator

```
FIVE    WORD    5      → Word constant
ALPHA   RESW    1      → Reserved word. Is Reserved for a word.
```

Symbolic labels

- Movement unit = 1 byte ⇒ similar approach

```
LDCH    CHARZ
STCH    C1

CHARZ   BYTE    C 'z'   ⇒ Byte constant
C1      RESB    1       ⇒ Reserved byte
```

⇒ Programming examples ⇒ SIC/XE

- Additional addressing modes.

```
LDA   # 5     ⇒ Immediate addressing ⇒ directly send the value of constant
STA   ALPHA
LDA   # 90
STCH  C1
ALPHA RESW    1
C1    RESW    1
```

⇒ Arithmetic operations

BETA ← (ALPHA + INCR - 1)

DELTA ← (GAMMA + INCR - 1)

① SIC

1st inst :-

```
LDA   ALPHA
ADD   INCR
SUB   ONE
STA   BETA
```

2nd inst :-

```
LDA   GAMMA
ADD   INCR
SUB   1
STA   DELTA
```

Creating "labels" :- Declaration ⇒

```
ALPHA   RESW   1
BETA    RESW   1
GAMMA   RESW   1
DELTA   RESW   1
INCR    RESW   1
ONE     WORD   1
```

How many words

② SIC/XE

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

S register

Declarations :-

```
ALPHA   RESW   1
BETA    RESW   1
GAMMA   RESW   1
DELTA   RESW   1
INCR    RESW   1
```

imp as other next skip will overwrite accumulator

classmate
Date
Page

⇒ Indexing & Loop management
⇒ Array
⇒ I/O
} Agenda for next class.

⇒ Lecture Notes Contd. (Dated :- 4-2-22)

⇒ Labels ⇒ 1st column
Directives ⇒ 2nd column
Operands ⇒ 3rd column

⇒ ADDR S, A is faster than ADD INCR
↳ operand fetch from memory ⇒ slower.
by one!

⇒ again SUB #1 is faster SUB ONE
↳ immediate addressing
↳ operand fetch

⇒ Looping, indexing

⇒ SIC :- program to copy a 11 byte string to another string

```
        LDX     ZERO      ⇒ Load index reg with const labelled zero
MOVECH  LDCH    STR1, X   ⇒ Load* STR1[X]
        STCH    STR2, X   ⇒ Indicates indexed addressing
        TIX     ELEVEN    ⇒ Increment X ⇒ add 1 to x & compare result to 11
        JLT     MOVECH    ⇒ Load STR1[X+1]   ↳ result of comparision will be stored in condition code
        :
STR1    BYTE    C'TEST STRING'
STR2    RESB    11
        :
ZERO    WORD    0
ELEVEN  WORD    11
```

⇒ SIC/XE

```
        LDT   #11
        LDX   #0
MOVECH  LDCH  STR1,X
        STCH  STR2,X
        TIXR  T        ⇒ TIX is also same except we're making use of register
        JLT   MOVECH
        :
STR1    BYTE  C'TEST STRING'
STR2    RESB  11
```

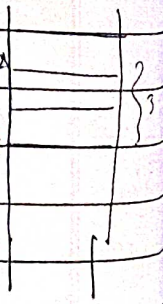
⇒ ALPHA[100] BETA[100] GAMMA[100]

GAMMA[i] = ALPHA[i] + BETA[i]

(1) SIC:

```
        LDA   #ZERO
        STA   INDEX
ADDLP   LDX   INDEX
        LDA   ALPHA,X
        ADD   BETA,X
        STA   GAMMA,X
        LDA   INDEX      } Not done by TIX
        ADD   THREE      } Here TIX only increments by 1
        STA   INDEX      } also TIX compares
        COMP  K300       ⇒ After 100 words, it should be 300
        JLT   ADDLP
INDEX   RESW  1
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100

ZERO    WORD  0
K300    WORD  300
THREE   WORD  3
```



⇐ 3 since memory is byte-addressable

(1) SIC/XE

```
        LDS   #3
        LDT   #300
        LDX   #0
ADDLP   LDX   ALPHA,X
        ADD   BETA,X
        STA   GAMMA,X
        ADDR  S,X
        COMPR X,T
        JLT   ADDLP
        :
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100
```

⇒ I/O

```
INLOOP  TD    INDEV
        JEQ   INLOOP
        RD    INDEV   ⇒ Read one byte into A
        STCH  DATA
        :
        TD    OUTDEV
        JEQ   OUTLP
        LDCH  DATA
        WD    OUTDEV
        :
INDEV   BYTE  X'F1'
OUTDEV  BYTE  X'05'
DATA    RESB  1
```

