



CDSS PROJECT REPORT

Submitted By:

Muskaan Goel[ENG18CS0177]

Muskaan Sinha[ENG18CS0178]

Meghana Shree M[ENG18CS0165]

Lysetti Lakshmi Poojitha[ENG18CS0150]

of

BACHELOR OF TECHNOLOGY

in

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

at

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING, BANGALORE-560068

6TH SEMESTER

(Course Code: 16CS)

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the CDSS Project report entitled “**TWO PASS ASSEMBLER IN C**” being submitted to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 6th semester B.Tech C.S.E of this university during the academic year.

2020-2021.

Date: _____

Signature of the Faculty in Charge

Signature of the Chairman

DECLARATION

We Muskaan Goel[ENG18CS0177],Muskaan Sinha[ENG18CS0178],Meghana Shree M[ENG18CS0165],Lysetti Lakshmi Poojitha[ENG18CS0150]students of 6th semester B.Tech in Computer Science and Engineering, Dayananda Sagar University, Bengaluru, hereby declare that titled

“TWO PASS ASSEMBLER IN C”

submitted to the Dayananda Sagar University during the academic year 20202021, is a record of an original work done by me under the guidance of Dr.Revathi V, Assistant Professor, Department of computer science engineering, Dayananda Sagar University, Bengaluru. This project work is submitted in partial fulfilment for the award of the degree of Bachelor of Technology in Computer Science. The result embodied in this thesis has not been submitted to any other university or institute for the award of any degree.

Team Members,

Muskaan Goel[ENG18CS0177]
Muskaan Sinha[ENG18CS0178]
Meghana Shree M[ENG18CS0165]
Lysetti Lakshmi Poojitha[ENG18CS0150]

ACKNOWLEDGEMENT

We are pleased to acknowledge _____, Department of Computer Science & Engineering for his invaluable guidance, support, motivation and patience during the course of this project work.

We extend our sincere thanks to **Dr. SANJAY CHITTNIS**, **Chairman**, Department of Computer Science & Engineering who continuously helped us throughout the project and without his guidance, this project would have been an uphill task.

We have received a great deal of guidance and co-operation from our family and we wish to thank one and all that have directly or indirectly helped us in the successful completion of this project work.

Muskaan Goel[ENG18CS0177]
Muskaan Sinha[ENG18CS0178]
Meghana Shree M[ENG18CS0165]
Lysetti Lakshmi Poojitha[ENG18CS0150]

ABSTRACT

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. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.

Consider an assembler instruction like the following

```
JMP LATER
```

```
...
```

```
...
```

LATER:

This is known as a forward reference. If the assembler is processing the file one line at a time, then it doesn't know where LATER is when it first encounters the jump instruction. So, it doesn't know if the jump is a short jump, a near jump or a far jump. There is a large difference amongst these instructions. They are 2, 3, and 5 bytes long respectively. The assembler would have to guess how far away the instruction is in order to generate the correct instruction. If the assembler guesses wrong, then the addresses for all other labels later in the program would be wrong, and the code would have to be regenerated. Or, the assembler could always choose the worst case. But this would mean generating inefficiency in the program, since all jumps would be considered far jumps and would be 5 bytes long, where actually most jumps are short jumps, which are only 2 bytes long.

So, what is to be done to allow the assembler to generate the correct instruction? Answer: scan the code twice. The first time, just count how long the machine code instructions will be, just to find out the addresses of all the labels. Also, create a table that has a list of all the addresses and where they will be in the program. This table is known as the symbol table. On the second scan, generate the machine code, and use the symbol table to determine how far away jump labels are, and to generate the most efficient instruction.

This is known as a two-pass assembler. Each pass scans the program, the first pass generates the symbol table and the second pass generates the machine code.

TABLE OF CONTENTS

Chapter No	Title	Page No
1	Introduction	
2	Problem Statement	
3	Objectives Of Project	
4	Methodology	
5	Software and hardware Requirement	
6	Results	
7	Conclusion	
8	Implementations	
9	References	

1.INTRODUCTION

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 (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divide these tasks in two passes:

- **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

2. PROBLEM STATEMENT

One-pass assembler cannot resolve issues of forward references of data symbols. It requires all data symbols to be defined before they are being used. A Two-pass assembler solves this confusion by devoting one pass exclusively to resolve all issues of forward referencing and then generates object code with no chaos in the next pass.

