**Problem Statement:**

Design and implement a version of SIC/XE assembler yourself to demonstrate major functions of a two-pass SIC/XE assembler: Pass 1 and Pass 2. Consider all the SIC/XE instructions (given below) from the textbook as the instruction set for your assembler design and implementation. Please ensure to include Format 2, 3 and 4

instructions in the set.

Input to your assembler: Assembler source programs (e.g., several control sections) using the instruction set of SIC/XE.

Output required:

a) Symbol Table from Pass 1.

b) Relocatable object program with object code assembled from Pass 2 (object code assembled from

each instruction to the assembler).

c) Each object program should include the following types of record: H, D, R, T, M, and E types.

The report contains the following:

1. Project introduction and background.
2. Architecture and design of the software
3. Algorithms and data structures used in the implementation
4. Design results (design analysis and flow charts and documentation)
5. Source code and implementation results
6. Conclusion (summary and insights gained from the work of project design and implementation)
7. **PROJECT INTRODUCTION AND BACKGROUND**

**Introduction:**

An **SIC/XE assembler** is a software program that translates assembly language source instructions into machine language instructions, on a one to one basis. This means that each source instruction is translated into exactly one target instruction. However, it is not a precise definition because an assembler can do (and usually does) much more than just translation. It offers a lot of help to the programmer in many aspects of writing the program. The many types of help offered by the assembler are grouped under the general term directives (or pseudo-instructions).The assembler directives are not translated into machine instructions. Instead, they provide instructions to assembler itself.  A modern assembler can support as many as a hundred directives. Like early programming languages such as Fortran, Algol, Cobol and Lisp, assemblers have been available since the 1950s and the first generations of text based computer interfaces. However, assemblers came first as they are far simpler to write than compilers for high-level languages.

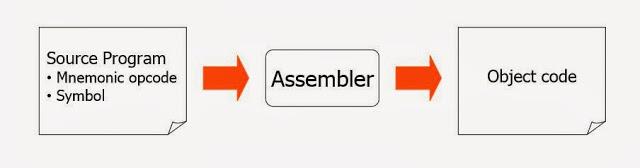
One reason for studying assemblers is that the operation of an assembler reflects the architecture of the computer. The assembler language depends heavily on the internal organization of the computer. Architectural features such as memory word size, number formats, internal character codes, index registers, and general purpose registers, affect the way assembler instructions are written and the way the assembler handles instructions and directives. This fact explains why there is an interest in assemblers today and why a course on assembler language is still required.

**Background:**

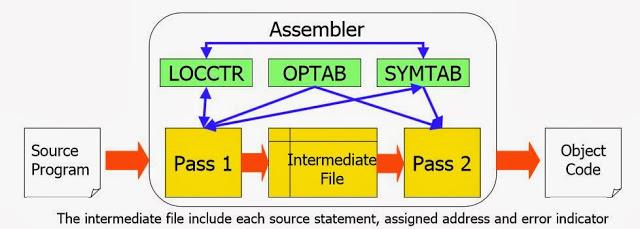
The history of development of an SIC/XE assembler goes way back in time. The Electronic Delay Storage Automatic Calculator (EDSAC) had an assembler (named "initial orders") integrated into its bootstrap program that used one-letter mnemonics. EDSAC was developed at Cambridge University in 1949 by Maurice Wilkes and W. Renwick. It was credited as first "assembler" by IEEE computer society. It was implemented in a read-only memory formed from a set of rotary telephone selectors, and it accepted symbolic instructions. Each instruction consisted of a one letter mnemonic, a decimal address, and a third field that was a letter. The third field caused one of 12 constants preset by the programmer to be added to the address at assembly time. Wilkes was also the first to use labels and macros and the first to develop a subroutine library. The IBM 650 computer(around 1953) used labels in assembler and this assembler was very similar to current assemblers. SOAP (Symbolic Optimizer and Assembly Program) did symbolic assembly in the conventional way, and was perhaps the first assembler to do so. However, its main feature was the optimized calculation of the address of the next instruction. The IBM 650 was based on a magnetic drum memory and the program was stored in that memory. Each instruction had to be fetched from the drum and had to contain the address of its successor. For maximum speed, an instruction had to be placed on the drum in a location that would be under the read head as soon as its predecessor was completed. SOAP calculated those addresses, based on the execution times of the individual instructions. One of the first commercially successful computers was the IBM 704. It had features such as floating-point hardware and index registers. It was first delivered in 1956 and its first assembler, the UASAP-1, was written in the same year by Roy Nutt of United Aircraft Corp. It was a simple binary assembler, did practically nothing but one-to-one translation, and left the programmer in complete control over the program. UASAP has pointed the way to early assembler writers, and many of its design principles are used by assemblers to this day. In the same year another assembler, the IBM Autocoder was developed by R. Goldfinger for use on the IBM 702/705 computers. This assembler was apparently the first to use macros. The Autocoder assemblers were used extensively and were eventually developed into large systems with large macro libraries used by many installations. Another pioneering early assembler was the UNISAP, for the UNIVAC I & II computers, developed in 1958 by M. E. Conway. It was a one-and-a-half pass assembler, and was the first one to use local labels. By the late fifties, IBM had released the 7000 series of computers. These came with a macro assembler, SCAT, that had all the features of modern assemblers. It had many directives, an extensive macro facility, and it generated relocatable object files. The SCAT assembler (Symbolic Coder And Translator) was originally written for the IBM 709 and was modified to work on the IBM 7090. The GAS (Generalized Assembly System) assembler was another powerful 7090 assembler.

