Topic

# Module - (1) ⇒

## (A) The Simplified Instructional Computer (SIC)

SIC comes in two versions →
- The standard model (SIC)
- extra equipment version (SIC/XE)

→ The two versions have been designed to be upward compatible — i.e an object program for the standard SIC m/c will also execute properly on a SIC/XE system.

### (A) SIC Machine Architecture :-

#### (1) Memory :-

→ Memory consists of 8 bit bytes, any 3 consecutive bytes form a word (24 bits)

→ All addresses on SIC are byte addresses,

→ There are total $2^{15}$ bytes in the computer memory.

#### (2) Registers :-

→ There are 5 registers.

→ Each register is 24 bits in length

The following table indicates the number, mnem and uses of these registers:-

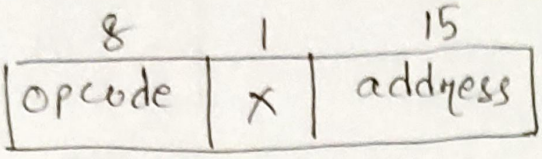
| Mnemonic | Number | Use |
|---|---|---|
| A | 0 | Accumulator, used for arithmatic operations. |
| X | 1 | Index register, used for addressing. |
| L | 2 | Linkage register; the jump to subroutine (JSUB) instruction stores return add. in this register. |
| PC | 8 | Program Counter; contains the address of the next instruction to be fetched for execution. |
| SW | 9 | Status word; contains a variety of information, including a condition code (CC) |

(3) Data Formats :-

- Integers are stored as 24-bit binary numbers;
- 2's complement representation is used for -ve 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 :-




The flag bit- x=1 is used to indicate indexed-addressing mode.

(5) Addressing Mode :-

There are 2 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.

→ ( ) is used to indicate the contents of a register or a memory location. ex: (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 basic set of instructions - like,

(i) load and store registers → LDA, LDX, STA, STX

(ii) Integer Arithmatic operations → ADD, SUB, MUL, DIV } involve register 'A' and a word in memory with the result being stored in memory.

(iii) COMP - compares the value in register 'A' with a word in memory. This instruction sets a condition code (CC) to indicate the result - (<, =, >).

(iv) Conditional Jump Instructions → JLT, JEQ, JGT } Can test the setting of CC & jump accordingly.

- Subroutine Linkage
  - JSUB (jumps to subroutine) placing the return address in reg. L
  - RSUB (returns by jumping to the address contained in register L)

## (7) Input and Output :-

I/O in SIC standard version is performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A. Each device is assigned a unique 8 bit code.

* There are 3 I/O instructions

Each of them specifies the device code as an operand.

- TD (Test device)
  Tests whether the addressed device is ready to send or receive a byte of data. CC is set to indicate the result.
  - A setting '<' means ready
  - A setting '=' means not ready
- RD (Read data)
- WD (write data)

# Module - (1)

Topic (13) * **SIC / XE Machine Architecture** :-

## (1) Memory :-

Same as SIC:
- → Memory consists of 8 bit bytes
- → Any 3 consecutive bytes form a word (24 bits)
- → All addresses are byte addresses.

→ Maximum memory available is $2^{20}$ bytes (1 mega-byte)

* This increase in memory leads to a change in instruction formats and addressing modes.

## (2) Registers :-

Explain them also

Along with Registers present in SIC (Register - A, X, L, PC, SW) the following additional registers are provided by SIC/XE: There are total 9 registers (A, X, L, PC, SW, B, S, T, F)

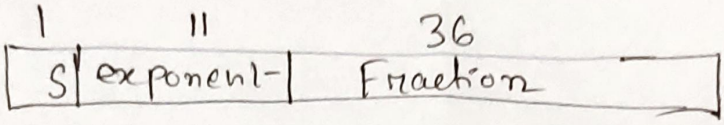
| Mnemonic | Number | Purpose |
|---|---|---|
| B | 3 | Base register; used for addressing. |
| S | 4 | General working register — no special use. |
| T | 5 | General working register — no special use. |
| F | 6 | Floating-point accumulator (48 bits) |

* Each register here is of 24 bit length except the floating pt. register

## (3) Data formats :-

→ SIC/XE provides the same data formats as the standard version (SIC), i.e - integer, -ve value, etc.
Explain them also

→ In addition, there is a 48 bit- floating point- data type with the following format- :-




→ The fraction is interpreted as a value between 0 & 1.

→ The exponent- is interpreted as an unsigned binary number between 0 and 2047.

