

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANA SANGAMA, BELAGAVI-590018**



2016 – 2017

A Report on

“Single Pass Macro Processor for 8086”

Submitted to RVCE (Autonomous Institution Affiliated to Visvesvaraya Technological University (VTU), Belgaum) in partial fulfillment of the requirements for the award of degree of

**BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING
by**

Madhusudhan Aithal M 1RV13CS078

Mohith M R 1RV13CS085

*Under the guidance
Of*

**Prof. Suma B,
Assistant Professor,
Department of CSE, R.V.C.E.,
Bangalore - 560 059**

**Prof. Srividya M S
Assistant professor,
Department of CSE, R.V.C.E.,
Bangalore - 560 059**



**R. V. College of Engineering,
(Autonomous Institution Affiliated to VTU)**

Department of Computer Science and Engineering
R V COLLEGE OF ENGINEERING
R.V.VIDYANIKETAN POST, MYSORE ROAD,
BENGALURU- 560059
2016-17

VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI

R.V. COLLEGE OF ENGINEERING,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Mysore Road, R.V.Vidyaniketan Post, Bengaluru - 560059



CERTIFICATE

Certified that the System Software and Compiler Design mini project entitled “**Single Pass Macro Processor for 8086**” carried out by **Madhusudhan Aithal M**, USN: **1RV13CS078** and **Mohith M R**, USN: **1RV13CS085**, a bonafide students of **R.V. College of Engineering, Bangalore** in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the **Visvesvaraya Technological University, Belgaum** during the year 2015-2016. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirement.

Prof. Suma B
Assistant Professor,
Department of CSE,
R.V.C.E., Bangalore –59

Prof. Srividya M S,
Assistant Professor,
Department of CSE,
R.V.C.E., Bangalore –59

Dr. Shobha G
Head of Department,
Department of CSE,
R.V.C.E., Bangalore –59

Name of the Examiners

Signature with Date

1. _____

2. _____

ACKNOWLEDGEMENTS

We would like to acknowledge R.V.C.E. for providing us the opportunity to work on this mini-project.

We also express our heartfelt thanks to Dr. Shobha G, Head of Department, Computer Science & Engineering, for providing us with the required facilities and information which helped us in the completion of our project.

We would personally like to thank Prof. Suma B and Prof. Srividya M S for providing timely assistance & guidance.

We would also like to acknowledge the co-operation shown by the lab administrators and lab assistants towards the completion of our project.

Madhusudhan Aithal M

1RV13CS078

Mohith M.R

1RV13CS085

ABSTRACT

A macro processor replaces each macro instruction with the corresponding group of source language statements. When it is required to include the macro text in your assembly program, all that has to be provided is the name of the macro. Normally, it performs no analysis of the text it handles. It does not concern the meaning of the involved statements during macro expansion. This project aims at implementing a single-pass macro processor for 8086. A single pass macro processor is one that can completely translate the source code into machine language in only one pass. The only restriction in the one pass macro processor is that the definition of a macro must appear in the source program before any statements that invoke that macro. The macro processor can alternate between macro definition and macro expansion to handle recursive macro definition. Nested macro definitions are allowed. An input file containing the 8086 source code with macro call and definition is read and the necessary tables are created from this. Finally, the output file containing the expanded code with macro definitions removed is created by the macro processor.

TABLE OF CONTENTS

<u>Topics</u>	<u>Page No.</u>
1. INTRODUCTION	1
1.1 Overview of Single Pass Macro Processor for 8086	1
1.2 Scope	1
1.3 Methodology	2
2. SYSTEM REQUIREMENTS	3
2.1 Functional Requirements	3
2.2 Non Functional Requirements	3
3. SYSTEM DESIGN	4
3.1 DFD	4
3.2 Algorithm	5
3.2 Data structures	7
4. IMPEMENATION	8
4.1 Source Code	8
4.1.1 Main Function	8
4.1.2 Using Lex	9
4.2 Result Discussion	12
5. CONCLUSION	13
5.1 Limitation	13
5.2 Scope for Further Enhancements	13
6. BIBLOGRAPHY	14
7. APPENDIX- A (Snapshots)	15

CHAPTER 1

INTRODUCTION

1.1 Overview of Single-Pass Macro Processor for 8086

A macro processor enables you to define and to use macros in your assembly programs. A macro represents a commonly used group of statements in the source programming language. Macros provide several advantages when writing assembly programs.