1. **ARCHITECTURE AND DESIGN**

**ASSEMBLER**

Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

An assembler is a translator that translates an assembler program into a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created.



**Design of the software:**

There are basically three types of assemblers:-

1. One Pass assembler
2. Two pass assembler
3. Multi pass assembler

•Initially, the design of assembler is developed so that it can perform these basic functions:

– Convert mnemonic operation codes to their machine language equivalents

– Convert symbolic operands to their equivalent machine addresses

– Decide the proper instruction format Convert the data constants to internal machine representations

– Write the object program and the assembly listing.

This Project focuses on design and implementation of a two pass SIC/XE assembler.  A two-pass assembler resolves the forward references and then converts into the object code. Hence the process of the multi-pass assembler can be as follows:

**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 assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
* During Pas s 1, each literal operand is identified and an address is assigned in the literal table.
* Defines the symbols in the symbol table (generate the symbol table). All the labels of the instructions are symbols. The table has entry for symbol name, address value.

**Pass-2**

* Assemble the instructions (translating operation codes and looking up addresses).
* Generate data values defined by BYTE, WORD etc.
* Perform the processing of the assembler directives not done during pass-1.
* During Pass 2, the operand address for use in generating object code is obtained by searching LITTAB and SYMTAB for each literal/symbol operand encountered.
* Write the object program and assembler listing.

The most important things which need to be concentrated is the generation of Symbol table and resolving forward references.

Forward reference:

– Symbols that are defined in the later part of the program are called forward referencing.

– There will not be any address value for such symbols in the symbol table in pass 1.

In addition to the translation to object program, the assembler has to take care of handling assembler directive. These directives do not have object conversion but gives direction to the assembler to perform some function. Examples of directives are the statements like BYTE and WORD, which directs the assembler to reserve memory locations without generating data values.

**SIC/XE Architecture:**

**Memory**

Memory consists of 8-bit bytes; any 3 consecutive bytes form a word (24 bits). The maximum memory available on a SIC/XE system is 1 megabyte.

**Registers**

There are total 9 registers in SIC/XE system. The following table indicates the numbers, mnemonics and uses of these registers. All the registers are 24 bits in length except register F.

|  |  |  |
| --- | --- | --- |
| Mnemonic | Number | Special use |
| A | 0 | Accumulator |
| X | 1 | Index Register |
| L | 2 | Linkage Register |
| B | 3 | Base Register |
| S | 4 | General Purpose Register |
| T | 5 | General Purpose Register |
| F | 6 | Floating-point Accumulator(48 bits) |
| PC | 8 | Program Counter |
| SW | 9 | Status Word |

**Data Formats**

There is a 48-bit floating-point data type.

1       11                                              36

|  |  |  |
| --- | --- | --- |
| s | exponent | fraction |

         The sign of the floating-point number is indicated by the value of s (0       = positive, 1 = negative). The fraction is interpreted as a value between 0 and 1. The exponent is interpreted as an unsigned binary number between 0 and 2047.

**Instruction Formats**

* Includes all SIC instructions.
* New instructions (e.g. LDB, STB).
* Floating point instructions (e.g. ADDF, MULF).
* Register-to-register instructions (e.g. ADDR, MULR)
* Addresses must be 20 bits long.
* Uses both relative addressing and extended addressing.
* Instructions may be 1, 2, 3 or 4 bytes long.

1-Byte format:

|  |
| --- |
| Opcode |

Example: FLOAT

2-Byte format:

|  |  |  |
| --- | --- | --- |
| Opcode | R1 | R2 |

Examples: RMO S, T MULR S

3-Byte format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | n | i | x | b | p | e | Displacement |

x – index bit (as before).

