## horizontal line



SIC/XE Assembler with Control section

CSN-252 Project

**─**

By:

Mudit Gupta

21114061

Batch- O3

CSE

# Overview

SIC/XE stands for Simplified Instructional Computer Extra Equipment or Extra Expensive . This computer is an advance version of SIC . Both SIC and SIC/XE are closely related to each other that’s why are upward compatible.

1. Memory

Memory consists of 8 bit-bytes and the memory size is 1 megabytes (220 bytes). Standard SIC memory size is very small. This change in the memory size leads to change in the instruction formats as well as addressing modes. 3 consecutive bytes form a word (24 bits) in SIC/XE architecture. All address are byte addresses and words are addressed by the location of their lowest numbered byte.

## Registers

It contains 9 registers:

### A: Accumulator

### X: Index Register

### L: Linkage register

### B: Base register

### S: General working register

### T: General working register

### F: Floating Point Accumulator

### PC: Program Counter register

### SW: Status Word register

## Data Formats

Integers are represented by binart numbers . Characters are represented using ASCII codes . Floating points are represented using 48 bits .

## Instruction Formats

In SIC/XE architecture there are 4 types of formats available. The Bit(e) is used to distinguish between Formats 3 and 4 . e = 0 means Format 3 and e = 1 means Format 4 .

# Assembler

• Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.

• It generates instructions by evaluating the mnemonics in operation field and find the value of symbol and literals to produce machine code .

## • Pass 1 :

1. Define symbols and literals and remember them in symbol table and literal table respectively.

2. Keep track of location counter .

3. Process pseudo – operations .

## • Pass 2 :

1. Generate object code by converting symbolic op code into respective numeric op-code.

2. Generate data for literals and look for values of symbols.

# Assembler: Design

## Utils.cpp

This file contains all useful functions that will be required throughout the assembling process.

Description of functions:

* 1. str\_to\_dec: function to convert decimal string to integer.
  2. str\_to\_hex: function to convert hexadecimal string to integer.
  3. is\_number: function to find if a string is a valid decimal number.
  4. is\_hex: function to determine if a string is a valid hexadecimal number.
  5. str\_to\_ascii: Function to convert string to ASCII integer.
  6. get\_words: get 3 parameters from an instruction line (label, mnemonic and operand).
  7. find\_parameters: finds parameters in operand which are separated by plus(‘+’) or minus(‘-’).
  8. to\_hex: function to convert integer to hexadecimal string of given length.
  9. split\_comma: function to split a string according to comma if present.
  10. pad: function to pad a string with spaces to make it of given length.

## tables.cpp

It has init\_tables() function which initializes all values of OPTAB and REGISTERS.

## structs.cpp

It contains all structures required in the assembler:

1. info\_reg: stores info of register
2. info\_literal: stores info of literal
3. info\_op: stores info of operand
4. info\_sym: stores info of symbols
5. info\_mod: stores info of modification records

## pass1.cpp

This file is used to implement first pass of the assembler. It contains the function run\_pass1() which contains all operations to be done in first pass. The final result of this function is the intermediate file (intermediate.txt) and the error file (error.txt) containing all errors. Here, we first read input file (input.txt) line by line and extract label, mnemonic and operand from it. For each line, it creates an instruction object and push it in the instruction vector of the current control-section object, only if it is a programming instruction (if it is comment then it is ignored).

There is locctr variable which maintains the current value of location counter and it is incremented for each instruction according to its size.

If-else is used to differentiate between different types of instructions. If the opcode is START, a new control-section object is created and we initialize start address and LOCCTR as 0. Then for END and CSECT opcodes, we set the end address of previous control-section and also create a literal pool (if literals are there). For END, we break from loop, and for CSECT, we create another control-section object.

Then, we handle assembler directives like WORD, BYTE, RESB, BASE, NOBASE, ORG, EQU, EXTREF and EXTDEF. For LTORG, we push all literals into LITTAB and also we create new instruction objects for each of them. For EQU, we check if the expression is valid or not and give error accordingly and then we push it to SYMTAB.

Then we check if there is a label in the line, if present we check if it is present in the SYMTAB, if found we print error saying ‘Duplicate symbol’ in the error file or else assign name, address and other required values to the symbol and store it in the SYMTAB.

