**CHAPTER 1**

**INTRODUCTION**

An assembler program is a language processor. It simply translates statements of a ‘source’ program into their machine language counterparts. It reads records from input device (code F1) and copies them to output device (code 05).At the end of the file, it writes EOF on the output device, then RSUB to the operating system.

**Types of Assemblers:**

1. Load and go one pass assembler: It produces object code directly in memory for immediate execution.

2. Store and go one pass assembler: It produces the usual kind of object program for later execution.

* 1. **Overview of One pass assembler for 8086**

An assembler is a program that creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("opcode") as well as other control bits. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. Assemblers are the simplest of a class of systems programs called *translators*. A translator is simply a program, which translates from one (computer) language to another (computer) language. In the case of an assembler, the translation is from assembly language to object language (which is input to a loader). Notice that an assembler, like all translators, adds nothing to the program which it translates, but merely changes the program from one form to another. The use of a translator allows a program to be written in a form which is convenient for the programmer, and then the program is translated into a form convenient for the computer. Assembly language is almost the same as machine language. The major difference is that assembly language allows the declaration and use of symbols to stand for the numeric values to be used for opcodes, fields, and addresses. An assembler inputs a program written in assembly language, and translates all of the symbols in the input into numeric values, creating an output object module, suitable for loading. The object module is output to a storage device, which allows the assembled program to be read back into the computer by the loader.

* 1. **Scope**

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 for many, perhaps even most, computer science degrees. The first assemblers were simple assemble-go systems. All they could do was to assemble code directly in memory and start execution. It was quickly realized, however, that linking is an important feature, required even by simple programs. The pioneers of programming have developed the concept of the routine library very early, and they needed assemblers that could locate library routines, load them into memory, and link them to the main program. It was this task of locating, loading, and linking—of assembling a single working program from individual pieces—that created the name assembler. Today, assemblers are translators and they work on one program at a time.

There are two types of assemblers based on how many passes through the source are needed to produce the executable program: A Single pass assembler and a Multi pass assembler. We will be focusing on the single-pass assembler. A Single-pass assembler scans the program only once and creates the equivalent binary program. The assembler substitutes all the symbolic instructions with machine code in one pass. The main advantage of using a single-pass assembler is that every source statement needs to be processed once. Here's how a general single pass assembler works:

● Most computers come with a specified set of very basic instructions that correspond to the basic machine operations that the computer can perform.

● The programmer can write a program using a sequence of these assembler instructions

● This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started.

● The assembler program takes each program statement in the source program and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).

● The output of the assembler program is called the object code or object program relative to the input source program.

● The object program can then be run (or executed) whenever desired.

In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Today, assembler programming is used only where very efficient control over processor operations is needed. It requires knowledge of a particular computer's instruction set, however. Historically, most programs have been written in "higher-level" languages such as COBOL, FORTRAN, PL/I, and C. These languages are easier to learn and faster to write programs with than assembler language. It‟s converted into machine code by a compiler.

* 1. **Methodology**

The assembler generates machine instructions as follows: 1. Evaluates the mnemonics to produce their machine code 2. Evaluates the symbols, literals, addresses to produce their equivalent machine addresses 3. Converts the data constants into their machine representations **Data Structures Used:** SYMTAB is used to store values (addresses) assigned to labels. OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalents.

We also need a Location counter LOCCTR. This is a variable that is used to help in the assignment of addresses. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instructions or date area to be generated is added to LOCCTR. Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label

**CHAPTER 2**

**SYSTEM REQUIREMENTS**

A modern assembler has two inputs and two outputs. The first input is short, typically a single line typed at a keyboard. It activates the assembler and specifies the name of a source file (the file containing the source code to be assembled). It may contain other information that the assembler should have before it starts. This includes commands and specifications such as: The names of the object file and listing file. Display (or do not display) the listing on the screen while it is being generated. Display all error messages but do not stop for any error. Save the listing file and do not print it (see below). This program does not use macros. The symbol table is larger (or smaller) than usual and needs a certain amount of memory. All these terms are explained elsewhere. An example is the command line that invokes Macro, the VAX assembler. The line: MACRO /SHOW=MEB /LIST /DEBUG ABC activates the assembler, tells it that the source program name is abc.mar (the .mar extension is implied), that binary lines in macro expansions should be listed (shown), that a listing file should be created, and that the debugger should be included in the assembly. The second input is the source file. It includes the symbolic instructions and directives. The assembler translates each symbolic instruction into one machine instruction. The directives, however, are not translated. The directives are our way of asking the assembler for help. The assembler provides the help by executing (rather than translating) the directives. A modern assembler can support as many as a hundred directives. They range from ORG, which is very simple to execute, to MACRO, which can be very complex.

The first and most important output of the assembler is the object file. It contains the assembled instructions (the machine language program) to be loaded later into memory and executed. The object file is an important component of the assembler-loader system. It makes it possible to assemble a program once, and later load and run it many times. It also provides a natural place for the assembler to leave information to the loader, instructing the loader in several aspects of loading the program. Note, however, that the object file is optional. The user may specify no object file, in which case the assembler generates only a listing. The second output of the assembler is the listing file. For each line in the source file, a line is created in the listing file, containing:

● The Location Counter.