P.T.O ⇒

⇒ • Reading a 100-byte record & store it in memory
• How return from subroutine is managed

```
        JSUB    READ    ⇒ jump to subroutine
        :
READ    LDX     ZERO
RLOOP   TD      INDEV
        JEQ     RLOOP
        RD      INDEV       ⇒ Subroutine
        STCH    RECORD,X
        TIX     K100
        JLT     RLOOP
        RSUB ⇒ goes back to previous call. ⇒ managed using linkage register
INDEV   BYTE    X'F1'
RECORD  RESB    100
```

⇓

H.W Corresponding SIC/XE Program

Q) ALPHA = BETA * GAMMA. ⇒ SIC Program

```
        LDA     BETA
        STA     BETA
        MUL     GAMMA
        STA     ALPHA
        :
BETA    RESW    1
GAMMA   RESW    1
ALPHA   RESW    1
```

Q) SIC/XE ALPHA = 4* BETA - 9

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

classmate
Date
Page

SIC :-
```
LDA  BETA
MUL  FOUR
SUB  NINE
STA  ALPHA.
```

Q) SIC program to SWAP @ ALPHA & BETA.

A)
```
temp = A
A = B
B = temp
```
~~LDA~~ ~~STA~~
```
LDA  ALPHA
STA  TEMP
LDA  BBETA
STA  BALPHA
LDA  TEMP
STA  B
```
Haha!

Q) a) SIC program to find sum of elements of an array. ALPHA that has 10 words.

A)
```
LDA   ~~SUM~~ ~~ZERO~~.
~~STA~~ LDX   ~~ZERO~~
LDA   ZERO
~~STA~~   ZERO.
ADDLP  ~~&~~ LDX
```

```
      LDA   SUM
      LDA   ~~ZERO~~
      LDX   ZERO
ADDLP ADD   ALPHA,X
      STA   SUM
```

```
      LDA   ~~ZERO~~SUM
      STA   INDEX
ADDP  LDX   INDEX
      ~~LDA~~
      ADD   ALPHA,X
      ~~LDA~~   ~~SUM~~
      TIX   INDEX
```

```
      LDA   ~~SUM~~ ZERO
      STA   INDEX
      LDA   ZERO
      STA   SUM
ADDLP LDX   INDEX
      LDA
      ADD   SUM ALPHA,X
      ~~LDX~~ LDA   INDEX
      ADD   THREE
      STA   INDEX
      COMP  KARRSIZE
      JLT   ADDLP
```

in SIC/XE!! write ⇒ H.W

PRACTICE!!

b)
```
        LDS  #3
        LDT  #30
        LDX  #0
ADDLP   LDX  ALPHA,X
```

⇒ GAMMA = ALPHA * BETA ⇒ without MUL ⇒ repeated addition.

⇒ AL —ASSEMBLER→ ML
(SIC, SIC/XE)
Source program → Object program

⇒ Lecture Notes Contd (Dated :- 7-2-22)

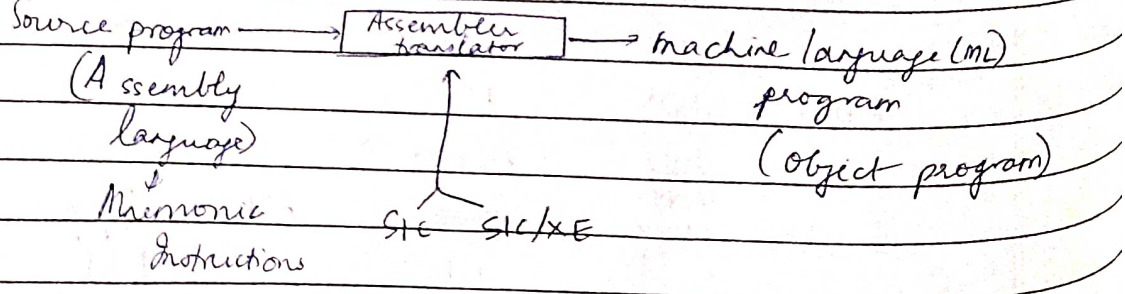
⇒ Appendix A from TB ⇒ no need to memorise.

