CHAPTER 1
INTRODUCTION

1.1 Problem Definition

The problem that was before us was to develop an assembler for, which will process assembly code for 8086 and output object code.

This software will be used by system engineers for assembling their assembly source program. Input to the software will be the source file containing assembly program, output would be in two forms: The first output is printed on the screen containing the symbol table entry .Symbol table consists of a list of labels used in the program along their name, address, size and value.

The second output has 2 files: one .txt file that contains the symbol table and a .obj file that contains all the object codes generated for the valid instructions in the program given as test input to the assembler.

1.2 Purpose

The main purpose of this project is to convert an assembly language program in 8086 to its equivalent machine code, only if the instructions used in the program use only the instructions, which are designed in the project.

Here analysis of the logic present in the program is not done but rather checks for the instruction if present in the project designed then its equivalent machine code will be the output, otherwise there will be an error message displayed. There are many instructions present in the working model of 8086. But, this project code is designed in such a way that it recognizes only a few instructions of 8086 model.

1.3 Scope

An assembly language is a machine dependent, low level programming language which is specific to a certain computer systems compared to the machine language of a computer system; it provides

3 basic features which simplify programming:

- Mnemonic opcodes: use of mnemonic opcodes for machine instructions eliminates the need to memorize numeric opcodes.
- Symbolic operands: symbolic names can be associated with data or instructions.
- Data declarations: data can be declared in a variety of notations, including the decimal notation. An assembler is a system program that translates the assembly program into equivalent object code.

1.4 Motivation

This 8086 assembler is chosen as the project since completion of the project would improve the system programming capabilities. Assembler is one of the basic system software which is widely used because the object code produced by this has lesser space requirements as compared to compiled code.

The assembler was coded in Lex and Yacc and hence enhanced our Lex and Yacc knowledge and programming skills. We chose to design the assembler for 8086, microprocessor because we had ample study material to study about the architecture, addressing modes and instruction formats of this processor. Moreover, such projects widen the approach on how to code by strictly confirming to a specification and how to integrate software components written by various software developers, each having their own different styles, into a single cohesive unit.

1.5 Literature Review

1.5.1 Existing System

The features and designs of an assembler depends upon the source language it translates and the machine language it produces. As far as existing system is concerned, we will consider the assembly language programming using MASM. MASM reads the source program as its input and produces an object program in 2 passes. A 2 pass assembler is one that goes through the source file two times. It does so that the addresses of any forward declaration which it encounters during the first pass are resolved during the second pass. If the declaration is not resolved an error is flagged.

MASM accepts the file name only with extension.asm. MASM contains directives and operations of 8086 microprocessors.

1.5.2 Proposed System

The Proposed system is designed to consider most of the assembler features in a limited fashion. The features implemented in this design are 1 byte, 2 byte, 3 byte instructions, register-register and register to memory instruction formats. It also performs specific task for the assembler directives as indicated in the module. Parsing is done for each instruction to determine the type and addressing modes of the operands. 'Jmp' instruction is also included which supports only near jump.

Only the following memory modes **are proposed to be implemented**:
[BX][SI], [BX][DI], [BP][SI], [BP][DI]

[SP], [BP], [SI] and [DI] with appropriate displacements if provided. For eg:
[BX][SI]23h refers to Target address = BX + SI + 0023h

Also all instructions are classified into :

(1).Arithmetic	eg: ADD,DAA,SUBB
(2). Logical	eg:AND,OR,XOR
(3).Data Transfer	eg:MOV,PUSH
(4). Control Transfer	eg: CALL,JMP,JNZ

CHAPTER 2
SOFTWARE REQUIREMENTS AND SPECIFICATIONS

2.1 Overall Description

A microprocessor works upon the hardware embedded in it. It accepts instructions to generate suitable output desired by programmer. These instructions exist natively in binary form-i.e., in a pattern of 0’s and 1’s.Every type of instruction that can be handled by microprocessor has a unique code- a sequence of binary numbers.

Thus for every operation serviceable by the microprocessor, a suitable binary stream has to be supplied. This is the fastest way to perform an operation on a microprocessor, because the microprocessor understands the only language of binary digits. Hence there is no translation required of the input.

Speed may be achievable in this procedure, but it is very inefficient form of a programmer’s point of view as highlighted by the following arguments.

- Very tedious procedure: Remembering the instruction format and encoding large volumes of binary information would take a lot of time.

- Error-prone: Due to the similarity in the machine codes of the numerous instructions, the procedure is extremely error prone.

- Unrecognizable language: As the entire program is hand -coded, the human interpretation of the program’s execution becomes virtually impossible. This leads to a lot of redundant documentation processes that only take more and memory.

Due to these inconveniences; which are very serious; the microprocessor manufacturers supply an instruction set along with the chip. The instruction set consists of English language equivalents of the instruction. But these opcodes and operands are not interpreted by microprocessor. It needs that the instructions are in the binary form. An assembler does the job of converting these mnemonics to a machine understandable format.

Assembler is a system software that converts mnemonic code to object code. Here the mnemonic code is also known as source program. The fundamental functions of assembler are:

- Translating mnemonic operation codes into their machine equivalents.
- Assigning machine addresses to symbolic labels used by the programmer, using a location counter.
- Using the above values, object code is written to a destination file.
- Builds each statement in proper format.
- Writes object program and assembly listing.

Often a symbol is encountered, which has not been defined. This is called forward reference problem. One way to solve this is to use a two pass assembler. Here the assembling operation is divided into iterations. The two passes are known as pass1 and pass2.

2.2 Specific Requirements

2.2.1 Functionality

When coded in assembly language, the program has to go through three phases of modification before finally appearing as a machine code.

The three phases are as shown in the Fig 2.2

The first phase is to type user program and save it in a suitable format.

Next step is to assemble the mnemonics and to convert them to machine code. The step is take care by the assembler. The assembler converts all the information in the text file to its machine format. This form of file, which is directly not executable, is called the object file containing the object code.

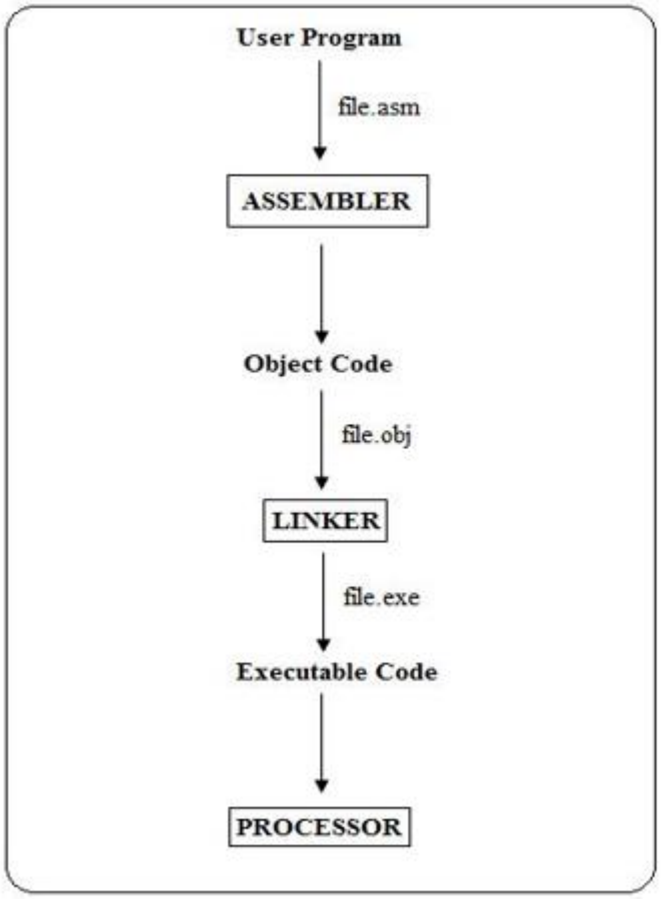


Fig.2.2: Three phases of assembly language program execution.

After the creation of the object code, the loader or linker is involved with the object as the parameter. These programs do the following: Loader - loads the program on to the memory; Linker- links the various files or related files that are a requisite to run the program onto the object code. Now these programs generate the executable code and load into memory. This code is now available for execution.

2.2.1.1 Operations performed in pass1

- Checks to see if the instructions are legal in the current assembly mode.
- Allocates space for instructions and storage areas that are requested.

- Assigns addresses to all statements the program, using a location counter.
- Builds a symbol table, also called cross reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement.

Save the values (addresses) assigned to all labels in the symbol table, for use in pass2.

- It also processes the assembler directives.

At the end of the first pass, all the necessary space has been allocated and each symbol defined in the program has been associated with a location counter in the symbol table. When there are no more statements to be read, the second pass starts at the beginning of the program.

2.2.1.2 Operations performed in pass2

- Examines the operands for symbolic references to storage locations and resolves these symbolic references using information in the symbol table.
- Ensures that no instructions contain an invalid instruction form.
- Translates source statements into machine code and constants, thus filling the allocated space with object code.
- Performs processing of assembler directives not done during pass 1.
- Write the object program to destination .obj file.
- It keeps track of errors and displays appropriate messages if the same occurs.

If an error is found in the first pass, the assembly process terminates and does not continue to the second pass. If this occurs the assembly listing only contains errors and warnings generated during the first pass of the assembler.

If only warnings are generated in the first pass, the assembly process continues to the second pass. The assembler listing contains errors and warnings generated during the second pass of the assembler. Any warnings generated during the first pass do not appear in the assembler listing.