PASS 1

- Assign addresses to all statements in the program.
- Addresses of symbolic labels are stored.
- Some assembly directives will be processed.

PASS 2

- Translate opcode and symbolic operands.
- Generate data values defined by BYTE,WORD etc.
- Assemble directives will be processed.
- Write the object program and assembly listing.

3. OBJECTIVES OF THE PROJECT

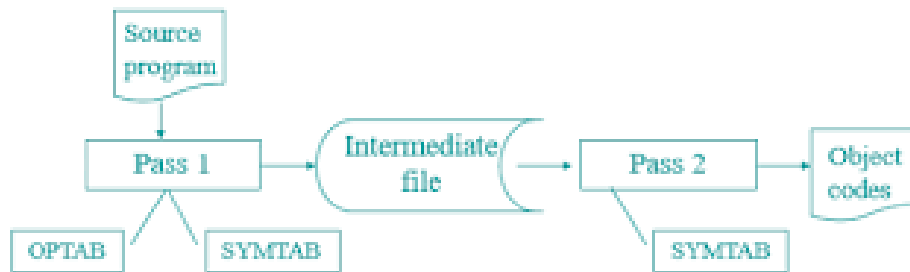
The one-pass assembler cannot resolve forward references of data symbols. It requires all data symbols to be defined prior to being used. A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass. If a data symbol depends on another and this another depends on yet another, the assembler resolved this recursively.

Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table).

Various Data bases required by pass-2:

1. MOT table(machine opcode table)
2. POT table(pseudo opcode table)
3. Base table(storing value of base register)
4. LC (location counter)

4.DESIGN METHODOLOGY



A

One pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references (the problem of forward referencing) and assemble code in one pass. The one pass assembler prepares an intermediate file, which is used as input by the two pass assembler.

A two pass assembler does two passes over the source file (the second pass can be over an intermediate file generated in the first pass of the assembler). In the first pass all it does is looks for label definitions and introduces them in the symbol table (a dynamic table which includes the label name and address for each label in the source program). In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations into machine codes and so on.

5. REQUIREMENT ANALYSIS

Hardware Requirements: Our computer architecture consists from CPU (Central Processing Unit), registers and Random Access Memory RAM, where part of the memory is being used as a stack. The size of each word in memory is 16 bits. Arithmetics is to be carried by the '2 complement' method. Our computer machine can only handle integers (Positives or negatives), it doesn't handle real numbers.