Then we check condition for literal, it is a literal, then we push it to a local vector which will be emptied in next literal pool.

Then we verify if the opcodes are present in OPTAB or not , if present we find its format and accordingly increment LOCCTR .

After the loop ends , we store the program length and then go on for printing the SYMTAB , LITTAB and intermediate file.

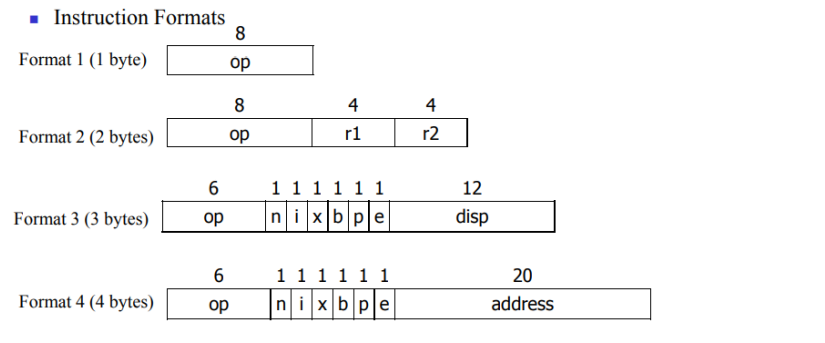
## Pass2.cpp

Here we use the addresses generated in first pass and generate the final output file (output.txt) which has the object program and also the error file. It contains a function run\_pass2() which implements second pass.

Similar to first pass, here also we process each instruction one-by-one and calculate its object code.

We first check if instruction is a literal and set its object code accordingly. Then we check for BYTE and WORD and evaluate the expression if needed and give error accordingly. Then we check for other assembler directives like EXTDEF and store its data in the def vector of our control-section object.

Then we check it the opcode is present in OPTAB, if not then we give error, else we find its object code according to its format:



For each format, we also check if the instruction is valid, for e.g: in format 3 and 4, we check if it is both immediate and index relative and give error accordingly.

## file\_handler.cpp

This file contains variables to access various file:

1. Input: For input file (input.txt)
2. Output: For output file (output.txt)
3. intermediate: for intermediate file (intermediate.txt)
4. Error: for error file (error.txt)
5. Symtab: for symbol table file (symtab.txt)
6. Littab: for literal table file (littab.txt)

It also contains functions to handle these files:

1. open\_files(): To open all files.
2. check\_files(): To check if all files are open.
3. close\_files(): To close all files.

## main.cpp

This is out main file form which we run both passes. It importes all necessary libraries and all the files mentioned above. It first runs init\_tables() to initialize all tables, then open\_files() to open all files and then check\_files() to check if all needed files are opened correctly. Then it runs first pass by run\_pass1() then second pass by run\_pass2(). We then close all files by close\_files().

# Assembler : Data Structures

## REGISTERS

Contains information of the registers like its numeric equivalent , character representing , whether such register exists or not .

## LITTAB

Contains information of literals like its value , address , block number , a character representing whether the literal exists in literal table or not .

## SYMTAB

Contains information of labels like , name , address , block number , character representing whether the label exists in the symbol table or not , an integer representing whether label is relative or not .