→ If the exponent- has value 'e' and the fraction has value 'f', the absolute value of the number represented is : $f * 2^{(e-1024)}$

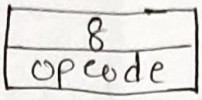
→ The sign (s) of the floating pt. number is indicated by the value of S (0 = +ve, 1 = -ve).

→ A value '0' is represented by setting all bits (including sign, exponent-, and fraction) to 0.

## (4) Instruction Formats :-

The larger memory available on SIC/XE means (2^20 bytes) that an address will no longer fit into a 15 bit field as was in SIC. Therefore the SIC instruction format is no longer suitable in SIC/XE. Following are the instruction formats for SIC/XE machine :-

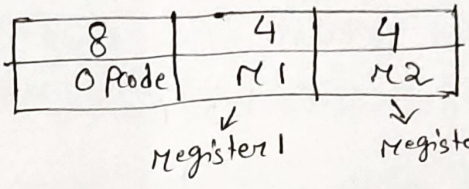
Format-1 (1 byte)




→ no operand or registers used.

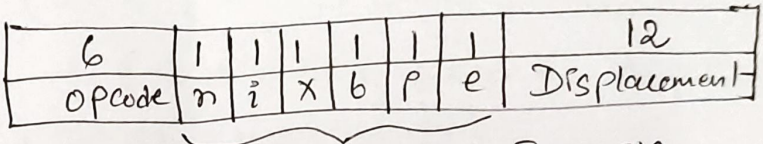
Example
RSUB

Format-2 (2 bytes)




Example
ADDR S, T

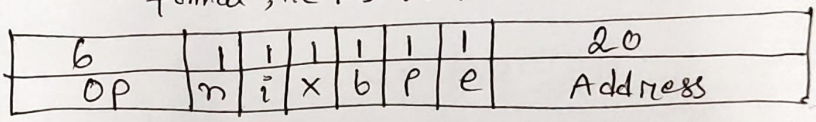
Format-3 (3 bytes)
F3




used to represent- → kind of addressing mode & what-type of format-, i.e F3 or F4.

Example
LDA #3

Format-3 (4 bytes)
F4




Example
+JSUB RDREC

Subroutine for reading record

Here e value = 1, i.e it is format-4.
e value = 0, i.e it is format-3.
n = 1, indirect- addressing mode.
n = 0, normal " "

Here + is used to represent- Format-4.

i = 1, immediate addressing mode.
i = 0, not immediate addressing mode

x = 0, not indexed mode
x = 1, indexed mode.

b = 1, base relative addressing mode
p = 1, Program counter relative addressing mode.

## (5) Addressing Mode and Flag bits :-

→ Base relative (n=1, i=1, b=1, p=0)
→ Program Counter relative (n=1, i=1, b=0, p=1)
→ Direct (n=1, i=1, b=0, p=0)
→ Immediate (n=0, i=1, x=0)
→ Indirect (n=1, i=0, x=0)
→ Indexed (both n & i=0 or 1, x=1)
→ Extended (e=1 for format-4, e=0 for format-3)

## (6) Instruction Set :-

→ Load & Store registers – LDA, LDX, STA, STX, LDB, STB (additional in SIC/XE)

→ Integer Arithmatic operations – ADD, SUB, MUL, DIV

→ Floating Point arithmatic OP- ADDF, SUBF, MULF, DIVF

→ COMP - comparison Instruction (Same purpose as in SIC) → explain as given in SIC

→ Conditional Jump instructions — JLT, JGT, JEQ (Same as in SIC) → explain as given in SIC

→ Subroutine Linkage — JSUB, RSUB (Same as in SIC) → explain as given in SIC

→ Register Move Instruction: RMO (move value from 1 register to another register)

→ Register-to-register arithmatic op — ADDR, SUBR, MULR, DIVR.

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

---

The operations are same as in SIC

TD (Test device)
RD (Read data)
WD (write data)

} ⇒ explain as is given in SIC

# A simple SIC Assembler →

Topic

## Basic Assembler Functions :-

The translation of source program to objeul-code requires us to accomplish the following functions :-

① Convert Mnemonic operation codes to their machine language equivalent - e.x - translate STL to 14.

② Convert symbolic operands to their equivaleul-m/c address. ex: translate LDA to 4003.

③ Build the machine instructions in the proper format.

④ Convert the data constaul-s specified in the source program into their internal machine representation.
ex: translate EOF to 454F46

⑤ Write the objeul-Program and assembly listing.