- The frequent use of macros can reduce programmer-induced errors.
- The scope of symbols used in a macro is limited to that macro.
- Macros are well-suited for creating simple code tables.

The macro processor replaces each macro instruction with the corresponding group of source language statements. When it is required to include the macro text in your assembly program, all that has to be provided is the name of the macro. A single pass processor can completely translate the source code into machine language in only one pass. Most processors take 2 passes as there may be some forward referencing where the return address won't be known.

1.2 Scope

This processor will be made to work as a one pass macro processor by ensuring all the macros are defined during the first pass, i.e. definition of a macro must appear before any statements that invoke that macro. The body of one macro can contain definitions of other macros. If the processor can alternate between macro definition and macro expansion, then it can easily handle recursive macro definition. The single pass macro processor will be expected to produce an output much faster than a multi pass processor as it only goes through the entire code once. This makes it smaller and far more efficient.

Expected input: Source Code(with macro) for 8086

Expected output: Expanded Code (with macro definitions removed)

1.3 Methodology

The assembler directives used in macro definition are:

MACRO: To identify the beginning of macro definition

ENDM: To identify end of macro definition

When a macro definition is encountered, it is entered in the DEFTAB and a corresponding entry is made in NAMTAB. The normal approach is to continue entering until ENDM is encountered. However, if there is a program having a macro defined within another macro, while defining in the DEFTAB, the very first ENDM is taken as the end of the Macro definition. This does not complete the definition as there is another outer macro which completes the definition of the macro as a whole. Therefore, the DEFINE procedure keeps a counter variable. Every time a macro directive is encountered, this counter is incremented by 1. The moment the innermost macro ends (as indicated by the directive ENDM), it starts decreasing the value of the counter variable by one. The last ENDM should set the counter value to zero. Once the counter becomes zero, the ENDM corresponds to the original MACRO directive.

After all the macro definitions have been analyzed in accordance with the above procedure, the rest of the input program is read. Whenever a macro call is read, the argument entries are made into ARG TAB according to their position in the argument list.

Finally, the contents of the three data structures are used to expand the program using all the necessary notations and rules.

CHAPTER 2

SYSTEM REQUIREMENTS

2.1 Functional Requirements

- A Data structure is required to store the name (NAMTAB), definition of the macro (DEFTAB) and arguments of the macro (ARGTAB).
- Macro processor should replace each macro instruction with the corresponding group of source language statements.
- It should handle various types of parameters like constants, parameters through memory locations etc.
- It should be able to handle nested macro i.e. a macro calling another macro.
- Input Requirements: Source code should contain the macro definition in accordance with the 8086 syntax (MACRO & ENDM).

2.2 Non-functional Requirements

- Validate the definition of macro: Comments in macro definition shouldn't be expanded in the main program, check for parameters and check for the registers used in the definition of the macro.
- Macros must be defined before the segment definitions i.e. macro definition should be done before calling the macro. Single pass macro processors cannot handle the macros that are defined after it is called.
- Macro definition cannot occur inside another macro.
- In addition, the factors of performance and reliability have to be taken into consideration. The Macro-processor should be efficient in expanding the macro-definitions in the source code.

CHAPTER 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 within a software system. The various levels of *data flow diagrams* are shown below (Fig.1-Fig.3)...