Bits b and p specify relative addressing.

b = 1 and p = 0: Relative to B.

b = 0 and p = 1: Relative to PC.

b = 0 and p = 0: Direct address.

i = 1 and n = 0: Immediate mode.

i = 0 and n = 1: Indirect mode.

i = n: Simple addressing.

Bit e is always 0 for 3-byte instructions.

In B-relative mode, displacement is treated as an unsigned integer. In PC-relative mode, displacement is treated as an integer in 2’s complement form

4-Byte format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | n | i | x | b | p | e | Address(20 bits) |

Bits b and p are always 0. (Thus, 4-byte format does not allow relative addressing.)

Bit e is always 1 (to distinguish between 3-byte and 4-byte instructions).

**Input/Output**

Each device is assigned a unique 8-bit code.There are three I/O instructions, each of which specifies the device code as an operand.  Test device(TD), Read device(RD) and Write device(WD).In addition, there are I/O channels that can be used to perform input and output while the CPU is executing other instructions. The instructions SIO, TIO, and HIO are used to start, test, and halt the operation of I/O channels.

1. **ALGORITHMS AND DATA STRUCTURES**

We have made use of the regular two pass-assembler (SIC/XE), as mentioned in the book **System Software: An Introduction to Systems Programming** by **L. Beck.**

The algorithms for the two passes can be understood as follows:



*Algorithm for Pass 1*



*Algorithm for Pass 2*

Of the data structures that we’ve used in our implementation of the two-pass assembler, the major ones are the Operation Code

Table (OPTAB) and the Symbol Table (SYMTAB).

OPTAB: It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length. In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language.

We search OPTAB in the first pass to find the instruction length for incrementing LOCCTR. In pass 2 we take the information from

OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

We have made use of String data structure implementation for Opcodes, with the use of StringTokenizer class of Java to tokenize the strings and make use of them.

SYMTAB: This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).

During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.

We’ve used the Symbol Table data structure (Java implementation) for storage of symbols (SYMTAB).

Apart from these, we have used:

1. Linked List data structure for Literal Table
2. Array List data structures (Java implementation) for Object Code generation and Line Counter
3. HashMap data structure (Java implementation) for storing Register values of the SIC/XE computer.
4. **DESIGN RESULTS**

**Analysis**

In this section, we will try to analyze the design of the assembler and the algorithm implemented to carry out the design in this project. We exploited ,here, two-pass assembler design ideology to make our own two-pass assembler to solve a major problem i.e the problem of forward referencing in the program and generate the actual opcode.