2.2.2 Performance

The main reason why most assemblers use a 2-pass system is to address the problem of forward references - references to variables or subroutines that have not yet been encountered when parsing the source code. A strict 1 -pass scanner cannot assemble source code which contains forward references.

Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code. Some assemblers have been written to use a 1.5 (one and a half) pass scheme, whereby the source is only scanned once, but any forward references are simply assumed to be of the largest size necessary to hold any native machine data type (for instance, assuming a reference to the address of an as-yet un-encountered subroutine could be very far away, necessitating the use of a far branch). The unknown quantity is temporarily filled in as zero during pass 1 of the assembler, and the forward reference is added to a 'fix -up list'. After pass 1, the '.5' pass goes through the fix-up list and patches the output machine code with the values of all resolved forward references. This can result in sub -optimal opcode construction, but allows for a very fast assembly phase.

2.2.3 Supportability

2.2.3.1 8086 Microprocessor

8086Microprocessor was developed by Intel Corporation in 1978. It is a 16 bit microprocessor, implemented in N – channel, depletion load, silicon gate technology (HMOS), and packaged in a 40 pin dual inline package.

8086 can be divided into 2 subunits:

Bus Interface Unit (BIU) and Execution Unit (EU)

BIU is concerned interfacing of 8086 with memory. I t has an instruction queue which can store up to 6 instruction

EU performs the execution of the instructions. It has a set of registers to store data values and a flag register to store status of various flags. ALU performs arithmetic and logic functions.

Registers In 8086

8086 supports both 16-bit and 8- bit registers. The various 16-bit registers supported are: Segment registers - ES, CS, SS, DS.

General purpose registers - AX, BX, CX, and DX.

Index registers - SI, DI.

Pointer register - SP, BP.

Various 8- bit registers supported are:

AL, AH, BL, BH, CL, CH, DL, DH.

Addressing Modes In 8086

The different ways that a processor can access data are referred to as addressing modes. There are 12 basic addressing modes in 8086 .This can be classified into 5 categories:

- Addressing modes for accessing immediate and register data (register and immediate modes).
- Modes for accessing data in memory (9 memory modes).
- Addressing modes for accessing I/O ports (I/O modes).
- Relative addressing mode.
- Implied addressing mode.

Memory modes can further be classified into the following types:

- Direct addressing mode.
- Register addressing mode.
- Based addressing mode.
- Indexed addressing mode.

Based indexed addressing mode.
- String addressing mode.

Instruction format:

Label Field	Operand field	Operand 1	Delimiter	Operand 2

Label:	**MOV**	**AX**	**,**	**BX**

Instruction Formats

Instructions in 8086 vary from 1 to 6 bytes in length. Operands may be either 8 - bit or 16 –bit long depending on the instructions. The opcode byte is followed by:

- No additional byte.
- Two byte EA (for direct addressing only).
- One or two byte displacement.
- One or two immediate operand.
- One or two displacements followed by a one or two byte immediate operand.
- Two – byte displacement and a two-byte segment address.

2.3 Interfaces

2.3.1 Hardware Interfaces:

- INTEL 80X86 or above Microprocessors

- 2 MB of memory.

2.3.2 Software Interfaces:

Operating System:

- Linux
- Unix

Language used:

- C

Compiler:

- Linux or Unix based Compilers(cc,gcc)

Editor:

- vi Editor

CHAPTTER 3
SYSTEM DESIGN

3.1 Data Flow diagrams

A data flow diagram is a representation of the movement of information between storage and processing steps with in a software system .The various levels of data flow diagrams are shown below (Fig. 3.1-3.3)

3.1.1 Level 0

The level 0 is the initial level data flow diagram and it’s generally called as the context level diagram. It is common practice for a designer to draw a context-level DFD first which shows the interaction between the system and outside entities. This context-level DFD is then exploded to show more detail of the system being modeled.

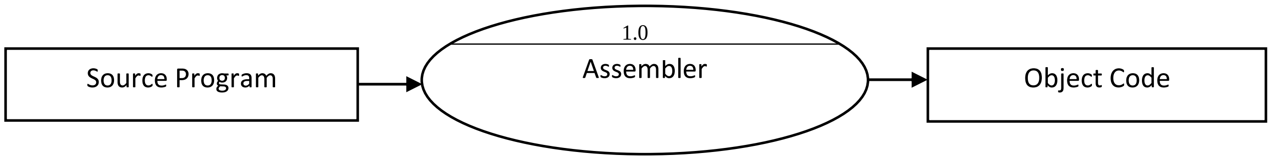


Fig 3.1 Level-0 DFD

3.1.2 Level 1

The level 1 data flow diagram gives more information than the level 0 Data Flow Diagram. The figure 4.3 shows the level 1 Data Flow Diagram. In this level 1 DFD, the process 2 shown in the level 0 DFD is broken down into detailed processes.

The Level 2 is divided into tokenization which helps in building symbol table and to assemble the instructions. Inorder to assemble the instructions aid is also taken from the Symbol Table and Optable. And if any errors are given, they are printed.

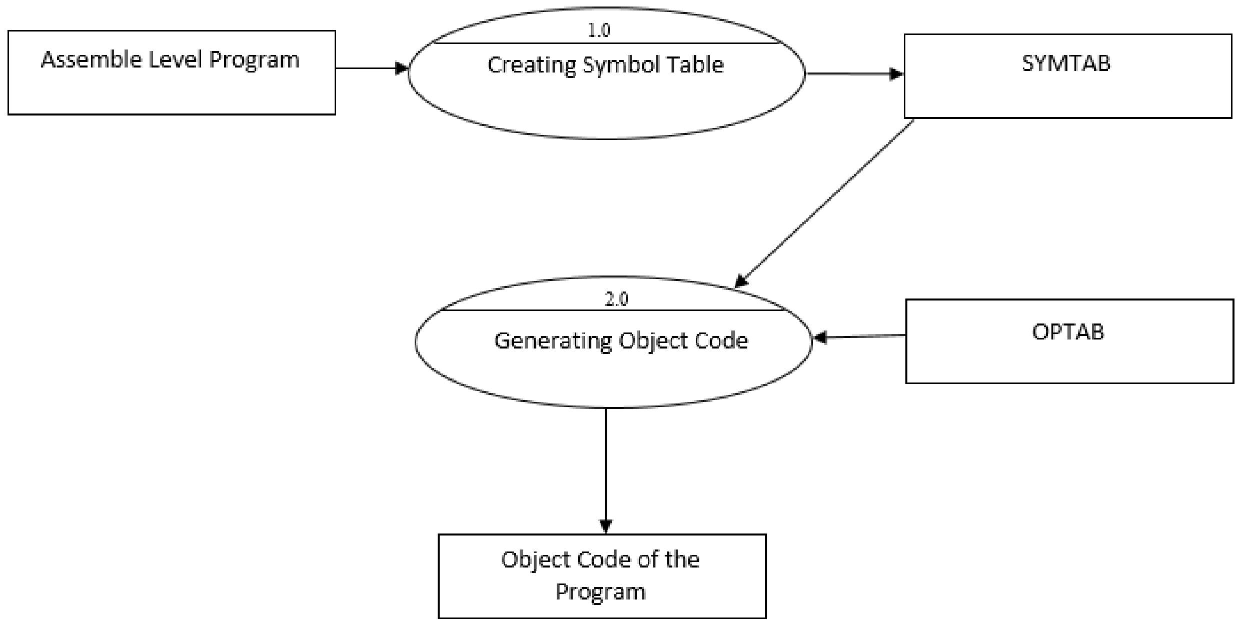


Fig 3.2 Level-1 DFD

3.1.3 Level 2

In level 2 more insight is given to assemble instructions (2.4) which first assembles a command by taking data from the optable and then the labels are assembles (if any), by using the Symbol Table or building the symbol table. If any references are found, they are resolved and the object code is thus then put into the object code file.

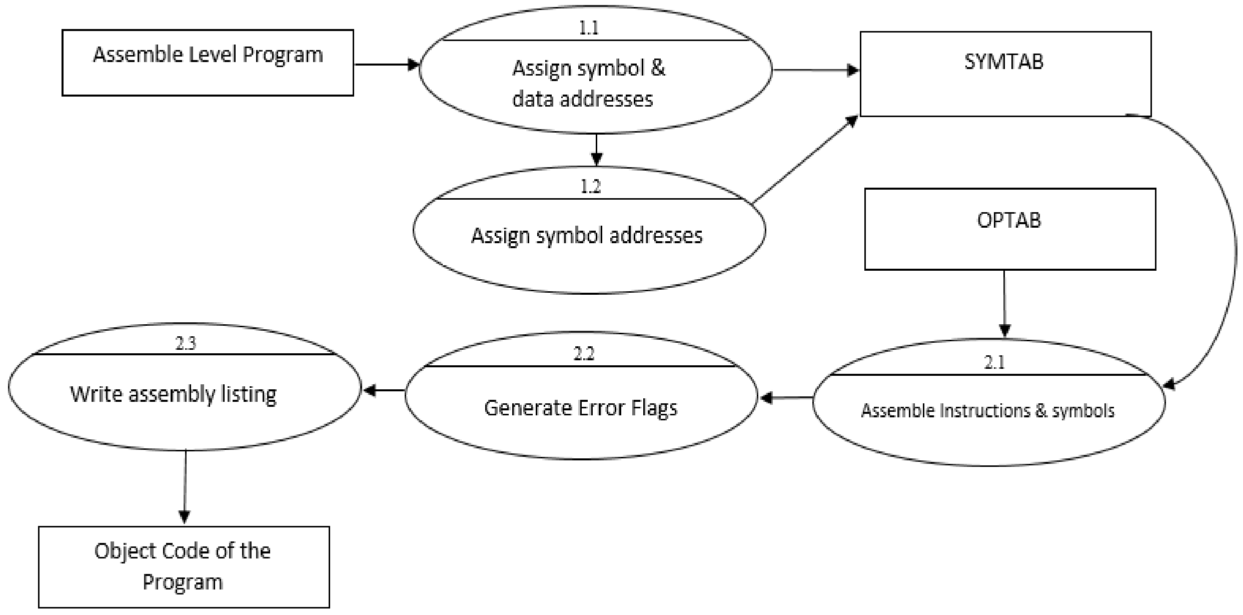


Fig 3.3. Level-2 DFD

3.2 Data Structures Used

Two major internal data structures are used in the assembling process:

- **Operation code table (OPTAB):** It is used to store the mnemonic operation code and its machine language equivalent. The information stored in this table is predefined when the assembler itself is written, rather than being loaded into the table at execution time. This is a static table in the sense that the values in this table are not modified during assembling process. During Pass 1, this table 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.