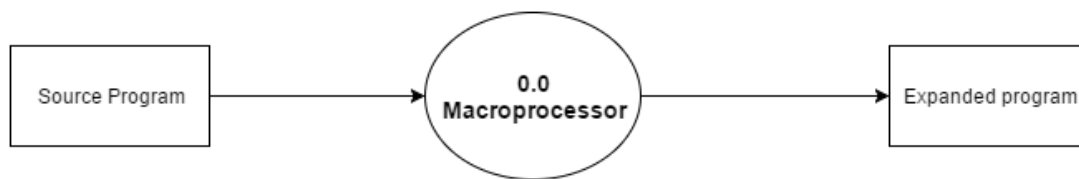


Fig.1: Level (0) Data Flow Diagram

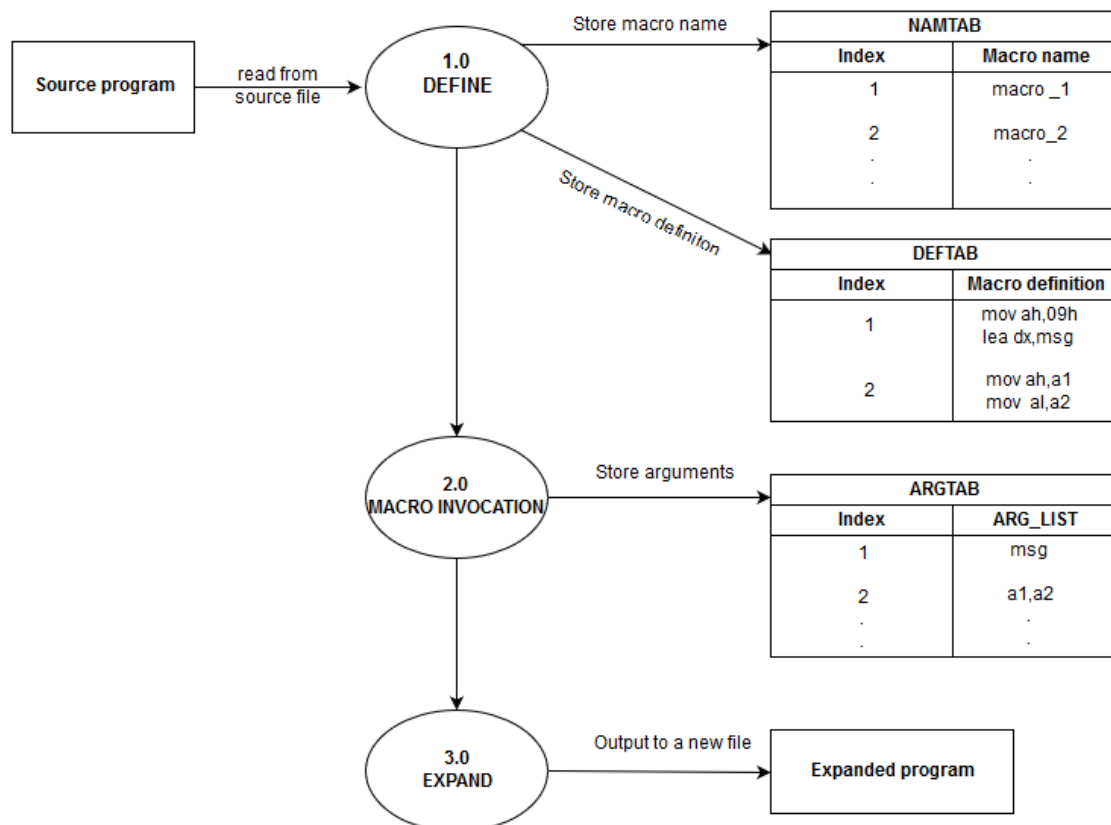


Fig.2: Level (1) Data Flow Diagram

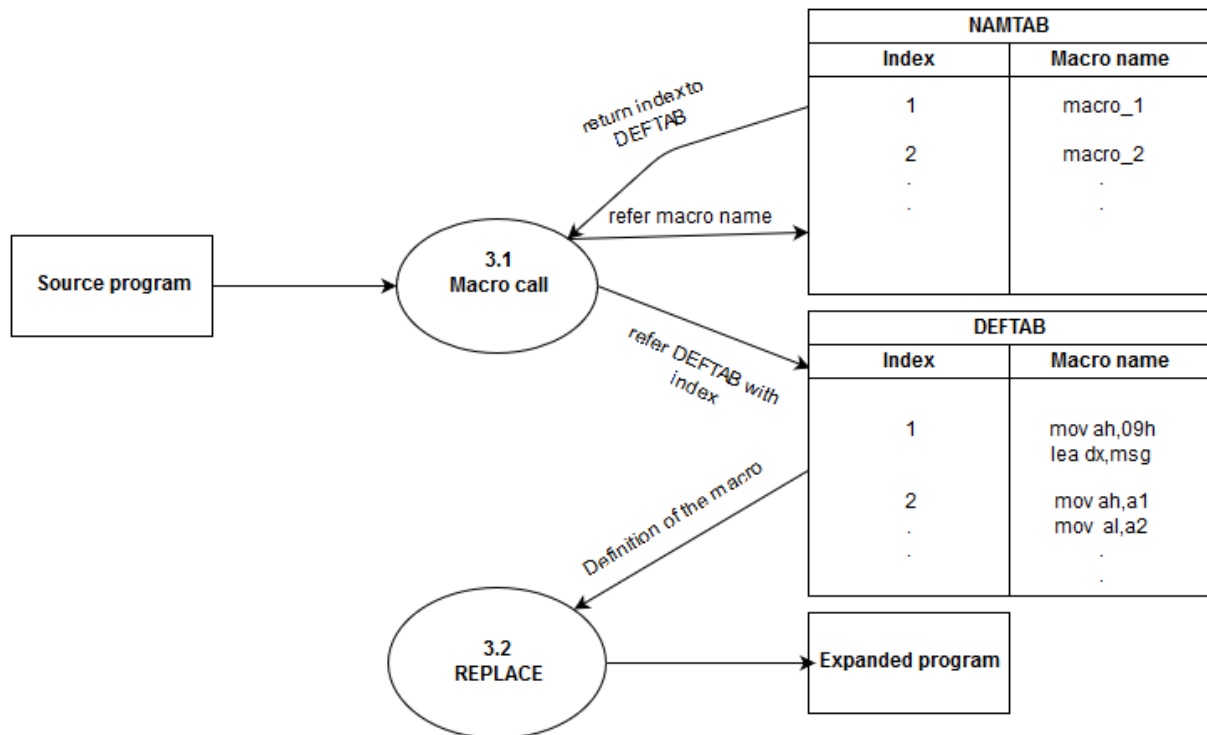


Fig.3: Level (2) Data Flow Diagram

3.2 Algorithm

The algorithm requires the following procedures:

- The procedure **DEFINE** makes the entry of *Macro Name* in the NAMTAB and *Macro Prototype* in DEFTAB.
- The procedure **EXPAND** is called to set up the argument values in ARGTAB and expand a *Macro Invocation* statement.

Algorithm:

```

Begin{macro processor}
  While OP CODE != 'END' do
    Read next line from input file
    Call PROCESSLINE
  End{While}

```

End{macro processor}

Procedure PROCESSLINE

Begin:

 Search NAMTAB for OPCODE

 If found then

 EXPAND

 Else if OPCODE= 'MACRO' then

 DEFINE

 Else write source line to expanded file

End{PROCESSLINE}

Procedure DEFINE

Begin:

 Enter macro name into NAMTAB

 Enter macro prototype into DEFTAB

 LEVEL: -1

 While LEVEL > 0, do

 Begin:

 GETLINE

 If this is not a comment line, then

 Begin:

 Substitute positional notation for parameters

 Enter line into DEFTAB

 If OPCODE = 'MACRO' then

 LEVEL: = LEVEL + 1

 If OPCODE = 'ENDM' then

 LEVEL:= LEVEL - 1

 End{if not comment line}

 End{While}

Store in NAMTAB pointers to beginning and end of definition

End{DEFINE}

Procedure EXPAND

Begin:

 Get first line of macro definition (prototype) from DEFTAB

 Set up arguments from macro invocation in ARGTAB

 While not end of macro definition do

 Begin:

 Get next line of macro definition from DEFTAB

 Substitute arguments from ARGTAB for positional notation

 Call PROCESSLINE

 End{While}

End{EXPAND}

3.3 Data Structures

The following data structures are required for the implementation –

DEFTAB (Definition Table)

- Stores the macro definition including *macro prototype* and *macro body*
- Comment lines are omitted.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

NAMTAB (Name Table)

- Stores macro names
- Serves as an index to DEFTAB
- Pointers to the beginning and the end of the macro definition (DEFTAB)

ARGTAB (Argument Table)

- Stores the arguments according to their positions in the argument list.
- As the macro is expanded, the arguments from the Argument table are substituted for the corresponding parameters in the macro body.