---

→ All of these functions exeept no. ② can easily be accomplished by sequential processing of the source program, one line at a time. Translation of address however presents a problem, Consider line no. ⑩ in foll. prog.

### Sample SIC program

| Line | Loc | Src. Statement- Label | Mnemonic opcodes | Symbolic operands | Objeul-code |
|---|---|---|---|---|---|
| 5 | 1000 | copy | START | 1000 | — |
| 10 | 1000 | FIRST | STL | RETADR | |
| ⋮ | ⋮ | | | | |
| 95 | 1033 | RETADR | RESW | 1 | |

forward reference

The instruction in line no. (10) contains a forward reference i.e, a reference to a label (RETADR) that is defined later in this sample prog. If we attempt to translate the program line by line, we will be unable to process this statement because we don't know the address that will be assigned to RETADR. Because of this, most assemblers make 2-passes over the source program.

Pass-(1) :→ Assign addresses to all the statements
→ Save the addresses assigned to all labels to be used in pass-2,
→ Perform some processing of the assembler directives such as - RESW, RESB to find the length of data areas for assigning the address value.

→ define the symbols in the symbol table

Pass-2 :→ Assemble the instructions (Translating the OP. codes & looking up the addresses)

→ Generate data values defined by BYTE, WORD etc.

→ Perform the processing of the assembler directives not done during Pass-1.

→ Write the object Program and assembler listing.

## Topic /SIC Assembler Directives/ :-

START : Specifies name & starting address for the program.

END : Indicates the end of the source program and specify the 1st executable instruction in the program.

BYTE : Generates character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

WORD : Generates one-word integer constant.

RESB : Reserves the indicated no. of bytes for a data area.

RESW : Reserves the indicated no. of words for a data area

## Format of object Program :-

Header Record :-

Col. 1 : H

Col. 2 - 7 : program name

Col. 8 - 13 : starting address of obj. program (hexadecimal)

Col. 14-19 : length of object program in bytes (hexadecimal)

Text Record :-

Col 1 : T

Col. 2-7 : starting address for object code in this record (hexadecimal)

Col: 8-9 : length of object code in this record in bytes (hexadecimal)

Col 10-69 : Object code, represented in hexadecimal (2 columns per byte of object code)

End Record :-

Col 1 : E

Col 2-7 : Address of first executable instruction in object program (hexadecimal)

## (*) Sample problem ①

Convert the following source program into an object program.

Source Program (SIC)

Given M/C code (or)

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

**Step ①: object code Generation for each statement.**

| | LOC | Label | Mnemonic opcode | Symbolic operand | Objal-code |
|---|---|---|---|---|---|
| 1) | 4000 | SUM | START | 4000 | — |
| 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 → | — |
| 10) | 5785 | COUNT | WORD | 10 (A) → | 00000A |
| 11) | 5788 | ZERO | WORD | 0 (0) → | 000000 |
| 12) | 578B | TOTAL | RESW | 1 → | — |
| 13) | 578E | | END | FIRST | |

↓ 2000*3 = 6000-byte = 1770-hexadecimal

Line no. (2) Object-code creation :-

| opcode (8) | x (1) | add. of operand (15) |
|---|---|---|
| LDX 04 — 0000 0100 | 0 | ZERO: 0101 0111 1000 1000 |

in the operand field no 'x' is there. so, x-indexed mode is 0. It is direct-addressing mode.

address of "ZERO" but we have space only for 15 bits, so neglect 1st bit.

Now group every 4 bits. ∴ object-code = 045788

Line no. (4) object-code creation :-

| opcode (8) | x (1) | address of operand (15) (TABLE) |
|---|---|---|
| ADD 18 — 0001 1000 | 1 | 0100 0000 0001 0101 |

Object-code = 18C015

In the operand field 'x' is there.

So, x-indexed add. mode is 1.

Now group every 4 bits. ∴ object-code = 18C015

Note

→ For RESW, RESB, object-codes are not-generated.

→ For WORD, the value of Symbolic operand is only represented as 6 digit (hexadecimal format) object-code

step ② Object Program Generation :-

H^SUM__^004000^00178E

T^004000^15^045788^005788^18C015^2C5785^38[illegible]

T^005785^06^00000A^000000 | ^0C5788^4C0000

[illegible]

E^004000

→ Here H is representing Header Record
T " " Text Record
E " " End Record.