Machine Architecture - SP

SIC → RISC
SIC/XE → CISC

Reading Assignment:
- UltraSPARC
- PowerPC
- Cray T3E
- VAX
- Pentium Pro

⇒ Our first system program ⇒ assembler translator



Source Program (high level language) → [COMPILER] → machine language (object program)

⇒ Lecture Notes Contd. (Dated :- 9-2-22)

⇒ Assembler

- SIC

Source Program (AL) → [Assembler (Translator)] → Object project (ML)

⇒ Fig 2.1 from textbook :- Comprises of :-

- Instructions
- Assembler directives ⇒ don't have a corresponding object code

⇒ Assembler directives :-

- START ⇒ specifies starting address of program in SIC
- END ⇒ end of program
- BYTE ⇒ for generating a character or hexadecimal constant.
  BYTE C 'N' ⇒ 'N' stored in C
  BYTE C 'NITGOA'
  ⇓ BYTE won't be there in object code, only the constant
- WORD ⇒ one word constant
- RESB ⇒ reserves indicated no. of bytes ~~for~~ for data
- RESW ⇒ reserves indicated no of words.

⇒ SIC Assembler

- Mnemonic instructions → object code } Build M/C instruction
- Symbolic operands → should be translated to their machine address

⇒ • RETADR 1033

• Conversion of constant to respective values

EOF → 45 4F 46

E O F

THREE WORD 3 → 000003

• Object program generator

⇒ Lecture Notes Contd (Dated :- 10-2-22)

⇒ In SIC :-

OPCODE	X	Address
8	1	15

① STL RETADR

⇒ no indexed, as no ',X'

⇒ STL ⇒ opcode = 14

address of RETADR = 1033

② JSUB RDREC

48 2039

③ LDA LENGTH

00 1036

④ COMP ZERO

28 1030

STCH BUFFER,X ⇒ 0101 0100 1001 0000 0011 1001

54

OPCODE X=1 address

5 4 9 0

eg:-

⇒ Sequential Process ⇒ reading file line-by-line
- But happens in multiple passes, as we don't know RETADR initially.
  ↳ as there's forward referencing being used

⇒ Object program
- consists of 3 different types of records

① Header record
Col 1 : H
Col 2-7 : 6 columns ⇒ program name
Col. 8-13 : Starting address of object program (hex)
Col. 14-19 : length of object program in bytes (hex)

2079
- 1000
1079 + 1 ⇒ 107A //

② Text record
Col 1 : T
Col 2-7 : Starting address for object code in this record
Col-8-7 : length of object code in this record
Col. 10-69 : object code, represented in hexadecimal
⇒ 2 columns per byte ⇒ cuz in nibble format
↳ 60 columns
= 30 bytes

⇒ 30 bytes ⇒ (1E) //

③ End record
Col 1 ⇒ E
2-7 ⇒ Address of 1st executable instruction.

Lecture Notes Contd. (Dated :- 11-2-22)

name will be truncated if greater than 6

				Object code
1000	ARRSUM	START	1000	
1000		LDA	ZERO	00 102D
1003		STA	INDEX	0C 1021
1006	ADDLP	LDX	INDEX	04 1021
1009		LDA	ALPHA, X	00 9024
100C		ADD	BETA, X	18 9030
100F		STA	GAMMA, X	0C 9027
1012		LDA	INDEX	00 1021
1015		ADD	THREE	18 13AE
1018		STA	INDEX	0C 1021
101B		COMP	K300	28 13AB
101E		JLT	ADDLP	38 1006
		⋮		
1021	INDEX	RESW	1	
1024	ALPHA	RESW	100	
1159	1027	BETA	RESW	100
129C	102A	GAMMA	RESW	
13A8	102D	ZERO	WORD	0 000000
13AB	1030	K300	WORD	300 000012C ?
13AE	1033	THREE	WORD	3 000003
		END	END	
			FIRST	

H^ ARRSUM ^

13B0
- 1000
3AE 3B0

00000000 | 1 | 0001 0000 0010 0100

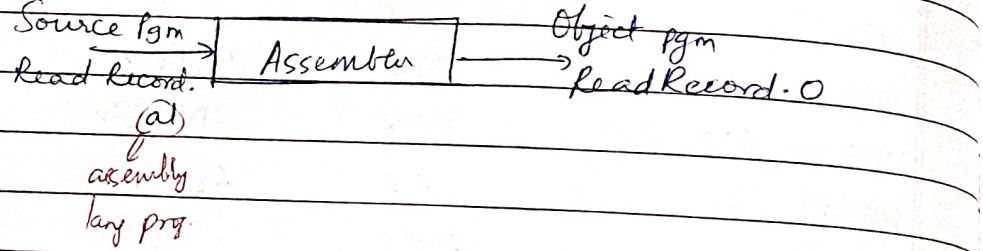
Object program:

H^ ARRSUM ^ 001000 ^ 0003B0

T^ 001000 ^ 1E ^

⇒ Lecture Notes Contd (Dated :- 17-2-22)