● The source line itself.

● The machine instruction (if the source line is an instruction), or some other relevant information (if the source line is a directive).

The listing file is generated by the assembler, sent to the printer, gets printed, and is then discarded. The user, however, can specify either not to generate a listing file or not to print it. There are also directives that control the listing. They can be used to suppress parts of the listing, to print page headers, or to control the printing of macro expansions. The cross-reference information is normally a part of the listing file, although the MASM assembler creates it in a separate file and uses a special utility to print it. The cross-reference is a list of all symbols used in the program. For each symbol, the point where it is defined and all the places where it is used are listed. Assembler

**2.1 Functional Requirements**

As mentioned in the introduction, the main input of the assembler is the source file. Each record on the source file is a source line that specifies either an assembler instruction or a directive. A typical source line has four fields. A label (or a location), a mnemonic (or operation), an operand and a comment. Example: LOOP ADD R1,ABC PRODUCING THE SUM In this example, LOOP is a label, ADD is a mnemonic meaning to add, R1 stands for register 1, and ABC is the label of another source line. R1 and ABC are two operands that make up the operand field. The example above is, therefore, a double operand instruction. When a label is used as an operand, we call it a symbol. Thus, in our case, ABC is a symbol.

Modern assemblers are more flexible and do not require any special format. If a label exists, it must end with a „:‟. Otherwise, the individual fields should be separated by at least one space (or by a tab character), and subfields should be separated by either a comma or parentheses. This rule makes it convenient to enter source lines from a keyboard, but is ambiguous in the case of a source line that has a comment but no operand. Example: EI ;ENABLE ALL INTERRUPTS The semicolon guarantees that the word ENABLE will not be considered an operand by the assembler. This is why many assemblers require that comments start with a semicolon.

**2.1.1 Software requirements:**

OPERATING SYSTEM: Red Hat Linux 9.0 version 2.4.2 0-8 or Fedora Core 2.4.22-1.2215.nptl

COMPILER USED : GCC version 3.2.2

EDITOR : VI Editor version 6.1

PROGRAMMING LANGUAGE : GNU C, Lex version 2.5.4

**2.1.2 Hardware requirements:**

MAIN PROCESSOR : Pentium IV (500MHz)

RAM SIZE : 128 MB

CACHE MEMORY:256KB

DISKETTE DRIVE:1.FFMB,3.5inches

**2.2 Non Functional Requirements**

**2.2.1 Robustness**

This program for the assembler focuses on handling unexpected termination and unexpected actions. This program handles all the terminations and actions gracefully by displaying accurate and unambiguous error messages. These error messages allow the user to more easily debug the program.

**2.2.2 Stability**

One pass assembler for 8086 will be stable over time and will not need changes.

**2.2.3 Response time**

Total amount of time taken to give the output for the respective input is minimal.

**2.2.4 Quality**

This assembler is reliable, maintainable and sustainable. This program has the ability to perform satisfactorily in service and is suitable for its intended purpose.