E.g. Structure used for optab:

```
struct instrn
 {
 char opname[10],code[10];
 struct instrn *link;
 }optab[150];
```

- **Symbol table (SYMTAB):** It is used to store name, address, and other attributes associated with different labels used in the source program. This is a dynamic table, that is, values are entered into the table during assembling process, and its length cannot be predicted. During pass 1 of the assembler, labels are entered into this table as they are encountered in the source program, along with their assigned addresses using the value of the location counter at the instance it is encountered.

During pass 2, symbols used as operands are looked up in this table to obtain the addresses to be inserted in the assembled instructions. This table acts as an input to the second pass

Structure used for symbol table:

```
struct symbol
{
 char label[30];
 int addrs;
 int flag;
 SYMBOL *next;
}symtab[26];
```

3.3 Design Specifications

3.3.1 Input specifications

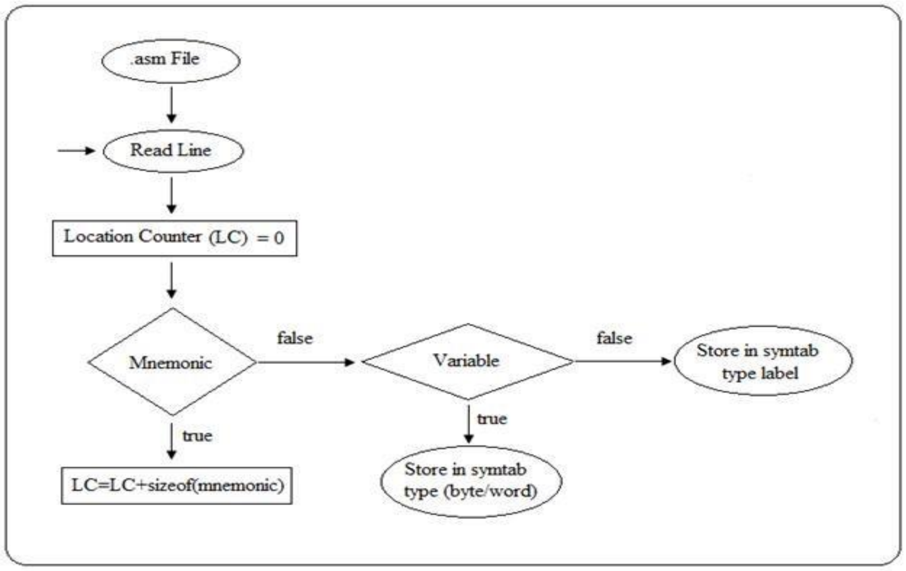
- The labels in the source program, if present must begin on a new line. The opcode field should be separated from the label field by a colon followed by \n. If no label is present, the opcode may begin on a new line.
- Labels should satisfy the following requirements. The first character must be an alphabet or numbers (_a-zA-Z0-9) and each of the remaining characters can be alphabets or Numbers. Label names should not match with keywords or reserved words (opcode and assembler directives).
- The opcode field must be from the instruction set supported or from one of the assembler directives (Pseudo instructions).
- An instruction operand may be either a symbol, which appears as a label in the program or a number (only HEXADECIMAL) that represents an immediate operand or an INTEL 8086 register.
- The source program should contain distinct labels.

3.3.2 Output Specifications

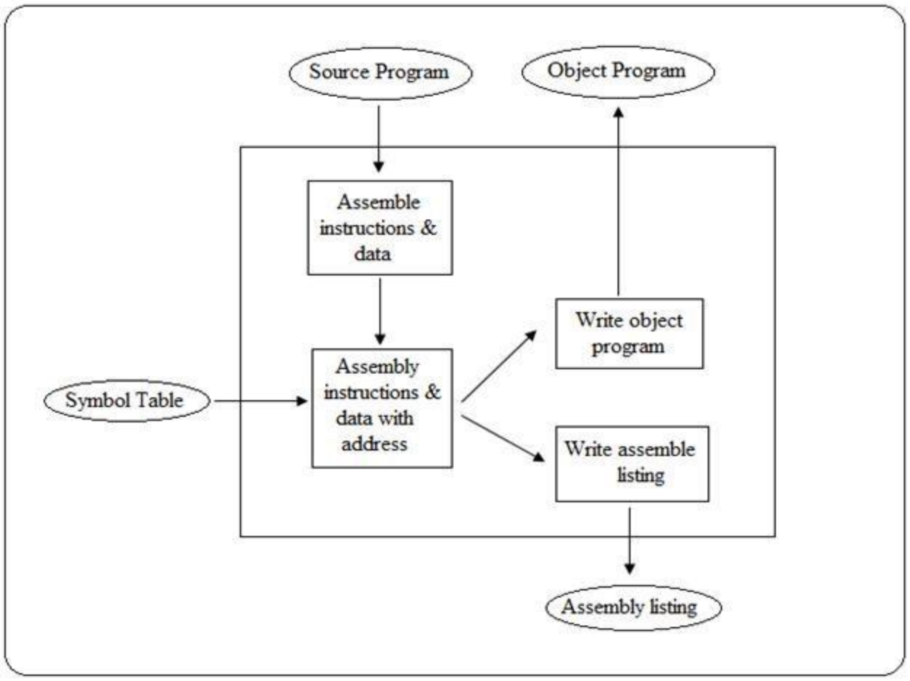
- The assembly listing should show each source program statement, together object code generated. This describes the form and contents of the desired output. The object program should not be generated if any assembly errors are reported. This specifies the conditions under which the output is generated.

3.4 Flowcharts

3.4.1 Pass 1:


.asm File
Read Line
Location Counter (LC) = 0
Mnemonic
false
Variable
false
Store in symtab type label
true
true
LC=LC+sizeof(mnemonic)
Store in symtab type (byte/word)

3.4.2 Pass 2:



CHAPTER 4
IMPLEMENTATION

4.1 Programming language selection

The programming language we selected is C as is very flexible in its use as well as lex and yacc natively support it. Lex and Yacc and useful for the verification of the syntax and checking the program and thus a programing language supported by them was of utmost importance. Also the library support available for C and the user community of C is very vast due to which if we stumbled onto any problem we could easily find the solutions.
Also we used shell scripting for the automatic compilation and running of the lex and yacc files that have been created.

4.2 Platform Selection

Linux Operating System is used for the development for this project since it is an open source and the sofwares lex, yacc and gcc compiler, needed for the production of this software are easily available and compatible with the Linux.

4.3 Code Conventions

Python code conventions explained in this section. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

4.3.1 Naming Conventions

Naming convention make programs more understandable by making them easier to read. They can also give information about the function of the identifier for example, whether it's a constant, method or package –which can be helpful in understanding the code. The conventions given in this selection are high level. The naming rules for the identifier are explained below.

- **Variable Names:** The variables are named as per their purpose. E.g. mode signifies mode of the instructions.
- **Method Names:** Use only lower case letter and connects multiple words with underscore. Example: binary_to_hex()

- **Indentation:** Indentation should be done using 1 tab per indentation level
- **White space:** White space should be used to separate binary operators and assignment statement from other elements. No white space between a function call and its argument list's first parenthesis

4.3.2 File Organization

Lex, Yacc and C files are with extension .l, .y and .c. File should be 31 character or less. The files are stored in the central repository so that the files can be accessed by anyone to run test for validating their requirements. A script file is also made that eases the execution of multiple commands by a sinfle command.

4.3.3 Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. Comments should be in complete sentences. If a comment is Phrase or sentence, its first word should be capitalized, unless it is an identifier begin with a lower case letter. C and Yacc uses // and /* … */ for single and multi-line comments.

4.4 Implementation of 2 Pass Assembler

4.4.1 Lex

Lex is used to tokenize the source program and pass only the valid tokens to yacc file.

4.4.2 Yacc

Yacc verified the syntax of the instruction and allowed to call syntax directed translations where necessary coding can be done to drive the program. In the program the commands are divided into the ones having zero, one or two arguments. The proper grammar has been written which when succeeds marks the syntax validity of the program.

4.4.3 C

C helped in doing all the backend development of the instructions. It provides with the necessary methods to create the data structures and execution logic to develop the logic of the program. All the required data goes into the C file from Yacc and the instruction is made and stored there.

4.5 Difficulties faced

We faced many difficulties during the generation of tokens in lex and the formulation of particular grammar to verify the grammar. The major problem that we faced was to calculate both the location address and the develop the object code of an instruction within one pass. The solution that we came up with was to calculate that location address using stack while going up the grammar and passing the all the relevant value to the backend necessary to generate the object code of the instruction.