⇒ SIC Assembler



- This doesn't happen in one go (sequential), rather, ma: assemblers take 2 passes

- Pass 1 :-
  - Assignment of addresses for instructions & symbols
  - Save the addresses assigned to symbols
    ⇓
    CLOOP
    ENDFIL etc.
  - Processing of assembler directives ⇒ not completed

- Pass 2 :-
  - Mnemonic instructions to object code
  - BYTE, WORD ⇒ in machine language
  - Completes processing of assembler directives
  - Object program

⇒ Data structures for SIC Assembler

(1) Location counter ⇒ LOCCTR
(2) Operation code table ⇒ OPTAB
(3) Symbol table ⇒ SYMTAB

(2) OPTAB

- It has the machine code corresponding to mnemonic instruction

Mnemonic Instruction	Machine code
LDA	00
STL	14
JSUB	48

: no. of instructions available in that hardware

Usage of OPTAB in the passes :-

⇒ Pass ①

- Validating mnemonic operation code.
  - ↳ 000h, if programmer makes a mistake
  - eg :- LDA LENGTH
    LFA LENGTH
    ↳ If you type this by mistake
    Reports error message.
- also used for instruction length

⇒ Pass ②

- For translation

③ SYMTAB

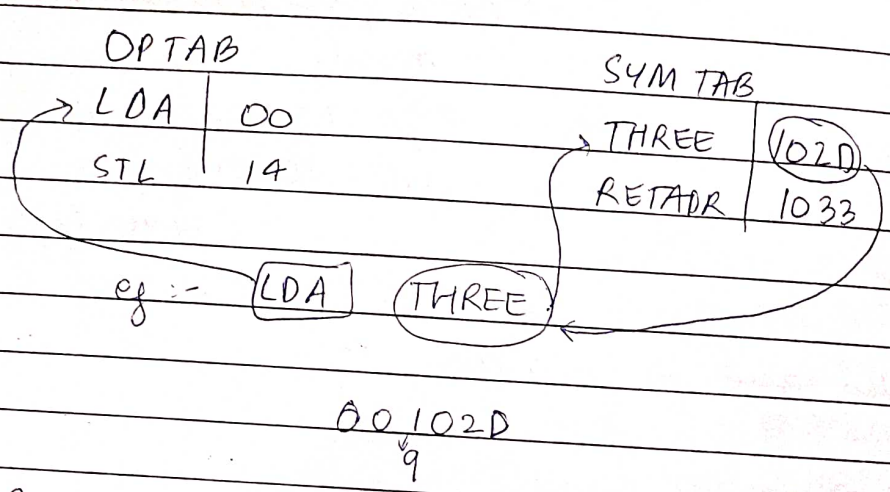
- Symbol table
- name & value for each symbol used
  - CLOOP, ENDFIL, FIRST
  - value → corresponding address

Symbol	Value
~~Copy~~ ~~COPY~~	
FIRST	1000
CLOOP	1003
ENFIL	1015

- Usage of SYMTAB in passes :-

(1) Pass - 1
- Name - value pairs are inserted
- In case CLOOP is assigned to another instruction
  ⇓
  ERROR

(2) Pass - 2
- Translation



⇒ Instead of linear search, we'll take a hash table ⇒ O(1)

⇒ Lecture Notes Contd. (dated :- 18-2-22)

⇒

Key	Value
LDA	00
STL	14

Key → Hash function → value

hf (LDA) → index of LDA

Challenge finding a hash table

Ref books for hash table - File structures, An object oriented approach with C++ by Michael J. Folk

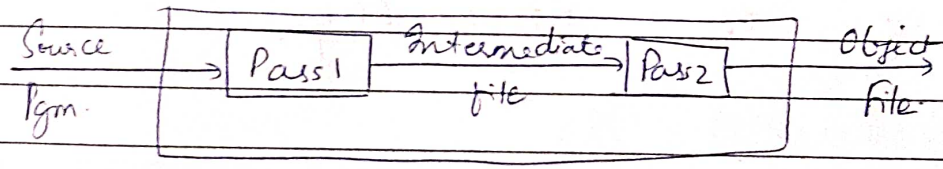
- Avoiding collision.

hf(k1) → index
hf(k2) → index

Eg :- hf(LDA)
hf(LDB)

based on characters' mapping.

⇒ Making a hash function is tougher for ~~OPTAB~~ SYMTAB since for OPTAB its predefined & many explorations would've been made for hashing function.