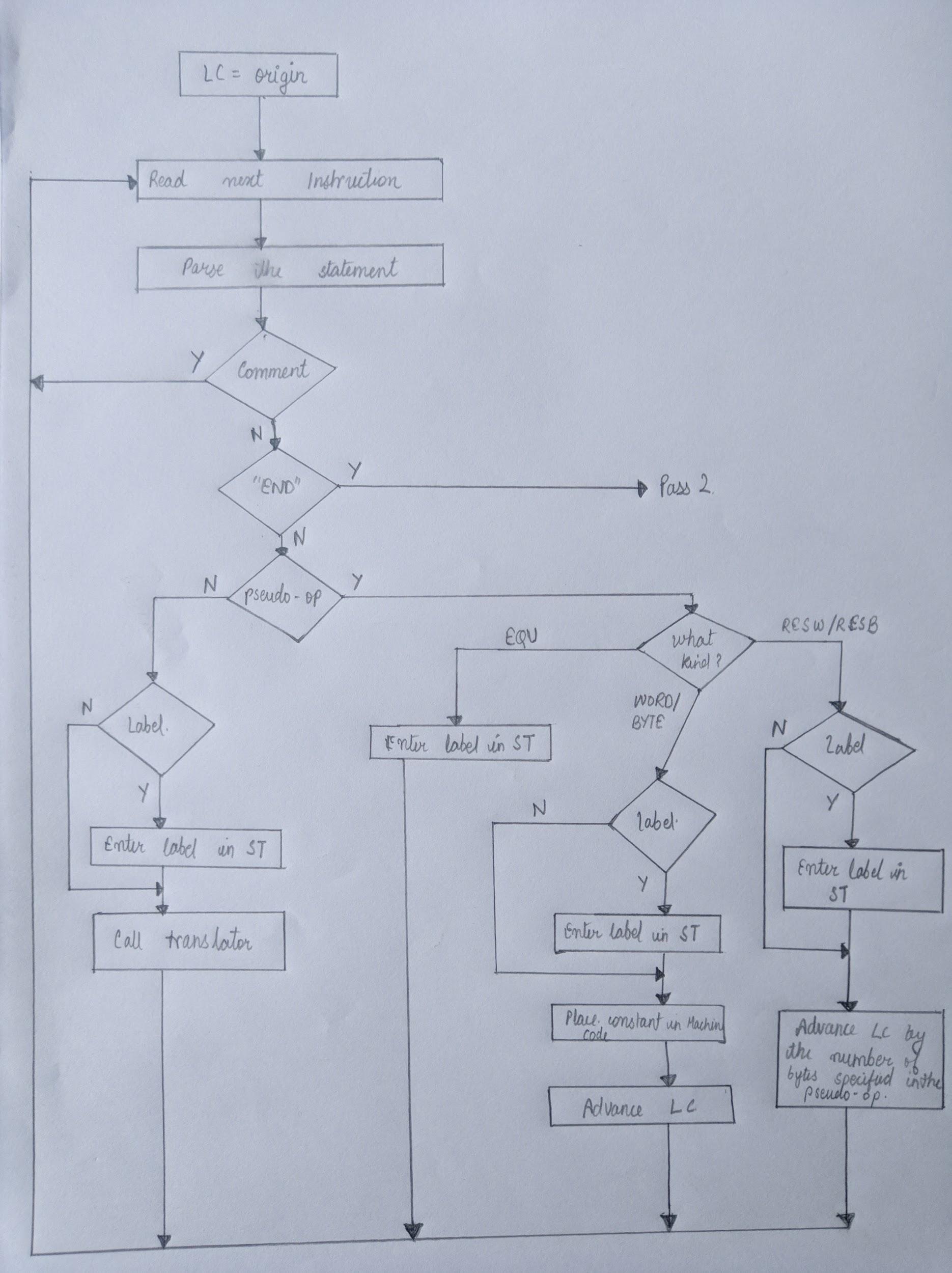
Our Pass 1 of the two-pass assembler while scanning the code assign addresses to all the statement in the program and save the address assigned to all the labels for use in Pass 2 in SYMTAB.The data structure used in algorithm i.e. op-code translation table (OPTAB) is used to lookup and validate operation code in the source program. Pass 1 algorithm also performs some processing of assembler directives. This includes processing that affects address assignment . such as defining the length of data areas defined by RESW, BYTE . For a machine like SIC/XE, where different instruction lengths are used ,it’s important to search the OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

And as its end product Pass 1 generates an intermediate file which is then passed to Pass 2 of our two-pass assembler to generate object code. First, Pass 2, assemble instruction i.e translates operation code and lookup for addresses in the SYMTAB, and generate data value defined by BYTE, WORD, etc and perform processing of assembler directives which were left during Pass 1. And at last , it writes the object code and assembly listing and it generates the relocatable object file. OPTAB helps in pass 2 by telling which instruction format to use in assembling the instruction and any peculiarities of the object code instruction.

