System Software [TCS-501]

Pre-requisite:- All the students must have basic understanding about TCS 301, TCS 307, TCS 403

Course outcomes (CO): After completion of this course students will be able to

CO1 – Define system s/w and differentiate ~~sw~~ system s/w with other softwares.

CO2 – Assess the working of Assembler, Loader/Linker and Macroprocessor

CO3 – Understand the concept of passes in translators.

CO4 – Determine the purpose of linking and types of linking.

CO5 – Develop the system s/w according to machine limitations.

CO6 – Compare and contrast the various text editors.

Syllabus: UNIT-I

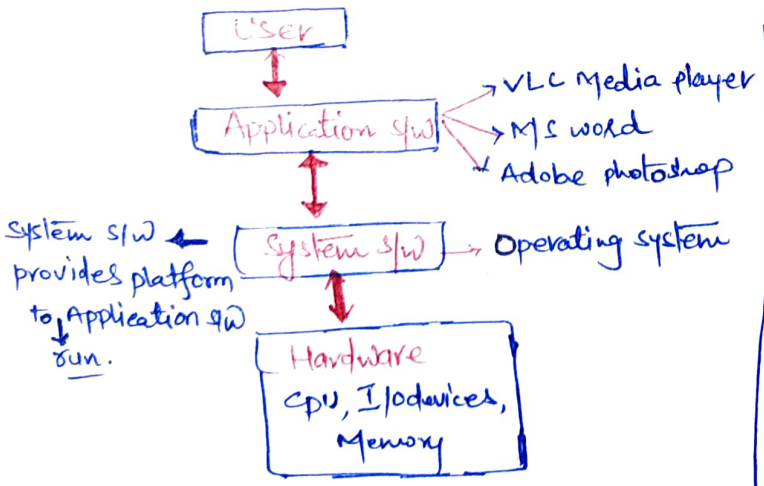
Books: Leland L. Beck: System software: "An Introduction to system programming" 3rd edition, Addison-wisley 1997.

Machine Architecture: Introduction, System software and its relation to machine Architecture, Simplified Instructional computer (SIC), Architecture of SIC Machine, SIC programming examples.

Introduction to Software → set of programs
↳ programs are set of instructions.

Types of software → system software
↳ Application s/w → designed to do some specific task.

① System Software – System software is an interface between the computer hardware and user Applications. Eg - Operating System, Drivers



* System s/w, Importance of System s/w.
↳ It works internally inside a computer
↳ It works without human knowledge
↳ It helps the user to do his/her work succesfully.

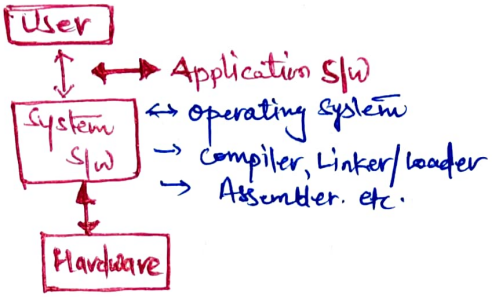
Eg → Operating system, compiler, Linker, Loader, Assembler etc.

Introduction to system software:

There are mainly two types of software: System s/w and Application s/w.

- System s/w includes the programs that are dedicated to managing the computer itself, such as the operating system, file management utilities, etc.
- System s/w ~~are the~~ is a software that provides platform to other softwares. Some example can be Operating system, compiler, Assembler, Linker, Loader, disk formatting s/w etc.
- System software act as an interface between the hardware and the end user.

- The most important features of system s/w includes:
  1. Closeness to the system
  2. Fast speed
  3. Difficult to manipulate
  4. Difficult to Design.

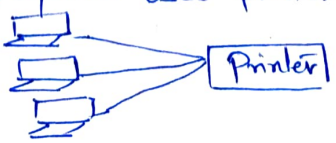


Operating System: An operating system (OS) is a type of system s/w that manages computer's hardware and software resources.

- It provides common services for computer programs.
- It acts as an interface between computer hardware and the end user.
- It controls and keeps a record of the execution of all other programs that are present in the computer, including application programs and other system software.

The Most important task performed by operating system are:

1. **Memory Management** – The OS keeps track of the primary memory (MM) and allocates the memory when a process requests it.
2. **Processor Management** – To which process cpu would be allocated and for how long time. (This thing can be done with the help of Scheduling")
3. **Resource Manager** – Resource may be cpu, I/o devices, Memory.
4. **I/o device Management** – which system uses printer at what time.
5. **File Management** – Allocates and de-allocates the resources and decides who gets the resources.



6. **Security** – prevents unauthorised access to programs and data by means of password.

(2) Compiler → It is also a type of system software.

- Compiler is a specific type of translator which is used to convert source code (written in high level language) into machine code or object code or target code (low level language)

Source code (HLL) → [Compiler] → Target code, machine code (LLL)

- Compiler performs all of the following operations during compilation of the source program.
  - Pre-processing, Lexical Analysis, Syntax Analysis (Parsing), Semantic analysis (SDT), Intermediate code generation, code optimization, and code generator.
- Examples of compiler may include gcc (C compiler), g++ (C++ compiler), javac (java compiler) etc.

(3) Interpreter → It is also a type of system software.

- Interpreter is a specific type of translator that can be translate / convert high level language into a low level language line by line.
  or
- In other words "Interpreter converts source code into machine code line by line or statement by statement.

Note:- From memory point of view

- Interpreter will take less amount of memory space.

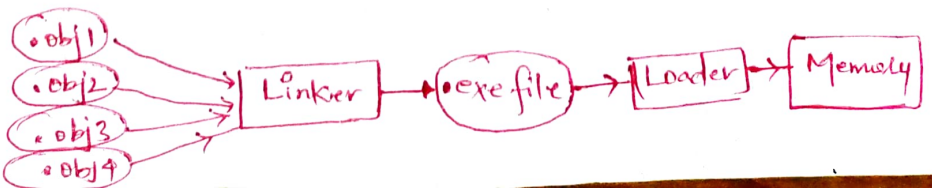
From Execution time.