CHAPTER 4

IMPLEMENTATION

4.1 Source Code

4.1.1 Main Function

```
main ()
{
    int i=0;
    for(i=0;i<100;i++)
    {
        args_tab[i]=NULL;
        def_tab[i]=NULL;
    }
    char inputfile[100] = "input.txt", outputfile[100] =
"output.txt";
    printf ("-----One Pass
Macro Processor for 8086-----\n");
    printf ("Enter the name of the input file : ");
    scanf ("%s",inputfile);
    printf ("Input file : %s \n",inputfile);
    printf ("Output file : %s \n",outputfile);
    yyin = fopen (inputfile, "r");
    yyout = fopen (outputfile, "w");
    yylex ();
    int cn,r;
    printf("-----NAMTAB-----\n");
    printf("Index\t Macro_name\n");
    for(cn=0;cn<last_entry;cn++)
    {
        r=cn+1;
        printf("%d\t\t%s\n",r,name_tab[cn]);
    }
    printf("-----ARGTAB-----\n");
    printf("Index\tARG_LIST\n");
    for(cn=0;cn<last_entry;cn++)
```

```
        {
        printf("%d\t",cn+1);
        display_params(cn);
        }
printf("-----DEFTAB-----\n");
        printf("Index\t Macro_definition\n");
        for(cn=0;cn<last_entry;cn++)
        {
        printf("%d\t",cn+1);
        display_def(cn);
        printf("\n");
        }
printf("-----One Pass Macro
Processor for 8086-----\n");
}
yywrap ()
{
return 1;}
```

4.1.2 Using Lex

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran.

In our code, we use Lex to generate an analyzer in C in order to detect the macro prototype and definition.

```
%%
^[ ]*("endm"|"ENDM") {

    if(flag==1)
    {
        flag=0;
    }
    else
    {
        displayerror("Detected endm before detecting macro name");
    }
}

^([ ]*)[a-zA-Z][a-zA-Z0-9]*([ ]*)macro[a-zA-Z0-9, ]* {
    if(flag==0)
    {
        flag=1;
        int i,j,k,l,m,n;
        n=0;
        char name[100]={};

        while(yytext[n]!=' ')
            n++;

        l=0;
        while(yytext[n]!=' ')

            name[l++]=yytext[n++];

        name[l]='\0';

        macro_ref=last_entry;
        last_entry++;

        name_tab[macro_ref]=malloc(sizeof(char)*l);
        strcpy(
            name_tab[macro_ref],name);

        while(yytext[n]!=' ')
            n++;
        char
            j=0;

        for(i=0;i<5;i++)
        {

            if(tolower(macrostring[i])!=tolower(yytext[n+i]))
```

```

        {
            j++;
            break;
        }

    }
    if(j!=0)
        displayerror("Error
in detecting the \"MACRO\" word in the macro prototype");
    n+=5;
    while(yytext[n]==' ')
        n++;

    if(n<strlen(yytext))
        if(strlen(yytext+n)>0)

get_param_list(yytext+n,macro_ref);

    }

    else
    {
        displayerror("Unknown flag Detected
in the macro prototype");

    }

}

^(.)*
    {
        char first_substring[100]={};
        int n=0,i,j,k,l,m,loc=-1;
        if(flag==0)
        {
            while(yytext[n]==' ')
                n++;
            i=0;
            while(yytext[n]!='
'&&yytext[n]!='&&yytext[n]!='\n')
                first_substring[i++]=yytext[n++];
            if((loc=macro_present(first_substring))>-1)
            {
                replace_with_macro(yytext+n,loc,first_substring);
            }
            else
            {
                fprintf(yyout,"%s",yytext);

                fprintf(yyout,"\n");
            }
        }
        else if(flag==1)
        {

```



```
add_def(yytext,macro_ref);
                                }
                                else
                                {
                                displayerror("Unknown
flag Detected in the macro prototype search");
                                }
                                }
                                }

\n {} ;

%%
```

4.2 Result Discussion

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. For testing the program, several basic assembly level programs were taken as the input files and analysis of the output files for each input was done. The implementation provides a functional macro processor that performs all basic substitution functions on assembly language input codes. It takes in an input file containing code with macro call and definition, creates the necessary tables needed for performing its functions, and also creates an output file that contains the expanded code. It is easy to use and the readability of the output files is good, thus making it easier to debug any issues along the way. The program has been tested for several cases including nested macros. It is found to work efficiently and accurately.