The retrieval for proper opcodes for the commands was a big problem so we decided to give an incremental opcode which can be replaced by true opcodes whenever user wants in the opcode file.

CHAPTER 5
TESTING

The defects and errors of the program are identified in this phase of the project. Testing was focused on establishing functional requirements and on system behavior in given scenario. The test cases are selected to ensure that the system behavior can be for all possible scenarios that can occur during the phase of execution.

Accordingly, expected behavior of the system under different conditions is given. Therefore test cases are selected which have inputs and the outputs are as expected

5.1 Testing Process

Testing is one of the core parts of software development. Testing process, in a way certifies, whether the product, that is developed, complies with the standards, that it was designed to. Testing process involves building of test cases, against which the product has to be tested. In some cases, one drives the test cases from the requirements of the product/software, which is to be developed.

5.2 Levels of Testing

Different levels of testing are used in the testing process, each level of testing aims to test different aspects of the system. The basic levels are unit testing, integration testing, system testing and acceptance testing.

5.2.1 Unit Testing

Unit testing certifies the working of the small units of the program. In this different modules are tested when they are created to certify the validity of the input that it takes and the output that it gives. It is typically done by the parameter of the module. Due to its close association with coding, the coding phase is frequently called ‘coding and unit testing ‘.The unit test can be conducted in parallel for multiple modules.

5.2.2 Integration Testing

The second level of testing is called integration testing. Integration testing deals finding the errors when a few modules are combined and tested together to check the integration of the modules. The output of one module is input of another module. The acceptance of the data that is output by another module is tested in this scenario and combines effect is checked.

5.2.3 System Testing and Acceptance Testing

Here the entire software is tested in the given condition. The input and output to the software is given by and to the user respectively. Thus, this unit should give the correct output that would be shown to the user and is valid.

5.3 Testing Environment

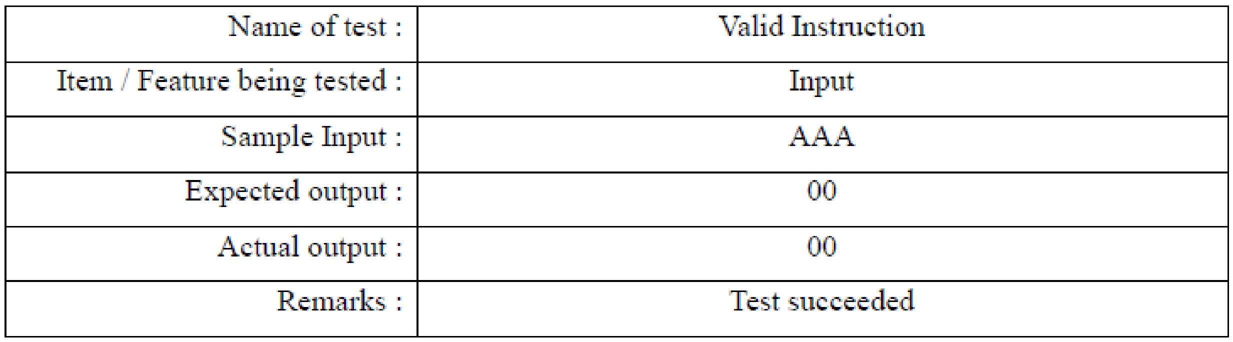
The software is tested on the following basis:

- Lexical acceptance of the assembly code
- The syntax verification of the assemble code
- The output of the object code

5.4 Test Cases

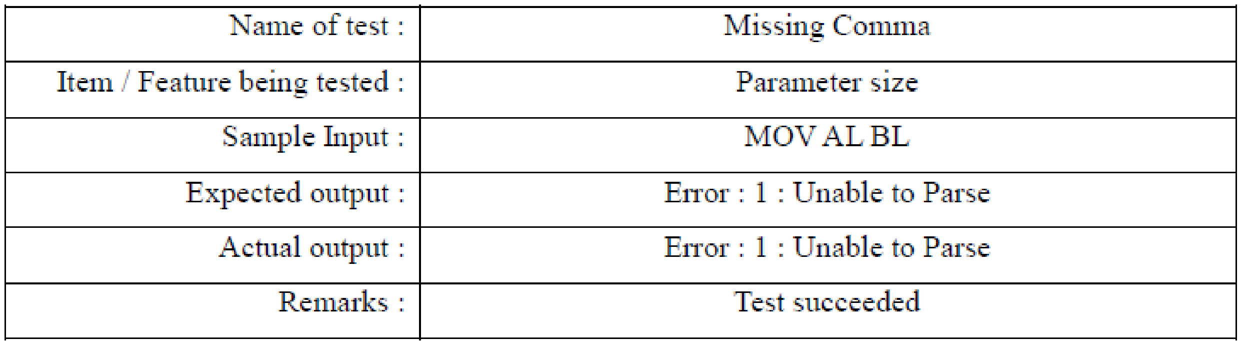
The following are the test cases that were applied to the program and the corresponding output were noted. The success and failure of the test cases are also mentioned.

Test Case 1: The program was input with a valid instruction and the output expected was the object code of the instruction. The input instruction was a no operand operation code AAA. The output was the object code of the instruction 00.



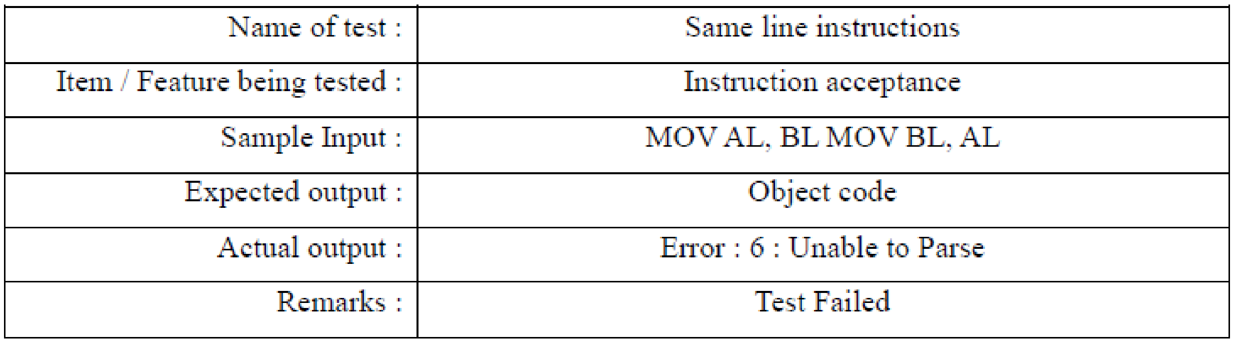
Name of test :	Valid Instruction
Item / Feature being tested :	Input
Sample Input :	AAA
Expected output :	00
Actual output :	00
Remarks :	Test succeeded

Test Case 2: The program was input with instruction that had a comma missing for an instruction between its parameters. The program was supposed to show up with an error and it did as expected.



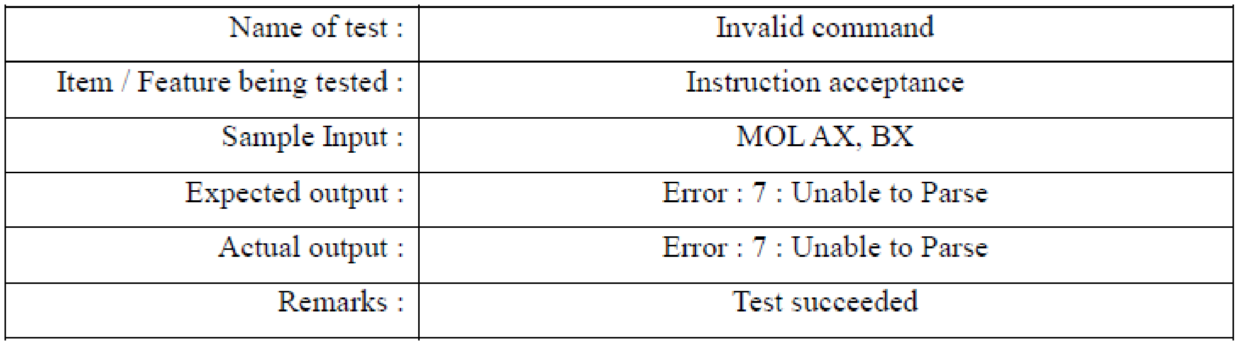
Name of test :	Missing Comma
Item / Feature being tested :	Parameter size
Sample Input :	MOV AL BL
Expected output :	Error : 1 : Unable to Parse
Actual output :	Error : 1 : Unable to Parse
Remarks :	Test succeeded

Test Case 3: When two instructions were given as input in the same line in the program the program should have taken the instructions as input but the software instead printed an error.