- ~~Interpreter will take~~ compiler will take less time to execute the source code as compared to interpreter.

(4) Assembler – An assembler is a program that converts assembly language into machine code.

(5) Linker / Loader – Linker is used to link all the library / object files into .exe file.

– Loader is used to load .exe file into memory.



* **Instruction Format:** A computer performs a task based on the instruction provided. Instruction in computer comprises groups called fields. These fields contain different information as for computers everything is in 0's and 1's. so each field has different significance based on which a CPU decides what to perform. The most common fields are:

Ex: ADD R_0 R_1
↳ opcode → operand

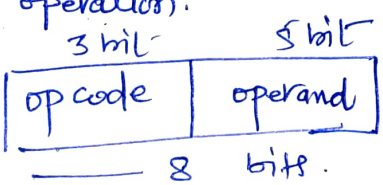
(1) **Opcode:** operation field specifies the operation to be performed like Addition, SUB, MUL etc.

Ex: A+B
↳ operator
↳ operand

(2) **Operand:** Address field which contains the location of the operand. i.e; Register or Memory location.

(3) **Mode:** Mode field specifies how operand is to be founded.

Instruction – A group of bits which instructs the computer to perform some operation.



Q → Suppose the length of the instruction is 8 bits. where opcode field contain 3 bits and operand field contains 5 bits. Then how many maximum opcode does a CPU can perform?

= CPU can perform 2^3 operations / opcode = 8 operations.

* **Instruction set Architecture** – collection of all instructions which a CPU can support.

Types of Instructions: Based on the operation.

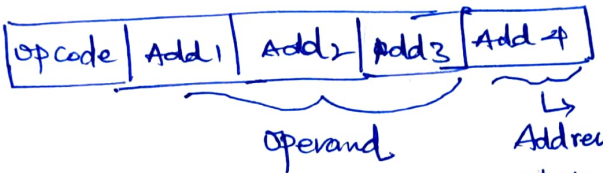
(1) Data transfer – MOV, LDI, LDA
(2) Arithmetic & logic – ADD, SUB, MUL, AND, OR
(3) Machine control – EI (Enable interupt), DI, PUSH, POP
(4) Iteralive – LOOP, LOOPE, LOOPZ
(5) Branch – JMP, CALL, RET, JZ, JNZ

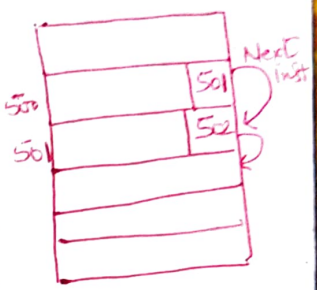
Types of Instruction: Based on operands

1. 4 - Address Instruction
2. 3 - " "
3. 2 - " "
4. 1 - " "
5. 0 - " "

(1) 4 Address Instruction – Maximum 4 addresses can be specified within an instruction.

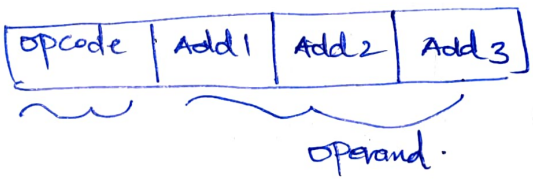
↳ Program counter is not available.





↳ Size of instruction is large.

(2) 3 Address Inst. → Maximum 3 Addresses can be specified within an instruction



It can also support 2 address, 1 address and 0 address instruction

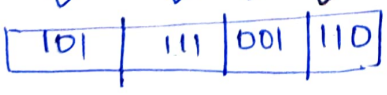
→ All modern day computer uses 3-Add. inst.

eg → ADD R_1, R_2, R_3 — Source ← Two Source
Destination – one desti

eg – A = B + C

$R_0 \leftarrow a$
$R_1 \leftarrow b$
$R_2 \leftarrow c$

⇒ ADD R_0, R_1, R_2



(3) 2 Address instruction – Maximum 2 addresses can be specified within an instruction.

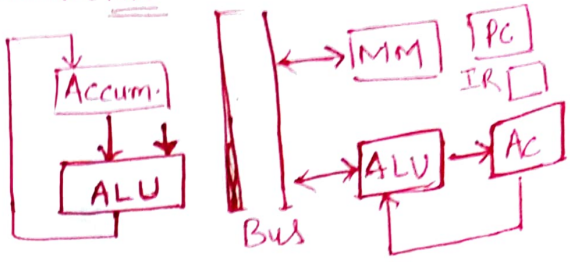
- One operand will be used as Source and destination both.
- Old value of common operand is overwritten by result.

(4) 1 Address instruction: Maximum one address can be specified within an instruction. It is used for Accumulator based architecture.

Opcode	Add1

eg → ADD R_1

$R_1 \leftarrow AC + R_1$



⇒ Accumulator is used as second operand implicitly.

(5) 0 Address instruction: No any address is specified within an instruction.

⇒ Supported by stack-based architecture.

Opcode

eg → ADD → ~~value~~ Two operands are taken from stack.

Addressing Mode. The term addressing modes refers to the way in which the operand of an instruction is specified.

The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

1. Implied mode
2. Immediate mode
3. Register Mode
4. Register Indirect
5. Auto increment
6. Auto Decrement
7. Indirect Add. mode.
8. Relative Add. mode
9. Base Register Add. mode.
10. Indexed Addressing mode.

* Simplified Instructional computer (SIC)

Why SIC – It is tough to learn/study about real machine. i.e; computer h/w. because of the following:

1. A real machine contains huge number of Addressing modes
2. Huge number of instruction set.

Introduction to SIC → A simplified instructional computer is a hypothetical (unreal) computer that has hardware features that are often found in real machine.

➡ It is easy to learn a system software using SIC machine architecture due to limited number of addressing modes and instruction sets.