→ Text record can occupy maximum 60 columns. (Col 10–69)
while writing the object codes in text record, if there is a break, (i.e if there is no object-code, for any statement in the source program), then for the next object-code sequence (after the break), new Text-Record need to be initiated.

→

Topic / Assembler Algorithm & Data structure /

Assembler uses 2 major internal data structures. The operation code Table (OPTAB) and the Symbol Table (SYMTAB). There is another variable - LOCCTR to help in assigning addresses.

⇒ **OPTAB:-** OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalent.

→ The OPTAB must contain (at least) the mnemonic operation code and its machine language equivalent. In more complex assemblers, this table may also contain information about instruction format & length.

→ During Pass-1, OPTAB is used to look up and validate operation codes in the source program.

→ In Pass-2, it is used to translate the operation codes to machine language.

→ OPTAB is usually organized as a hash table, with mnemonic operations code as the key and in most cases it is a static table.

⇒ **SYMTAB** :- The SYMTAB includes the name & value (address) for each label in the source program together with flags to indicate error conditions (ex: a symbol defined in 2 different places)

→ During pass-1, labels are entered into SYMTAB as they are encountered in the source program along with their assigned addresses (From LOCCTR)

→ During pass-2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

→ SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.

⇒ **LOCCTR** :- LOCCTR is a variable that is used to help in the assignment of addresses.

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

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

Topic (V. Imp) ***

## Algorithm for Pass (1) of a 2-pass Assembler

```
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
```

* Pass (1) is for Generating Object-code

```
else if OPCODE = 'RESB' then
    add #[OPERAND] to LOCCTR
else if OPCODE = 'BYTE' then
    begin
        find length of constant in bytes
        add length to LOCCTR
    end {if BYTE}

else
    set error flag (invalid operation code)

end {if not a comment}
write line to intermediate file
read next input line
end {while not END}
write last line to intermediate file
save (LOCCTR - starting address) as program length
end {pass 1}
```

Topic

# Algorithm for Pass-② of a 2-pass Assembler

```
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 address
                                    else
                                        begin
                                            store 0 as operand address
                                            set error flag (undefined symbol)
                                        end
                                end { if symbol }
```

* Pass-② is for generating object program

```
            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}
```

(*) Problem – (2)

Imp Convert the following source Program into an (SIC/XE) object Program.

Given, CLEAR = B4, LDA = 00, LDB = 68, JLT = 38, RSUB = 4C
ADD = 18, TIX = 2C, STA = 0C

| Line no. | LOCCTR | LABEL | OPCODE | OPERAND | OBJECT CODE |
|---|---|---|---|---|---|
| 1) | | SUM | START | 0 | |
| 2) | 0000 | FIRST | CLEAR | X | — B410 |
| | | | LDA | #0 | — 010000 |
| 3) | 0002 | | | | |
| 4) | 0005 | | +LDB | #TOTAL | — 6910178B |
| 5) | | | BASE | TOTAL | — |
| 6) | 0009 | LOOP | ADD | TABLE,X | — 1BA00F |
| 7) | 000C | | TIX | COUNT | — 2F2009 |
| 8) | 000F | | JLT | LOOP | — 3B2FF7 |
| 9) | 0012 | | STA | TOTAL | — 0F4000 |
| 10) | 0015 | | RSUB | | — 4F0000 |
| 11) | 0018 | COUNT | RESW | 1 | |
| 12) | 001B | TABLE | RESW | 2000 | |
| 13) | 178B | TOTAL | RESW | 1 | |
| 14) | 178E | | END | FIRST | |

0000
Starting add.

## Instruction Formats in SIC/XE

Type 1 = 1 byte,
Type 2 = 2 byte
Type 3 = 3 byte
Type 4 = 4 byte

→ Line Sl. NO. - (2) is a type 2 instruction as there is a register 'X' involved in this instruction. & no other operand

→ Line no. (3) is a type 3 instruction as there is no register involved in the instruction & also there is no '+' symbol before the opcode/Mnemonic.

→ Line no. (4) is a type 4 instruction as '+' is there.

→ Line no. (5) involves assembler directive 'BASE', ∴ no address need to be calculated for this line.