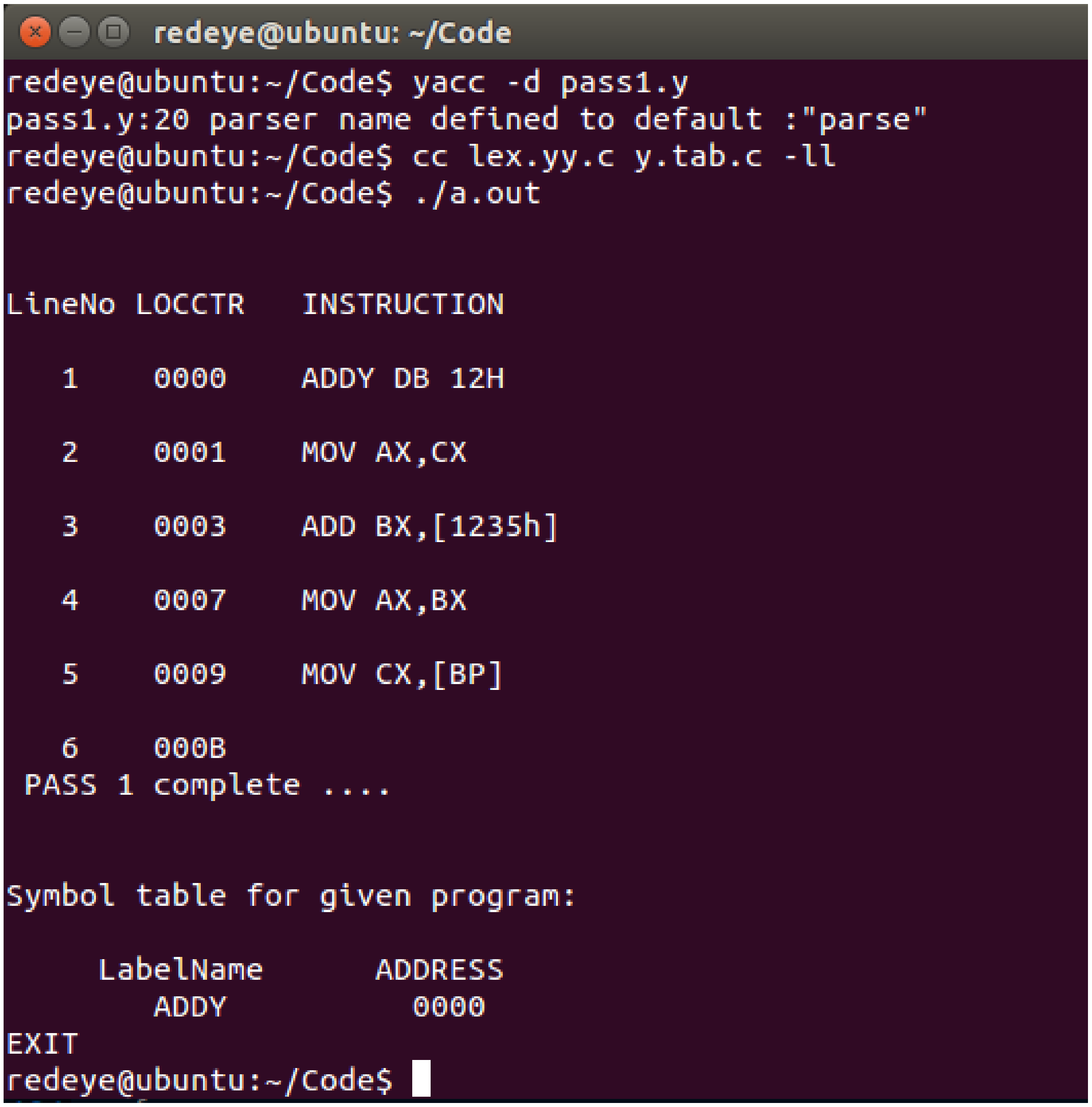
Name of test :	Same line instructions
Item / Feature being tested :	Instruction acceptance
Sample Input :	MOV AL, BL MOV BL, AL
Expected output :	Object code
Actual output :	Error : 6 : Unable to Parse
Remarks :	Test Failed

Test Case 4: When a command doesn’t exit then program shows an error which is expected.

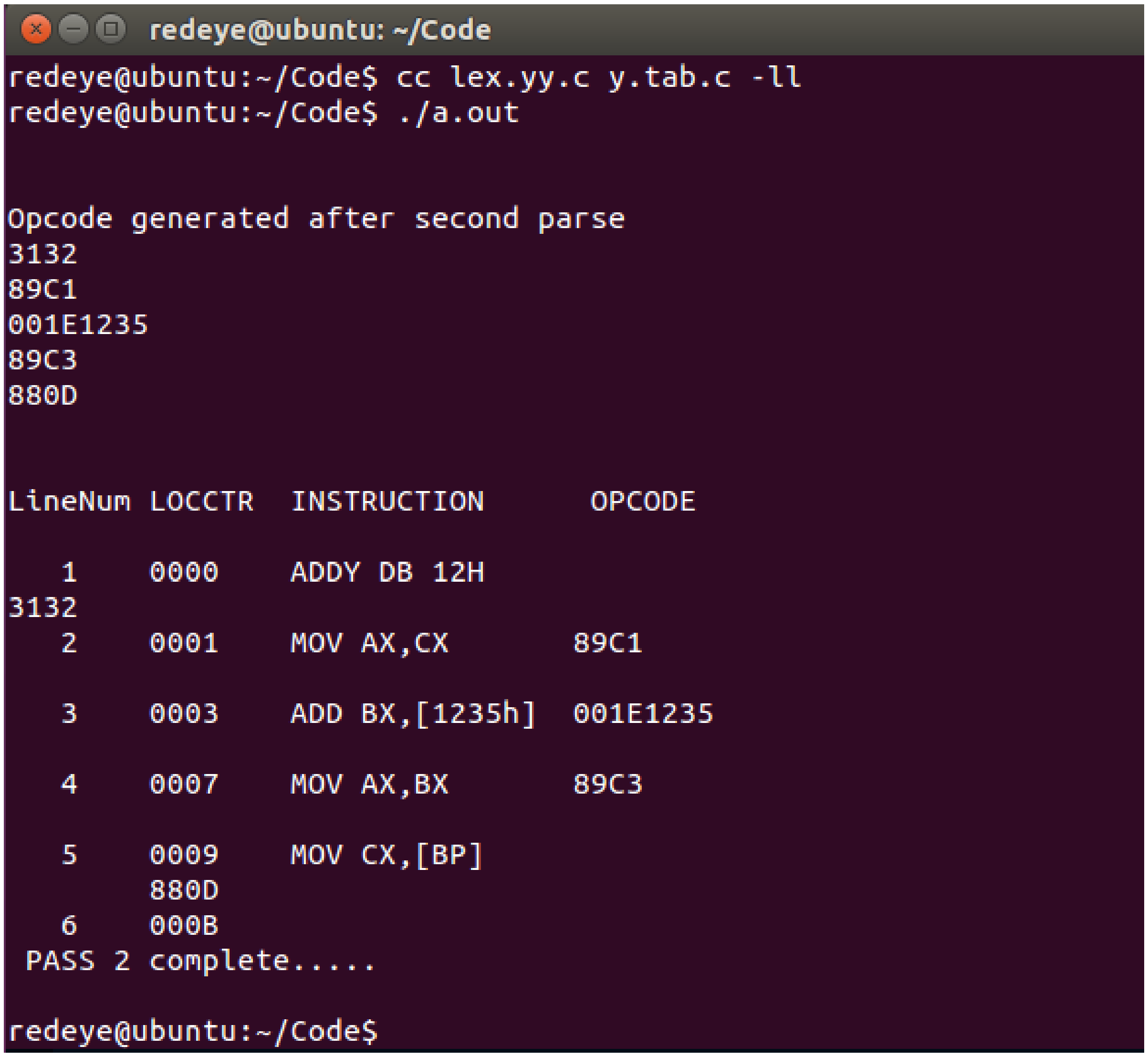


Name of test :	Invalid command
Item / Feature being tested :	Instruction acceptance
Sample Input :	MOL AX, BX
Expected output :	Error : 7 : Unable to Parse
Actual output :	Error : 7 : Unable to Parse
Remarks :	Test succeeded

CHAPTER 6
RESULTS AND INFERENCE



```
redeye@ubuntu:~/Code$ yacc -d pass1.y
pass1.y:20 parser name defined to default :"parse"
redeye@ubuntu:~/Code$ cc lex.yy.c y.tab.c -ll
redeye@ubuntu:~/Code$ ./a.out


LineNo LOCCTR   INSTRUCTION

   1     0000     ADDY DB 12H

   2     0001     MOV AX,CX

   3     0003     ADD BX,[1235h]

   4     0007     MOV AX,BX

   5     0009     MOV CX,[BP]

   6     000B
 PASS 1 complete ....


Symbol table for given program:

     LabelName      ADDRESS
        ADDY          0000
EXIT
redeye@ubuntu:~/Code$
```



```
redeye@ubuntu:~/Code$ cc lex.yy.c y.tab.c -ll
redeye@ubuntu:~/Code$ ./a.out


Opcode generated after second parse
3132
89C1
001E1235
89C3
880D


LineNum LOCCTR  INSTRUCTION      OPCODE

   1     0000    ADDY DB 12H
3132
   2     0001    MOV AX,CX       89C1

   3     0003    ADD BX,[1235h]  001E1235

   4     0007    MOV AX,BX       89C3

   5     0009    MOV CX,[BP]
         880D
   6     000B
 PASS 2 complete.....

redeye@ubuntu:~/Code$
```

In this project, the program has generated 2 files, namely, the object file and symbol table file. The Object file consists of the object module, formed according to the given set of assembly level instructions, which is then to be loaded into memory for execution. The Symbol table file consists of a list of all the labels used and their respective address value.

The input to the assembler is a source file. This file consists of a set of symbolic instruction and directives. The 2 pass assembler translates each symbolic instruction into one machine instruction. The directives, however, are not translated. Using directives is a way of asking the assembler for help. The assembler provides help by executing (rather than translating) the directives. The most important output of the assembler is the object file. It contains the assembled instructions to be loaded into memory and executed.