➡ There are two versions of this machine. and both are upward compatible. Means object program for SIC can be properly executed on SIC/XE

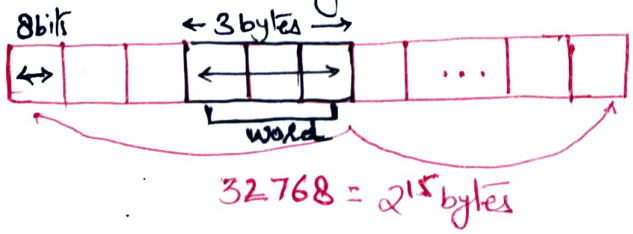
(i) SIC standard Model
(ii) SIC/XE (Extra equipment or expensive)

* **SIC Machine Architecture:** Following features are supported by SIC →

(1) Memory	(4) Data formats	(7) Input and output
(2) Registers	(5) Instruction sets	
(3) Addressing Modes	(6) Instruction formats	

(1) **Memory:** (i) Memory in SIC consists of 8 bit bytes
(ii) Any 3 consecutive byte form a word (24 bits)
(iii) All addresses on SIC are byte addresses.
(iv) words are addressed by ~~their~~ the location of their lowest numbered byte.
(v) There are a total of 32,768 bytes (2^{15} bytes) in the computer memory.

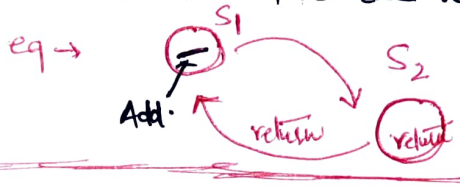
eg. word

② **Registers** :→ There are five registers in SIC, all of which have special uses.

→ Each register is 24 bits in length. (3 bytes)

→ The following Table indicates the numbers, Mnemonics, and uses of these registers (The numbering scheme has been chosen for compatibility with the XE version of SIC)

Mnemonic	No.	Special use
A	0	Accumulator, used for Arithmetic operation
X	1	Index register, used for Addressing.
L	2	Linkage register, the jump to subroutine (JSUB) instruction stores the return address in this register. eg →  ⇒ It contains the return address whenever control transferred to Subroutines
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 such as <, <=, >, >=, == etc.

Note: ⇒ SIC does not have any stack. It uses the linkage register to store the return address.

⇒ It is difficult to write the recursive program. A programmer has to maintain memory for return addresses when he write more than one layer of function call.

③ **Data Formats**

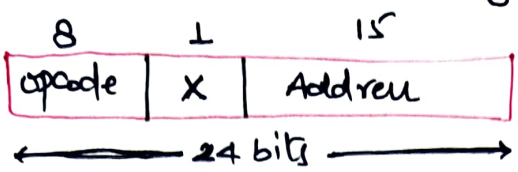
⇒ Integers are stored as 24 bits binary numbers.

⇒ 2's complements for storing Negative values.

⇒ Characters are stored using their 8-bit ASCII Codes.

⇒ There is no floating point hardware on the Standard version of SIC.

④ **Instruction Formats**: All machine instructions on the standard version of SIC have the following 24 bit format:



Note that the memory size of SIC is 2^{15} bytes.

Here, x is used to indicate index-address mode.

⑤ **Addressing Mode**: There are two Addressing modes available, indicated by the setting of the x bit in the instruction. The following Table describes how the target address is calculated from the address given in the instruction. Parenthesis are used to indicate the contents of a register or a memory location.
For example, (X) represents the contents of register X.

Mode	Indication	Target Address Calculation
Direct	X = 0	TA = Address
Indexed	X = 1	TA = Address + (X)

⑥ **Instruction Set**: SIC provides a basic set of instructions that are sufficient for most simple tasks. These include instructions:

- Load and Store registers – LDA, LDX, STA, STX etc.
  - eg → LDA #3 – Load 3 into Accumulator A.
  - → STA LABLE → Store the value of Accumulator A into LABLE
- Integer Arithmetic operations – ADD, SUB, MUL, DIV.
  - All arithmetic operations involve register A and a word in memory, with the result being left in the register.
- Comparison instruction → COMP is used to compare the value in register A with a word in memory; This instruction sets a condition code ~~value in register~~ CC to indicate the result (<, <=, >, >=, ==).
- Conditional Jump instructions – JLT (Jump less than), JEQ (Jump equal to), JGT (Jump Greater than) Can test the setting of CC
- Subroutine Linkage – JSUB, RSUB

⑦ Input/Output : I/O (transferring 1 byte at a time to/from the rightmost 8 bits of register A)

- Test device instruction (TD)
- Read data (RD)
- Write data (WD)

Based on this Test device inst. (TD) we can analyse whether the device is ready to read or write.

Example of SIC : ① SIC program for Data movement

```
LDA   FIVE    → Load value from FIVE into Accumulator A
STA   ALPHA   → Store the content of Accumulator into ALPHA
LDCH  CHARZ   → Load character Z into Accum.
STCH  C1      → Store the value of Accum. into C1
.
.
.
ALPHA  RESW  1     → ALPHA  [ ][ ][ ]   - Reserve word
FIVE   WORD  5     → FIVE   [0][0][5]
CHARZ  BYTE  C'Z'  → CHARZ  [Z]         - 8 bit ASCII code
C1     RESB  1     → C1     [ ]         - Reserve byte
```

Memory location — Name of variable

LDA = 5 → ALPHA = 5
LDCH = Z → C1 = Z

Note :
RESW (Reserve word) – Reserving a word – 3 consecutive bytes
RESB (Reserve byte) – Reserving a byte – 8 bits (for ASCII character)

WORD → [ ][ ][ ] → contain data → int a = 5;
RESW → [ ][ ][ ] → Reserving a word → int a;

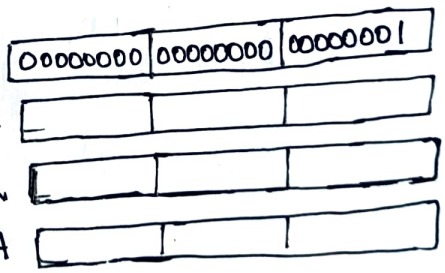
RESW is like declaring a variable in C progr.
WORD is like initializing a variable in C program.