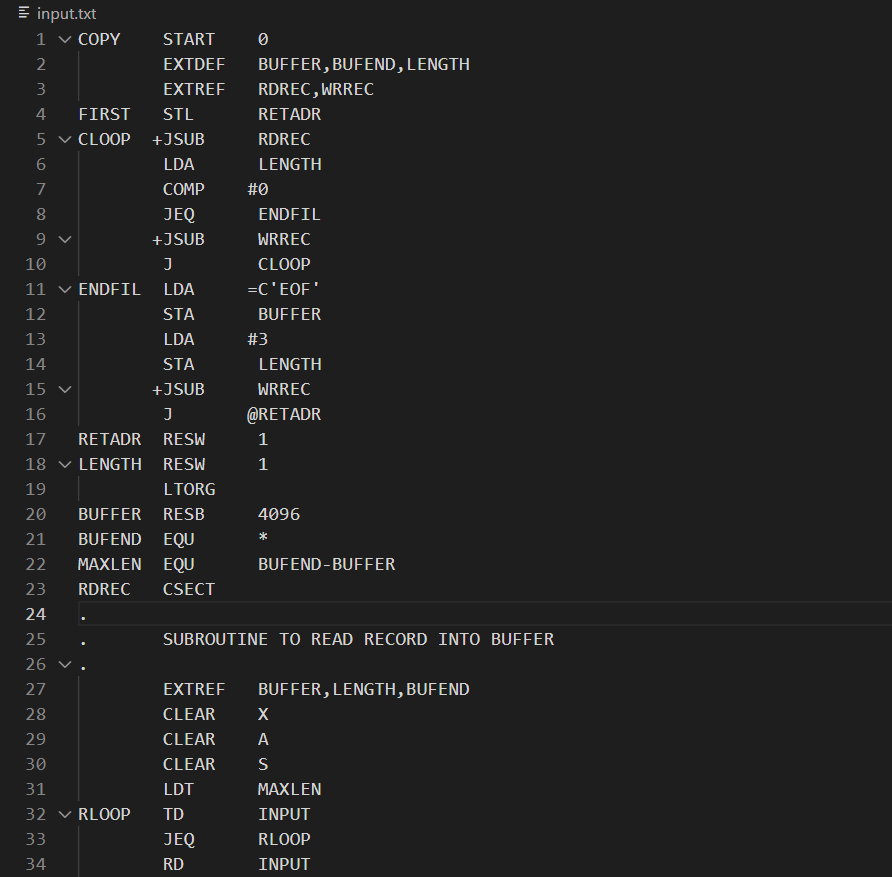
## OPTAB

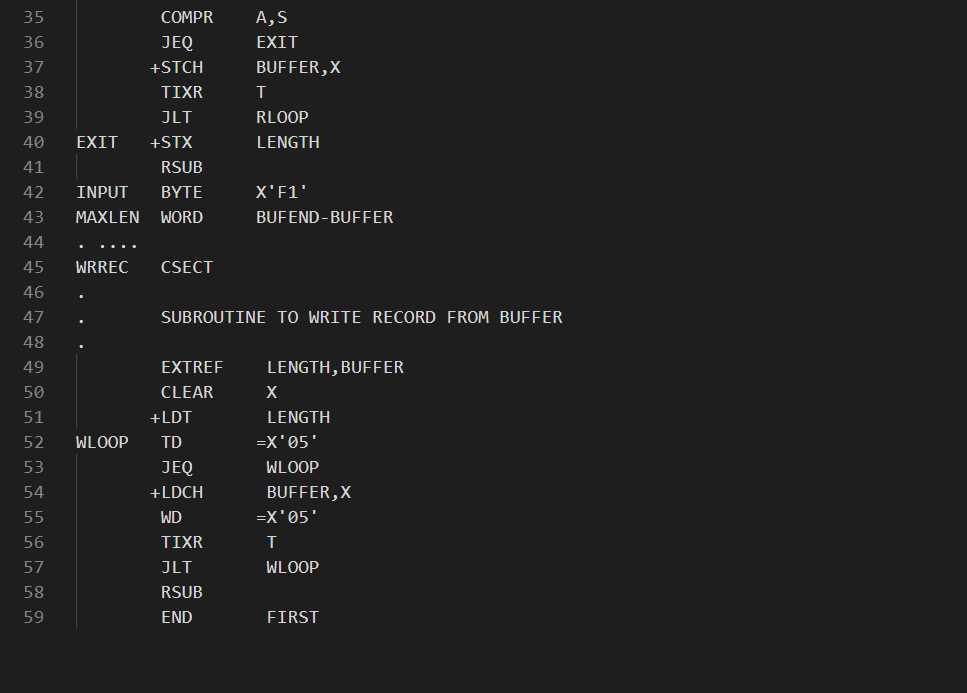
Contains information of opcode like name , format , a character representing whether the opcode is valid or not .

# Steps to compile and execute the assembler

1. Download and Extract the zip file.
2. Write the input in input file (input.txt)
3. Run the file main.cpp:
4. Desired files- output.txt, intermediate.txt, error.txt, symtab.txt and littab.txt will be created.

# Input Format



Each part of an instruction should be space separated and if the instruction does not have any label, then a space at beginning is needed.

For comments, start the line with dot (‘.’) and then space.

# Output format

# Intermediate file

# Symbol Table

# Literal Table