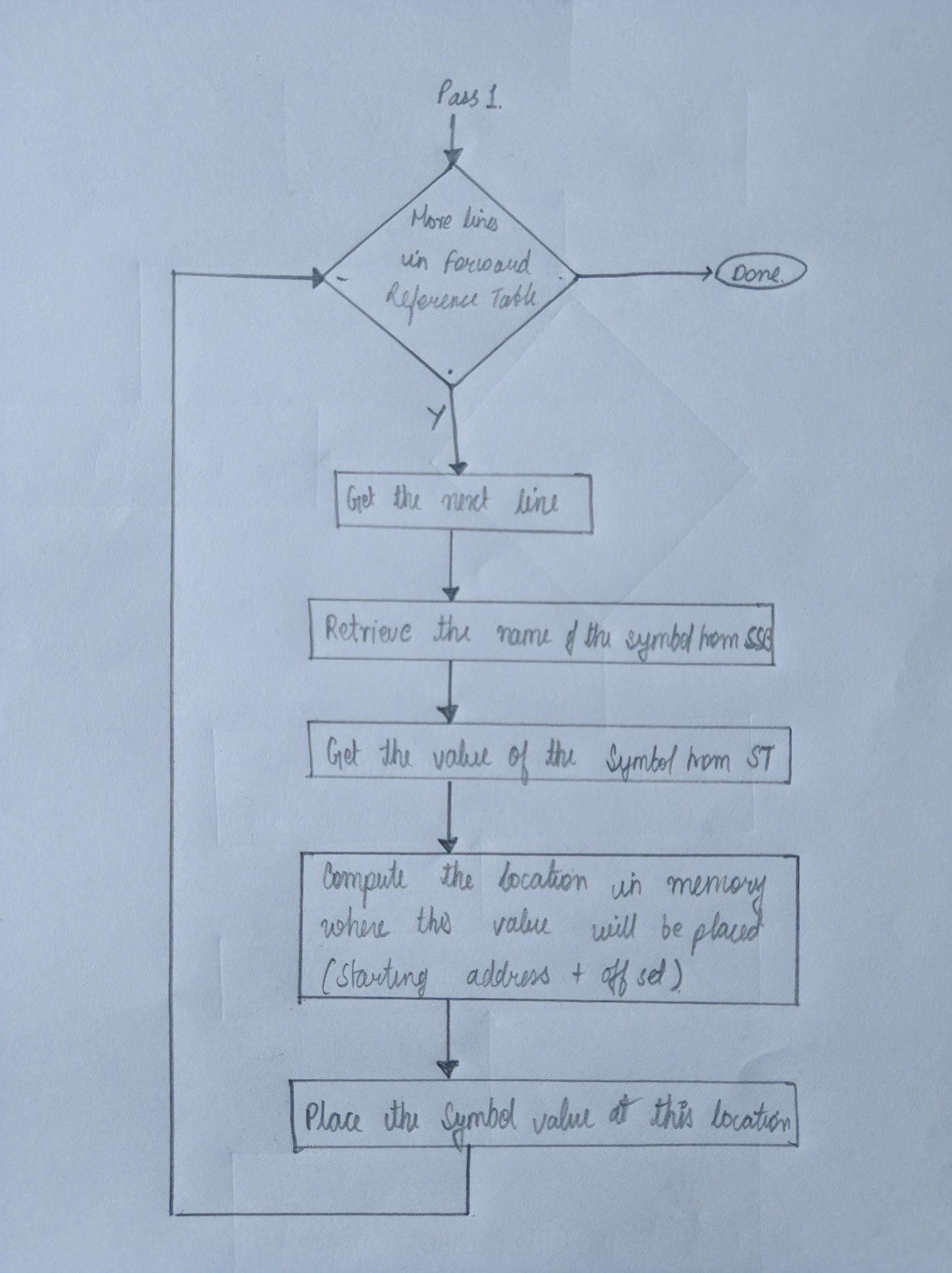
Now , we will analyze the Pass 1 algorithm, it quite simple and straightforward implementation of the design. It scans the first instruction line and see if the opcode is ‘START’ then it assigns the Location counter this address otherwise 0, and then write line to intermediate file and then in every subsequent loop till the ‘END’ is encountered it stores the value of the symbol in the label field with its address to use in Pass 2, otherwise raise error flag on encounter with the duplicate symbol. And if opcode is present, it searches for it in the OPTAB and increment the Location counter otherwise raise error file. The instruction is then written into the intermediate file . At last the length of the program is calculated.

Pass 2 algorithm first read the first line from intermediate file if opcode is ‘START’ then it write the listing line and read next line and write header record to the object program and initialize first text record. Then till the opcode is not equal to ‘END’ , it converts and write each instruction to object file considering all possible error and exception . At last ,it writes last text record and the End record.