CHAPTER 5

CONCLUSION

5.1 Limitation

The major drawback of a single pass macro processor is the problem of forward referencing. This is not possible as the macro processor only scans the code once and carries out macro identification as well as expansion in one pass. In the above implementation, all macro definitions must be added before any statement invokes a macro.

5.2 Scope for Future Enhancements

The implementation works well for many test cases. It correctly recognizes when the number of arguments is too less and displays an appropriate message. The macro processor is also able to handle nested macro definitions. However, code optimization may be required for simplicity purposes. Although nested macro definition has been taken care of, nested macro invocation (recursive macro expansion) may be implemented as an improvement.

CHAPTER 6

BIBLIOGRAPHY

[1] SOFTWARE DESIGN WITH MACRO PROCESSORS Dr. T.B. Williams Electrical Engineering University of Arizona Tucson, Az 85721 125

[2] The ML/1 macro processor. Communications of the ACM , 10, 618-23. [Brown 1974] Brown, P. J. (1974).

[3] MACROS- A COMMON SENSE APPROACH TO MICROCOMPUTER SOFTWARE CONTROL R. Bousley C. Deiotte K. Pratt Boeing Aerospace Company Boeing Military Airplane Company

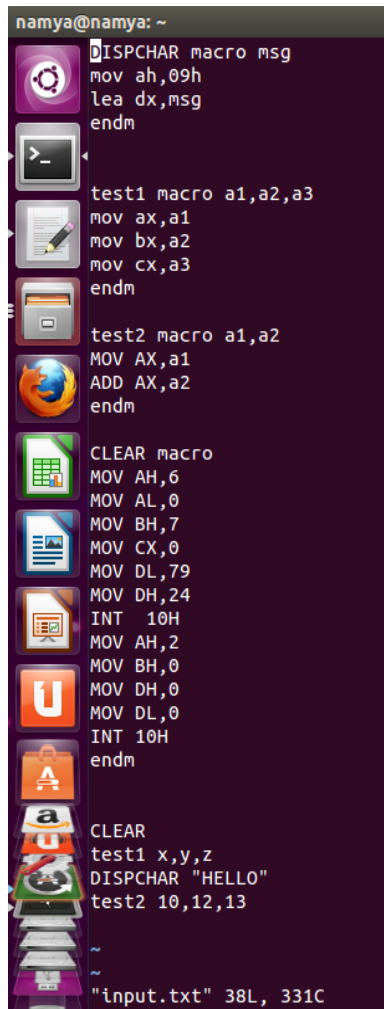
[4] A Language-Independent Macro Processor, W. M. Waite (U. of Colorado); Commun. ACM, vol. 10, pp. 433-440, July 1967.

[5] IEEE Transactions on Computer- Aided Design, 9(9):925-937, Sept. 1990., R. E. Griswold.