Example of SIC: ②: simple Acithmetic operation performed by SIC

```
LDA     ALPHA   → Load ALPHA into Register A
ADD     INCR    → Add the value of INCR
SUB     ONE     → subtract 1
STA     BETA    → Store the contents of Accumulator/register
                  in BETA
.
.
.
ONE     WORD 1  → one word constant
ALPHA   RESW 1  → one word variable.
INCR    RESW 1
BETA    RESW 1
```

Explaination

```
LDA   ALPHA  → A = ALPHA
ADD   INCR   → A = ALPHA + INCR
SUB   ONE    → A = ALPHA + INCR - 1
STA   BETA   → BETA = ALPHA + INCR - 1   Final output
.
.
.
ONE   WORD 1 → ONE
ALPHA RESW 1 → ALPHA
INCR  RESW 1 → INCR
BETA  RESW 1 → BETA
```



Example of SIC/XE – Data movement in SIC/XE

Immediate Addr. mode

```
LDA   #5      Load value 5 into Register A (using Immediate Addr. mode)
STA   ALPHA   store in ALPHA
LDA   #90     load ASCII code for 'Z' into A Regis.
STCH  C1      Store in character variable C1
 .
 .
 .
ALPHA  RESW  1     one word variable - ALPHA
C1     RESB  1     one byte variable.
```

WORD

- A = 5
- ALPHA = | 0 | 0 | 5 |
- A = Z
- C1 = | Z |

SIC/XE

Memory consists of 8 bit-bytes
3 consecutive bytes for a WORD = 24 bits

① Memory → Total size is 1MB (2^{20} bytes) — More as SIC

↳ This increase leads to a change in instruction format and addressing modes.

② Register → 5 in SIC — A(0), X(1), L(2), PC(8), SW(9)

① The following Additional registers Provided by SIC/XE:

B — 3 — Base register, used for addressing

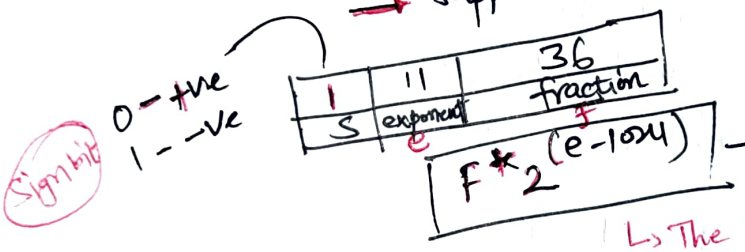
S — 4 — General working register, no special use.

T — 5 — General working register — no special use

F — 6 — Floating point Accumulator (48 bits)

③ Data formats — Integer — 24 bits

→ support 48 bit floating point no.



$2^{10} = 1024$
$2^{11} = 2048$
(e - 2047)

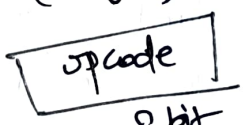
→ to get no. The absolute value of the no. if exponent is e and fraction is f.

↳ The exponent is interpreted as an unsigned binary number between 0 to 2047.

↳ The value zero is represented by setting all bits (s, e, f) to 0

④ Instruction formats

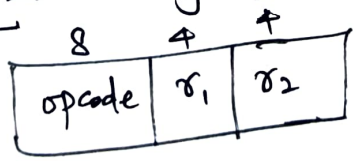
① Format 1 (1 byte)



eg - RSUB

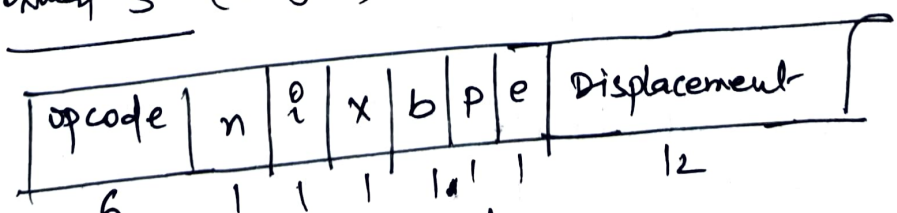
② format 2 - 2 bytes)

Adding Two Registers



eg - ADDR S, T

③ Format 3 (3 bytes)

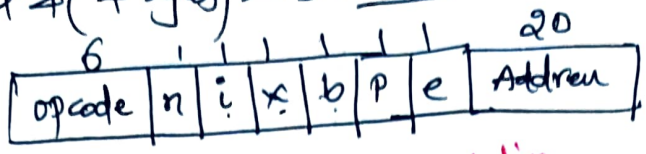


where. n → Indirect Add. mode
i → immediate Add. mode.
x → index Add. mod
b → base relative Add. mode

p → Program counter relative Add.

e → Whether format 3, 4
e = 0 - format ③
e = 1 - format ④

④ format 4 (4 bytes) — 32 bits



④ Addressing Mode - Two new relative Addressing modes are available for use with instructions assembled using format 3.

Mode	Indication	TA 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)

① If n=0 and i=1 → Then the target Address itself is used, as the operand value; No memory reference is performed. This is called Immediate Addressing mode.

② If n=1 and i=0 → Direct Addressing.

* Instruction set: SIC/XE provides all of the instruction sets that are available on the standard version.

① Load and Store the new register - LDB, STB etc (Load into Base register)

② Floating point arithmetic opn - ADDF, SUBF, MULF, DIVF

③ Register Move inst → Taking the operand from register ⇒ RMO (Register Move) inst.

④ Register to Register Arithmetic opn → ADDR, SUBR, MULR, DIVR

⑤ A Special Supervisor call inst → used to generate an interrupt that can be used for communication with the operating system (OS)

Input and output → I/O channels used to perform I/O while the CPU is executing other inst.