CHAPTER 7
CONCLUSION

7.1 Summary

This Project was an effort at learning the development of a system software which are the building blocks of programming. As we go through this project we come to understand the processes going on “behind the scenes” like how the object code is generated for the given source program in the assembly language and how the entries are made into the symbol table etc.

Although a lot of effort has been put in building this system software, there might be deficiencies. To the best of our knowledge this system takes care of most of the conditions that may occur at runtime environment.

To make the best of the designed system, the user has to work on the system and to train himself this would help the user in better understanding of the system and in turn increases the users’ efficiencies and skills in using the system.

This Project is developed such that anybody who has the basic idea about the assembly language can easily manage. There might be minor changes in the proposed system from time to time to update changes.

7.2 Limitations

- This assembler does not support the complete instruction set of 8086
- Macros have not been implemented.
- The assembly listing has been produced to be user friendly and cannot be directly loaded and run.
- This assembler does not support all the addressing modes.

7.3 Future Enhancements

There is scope for further improvement such as:

- Ability to handle all the remaining instructions along with their addressing modes.
- Ability to handle macros.

Generate a full-fledged object program that can be directly linked and executed by a linker.

BIBLIOGRAPHY

[1] Microprocessor and Interfacing, Tata McGraw-Hill, Revised second edition, Douglas V Hall

[2] System Software, Pearson Education, Third Edition, 2004, Leland L Beck, Manjula

[3] Advanced Microprocessors And Peripherals, Tata McGraw -Hill, 2004, A K Ray & K M Bhurchandi.

[4] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [2006]. Compilers, Principles, Techniques and Tools (2nd edition). Addison-Wesley, Reading, Massachusetts.

[5] John R. Levine, Tony Mason and Doug Brown, Lex and Yacc, O’Reilly, 2nd Edition 1995

APPENDIX-A : SOURCE CODE

Source Code Files required:

1	functions.h	(header file containing all common methods and modules used)
2	assemble.l	(Common lex file for pass1 and pass2)
3	pass1.y	(performs pass 1 operations)
4	pass2.y	(performs pass 2 operations)
7	optable.txt	(contains all the necessary instructions and their opcode equivalents)

[1]. Functions.h

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void tohex( unsigned int num,char hex[],int len)
{
   int i,j=len-1;
   while(j>=0)
  {
          i=num%16;
          num/=16;
          hex[j--]=((i<10)? ('0'+i):('A'+i-10));
  }
  hex[len]='\0';
}

void convert(char str[],char tar[])
{
  int i=0;
  while(str[i]!='\0')
  {
         tohex((int)str[i],tar+i*2,2);
         i++;
  }
}

void addhex(char hx1[],char hx2[],char sum[])
{
  int i=0,f1=1,f2=1;
  while(f1||f2)
  {
        if(hx1[i]=='\0') f1=0;
        if(hx2[i]=='\0') f2=0;
        sum[i]=f1*hx1[i]+f2*hx2[i]-(f1&&f2)*('0');

        if(sum[i]>'9'&&f1*hx1[i]<='9'&&f2*hx2[i]<='9')
              sum[i]+=7;
        i++;
  }
  //printf("\nSum = %s",sum);
  sum[i]='\0';
}


int toint(char hex[],int size)
{
  int num=0,i;
  for(i=0;i<size;i++)
  {
        num+=((hex[i]>='0'&&hex[i]<='9')? (hex[i]-'0'):(hex[i]-'A'+10));
        if(i<size-1) num*=16;
  }
  return num;
}

void getrm(int rorm,int reg,int mod,char rm[])
{
  int v=0,f=rorm/8;
  if(f) rorm-=8;
  if(reg>=8) reg-=8;
  v=(f<<8)+rorm+(reg<<3)+(mod<<6);
  tohex(v,rm,4);
}

struct instrn
{
       char opname[10],code[10];
       struct instrn *link;
}optab[150];

typedef struct instrn Instr;

void initopcodes()
{
  FILE *opcf;
  int c=0;
  char n[10]=" ",cur[10],ccode[10];
  Instr *temp,*cr;
  opcf=fopen("optable.txt","r");
  while(!feof(opcf))
  {
     fscanf(opcf," %s %s ",cur,ccode);
     optab[c].link=NULL;
     if(strcmp(n,cur)==0)
     {
             temp=((Instr *)malloc(sizeof(Instr)));
             cr=optab[c-1].link;
             if(cr!=NULL) {
                    for(;cr->link!=NULL;cr=cr->link);
                           cr->link=temp;
     }
       else optab[c-1].link=temp;
       strcpy(temp->opname,cur);
       strcpy(temp->code,ccode);
     }
     else
     {
             strcpy(optab[c].opname,cur);
             strcpy(optab[c].code,ccode);
             strcpy(n,cur);
             c++;
     }
     //getc(opcf);
  }
  fclose(opcf);
}


void get_code(char name[],char tar[],int d)
{
  int l=0,h=150,m=(l+h)/2,i;

  while(1)
  {
        int cmp=strcmp(name,optab[m].opname);
        if(l>h) break;
        if(cmp==0) break;
        else if(cmp>0) l=m+1;
        else h=m-1;
        m=(l+h)/2;
  }
  Instr *cur=optab[m].link;
  for(i=0;i<d-1;i++) cur=cur->link;
  if(d>0&&cur!=NULL) strcpy(tar,cur->code);
  else strcpy(tar,optab[m].code);
}

void displaycodes()
{
  int i=0;
  Instr *cur;
  for(;i<150;i++)
  {
       printf("\n%10s %10s",optab[i].opname,optab[i].code);
        cur=optab[i].link;
        if(cur!=NULL)
              while(cur) {printf("\n,%10s %10s",cur->opname,cur->code);cur=cur->link;}
  }
}

typedef struct symbol SYMBOL;
typedef struct symbol Symbol;
struct symbol
{
  char label[30];
  int addrs;
  int flag;
  SYMBOL *next;
 }symtab[26];

SYMBOL * search_symbol(char NAME[])
{
 int i = tolower(NAME[0]) -'a';
 SYMBOL *ptr1 = symtab[i].next;
 while(ptr1!=NULL && strcmp(ptr1->label,NAME)!=0)
       ptr1= ptr1->next;
 return ptr1;
 }
void insert_symbol(char NAME[],int addrs,int flg)
{
 SYMBOL *ptr,*temp;
 ptr = (SYMBOL *)malloc(sizeof(SYMBOL));
 ptr->next = NULL;
 ptr->flag=0;
 int index = tolower(NAME[0]) -'a';
 /*check if the symbol already exists in the hash table or not
  * if it is not present, just add the symbol
  * else increment the correspondin flag to represent the >1 point where the name is
repeated */
 if((temp= search_symbol(NAME))==NULL)
 {        //symbol not found, ok
       strcpy(ptr->label,NAME);
       ptr->addrs = addrs;
       //ptr->flag=0;
       ptr->next = symtab[index].next;
       symtab[index].next = ptr;

  if(flg==0)
        ptr->flag=0;
   else
       ptr->flag=flg;
 }
 else
 {
   //we have a repeated symbol error,set the flag
    temp->flag=flg+1;
  }
}
void write_symtab()
{
 char hex[4];
 FILE *ptr;
  ptr =fopen("symtab.txt","w");
  if(ptr==NULL)
       {
              printf("\n File cannot be created. Exiting....");
              exit(0);
    }
  SYMBOL *temp;
  int i=0;
  while(i<26)
  {
     temp = symtab[i].next;
              while(temp!=NULL)
              {
                     tohex(temp->addrs,hex,4);
                     fprintf(ptr,"%s %s %d\n",temp->label,hex,temp->flag);
                     temp =temp->next;
              }
        i++;
 }
 fclose(ptr);
 printf("\nEXIT");

}
void print_symtab()
{
  int i=0;
  char hexaddr[4];
  SYMBOL *node;
  printf("\nSymbol table for given program: \n");
  printf("\n     LabelName      ADDRESS");
  for(; i<26;i++)
  {
      node = symtab[i].next;
         while(node)
         {
               tohex(node->addrs,hexaddr,4);
                printf("\n%12s\t%10s",node->label,hexaddr);
                node =node->next;
       }
   }
 }


int intfl(char c)
{
  switch(c)
  {
    case '1': return 1;
       case '0': return 0;
       case '2':return 2;
       case '3': return 3;
       case '4': return 4;
       case '5': return 5;
       case '6':return 6;
       case '7': return 7;
       case '8': return 8;
       case '9': return 9;
  }
 }


void read_symtab()
{
  char buf[5],loc[30];
  int num;  FILE *ptr; char c[1];
  ptr= fopen("symtab.txt","r");
  if(ptr==NULL)
  {
        printf("\n Invalid file, exiting..");
        exit(0);
   }
   else
   {
       int i=0;int x;
       while(!feof(ptr))
       {
              fscanf(ptr,"%s",buf);
              fscanf(ptr,"%s" ,loc);
              fscanf(ptr," %d ",&x);
              //printf("\nAddrs  =%s",loc);
              insert_symbol(buf,toint(loc,4),x);//intfl(c[0]));

              // printf("\n %d",intfl(c[0]));
              i++;
       }
   }
  }
```

[2] assemble.h

```
 %{
       #include"y.tab.h"
       #include<string.h>
       int locctr ;
%}
Arith
("ADD"|"ADC"|"INC"|"AAA"|"DAA"|"SUB"|"SBB"|"DEC"|"NEG"|"CMP"|"AAS"|"DAS"|"MUL"|"IMUL"|
"AAM"|"DIV"|"IDIV"|"CBW"|"CWD")
Log ("NOT"|"SHL"|"SAH"|"ROL"|"ROR"|"RCL"|"RCR"|"AND"|"TEST"|"OR"|"XOR"|"SHR")