**CHAPTER 3**

**SYSTEM DESIGN**

**3.1 Data Flow Diagram Of A Single Pass Assembler:**

**LEVEL-0 DFD**

Single Pass Assembler

Source Code

Object code

Figure 3.1

**LEVEL-1 DFD**

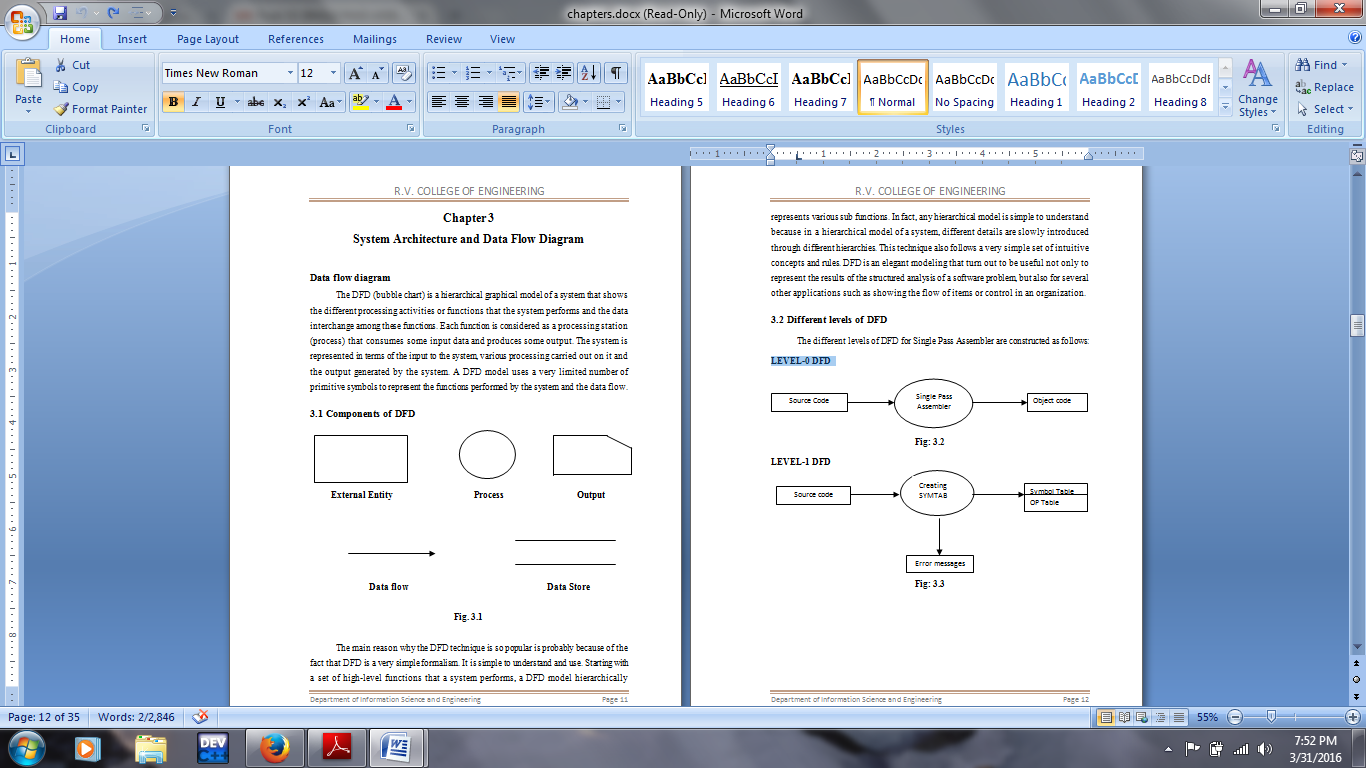
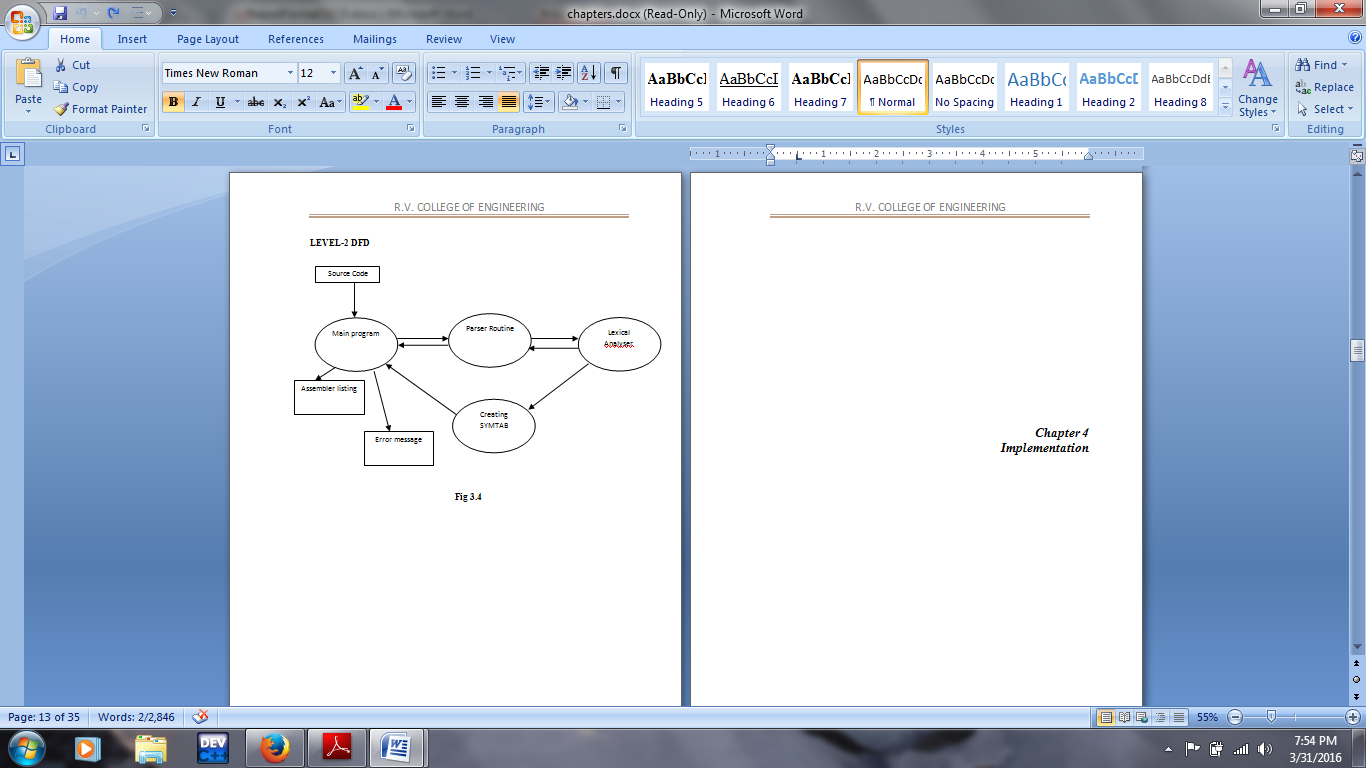
****

Figure 3.2

**LEVEL-2 DFD**

**Figure 3.3**

**3.2 Algorithm**

\\Input: source file containing the assembly level program for 8086

\\Output: object file containing the assembler listing.