① Start IO (SIO), Test IO (TIO) and HIO (Halt I/O)

Programming examples of SIC and SIC/XE

① SIC

```
LDA     ALPHA     Load ALPHA into register A  - A = ALPHA
ADD     INCR      Add the value of INCR       - A = ALPHA+INCR
SUB     ONE       Subtract 1                  - A = ALPHA+INCR-1
STA     BETA      Store in BETA               - BETA = ALPHA+INCR-1
LDA     GAMMA     Load GAMMA into register A  - A = GAMMA
ADD     INCR      Add the value of INCR       - A = GAMMA+INCR
SUB     ONE       Subtract 1                  - A = GAMMA+INCR-1
STA     DELTA     Store in DELTA              - DELTA = GAMMA+INCR-1
  .
  .
  .
ONE     WORD  1   One-WORD constant           - ONE  [0][0][1]  WORD
                  One-WORD variable
ALPHA   RESW  1   - ALPHA   [ ][ ][ ]  Reserve WORD
BETA    RESW  1   - BETA    [ ][ ][ ]  Reserve WORD
GAMMA   RESW  1   - GAMMA   [ ][ ][ ]  Reserve WORD
DELTA   RESW  1   - DELTA   [ ][ ][ ]  Reserve WORD
INCR    RESW  1   - INCR    [ ][ ][ ]  Reserve WORD
```

② SIC/XE

```
LDS     INCR      Load value of INCR into Register S     S = INCR
LDA     ALPHA     Load ALPHA into register A             A = ALPHA
ADDR    S, A      Add the value of INCR                  R = ALPHA+INCR
SUB     #1        Subtract 1                             R = ALPHA+INCR-1
STA     BETA      Store in BETA                          BETA = ALPHA+INCR-1
LDA     GAMMA     Load GAMMA into Register A - A = GAMMA
ADDR    S, A      Add the value of INCR                  R = GAMMA+INCR
SUB     #1        Subtract 1                             R = GAMMA+INCR-1
STA     DELTA     Store in DELTA                         DELTA = GAMMA+INCR-1
  .
  .
  .
ALPHA   RESW  1   - ALPHA   [ ][ ][ ]  (24 bits)  Reserve WORD
BETA    RESW  1   - BETA    [ ][ ][ ]             Reserve WORD
GAMMA   RESW  1   - GAMMA   [ ][ ][ ]             Reserve WORD
DELTA   RESW  1   - DELTA   [ ][ ][ ]             Reserve WORD
INCR    RESW  1   - INCR    [ ][ ][ ]             Reserve WORD
```

Addressing Modes of SIC/XE

	n (Indirect)	i (Immediate)	x (Indexed)	b (Base relative)	p (Program counter)	e (Format specifier)
Base relative Addressing Mode (TA = [B]+Disp.)	1	1	0	1	0	0
Program counter relative Add. Mode (TA = [PC]+Disp.)	1	1	0	0	1	0
Indirect (@)	1	0	0	0	0	0
Immediate (#) Addressing Mode	0	1	0	0	0	0
Indexed Add. mode	1	1	1	0	0	0

Here n=1, i=1 Mean neither Indirect nor Immediate

Ex: Target Address calculation → calculate the target Address for the following SIC/XE m/c instruction and Also find the value loaded into register

Given That (B) = 006000(h), (PC) = 003000(h), (X) = 000090(h)

Where B - Base relative, PC - program counter, X - index register

(I) 032600(h)

Soln

0000 0011 0010 0110 0000 0000

opcode n i x b p e — Displacement

here n=1, and i=1 - neither indirect nor immediate

P=1 it is program counter relative.

TA = (PC) + Displacement = 003000 + 600

TARGET ADD. = 3600

VALUE Loaded into Register A = 103000

(II) 03C300(h)

n i x b p e — Disp.

0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0

opcode

(IV) 0310C303

TA = Displac

TA = 0C303

value loaded 003038

n=1, i=1 - Neither 'indirect nor 'immediate'

TA = (X) + (B) + disp = 000090 + 006000 + 300

TA = 6390, Value loaded into Register A = 00C303

(III) 022030(h)

0000 0010 0010 0000 0011 0000

opcode n i x b p e — displacement

Indirect Add. mode — Indirect Add.

TA = PC + displacement = 003000 + 030 = 3030

TA = 3030, value loaded = ~~003600~~ 103000

Memory	
3030	003600
3600	103000
6390	00C303
C303	003030

Addrss

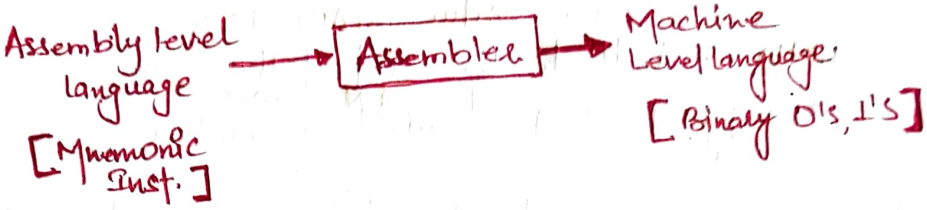
Value

UNIT - II

Introduction to Assembler

- Definition of Assembler
- Assembler Directives
- Data structure used by Assembler
- Features of an Assembler.
- Basic Assembler Functions

Definition: An Assembler is a kind of translator that accepts the input in assembly language program and produces its machine language equivalent.



Basic Assembler Functions

(1) Assembler convert mnemonic operations code to their equivalent machine language.

Ex: STL → 14, JSUB → 48,

Ex: LDA #3 – Load value 3 into Accumulator.

LDA – machine inst is 00

(2) Convert symbolic operands to their equivalent machine address.

For eg→ STA [redacted] TABLE – Whatever the value present in the Accumulator register, ~~should~~ that should be store into TABLE, but TABLE is not an address, instead it is a symbolic operand here. Then we need to suppose Find the address of the symbolic operand i.e; TABLE and then store the content of Accumulator into that address.