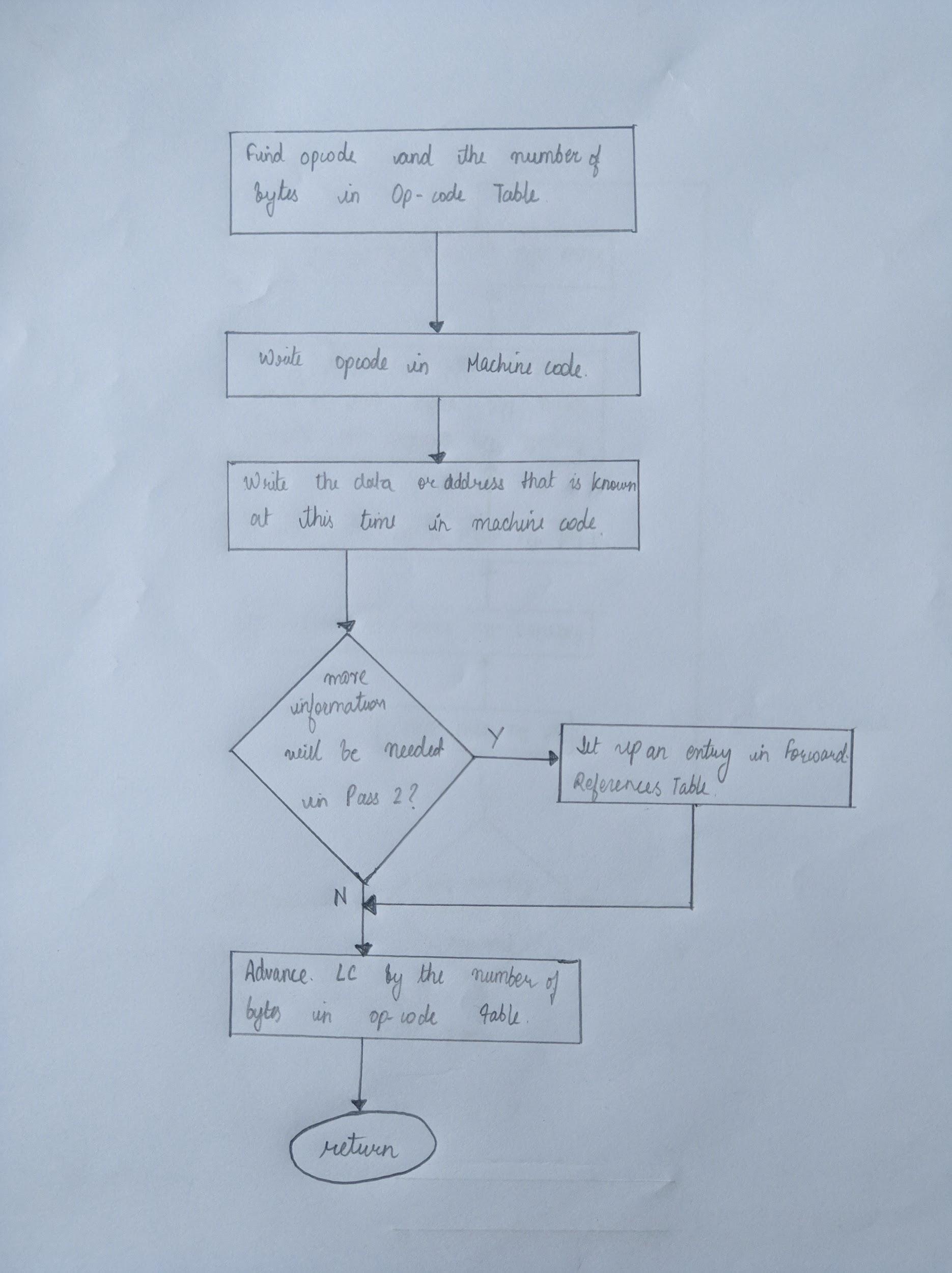
**Below are the flow Charts depicting the design and algorithm we implemented**



1. *Flowchart of the first pass of a two-pass assembler*



*b) Flowchart of the second pass of a two-pass assembler*



*c) Flowchart of a translation routine*

**MACHINE DEPENDENT ASSEMBLER FEATURES:**

* Instruction format and Addressing modes.

SIC/XE

– PC-relative/Base-relative addressing op m

– Indirect addressing op @m

– Immediate addressing op #c

– Extended format +op m

– Index addressing op m, X

– register-to-register instructions COMPR

– larger memory → multi-programming (program

allocation)

* Program Relocation

**Relocation** is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses. Prior to the advent of multiprocess systems, and still in many embedded systems the addresses for objects were absolute starting at a known location, often zero. Since multiprocessing systems dynamically link and switch between programs it became necessary to be able to relocate objects using position independent code. A linker usually performs relocation in conjunction with **symbol resolution**, the process of searching files and libraries to replace symbolic references or names of libraries with actual usable addresses in memory before running a program.

**MACHINE INDEPENDENT ASSEMBLER FUNCTIONS**

* Literals
* Symbol defining statements
* Expressions
* Program Blocks
* Control Sections

1. **SOURCE CODE AND IMPLEMENTATION RESULTS**

Commands to run assembler:

**javac Assembler/Main.java**

**java Assembler.Main InputTests/filename.asm**

Eg. java Assembler.Main InputTests/bsrch.asm

Commands to run SicTools (simulator):

**java -jar out/make/sictools.jar**

After running the assembler in terminal you’ll get the intermediate file, symtab (of pass1 and pass2), littab(of pass1 and pass2) and the object program.