Main routine ()

{

Declare required data and Data Structures;

Open source file;

Display error messages if file cannot be accessed;

Open a new file for writing assembler listing;

AA: Scan source file to find tokens;

Write the token to a temporary file;

Call parser routine;

Check for error flags if so display error messages;

Take care to increment location pointer and line count;

Write into object file the assembler listing;

Repeat AA for all instructions in source file;

}

**3.3 Data Structures**

**Operation Code Table (OPTAB**)

Used to look up mnemonic operation codes and translate them to their machine language. It contains instruction format& length. OPTAB is used to look up and validate operation codes.In SIC/XE, assembler search OPTAB in Pass 1 i.e. single pass to find the instruction lengthfor incrementing LOCCTR. It organizes as a hash table (static).

**Symbol Table (SYMTAB)**

Used to store values assigned to labels. It also includes the nameand value(address) for each label. It also includes flagsto indicate error conditions. It contains type, length.Labels are entered into SYMTAB, along with assigned addresses (from LOCCTR). It also organizes as a hash table. The entries are rarely deleted from the table.

**Location Counter (LOCCTR):**

Used to help in the assignment of address. LOCCTR is initialized to address specified in START. When reach a label, the current value of LOCCTR gives the address to be associated with that label.

**CHAPTER 4**

**IMPLEMENTATION**

**4.1 Source Code**

**Includefile.h file:**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int check\_status = 0;

char s[32];

char object\_code[100000][32];

int noOfInstructions = 0;

extern int lineno;

char opcode[7], d[2], w[2], mode[3], reg[4], rm[15], addr[17];

int op1\_type, op2\_type, arg;

char bin\_hex[16][5] = {"0000","0001","0010","0011","0100","0101","0110","0111","1000","1001","1010","1011","1100","1101","1110","1111"};

int n = 0;

struct symbol

{

int noOfLinks;

int link\_addr[100];

int used;

char label[25];

int addrs;

struct symbol \*link;

};

struct optable\_type

{

char op[15];

char op\_code[8];

}optable[100];

typedef struct symbol Symbol;

Symbol \*head = NULL;

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';

}

Symbol \*search\_symbol(char label[])

{

if(head == NULL)

return NULL;

Symbol \*cur = head;

while(cur&&strcmp(cur->label,label)!=0)

cur=cur->link;

return cur;

}

void insert\_symbol(char label[],int addrs, int used)

{

Symbol \*temp;

temp=(Symbol \*)malloc(sizeof(Symbol));

strcpy(temp->label,label);

temp->addrs=addrs;

temp->noOfLinks = 0;

temp->link=head;

temp->used = used;

head = temp;

}

void final\_errors()

{

Symbol \*cur;

char hex[5];

int c=0;

cur=head;

while(cur) {

if(cur->addrs == -1)

{

printf("Error: Label not found! : %s\n", cur->label);

}

if(cur->used == 0)

{

printf("Warning: Label not used! : %s\n", cur->label);

}

cur=cur->link;

}

}

void print\_symtab()

{

Symbol \*cur;

char hex[5];

int c=0;

printf("\nSymtab\n");

cur=head;

while(cur) {

if(cur->addrs == -1)

{

cur=cur->link;

continue;

}

tohex(cur->addrs,hex,4);

printf("%s %s\n",cur->label,hex);

cur=cur->link;

}

}

void write\_symtab()

{

FILE \*sym;

int i;

sym=fopen("symtab.txt","w");

Symbol \*cur;

char hex[5];

cur=head;

while(cur!=NULL)

{

tohex(cur->addrs,hex,4);

fprintf(sym,"%s %s\n",cur->label,hex);

cur=cur->link;

}

fclose(sym);

}

void write\_object()

{

FILE \*fp = fopen("assembly\_code.obj", "w");

int i;

for(i = 0; i < noOfInstructions; i++)

{

fprintf(fp,"%s\n", object\_code[i]);

}

}

void print\_object()

{

int i;

printf("\n\n\nObject Code: \n");

for(i = 0; i < noOfInstructions; i++)

{

printf("%s\n", object\_code[i]);

}

}

void load\_opfile()

{

int i = 0;

FILE \*fp = fopen("newoptable.txt", "r");

if(!fp)

{

printf("Optable doesn't exist!!!");

exit(0);

}

while(fscanf(fp, "%s%s", optable[i].op, optable[i].op\_code) > 0)

{

i++;

}

n = i;

}

void initialize()

{

int i;

noOfInstructions = 0;

for(i = 0; i < 1000; i++)

{

strcpy(object\_code[i], "\0");

}

}

void flush\_string()

{

op1\_type = -1;

op2\_type = -1;

arg = -1;

strcpy(s, "\0");

strcpy(opcode, "\0");

strcpy(d, "\0");

strcpy(w, "\0");

strcpy(reg, "\0");

strcpy(w, "\0");

strcpy(mode, "\0");

strcpy(addr, "\0");

strcpy(rm, "\0");