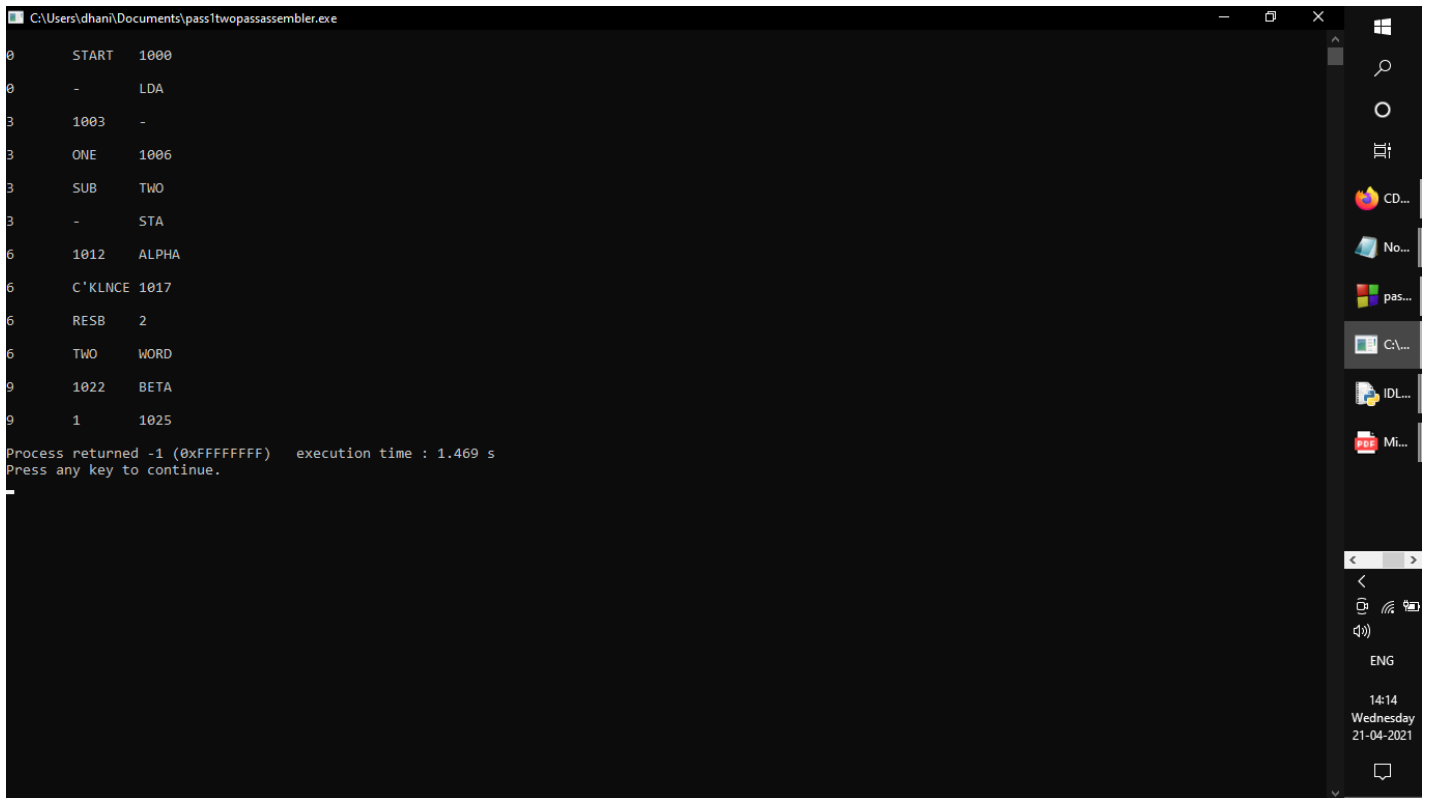
Software Requirements: Code blocks

6. RESULTS

```
C:\Users\dhani\Documents\passtwowopassassembler.exe

0      START  1000
0      -      LDA
3      1003  -
3      ONE   1006
3      SUB   TWO
3      -      STA
6      1012  ALPHA
6      C'KLNCE 1017
6      RESB  2
6      TWO   WORD
9      1022  BETA
9      1      1025

Process returned -1 (0xFFFFFFFF)   execution time : 1.469 s
Press any key to continue.
```



7. CONCLUSION

The two pass assembler performs two passes over the source program. In the first pass, it reads the entire source program, looking only for label definitions. All the labels are collected, assigned address, and placed in the symbol table in this pass, no instructions are assembled and at the end the symbol table should contain all the labels defined in the program. To assign address to labels, the assembler maintains a Location Counter (LC).

In the second pass the instructions are again read and are assembled using the symbol table. 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. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.

8. IMPLEMENTATIONS

- Convert mnemonic opcodes to their machine language equivalent.(e.g, LDA to 00)
- Convert symbolic operands to their equivalent machine address (eg, LOOP to 2045)
- Allocate necessary memory.
- Convert data constants to internal machine equivalents.
- Write the object program and assembly listing.

9.REFERENCES

<http://users.cis.fiu.edu/~downeyt/cop3402/two-pass.htm>

<https://github.com/ayushoriginal/Two-pass-Assembler/blob/master/README.md>

<https://stackoverflow.com/questions/10244422/how-is-a-2-pass-assembler-different-from-a-one-pass-assembler-in-resolving-the-f>

<https://www.slideshare.net/ShubhamShah001/two-pass-assembler>