(3) Build machine instruction in the proper format (Format 1, 2, 3, or 4)

(4) Convert the data constant into internal machine representation.

eg → EOF → 4546

(End of file)

(5) Write the object program, and the assembly listing.

eg→ Head record, Text record, End record.

Assembler Directives → Assembler directives are Pseudo instructions or Pseudo opcodes. and they provide definition to the assembler itself.

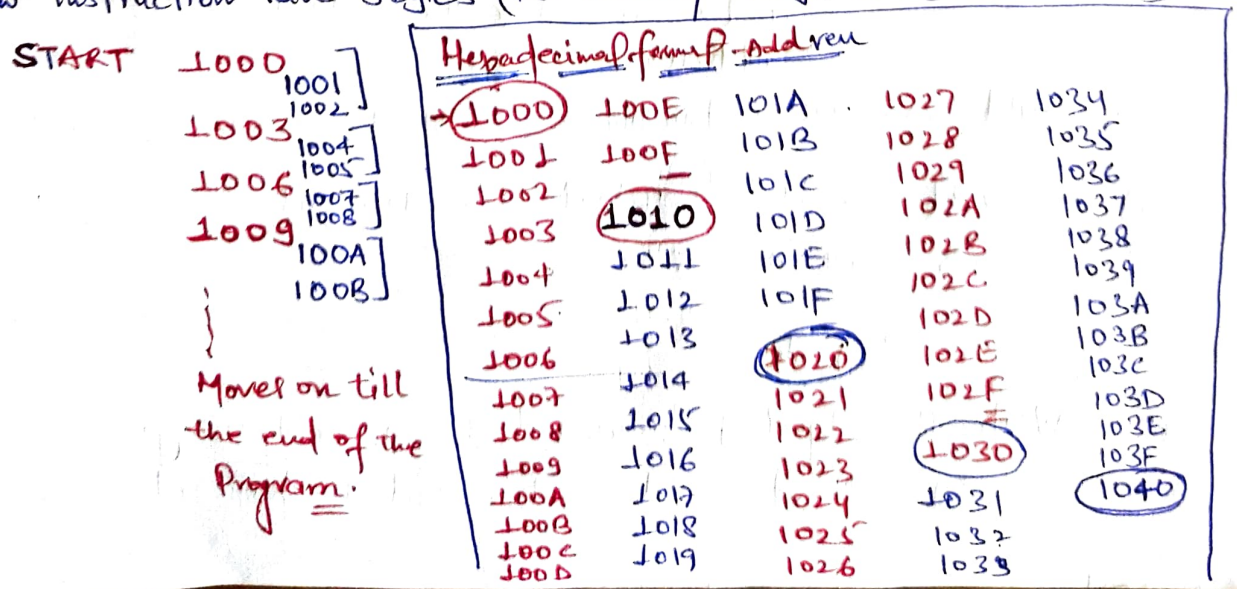
- They are not translated into machine operation code.
- In addition to the mnemonic machine instructions, we have used the following ~~ans~~ Assembler directives.

1. **START** – Specify the name and starting Address of the program.
   eg → value START 4000
2. **END** – Indicate the end of the source program and (optionally) specify the first executable instruction in the program.
   eg → END 400
3. **BYTE** – Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant
4. **WORD** – Generate one-WORD integer constant. (3 bytes)
5. **RESB** – Reserve the indicated number of bytes for a Data area.
   ~~eg RESB 10~~
6. **RESW** – Reserve the indicated number of words for a data area.
   RESW 10 – Reserve 10 WORDS (30 bytes)
   RESB 10 – Reserve 10 bytes

Example of START : COPY START 1000

Where → COPY is the name of the program.
→ START is the Assembler directives.
→ 1000 is the First executable instruction of the program.

- All the addresses are represented in hexadecimal format.
- Each instruction have 3 bytes (i.e. memory is ~~byte~~ WORD Addressable)



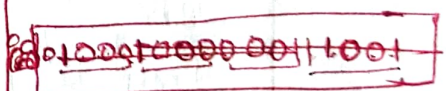
Line	Address (Loc)	Source Statements			Object code
5	1000	COPY	START	1000	→
10	1000	FIRST	STL	RETADR	→ 141033
15	1003	CLOOP	JSUB	RDREC	→ 482039
20	1006		LDA	LENGTH	→ 001036
25	1009		COMP	ZERO	→ 281030
30	100C		JEQ	ENDFIL	→ 301015
35	100F		JSUB	WRREC	→ 482061
40	1012		J	CLOOP	→ 3C1003
45	1015	ENDFIL	LDA	EOF	→ 00102A
50	1018		STA	BUFFER	→ 0C1039
55	101B		LDA	THREE	→ 00102D
60	101E		STA	LENGTH	→ 0C1036
65	1021		JSUB	WRREC	→ 482061
70	1024		LDL	RETADR	→ 081033
75	1027		RSUB		→ 4C0000
80	102A	EOF	BYTE	C'EOF'	→ 454F46
85	102D	THREE	WORD	3	→ 000003
90	1030	ZERO	WORD	0	→ 000000
95	1033	RETADR	RESW	1	→
100	1036	LENGTH	RESW	1	→
105	1039	BUFFER	RESB	4096	
110					
115					
120					
125	2039	RDREC	LDX	ZERO	→ 041030
130	203C		LDA	ZERO	→ 001030
135	203F	RLOOP	TD	INPUT	→ E0205D
140	2042		JEQ	RLOOP	→ 30203F
145	2045		RD	INPUT	→ D8205D
150	2048		COMP	ZERO	→ 281030
155	204B		JEQ	EXIT	→ 302057
160	204E		STCH	BUFFER, X	→ 549039
165	2051		TIX	MAXLEN	→ 2C205E
170	2054		JLT	RLOOP	→ 38203F
175	2057	EXIT	STX	LENGTH	→ 101036
180	205A		RSUB		→ 4C0000
185	205D	INPUT	BYTE	X'F1'	→ F1
190	205E	MAXLEN	WORD	4096	→ 001000
195					
200					
205					

SUB: Subroutine to Read record into Buffer.