}

void first\_to\_hex()

{

int i = 0, j;

char p[5], hex\_val[2];

hex\_val[1] = '\0';

while(i < 16)

{

for(j = 0; j < 4; j++)

p[j] = s[i + j];

p[j] = '\0';

for(j = 0; j < 16; j++)

{

if(!strcmp(bin\_hex[j], p))

{

if(j < 10){

hex\_val[0] = j + 48;

strcat(object\_code[noOfInstructions], hex\_val);

}

else

{

hex\_val[0] = j + 55;

strcat(object\_code[noOfInstructions], hex\_val);

}

}

}

i += 4;

}

for(i = 16; i < strlen(s); i++)

p[i - 16] = s[i];

p[i - 16] = '\0';

strcat(object\_code[noOfInstructions], p);

noOfInstructions++;

flush\_string();

}

void hex\_to\_binary\_opcode(char hex[10])

{

int i;

for(i = 0; hex[i] != '\0'; i++)

{

if(i == 0){

if(hex[i] == '0')

strcat(s, "0000");

else if(hex[i] == '1')

strcat(s, "0001");

else if(hex[i] == '2')

strcat(s, "0010");

else if(hex[i] == '3')

strcat(s, "0011");

else if(hex[i] == '4')

strcat(s, "0100");

else if(hex[i] == '5')

strcat(s, "0101");

else if(hex[i] == '6')

strcat(s, "0110");

else if(hex[i] == '7')

strcat(s, "0111");

else if(hex[i] == '8')

strcat(s, "1000");

else if(hex[i] == '9')

strcat(s, "1001");

else if(hex[i] == 'A')

strcat(s, "1010");

else if(hex[i] == 'B')

strcat(s, "1011");

else if(hex[i] == 'C')

strcat(s, "1100");

else if(hex[i] == 'D')

strcat(s, "1101");

else if(hex[i] == 'E')

strcat(s, "1110");

else if(hex[i] == 'F')

strcat(s, "1111");

else{

printf("Wrong opcode!");

exit(1);

}

}

else

{

if(hex[i] == '0')

strcat(s, "00");

else if(hex[i] == '1')

strcat(s, "01");

else if(hex[i] == '2')

strcat(s, "10");

else if(hex[i] == '3')

strcat(s, "11");

else if(hex[i] == '4')

strcat(s, "00");

else if(hex[i] == '5')

strcat(s, "01");

else if(hex[i] == '6')

strcat(s, "10");

else if(hex[i] == '7')

strcat(s, "11");

else if(hex[i] == '8')

strcat(s, "00");

else if(hex[i] == '9')

strcat(s, "01");

else if(hex[i] == 'A')

strcat(s, "10");

else if(hex[i] == 'B')

strcat(s, "11");

else if(hex[i] == 'C')

strcat(s, "00");

else if(hex[i] == 'D')

strcat(s, "01");

else if(hex[i] == 'E')

strcat(s, "10");

else if(hex[i] == 'F')

strcat(s, "11");

else

{

printf("Wrong Opcode!");

exit(0);

}

}

}

}

void reverse(char bin[8])

{

int i;

char temp;

for(i = 0; i < 4; i++)

{

temp = bin[i];

bin[i] = bin[7-i];

bin[7-i] = temp;

}

}

void decimal\_to\_hex(int dec, char a[3])

{

int i = 0, m;

char temp;

while(dec > 0)

{

m = (dec % 16);

if(m < 10)

a[i++] = m + '0';

else

a[i++] = m + 55;

dec /= 16;

}

if(dec == 0)

{

a[0] = '0';

a[1] = '0';

}

else if(i == 1)

a[1] = '0';

temp = a[0];

a[0] = a[1];

a[1] = temp;

a[2] = '\0';

}

void resolve\_links(Symbol \*temp)

{

int i;

char a[2];

decimal\_to\_hex(temp->addrs, a);

for(i = 0; i < temp->noOfLinks; i++)

{

strcat(object\_code[temp->link\_addr[i]], a);

}

temp->noOfLinks = 0;

}

void make\_instruction()

{

//printf("opcode %s\nmode %s\nreg %s\nrm %s\naddr %s", opcode, mode, reg, rm, addr);

if(!strcmp(reg, "AL") || !strcmp(reg, "BL") || !strcmp(reg, "CL") || !strcmp(reg, "DL") || !strcmp(reg, "AH") ||!strcmp(reg, "BH") ||!strcmp(reg, "CH") ||!strcmp(reg, "DH") ){

op1\_type = 0;

}

else

{

op1\_type = 1;

}

if(!strcmp(rm, "AL") || !strcmp(rm, "BL") || !strcmp(rm, "CL") || !strcmp(rm, "DL") || !strcmp(rm, "AH") ||!strcmp(rm, "BH") ||!strcmp(rm, "CH") ||!strcmp(rm, "DH") )

{

op2\_type = 0;

}

else

{

op2\_type = 1;