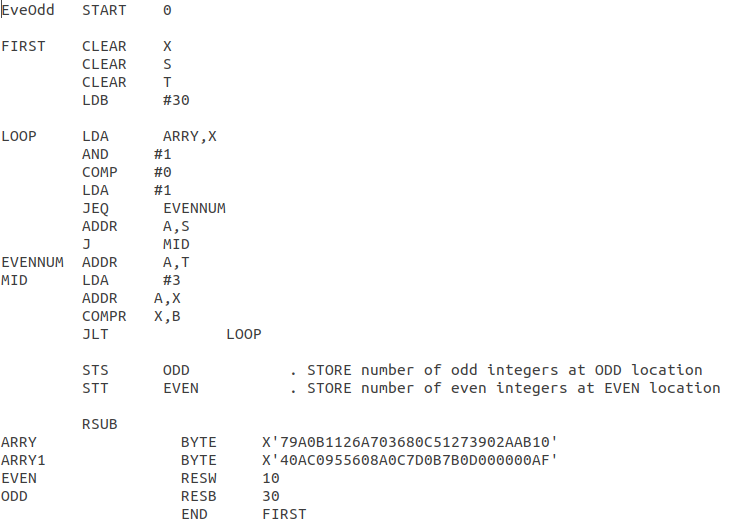
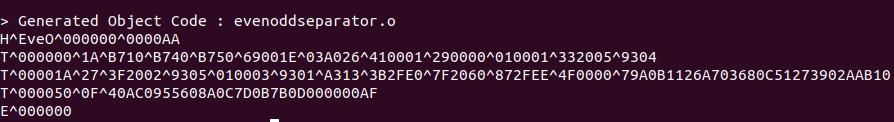
Following files are also created in the src folder along side the .asm file :

* .int file (intermediate file)
* .o file (object program)
* .txt (intermediate along with object code)

SIC/XE programs(.asm files to be found in src folder) :

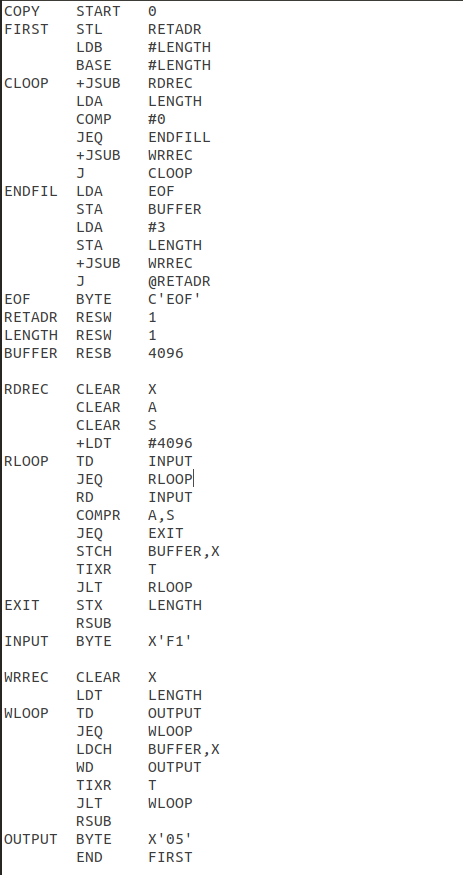
1. evenoddseparator.asm

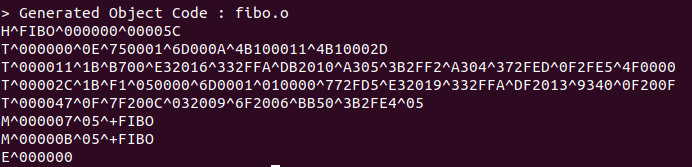
Source code:

Object program:

1. fibo.asm

Source code:

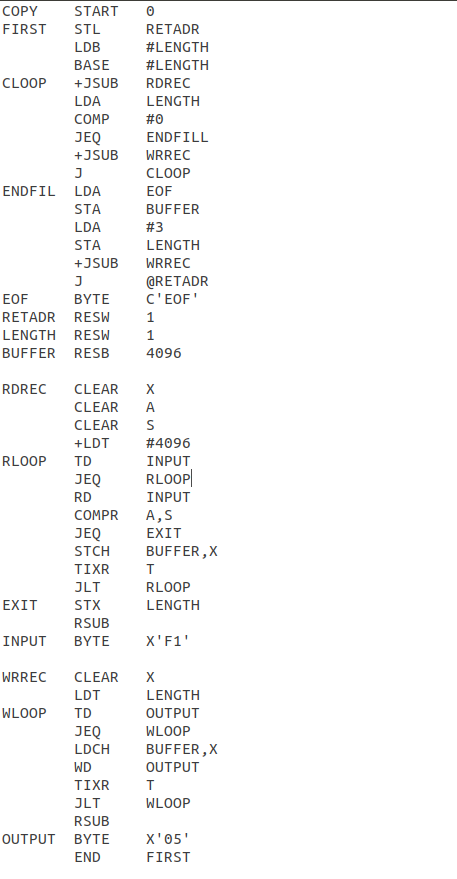




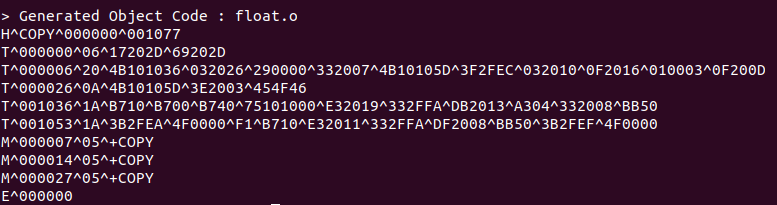
Object program:

3. float.asm

Source code

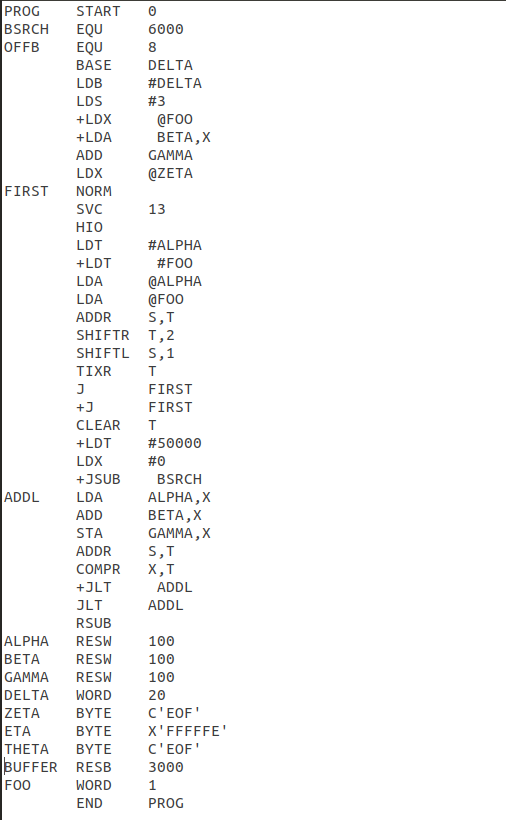


Object program:

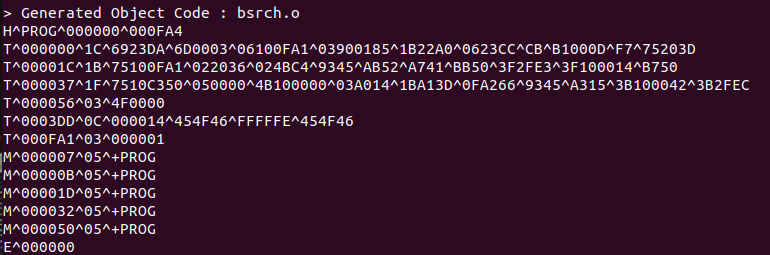


4. bsrch.asm

Source code:

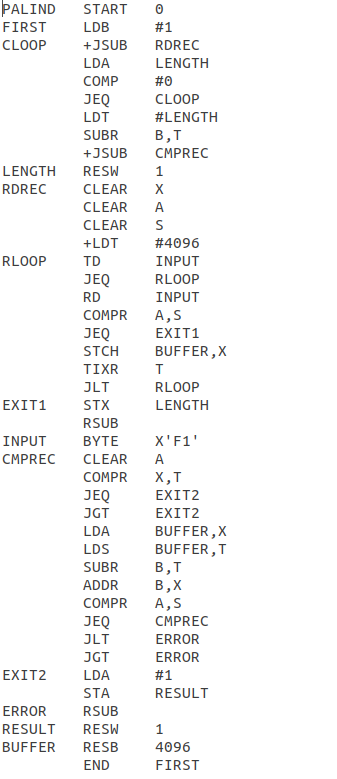


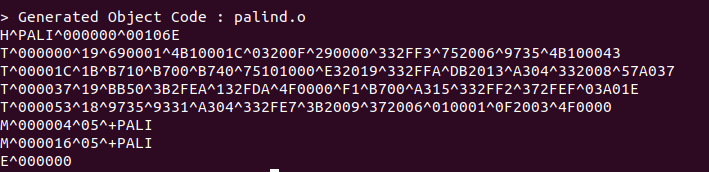
Object program:



5. palind.asm

Source code:



Object program:

1. **CONCLUSION**

From the design and suitable implementation of the two-pass assembler, we gained better insights into the minute details of how the assembler works and makes use of different components and data structures.

Also, we learned about the background of how assemblers underwent enhancements and design changes over the years.

The flow of logic and division of the assembly process into two passes was understood in a better way, along with learning new implementations of the data structures we were already familiar with.

In addition to this, we came up with the assembly codes of the programs needed to test our assembler, which helped us further in

debugging our assembler’s code and ensure proper output of the object code.