DataTrnsfr ("MOV"|"PUSH"|"POP"|"XCHG"|"XLAT"|"LEA"|"LDS"|"LAHF"|"SAHF"|"PUSHF"|"POPF")

CtrlTrnsfr
("CALL"|"JMP"|"RET"|"JE"|"JZ"|"JL"|"JNGE"|"JLE"|"JNG"|"JB"|"JNAE"|"JBE"|"JNA"|"JP"|"JP
E"|"JO"|"JS"|"JNE"|"JNZ"|"JML"|"JGE"|"JNLE"|"JG"|"JNB"|"JAE"|"JNBE"|"JA"|"JNP"|"JNO"|"
JNS"|"LOOP"|"LOOPZ"|"LOOPE"|"LOOPNZ"|"LOOPNE"|"JCXZ"|"INT"|"INTO"|"IRET"|"CLC"|"CMC"|"
STC"|"CLD"|"STD"|"CLI"|"STI"|"HLT"|"WAIT"|"ESC"|"LOCK")
```

```
Inout ("IN"|"OUT")
Reg8 ("AL"|"AH"|"BL"|"BH"|"CL"|"CH"|"DL"|"DH")
Reg16 ("AX"|"BX"|"CX"|"DX"|"SP"|"BP"|"SI"|"DI")
Segreg ("CS"|DS"|"ES"|"SS")

Mem  ("[BX][SI]"|"[BX][DI]"|"[BP][SI]"|"[BP][DI]"|"[BP]"|"[SP]"|"[DI]"|"[SI]")

Hex [0-9|a-f|A-F]
Hex8 {Hex}{2}[hH]
Hex16 {Hex}{4}[hH]
word ([a-zA-Z])([a-zA-Z0-9]*)
String ['"'][a-z|A-Z|0-9|,|"."|;|:|/|-|+|*|?|!|*|"("    |")"|_|=|"$"| ]+['"']
Spaces [ \t]*
%%

{Arith}              {ECHO; strcpy(yylval.str,yytext); return ARITH;}
{Log}                {ECHO; strcpy(yylval.str,yytext);return LOG; }
{DataTrnsfr}         {ECHO; strcpy(yylval.str,yytext);return DTTF;}
{CtrlTrnsfr}         {ECHO; strcpy(yylval.str,yytext);return CTTF;}
{Inout}              {ECHO;strcpy(yylval.str,yytext); return IO;}
{Hex8}               {ECHO; strcpy(yylval.str,yytext);return IBYTE;}
{Hex16}               {ECHO; strcpy(yylval.str,yytext);return IWORD;}
{Mem}                 {ECHO;
                       if(strcmp(yytext,"[SP]")==0) yylval.num=4;
                       else if(strcmp(yytext,"[BP]")==0) yylval.num=5;
                       else if(strcmp(yytext,"[SI]")==0) yylval.num=6;
                       else if(strcmp(yytext,"[DI]")==0) yylval.num=7;
                       else
                 yylval.num=(yytext[1]=='B')*((yytext[5]=='D')+2*(yytext[2]=='P'));
                      return MEM;}
{Spaces}             {ECHO;}
"DB"                  {ECHO; return DB;}
"DW"                  {ECHO; return DW;}
{Reg8}               {ECHO; yylval.num = ((yytext[1]=='L')?0:4)+ (yytext[0]-'A') +
((yytext[0]=='B')?3:0);
                     if(yytext[0]>='B') yylval.num-=1;
                     if(yytext[0] =='C' && yytext[1] =='L') {return CL;}
                             return REG8;}
{Reg16} {ECHO;
              yylval.num=(yytext[1]=='X')*(yytext[0]-'A'-
(yytext[0]>='B')+((yytext[0]=='B')? 3:0))+8;

                     if(strcmp(yytext,"SP")==0) yylval.num=12;
                     else if(strcmp(yytext,"BP")==0) yylval.num=13;
                     else if(strcmp(yytext,"SI")==0) yylval.num=14;
                     else if(strcmp(yytext,"DI")==0) yylval.num=15;

                     return REG16;}
{word}               {ECHO;strcpy(yylval.str,yytext);return LABEL;}
{String}             {ECHO;strcpy(yylval.str,yytext); return STR;}
":"                  {ECHO; return COLON; }
(";"[^\n]*)          {;}
\n                   {ECHO; return E;}
.                    {ECHO; return yytext[0]; }
%%
```

[3] pass1.y

```
%{
#include "functions.h"
extern FILE *yyin;
extern int locctr;
int lineno=1;
FILE *op;
SYMBOL *temp;
char hex[5];
%}

%token ARITH LOG DTTF CTTF IO REG8 REG16 MEM DB DW IBYTE IWORD LABEL STR COLON E
%token CL

%nonassoc CON
%nonassoc ','
%union
 {
    int num;
    char str[30];
  }

%start S
%type <str> LABEL
%type <str> STR
%type <num> VAL
%type <num> VAR
%type <num> NBYTES
%type <num> NWORDS
%type <num> IBYTE
%type <num> IWORD
%type <num> MEMORY
%type <num> RM
%type <num> CONST
%type <num> IMMED
%type <num> REGMEM
%type <num> INSTR
%type <num> ARITHMETIC
%type <num> DATATRANS
%type <num> LOGICAL
%type <num> CTRLTRANS

%%
S: N S
 |                   {printf("\n PASS 1 complete ...."); }
 ;

N: VAR E             {tohex(locctr,hex,4);printf("\n%4d\t%s\t",++lineno,hex);
                     fprintf(op,"%s     ",hex);}
| INSTR E
                  {locctr+=$1;tohex(locctr,hex,4);printf("\n%4d\t%s\t",++lineno,hex);
                     fprintf(op,"%s ",hex);}
| LABEL COLON E {if((temp=search_symbol($1))==NULL)
                      insert_symbol($1,locctr,0);
                     else {insert_symbol($1,locctr,temp->flag);}
                     tohex(locctr,hex,4);printf("\n%4d\t%s\t",++lineno,hex);
                     fprintf(op,"%s ",hex);
                     }
| E                  {printf("\n%4d",++lineno);}
;
VAR: LABEL VAL      {if((temp = search_symbol($1))==NULL)
                       insert_symbol($1,locctr,0);
                     else {insert_symbol($1,locctr,temp->flag);}
                             locctr+=$2;}
;
NBYTES: IBYTE ',' NBYTES  { $$ = 1+$3;}
| IBYTE                    { $$ = 1; }
;


NWORDS: IWORD ',' NWORDS   { $$ = 2+$3;}
       | IWORD             {$$ = 2;}
       ;


VAL:  DB NBYTES      { $$  = $2; }
   |  DW NWORDS      {$$ = $2; }
   | DB STR          {$$ = strlen($2)-2;}
   ;


INSTR: ARITHMETIC          {$$= $1;}
       | LOGICAL            {$$=$1;}
       | DATATRANS         {$$=$1;}
       | CTRLTRANS         {$$=$1;}
            ;

       STEP: '1'
       | CL
       ;

ARITHMETIC: ARITH REGMEM          {$$ = 2+$2;}
       | ARITH IMMED               {$$ = 2+$2;}
       | ARITH RM                 {$$ = 2+ $2;}
       |ARITH                     {$$= 1;}
            ;

LOGICAL:LOG REG ',' STEP          {$$= 2;}
            | LOG MEMORY ',' STEP {$$=2+ $2;}
            | LOG REGMEM          {$$=2+$2;}
            | LOG IMMED           {$$=2+$2;}
            ;


DATATRANS: DTTF REGMEM     {$$ = 2+$2;}
       | DTTF IMMED        {$$ = 2+$2;}
       | DTTF RM           {$$ = 2+$2;}
             ;

CTRLTRANS: CTTF LABEL    {Symbol *temp=search_symbol($2);
                           if(temp==NULL)
                                   $$=3;
                            else {
                                 if(abs(locctr+4-temp->addrs) > 255) $$=3;
                           else $$=2;
                            }
                           }
          | CTTF CONST            {$$=1+$2;}
          ;

REGMEM: REG ',' MEMORY            {$$=$3;}
       | MEMORY ',' REG           {$$=$1;}
       | REG ',' REG              {$$=0;}
       ;

IMMED: RM ',' CONST {$$=$1+$3;}
        ;

RM: REG       %prec CON           {$$=0;}
  | MEMORY    %prec CON            {$$=$1;}


MEMORY: MEM CONST                 {$$=$2;}
       | MEM                      {$$=0;}
       | '[' CONST ']'            {$$=$2;}
       | CONST                    {$$=$1;}
       | LABEL                    {Symbol *temp=search_symbol($1);
                                  if(temp==NULL)
                                          $$=2;
                                   else {
                                          if(abs(locctr+4-temp->addrs) > 255) $$=2;
                                          else $$=1;
                                          }
                                  }
       ;


CONST: IWORD                      {$$=2;}
       | IBYTE                    {$$=1;}
       ;

REG: REG8
   | REG16
    ;

%%

int main()
{
  locctr=0;
  yyin=fopen("prog.asm","r");
  op=fopen("inter.txt","w");
  tohex(locctr,hex,4);
  printf("\n\nLineNo LOCCTR   INSTRUCTION\n\n");
  printf("%4d\t%s\t",lineno,hex);
  yyparse();
  printf("\n\n");
  print_symtab();
  write_symtab();
  printf("\n");
  return 0;
}

int yyerror()
{
  printf("\nFatal Error %d:Unable to parse\n",lineno);