}

if(op1\_type != op2\_type)

{

printf("Error: %d : Parameters are of different sizes\n", lineno);

noOfInstructions++;

flush\_string();

return;

}

int i, j;

for(i = 0; i < n; i++)

{

if(!strcmp(opcode, optable[i].op))

break;

}

hex\_to\_binary\_opcode(optable[i].op\_code);

strcat(s, "0");

// if instruction is of only one byte

if(!strcmp(mode,"\0"))

{

strcat(s,"0");

char p[5], hex\_val[2];

hex\_val[1] = '\0';

for(j = 0; j < 4; j++)

p[j] = s[j];

p[4] = '\0';

for(j = 0; j < 16; j++)

{

if(!strcmp(bin\_hex[j], p))

{

if(j < 10){

hex\_val[0] = j + 48;

strcat(object\_code[noOfInstructions], hex\_val);

}

else

{

hex\_val[0] = j + 55;

strcat(object\_code[noOfInstructions], hex\_val);

}

}

}

for(j = 4; j < 8; j++)

p[j - 4] = s[j];

p[4] = '\0';

for(j = 0; j < 16; j++)

{

if(!strcmp(bin\_hex[j], p))

{

if(j < 10){

hex\_val[0] = j + 48;

strcat(object\_code[noOfInstructions], hex\_val);

}

else

{

hex\_val[0] = j + 55;

strcat(object\_code[noOfInstructions], hex\_val);

}

}

}

noOfInstructions++;

flush\_string();

return;

}

//for w

if(op1\_type == 1)

strcat(s, "1");

else

strcat(s, "0");

strcat(s, mode); // for mode

if(!strcmp(rm, "DIRECT"))

strcat(s,"000");

// for reg

if(!strcmp(reg, "AL") || !strcmp(reg, "AX"))

strcat(s,"000");

else if(!strcmp(reg, "BL") || !strcmp(reg, "BX"))

strcat(s,"011");

else if(!strcmp(reg, "CL") || !strcmp(reg, "CX"))

strcat(s,"001");

else if(!strcmp(reg, "DL") || !strcmp(reg, "DX"))

strcat(s,"010");

else if(!strcmp(reg, "AH") || !strcmp(reg, "SP"))

strcat(s,"100");

else if(!strcmp(reg, "CH") || !strcmp(reg, "BP"))

strcat(s,"101");

else if(!strcmp(reg, "DH") || !strcmp(reg, "SI"))

strcat(s,"110");

else if(!strcmp(reg, "DI") || !strcmp(reg, "BH"))

strcat(s,"111");

else

strcat(s,"000");

// for rm

if(!strcmp(rm, "INDIRECT") || !strcmp(rm, "LABEL"))

strcat(s,"110");

else if(!strcmp(rm, "DIRECT"));

else if(!strcmp(rm, "AL") || !strcmp(rm, "AX"))

strcat(s,"000");

else if(!strcmp(rm, "BL") || !strcmp(rm, "BX"))

strcat(s,"011");

else if(!strcmp(rm, "CL") || !strcmp(rm, "CX"))

strcat(s,"001");

else if(!strcmp(rm, "DL") || !strcmp(rm, "DX"))

strcat(s,"010");

else if(!strcmp(rm, "AH") || !strcmp(rm, "SP"))

strcat(s,"100");

else if(!strcmp(rm, "CH") || !strcmp(rm, "BP"))

strcat(s,"101");

else if(!strcmp(rm, "DH") || !strcmp(rm, "SI"))

strcat(s,"110");

else if(!strcmp(rm, "DI") || !strcmp(rm, "BH"))

strcat(s,"111");

else if(!strcmp(rm, "[BX+SI]"))

strcat(s, "000");

else if(!strcmp(rm, "[BX+DI]"))

strcat(s, "001");

else if(!strcmp(rm, "[BP+SI]"))

strcat(s, "010");

else if(!strcmp(rm, "[BP+DI]"))

strcat(s, "011");

else if(!strcmp(rm, "[SI]"))

strcat(s, "100");

else if(!strcmp(rm, "[DI]"))

strcat(s, "101");

else if(!strcmp(rm, "[BX]"))

strcat(s, "111");

Symbol \*sym;

char a[3];

if(!strcmp(rm, "DIRECT"))

{

strcat(s,addr);

s[strlen(s) - 1] = '\0';

}

else if(!strcmp(rm, "LABEL"))

{

sym = search\_symbol(addr);

if(sym == NULL)

{

insert\_symbol(addr, -1, 1);

sym = search\_symbol(addr);

sym->link\_addr[sym->noOfLinks++] = noOfInstructions;

}

else if(sym->addrs != -1)

{

sym->used = 1;

decimal\_to\_hex(sym->addrs, a);

strcat(s, a);

}

else

{

sym->used = 1;

sym -> link\_addr[sym->noOfLinks++] = noOfInstructions;

}

}

//Converting first 8 bits to hex

first\_to\_hex();

}

**Lex file:**

%{

#include "y.tab.h"

char hex[5];

int locctr;

int i = 0;

%}

END ("END")