Subroutine to write Record from Buffer.

STCH X BUFFER

54 1 1039



15 bits.

opcode	x	BUFFER
STCH	1	1039
54	1	1039

```
2010 - 2061   WRREC   LDX    ZERO     → 041030
2015 - 2064   WLOOP   TD     OUTPUT   → E02079
2020 - 2067           JEQ    WLOOP    → 302064
2025 - 206A           LDCH   BUFFER,X → 509039
2030 - 206D           WD     OUTPUT   → DC2079
2035 - 2070           TIX    LENGTH   → 2C1030
2040 - 2073           JLT    WLOOP    → 382064
2045 - 2076           RSUB            → 4C0000
2050 - 2079   OUTPUT  BYTE   X'05'    → 05
2051 - 2080           END    FIRST
```

SIC Assembler language program with object code.

Mnemonics	Machine code.
STL	14
JSUB	48
LDA	00
COMP	28
JEQ	30
J	3C
STA	0C
LDL	08
RSUB	4C
LDX	04
TD	E0
RD	D8
STCH	54
TIX	2C
JLT	38
STX	10
LDCH	50
LDL	08
WD	DC

Object code for inst no. 160

STCH	BUFFER, X

- index Register

Inst. format

8	1	15
Opcode	x	Buffer

24 bits in SIC

8	1	15
STCH	1	1039

m/c - 54

↳ we need to convert this no. into 15 bit binary.

0101 0110 1 0010000 0011 1001

5 4 9 0 3 9

object code.

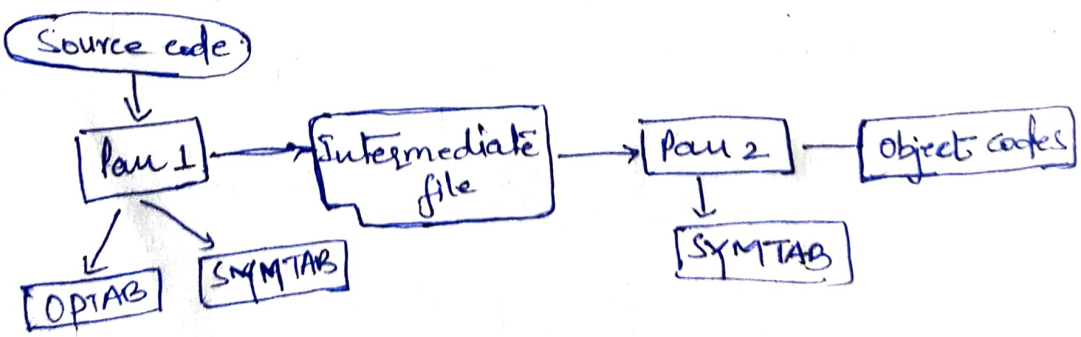
Q Generate the complete object program for the following Assembly language program. Assume standard SIC machine and codes in hexadecimal.

Given

LDA = 00 LDX = 04, STA = 0C, ADD = 18, TIX = 2C, JLT = 38, RSUB = 4C, JSUB = 48

Line no	Address (Location counter)	Label	Opcode	Operand	Object code
1		SUM	START	4000(h)	
2	4000	FIRST	LDX	ZERO	045788
3	4003		LDA	ZERO	005788
4	4006	LOOP	ADD	TABLE,X	18C015
5	4009		TIX	COUNT	2C5785
6	400C		JLT	LOOP	384006
7	400F		STA	TOTAL	0C578B
8	4012		RSUB		4C0000
9	4015	TABLE	RESW	2000 → 2000x3 = 6000 bytes	-
10	5785	COUNT	RESW	1 (0x1770)	-
11	5788	ZERO	WORD	0	-
12	578B	TOTAL	RESW	1	-
13	578E		END	FIRST	-

Pass 1 and Pass 2 Assembler



Object program

Header Record – only one
Text Record – Any no. depends on Program length
END Record – only one.

① Head Record

- Col 1 – H
- Col 2-7 Program Name
- Col 8-13 Starting Address of object program (Hexadecimal)
- Col 14-19 Length of object program in bytes (Hexadecimal)
  ↳ 578E
  4000
  178E – Length of the program.

② Text Record

- Col 1 – T
- Col 2-7 Starting Address of object code in this record (hexa)
- Col 8-9 Length of object code in this record in bytes (hexa)
- Col 10-69 object code, represented in hexadecimal (2 columns per byte of object code)

③ End Record

- Col 1 E
- Col 2-7 Address of First executable Instruction in the object program (hexa)

① H^SUM---^004000^00178E {2 col = 1 byte}

② T^004000^15^ (15 – Length of object code.) 045788^005788^18C015^2C5785^384006^0C5788^4C0000

Total object code = 7 and every code needs 6 bytes = 7×6 = 42 bytes = 21 bytes
Column ∵ 2 Column = 1 byte
42 col =
Total no of col = 42 — 2 column size 1 byte