### (*) Object-Code Generation

* line no. (2) :- CLEAR X → B4 → register no. = 1

Format (2)

| opcode | | R1 | R2 |
|---|---|---|---|
| 1011 | 0100 | 0001 | 0000 |
| B | 4 | 1 (X=1) | 0 (no data.) |

### Register Mnemonic

| Reg. | NO |
|---|---|
| A | 0 |
| X | 1 |
| L | 2 |
| B | 3 |
| S | 4 |
| T | 5 |
| F | 6 |
| PC | 8 |
| SW | 9 |

* line no. (3) :- LDA #0 → 00

Format (3)

| opcode | (1) n | (1) i | (1) x | (1) b | (1) p | (1) e | displacement (12) |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 0 | 0 | 0 | 0000 0000 0000 |

0000 0000 → take most significant bits → 0

i → immediate addressing

0 → type 3

0 → No displacement when there is immediate addressing.

Now take 4 bits together to form Object-code = 010000

line no. ④ : +LDB #TOTAL (68)

Format-④

| opcode(6) | n (1) | i (1) | x (1) | b (1) | P (1) | e (1) | address (20) |
|---|---|---|---|---|---|---|---|
| 0110 10ØØ | 0 | 1 | 0 | 0 | 0 | 1 | 0178B |
| 6 | 8 | | | | | | |

each digit occupies 4 bits.
∴ 4 bits × 5 digits = 20 bits.

Now take 4 bits together & form object code.

6910178B

* line no. ⑤ - for BASE no object code Generation.

line no. 6 : ADD TABLE, X (18)

| opcode(6) | n | i | x | b | P | e | displacement |
|---|---|---|---|---|---|---|---|
| 000110ØØ | 1 | 1 | 1 | 0 | 1 | 0 | 00F |

TABLE add. — PC value

Disp = 001B — 000C = F (falls in range.)

F is present in the range = −2048 to 2047

∴ The instruction is PC relative mode.

∴ object code = 1BA00F

line no. 7 :

2C-TIX COUNT

| opcode(6) | n | i | x | b | P | e | displacement |
|---|---|---|---|---|---|---|---|
| 0010 11ØØ | 1 | 1 | 0 | 0 | 1 | 0 | 009 |

Disp = COUNT − 000F = 0018 − 000F = 9 (hexa) (falls in range (binary value)

PC relative :- −2048 to 2047

object code : 2F2009

line no. 8 /

3E
JLT LOOP

| opcode | n | i | x | b | p | e | displacement |
|---|---|---|---|---|---|---|---|
| 0011 10ØØ | 1 | 1 | 0 | 0 | 1 | 0 | FF7 |

Disp: 0009 − 0012 = −9 → FF7

As −9 is negative; 2's complement = FF7 (binary value)

PC relative : −2048 to 2047.

~~Dis~~. objel- code :- 3B2FF7

---

line no. 9 /

0C
STA TOTAL

| opcode | n | i | x | b | p | e | displacement |
|---|---|---|---|---|---|---|---|
| 000011ØØ | 1 | 1 | 0 | 1 | 0 | 0 | 000 |

Displacement = 178B − 0015 = 1776 (hexa)

if we convert this to binary / decimal value. it wont fall in the range of PC relative add. mode. −2048 − to − 2047.

Another reason is, 1776 is of 4 digit, if we convert to binary, it will occupy 4 digit × 4 bit = 16 bits but in displacement we have only 12 bit-position.

∴ Considering base relative mode,

displacement = TOTAL − BASE (operand is again TOTAL)

= 178B − 178B

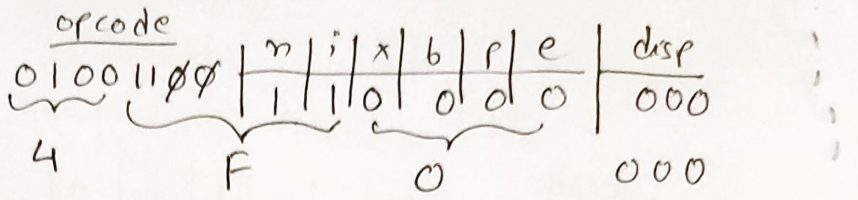
= 0

∴ Considering 4 bits together,

objel- code = 0F4000

line no. 10 / 4C

RSUB –




Object-code - 4F0000, There is no operand. ∴ direct address

---

line no. 11, 12, 13, 14 are reserved words and assembler directives, so we need not write object-code.

---

/Object-Program/ : *

H^SUM---^000000^178E

length = 178E – 0000
= 178E

T^000000^18^B410^010000^69101788^1BA00F^2F2009
^3B2FF7^0F4000^4F0000

E^000000

tot 48 columns are there.
2 col = 1 byte
∴ tot. 24 bytes are there.
which is 18 in hexadecimal.