ZERO\_ARG ("RET"|"AAA"|"AAD"|"AAM"|"AAS"|"CLC"|"CLD"|"CLI"|"CMC"|"CWD"|"DAA"|"CBW"|"CMPSB"|"CMPSW"|"DAS"|"HLT")

ONE\_ARG ("PUSH"|"MUL"|"POP"|"DIV"|"IDIV"|"IMUL"|"INC"|"DEC"|"CALL"|"INT"|"JAE"|"JMP"|"INT")

TWO\_ARG ("MOV"|"ADC"|"AND"|"ADD"|"CMP"|"IN")

Reg8 ("AL"|"AH"|"BL"|"BH"|"CL"|"CH"|"DL"|"DH")

Reg16 ("AX"|"BX"|"CX"|"DX"|"SP"|"BP"|"SI"|"DI")

MM ("[BX+SI]"|"[BX+DI]"|"[BP+SI]"|"[BP+DI]"|"[SI]"|"[DI]")

Hex16 [0-9a-fA-F]{4}"H"

Hex8 [0-9a-fA-F]{2}"H"

Name ([a-zA-Z])([a-z|A-Z|0-9])+

String ['"'][a-z|A-Z|0-9|,|"."|;|:|/|-|+|\*|?|!|\*|"(" |")"|\_|=|"$"| ]+['"']

Spaces [ \t]\*

%%

{END} {return END;}

{ZERO\_ARG} { strcpy(yylval.str, yytext); return OP\_ZERO; }

{ONE\_ARG} { strcpy(yylval.str, yytext); return OP\_ONE; }

{TWO\_ARG} { strcpy(yylval.str, yytext); return OP\_TWO; }

"DW" {return DW;}

"DB" {return DB;}

{Hex16} {strcpy(yylval.str,yytext);return WCONST;}

{Hex8} {strcpy(yylval.str,yytext);return BCONST;}

{MM} {

strcpy(yylval.str, yytext);

return MEMORY;}

{Reg8} {strcpy(yylval.str, yytext);

return REG8;}

{Reg16} { strcpy(yylval.str, yytext);

return REG16;}

{Name} {strcpy(yylval.str,yytext);return LABEL;}

{String} {strcpy(yylval.str,yytext);return STR;}

\n {return NL;}

{Spaces} {}

. {return yytext[0];}

%%

int yywrap()

{return 1;}

**Yacc file:**

%{

#include "includefile.h"

extern FILE \*yyin;

extern int locctr;

int lineno=1;

FILE \*op;

char hex[5];

int flag = 0;

%}

%union {

int val;

char str[40];

}

%token OP\_ZERO OP\_ONE OP\_TWO END

%token REG8 REG16 BCONST WCONST

%token MEMORY DB DW NL STR LABEL

%nonassoc CON

%nonassoc ','

%type <str> OP\_ZERO

%type <str> OP\_ONE

%type <str> OP\_TWO

%type <str> LABEL

%type <str> REG16

%type <str> REG8

%type <str> REG

%type <str> MEMORY

%type <str> STR

%type <val> VAL

%type <val> VAR

%type <val> INSTR

%type <val> WLIST

%type <val> BLIST

%type <str> WCONST

%type <str> BCONST

%type <val> REGMEM

%type <val> IMMEDIATE

%type <val> CONST

%type <val> RM

%type <val> MEM

%type <val> ZERO\_ARG

%type <val> ONE\_ARG

%type <val> TWO\_ARG

%%

PROG: Stat {flag = 0;};

Stat: NS {lineno++;} Stat

|

| error NL {lineno++;flush\_string();} Stat

NS: VAR NL

| INSTR {make\_instruction(); } NL {locctr+=$1;}

| LABEL INSTR NL {

Symbol \*sym = search\_symbol($1);

if(sym==NULL){

insert\_symbol($1,locctr,0);

locctr+=$2;

make\_instruction();

}

else if(sym ->addrs == -1){

sym->addrs = locctr;

resolve\_links(sym);

locctr+=$2;

make\_instruction();

}

else

{

flush\_string();

noOfInstructions++;

printf("Error Same label found! at line no. %d\n", lineno);

}

}

| NL

| END {return;}

VAR: LABEL VAL {

$$ = $2;

Symbol \*sym = search\_symbol($1);

if(sym==NULL)

insert\_symbol($1,locctr,0);

else if(sym ->addrs == -1)

{

sym->addrs = locctr;

resolve\_links(sym);

}

else

{

printf("Error: Same label found! at line no. %d\n", lineno);

}

locctr+=$2;