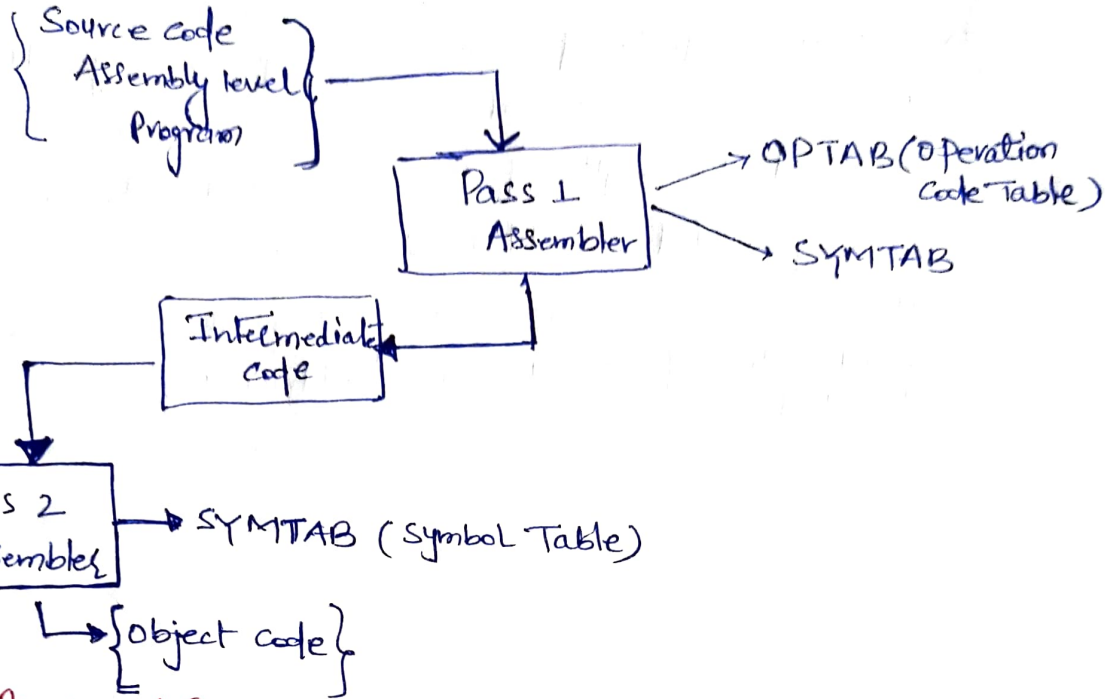
$\frac{42}{2}$ = 21 bytes

16	21
1	5

15 ⇒ hexadecimal of 21 is 15

③ E^004000

* Assembler Algorithm and Data structures



Functions of Pass 1 and Pass 2 Assembler

Pass 1 (Define symbols)

1. Assign Addresses to all Statements in the program.
2. Save the values (Addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of Assembler directives (This includes processing that affects addres assignment, such as determining the length of data areas defined by BYTE, RESW etc)

Pass 2 (Assemble instructions and generate object program)

1. Assemble instructions (translating operation codes and looking up addresses)
2. Generate data values define by BYTE, WORD etc.
3. Perform processing of Assembler directives not done during Pass 1.
4. Write the object program and the assembly listing.

- Our simple assembler uses two data structures / Two major internal data structures:
  - => The operation code Table (OPTAB) and The
  - => Symbol Table (SYMTAB)
  - ↳ OPTAB is used to Look up mnemonic operation codes and translate them to their machine language equivalents.
  - ↳ SYMTAB is used to store values (Addresses) assigned to labels

↳ We also need a location counter (LOCCTR). This is a variable that is used to help in the assignment of addresses.

- LOCCTR is initialized to the ~~beginning~~ beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR.

Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

Data structures used by Assembler

① LOCCTR (Location counter)

1. It is used to be a variable and help in the assignment of addresses.
2. LOCCTR initialized to be beginning address specified in the START statement.
3. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR.

② OPTAB (Operation Table)

1. It is used to look up mnemonics operation code and translate them to their machine language equivalent.

   eg → LDA 00 RSUB 48 etc.

2. In more complex assembler, this table also contains information about instruction format and length.
3. During pass 1, OPTAB is used to look up and validate operation codes in the source program.
4. During pass 2, it is used to translate the operation codes to machine language.

③ SYMTAB (Symbol Table)

SYMTAB is used to store values assigned to labels.

STL TABEL

Symbol	Address
TABEL	4015
LOOP	4006
FIRST	4000
COUNT	5785
ZERO	5788
TOTAL	5788

Line no	Location counter	Label	Opcode	Operand	Object code
1		SUM	START	3000(h)	
2	3000	FIRST	LDX	ZERO	043180
3	3003		LDA	THREE	003015
4	3006	LOOP	ADD	TABLE,X	18B018
5	3009		TIX	COUNT	2C3144
6	300C		JLT	LOOP	383006
7	300F		STA	TOTAL	0C3183
8	3012		RSUB		4C0000
9	3015	THREE	WORD	3	000003
10	3018 + (12C)	TABLE	RESW	100 — 12C — Byte	
11	3144 + (3C)	COUNT	RESW	20	100W = 300 Bytes
12	3180	ZERO	WORD	0	—
13	3183	TOTAL	RESW	1	—
14	3186		END	FIRST	—

LDA - 00
LDX - 04
STA - 0C
ADD - 18
TIX - 2C
JLT - 38
RSUB - 4C

3186
3000
0180

Object Program

* Head Record
  - Col 1 - H
  - Col 2-7 - Name of the program
  - Col 8-13 - Address (Starting Add.)
  - col 14-19 - length of the object code.

(Only one head record)

H^SUM---^003000^000186

* Text Record
  - col 1-T
  - col 2-7 starting Add
  - col 8-9 - object code length.
  - 10-69 - object codes. (2 columns per byte of object code)

(Many Text Record may be possible)

T^003000^18^043180^003015^18B018^2C3144^383006^
0C3183^4C0000^000003.

Object codes = 8×6 = 48 col. 48/2 = 24 → (18) In hexadecimal

* End Record
  - Col 1 - E
  - Col 2-7 - Address of the first executable instruction in the program.

(Only one)

E^003000

SYMTAB

LABEL	Add.
FIRST	3000

Algorithm for 2 pass Assembler (2 Pass Assembler)

Pass 1:

```
begin
read first input line
if OPCODE = 'START' then
    begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line
    end {if START}
else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
        if this is not a comment line then
            begin
                if there is 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 the 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}
```

* Pass 2 Assembler Algorithm

```
begin
  read first input line (from intermediate file)
  if OPCODE = 'START' then
  begin
      write listing line
      read next input line
    end {if START}
  write Head 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 address
        else
            begin
            store 0 as operand address
            set undefined 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}
```