- Source Pgm + location counter value
error-code

⇒ Lecture Notes Contd. (Dated :- 21-2-22)

⇒ SIC/XE Assembler

- Wherever possible register-to-register, immediate addressing } faster

A	0	T	5
X	1	F	6
L	2	PC	7
B	3	SW	8
S	4		

⇒ Instruction formats

- Format 1

OPCODE
8 - bit

- format 2

OPCODE	R1	R2
← 8 →	← 4 →	← 4 →

eg:- CLEAR X ⇒ only 1 register
B4 1 0

- format 4

OPCODE	n	i	x	b	p	e	Address
	1			0			20

Simple

eg:- CLOOP +JSUB RDREC
48 1036

0100	1000	1	1	0	0	0	1	0000	0001	0000	0011	0110
4	B			1				0		1	0	3 6

= 4B101036

+JSUB WRREC

0100	10	11	0001	0000	0001	0101
4	B		1	0		5 D

4B1010 5D

+LDT #4096
74
n=0 x=0
i=1

0111	0101	0001	0001	0000	0000
7	5	1	0	0	0

Format-3

OPCODE	n	i	x	b	p	e	displacement
6							

Reference
- PC → PC-relative: b=0 p=1
- B → Base relative: b=1 p=0

$EA|TA = (PC) + disp \Rightarrow disp = EA - (PC)$

$disp = EA - (B)$

Pg ⇒ 60

- TA = 0030
  PC = 0003
  disp = 0030 - 0003
  0030
  -0003
  disp = 002D

⇒ 0001 0011 0010 0000 0000 0011 1101
1 3 2 0 0 3 D

⇒ Lecture Notes Contd.(Dated :- 23-2-22)

⇒ Relative addressing (F3) ⇒ extra overhead or not?
↳ necessity as address is 12 bits & there are 2^{20} addresses

⇒ Format 1 & 2 ⇒ no problem cuz they have no addresses.

⇒ Case-1: Is the instruction ⇒ F4 ⇒ the plus prefix

⇒ Case 2 ⇒ • PC relative addressing

Disp = TA/EA − (PC)

if more than 12 bits ⇒ not PC relative

• Base relative addressing

Initialization using BASE assembly directive is necessary.

if both are not possible :- 2 choices ⇒

not possible, while assigning addresses initially, it was assumed to be F3, so now you can't change it.

① put it in F4 type.

② throw error ↳ throws error

eg :- STCH BUFFER, X

① PC relative

TA = Buffer's address. 0036

PC = 1051

⇒ disp = 0036 − 1051

ffff
1051
EFAE
1
EFAF

0036
+ EFAF
EFE5 ⇒ 2's comp.
1 0 1 A ans.
+ 1
-101B ⇒ disp

Can't fit it in 12 bits ⇒ PC relative not feasible

② Base relative

Disp = TA − [B]

[B] = 0033

Disp = 0036
− 0033
0003

→ Indexed.

0 1 0 1	0 1	1	1	1	1 0 0	0 0 0 0	0 0 0 0	0 0 1 1
5	7	C			0	0	0	3

= 57C003

☆☆ the DECODER sees the flags, not assembler ⇒ WE'RE setting those flags

So while decoding n=1 i=1 x=1
b=1 } BR
p=0 }
⇒ TA = [B] + disp + (X)
= 0033 + 0003
= 0036 + (X)

⇒ STL RETADR

① PC relative ⇒ TA = 0030
PC = 0003
⇒ disp = 0030 - 0003
FFFF
0003 = 0030
FFFD + FFFD
002D = disp.

= 0001 0 1 1 0 0 1 0 0000 0010 1101
1 7 2 0 2 D

= 17202D

WE USE PC RELATIVE & BASE R in modern assemblers like SIC/XE SINCE THE programs are relocatable

⇒ LDB #LENGTH ⇒ n=0, i=1.
68 val= 0033

① PC ~~ret~~ ⇒ 0006
TA = 0033
⇒ disp = 0033 - FFFF / 0006 = FFF9.

&
1 1
0033
+ FFFA
002D

0	0	1	1	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	1
6				9			2			0			2			D			

⇒ LDA LENGTH

00 ‖0033

PC ⇒ 000D

TA = dd33

FFF63

0026

FFFF
- 000D
FFF2

FFF3

0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0
0				3				2				0				2				6			

⇒ COMP #0 → 2900000

⇒ JEQ ENDFIL

[TA] = 001A

PC = 0013

FFFF
- 0013
FFEC
001A
0007

⇒ J CLOOP

3C

TA = 0006

PC = 001A

0006
- FFFA
FFEC

0006
FFF6
FFFC
FEC

‖
3F2FEC