}
```

[4] pass2.y

```
%{
#include "functions.h"
FILE *op,*intr;
extern FILE *yyin;
char *c,h[14],h1[14];
int f=0,locctr=0,mod=0,lineno=1,s=0;
int err[20],front=0;
extern struct instrn t;
int yyerrstatus,errf=0;
SYMBOL *temp;
%}

%union {
  int num;
  char str[30];
}
```

```
%token LABEL STR ARITH LOG DTTF CTTF IO REG8 REG16 MEM IBYTE IWORD CL
%start S
%nonassoc CON
%nonassoc ','
%type <str>LABEL
%type <str>STR
%type <str>ARITH
%type <str>LOG
%type <str>DTTF
%type <str>CTTF
%type <str>IO
%type <num>REG8
%type <num>REG16
%type <num>MEM
%type <str>IBYTE
%type <str>IWORD
%type <num>CL
%token COLON E DB DW


%type <str> ARITHMETIC
%type <str> LOGICAL
%type <str>DATATRANS
%type <str> CTRLTRANS
%type <str> CONST
%type <str> REGMEM
%type <str> RM
%type <str> STEP
%type <str> NWORDS
%type <str> NBYTES
%type <num>REG
%type <num>MEMORY

%%
S: N S
  |                  {printf("\n PASS 2 complete.....");}
  ;

N: VAR E            {tohex(locctr,h,4);printf("%4d\t%s\t",++lineno,h);
                          fscanf(intr," %s ",h);locctr=toint(h,4);}
  | INSTR E         {tohex(locctr,h,4);printf("%4d\t%s\t",++lineno,h);
                          fscanf(intr," %s ",h);locctr=toint(h,4);}
  | LABEL COLON E   { if((temp =search_symbol($1))!=NULL&& temp->flag!=0)
                                 printf("\t Symbol declared/repeated error\n");

                                 tohex(locctr,h,4);
                                 printf("%4d\t%s\t",++lineno,h);
                                 fscanf(intr," %s ",h);
                                 locctr=toint(h,4);
                        }
  | E               {printf("%4d\t",++lineno);}

VAR: LABEL VAL      {if((temp =search_symbol($1))!=NULL&& temp->flag!=0)
                       printf("\n Symbol repeated here error\n");
                       }
VAL: DW NWORDS         {convert($2,h);printf("%s\n",h);fprintf(op,"%s\n",h);}
   | DB NBYTES             {convert($2,h);printf("%s\n",h);fprintf(op,"%s\n",h);}
   | STR                   {convert($1,h);printf("%s\n",h);fprintf(op,"%s\n",h);}

NWORDS: IWORD',' NWORDS   {$1[4]='\0';strcat($1,$3);strcpy($$,$1);}
       | IWORD             {$1[4]='\0'; strcpy($$,$1);}

NBYTES: IBYTE ',' NBYTES   {$1[2]='\0';strcat($1,$3);strcpy($$,$1);}
```

```
       | IBYTE             {$1[2]='\0';strcpy($$,$1);}

INSTR: ARITHMETIC          {if(!errf) printf("\t%s\n",$1);
                                    else printf("\tERROR\n");
                                  fprintf(op,"%s\n",$1);mod=f=s=errf=0; }
       | LOGICAL           {if(!errf) printf("\t%s\n",$1);
                            else printf("\tERROR\n");
                            fprintf(op,"%s\n",$1);mod=f=s=errf=0;}
       | DATATRANS         {if(!errf) printf("\t%s\n",$1);
                           else printf("\tERROR\n");
                           fprintf(op,"%s\n",$1);mod=f=s=errf=0;}
       | CTRLTRANS         {if(!errf) printf("\t%s\n",$1);
                            else printf("\tERROR\n");
                            fprintf(op,"%s\n",$1);mod=f=s=errf=0;}
       ;
ARITHMETIC: ARITH REGMEM          {get_code($1,h,0+(f==2));addhex(h,$2,$$);}
       | ARITH CONST              {get_code($1,h,2);addhex(h,$2,$$);}
       | ARITH RM                { get_code($1,h,0);addhex(h,$2,$$);}
       | ARITH                    {get_code($1,h,0);strcpy($$,h);printf("\t");}
       ;
LOGICAL: LOG REG ',' STEP        {getrm($2,0,3,h);addhex(h,$4,h);
                                  get_code($1,h1,0);addhex(h,h1,$$);}
       | LOG MEMORY ',' STEP      {getrm($2,0,mod,h);addhex(h,$4,h);
                                 get_code($1,h1,0);addhex(h,h1,$$);}
       | LOG REGMEM              {get_code($1,h,0+(f==2));addhex(h,$2,$$);}

       ;
STEP: CL                        {strcpy($$,"02");}
       | '1'                      {strcpy($$,"00");}
       ;
DATATRANS: DTTF REGMEM     {get_code($1,h,0+(f==2));addhex(h,$2,$$);}
       | DTTF RM           {get_code($1,h,2);addhex(h,$2,$$);}
       ;

CTRLTRANS: CTTF LABEL      {get_code($1,h,0);
                     Symbol *temp=search_symbol($2);
                     if(temp==NULL)
                      {strcpy(h1,"");yyerror(2);}
                     else
                      {
                           tohex(locctr-temp->addrs,h1,4); //find out the difference
                                                   if(h1[0] =='F' && h1[1]=='F')
                           //in case of forward   jump, addrs= FFF... so the first 2F
                           get added with opcode **
                       { h1[0]='0'; h1[1]='0'; }   //so make them 00 so that 0+*=*
                       }
                       addhex(h,h1,$$);  //concatenating the hex values here
                      }
       | CTTF CONST        {get_code($1,h,0);addhex(h,$2,$$);}
       ;
REGMEM: REG ',' MEMORY     {if(f%10==2)
                     {getrm($1,0,3,$$);
                      tohex($3,h,4-f/5-s);
                      strcat($$,h);
                      if((f/5==0)&&(($1<8&&s==0)||($1>=8&&s==2))) yyerror(2);}
                     else
                      {getrm(((f%10==1)? 6:$3%8),$1,mod,$$);
                       if(f/10==1)
                        {tohex($3/8,h,4-s);strcat($$,h);}
                       }
                     mod=0;
                    }
       | MEMORY ',' REG   {getrm(((f%10==1)? $1%8:6),$3,mod,h);addhex(h,"02",$$);
                           if(f/10==1)
```

```
                            {tohex($1/8,h,4-s);strcat($$,h);}
                          mod=0;f=0;
                          }
       | REG ',' REG    {getrm($3,$1,3,h);strcpy($$,h);
                         if(f<=1&&(($1<8&&$3>=8)||($3<8&&$1>=8)))
                                   yyerror(1);
                       }
       | MEMORY ',' MEMORY {if(f!=2) yyerror(3);}
       ;
       RM: REG        %prec CON    {getrm($1,0,3,h);strcpy($$,h);}
       | MEMORY       %prec CON    {getrm($1,0,mod,h);
                                           strcpy($$,h);
                                           mod=0;}

       MEMORY: MEM CONST   {$$=$1+(toint($2,strlen($2)-1)<<3);
                                          mod=(strlen($2)-1)/2;
                                          f=10;}
       | MEM         {$$=$1;}
       | '[' CONST ']'            {$$=toint($2,strlen($2)-1)<<3;
                                            f=11;}
       | LABEL                    {Symbol *temp=search_symbol($1);
                      if(temp==NULL)
                        {$$=0;yyerror(2);}
                     else
                       $$=locctr-temp->addrs;
                      f=12;
                     }
       | ARITH       {yyerror(0);}
       | LOG  {yyerror(0);}
       | DTTF {yyerror(0);}
       | CTTF {yyerror(0);}
       ;

CONST: IWORD %prec CON       {strcpy($$,$1);s=0;}
       | IBYTE     %prec CON {strcpy($$,$1);s=2;}
       ;
REG: REG8           {$$=$1;}
       | REG16      {$$=$1;}
       ;
       %%

int yyerror(int errcode)
{
  if(!errf)
       err[front++]=errcode+(lineno<<3);
  errf=1;
  return 0;
}

void disperr()
{
  int i=0;
  if(front) printf("\n\n%d errors found!!",front);
  for(;i<front;i++)
        switch(err[i]%8)
         {
            case 0:printf("\nError %d:Invalid operand",err[i]/8);break;
            case 1:printf("\nError %d:Missmatch in size of operands",err[i]/8);break;
            case 2:printf("\nError %d:Label not defined",err[i]/8);break;
            case 3:printf("\nError %d:Atleast 1 operand must be a
            register",err[i]/8);break;
           default:printf("\nFatal Error %d:Unable to parse",err[i]/8);break;
       }
}
```

```
int main()
{


FILE *fp;
    char line[10];
    fp =fopen("opcode.obj","r");
    printf("\n\nOpcode generated after second parse\n");
    while(fgets(line,10,fp))
    printf("%s",line);




  read_symtab();
  //print_symtab();
  printf("\n\nLineNum LOCCTR  INSTRUCTION      OPCODE\n\n");
  initopcodes();
  intr=fopen("inter.txt","r");
  fscanf(intr," %s ",h);
  locctr=toint(h,4);
       printf("%4d\t0000\t",lineno);
  op=fopen("opcode.obj","w");
  yyin=fopen("prog.asm","r");
  yyparse();
  disperr();
  printf("\n\n");


}
```