}

VAL: DW WLIST {printf("WLIST:%d\n",$2);$$=$2;}

| DB BLIST {printf("BLIST:%d\n",$2);$$=$2;}

| DB STR {printf("STr:%d\n",$2);$$=strlen($2)-2;}

WLIST: WCONST ',' WLIST {$$=2+$3;}

| WCONST {$$=2;}

BLIST: BCONST ',' BLIST {$$=1+$3;}

| BCONST {$$=1;}

INSTR: ZERO\_ARG {$$=$1;}

| ONE\_ARG {$$=$1;}

| TWO\_ARG {$$=$1;}

ZERO\_ARG : OP\_ZERO {$$ = 1; strcpy(opcode, $1);}

ONE\_ARG : OP\_ONE RM {$$ = 1 + $2; strcpy(opcode, $1); }

TWO\_ARG : OP\_TWO IMMEDIATE {$$ = 1 + $2; strcpy(opcode, $1); }

| OP\_TWO REGMEM {$$ = 1 + $2; strcpy(opcode, $1); }

REGMEM: REG ',' MEM {$$=$3; strcpy(reg, $1); strcpy(mode, "00");}

| REG ',' REG {$$=0; strcpy(reg, $1); strcpy(rm, $3); strcpy(mode, "11");}

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

RM: REG %prec CON {$$=0; strcpy(rm, $1); arg = 1; strcpy(mode, "11");}

| MEM %prec CON {$$=$1; arg = 1; strcpy(mode, "00");}

MEM: MEMORY {$$=0; strcpy(rm, $1);}

| '[' CONST ']' {$$=$2; strcpy(rm, "INDIRECT");}

| CONST {$$=$1; strcpy(rm, "DIRECT");}

| LABEL {

$$ = 1;

strcpy(rm, "LABEL");

strcpy(addr, $1);

}

CONST: WCONST {$$=2; strcpy(addr, $1);}

| BCONST {$$=1; strcpy(addr, $1);}

REG: REG8 {strcpy($$, $1);}

| REG16 {strcpy($$, $1);}

;

%%

int main(int argc, char \*argv[])

{

head = NULL;

flush\_string();

load\_opfile();

locctr=0;

yyin=fopen(argv[1],"r");

if(!yyin) printf("Input file not found");

yyparse();

if(flag)

final\_errors();

else

{

print\_symtab();

print\_object();

write\_symtab();

write\_object();

}

printf("\n");

return 0;

}

void yyerror(char \*s)

{

flag = 1;

printf("\nError: %d : %s\n",lineno, s);

}

**4.2 Result Discussion**

In this project, the program has generated the two tables, namely, the object table, symbol table. The Object table 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 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 single 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 table. It contains the assembled instructions to be loaded into memory and executed. On completion of this project, it can be inferred that a single pass assembler can be used for a simple set of instructions where all references are defined before that symbol is used (to avoid forward references). To resolve forward referencing linked lists are used.

**CHAPTER 5**

**CONCLUSION**

It has been an important area of research of computer science on working this project. This project provides us great learning experience and great knowledge about the working of the single pas assembler.

With the help of this project, we can better understand the Integrate software components written separately into a single working unit. We can also better understand about the different aspects of assembler.

An assembler is a language processor. It generates object program equivalent to source code. It is obvious that an assembler which does more than one pass over the source program would create an intermediate file which it would modify during the further passes. Resolving references and other complex constructs in every pass until the target code is ready and can be loaded into the physical memory for execution with or without the help of a linker/loader.

**5.1 Limitation:**

The single pass assembler has one major drawback, which is dealing with forward references. The forward referencing problem arises in a very simple way: the expression evaluator attempts to evaluate an operand which is a symbol by searching the symbol table, and finds that the symbol is not in the symbol table.

**5.2 Future scope:**

1. Some amount of output code optimization have to be implemented.

2. To have more op table contents.

3. To generate more enhanced and proper object code.

**CHAPTER 6**

**REFERENCES**

[1] Mealy, George H. [1963]. A generalized assembly system. Technical Report RM-3646-PR, Rand Corporation, Santa Monica, CA.

[2] D. Solomon, Assemblers and Loaders: Prentice Hall, 1993.

[3] A. V. Aho,M. S. Lam,R. Sethi,J. D. Ullman Compilers: Principles, Techniques, & Tools 2nd ed.: Addison Wesley 2007.

[4] elearning.vtu.ac.in/P7/enotes/CS51/assem-SS.pdf

[5] K. Cooper and L. Torczon, Engineering a Compiler, 2nd ed.: Morgan Kaufmann, 2011.

[6] Leland L Beck; “System Software: An Introduction to System Programming”; Pearson Education; 3rd Edition

[7] Douglas V Hall; “Microprocessors and Interfacing; Tata McGraw-Hill; 2nd Edition.

[8] Jeff Duntemann - Assembly Language Step-by-Step: Programming with DOS and linux.

[9] Thomas Niemann; “A Guide to Lex and Yacc”; E-papers [10] A.K. Ray & K.M. Burchandi; Advanced microprocessors and Peripherals; Tata McGraw- Hill; 2nd Edition.

**CHAPTER 7**

**APPENDIX**

We compiled the code on Xcode compiler on Mac OSX 10.11.3 . In case of any errors, the corresponding error statement will be displayed.

**Fig 6.1: First example’s Input asm file**

****

**Fig 6.2: First example’s Output in the terminal**

****

**Fig 6.3: Second example’s Input asm file**

****

**Fig 6.4: First example’s Output in the terminal**

****

**Fig 6.5: Third example’s Input asm file**

****

**Fig 6.6: Third example’s Output in the terminal**

****