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

CHAPTER 7

APPENDIX-A



```

namya@namya: ~
DISPCHAR macro msg
mov ah,09h
lea dx,msg
endm

test1 macro a1,a2,a3
mov ax,a1
mov bx,a2
mov cx,a3
endm

test2 macro a1,a2
MOV AX,a1
ADD AX,a2
endm

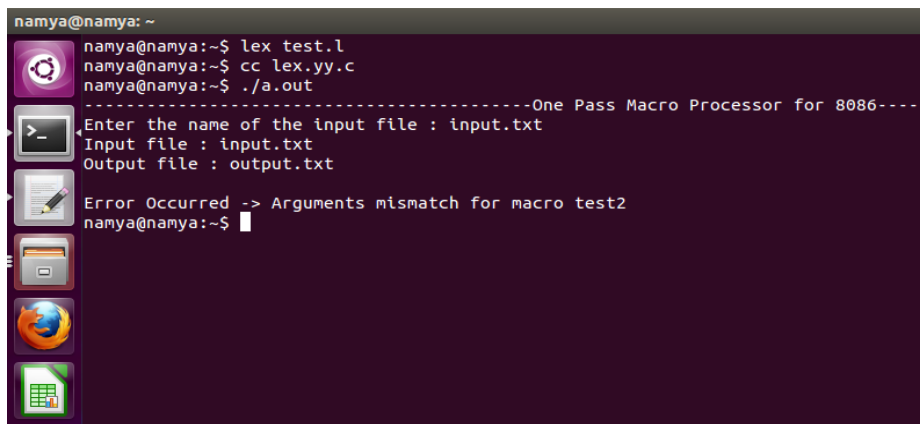
CLEAR macro
MOV AH,6
MOV AL,0
MOV BH,7
MOV CX,0
MOV DL,79
MOV DH,24
INT 10H
MOV AH,2
MOV BH,0
MOV DH,0
MOV DL,0
INT 10H
endm

CLEAR
test1 x,y,z
DISPCHAR "HELLO"
test2 10,12,13

"input.txt" 38L, 331C

```

Figure 4a - Input file with too few arguments for test2 macro



```

namya@namya: ~
namya@namya:~$ lex test.l
namya@namya:~$ cc lex.yy.c
namya@namya:~$ ./a.out
-----One Pass Macro Processor for 8086-----
Enter the name of the input file : input.txt
Input file : input.txt
Output file : output.txt

Error Occurred -> Arguments mismatch for macro test2
namya@namya:~$

```

Figure 4b – Output on running the program for input file in 4a (Error message)

```

namya@namya: ~
namya@namya:~$ ./a.out
-----One Pass Macro Processor-----
Enter the name of the input file : input2.txt
Input file : input2.txt
Output file : output.txt
-----NAMTAB-----
Index      Macro_name
1          DISPCHAR
2          test1
3          EXITTO
4          test2
5          CLEAR
-----ARGTAB-----
Index      ARG_LIST
1          msg
2          a1,a2,a3
3
4          a1,a2
5
-----DEFTAB-----
Index      Macro_definition
1          mov ah,09h
           lea dx,msg
2          mov ax,a1
           mov bx,a2
           mov cx,a3
3          MOV AH,4CH
           INT 21H
4          MOV AX,a1
           ADD AX,a2
5          MOV AH,6
           MOV AL,0
           MOV BH,7
           MOV CX,0
           MOV DL,79
           MOV DH,24
           INT 10H

```

Figure 5a – Output on running program with input file in 5b (NAMTAB, ARGTAB, DEFTAB Displayed)

```

namya@namya: ~
DISPCHAR macro msg
mov ah,09h
lea dx,msg
endm

test1 macro a1,a2,a3
mov ax,a1
mov bx,a2
mov cx,a3
endm

EXITTO macro
MOV AH,4CH
INT 21H
endm

test2 macro a1,a2
MOV AX,a1
ADD AX,a2
endm

CLEAR macro
MOV AH,6
MOV AL,0
MOV BH,7
MOV CX,0
MOV DL,79
MOV DH,24
INT 10H
MOV AH,2
MOV BH,0
MOV DH,0
MOV DL,0
INT 10H
endm

.model small
.data
.code
"input2.txt" 49L, 424C

```

```

namya@namya: ~
.model small
.data
.code
MOV AX,@data
MOV DS,AX
MOV AH,6
MOV AL,0
MOV BH,7
MOV CX,0
MOV DL,79
MOV DH,24
INT 10H
MOV AH,2
MOV BH,0
MOV DH,0
MOV DL,0
INT 10H
mov ax,x
mov bx,y
mov cx,z
mov ah,09h
lea dx,"HELLO"
MOV AX,10
ADD AX,12
MOV AH,4CH
INT 21H
END
~
~
~
~

```

Figure 5b – Input file with macro definitions Figure 5c – Output file (No macro definitions)

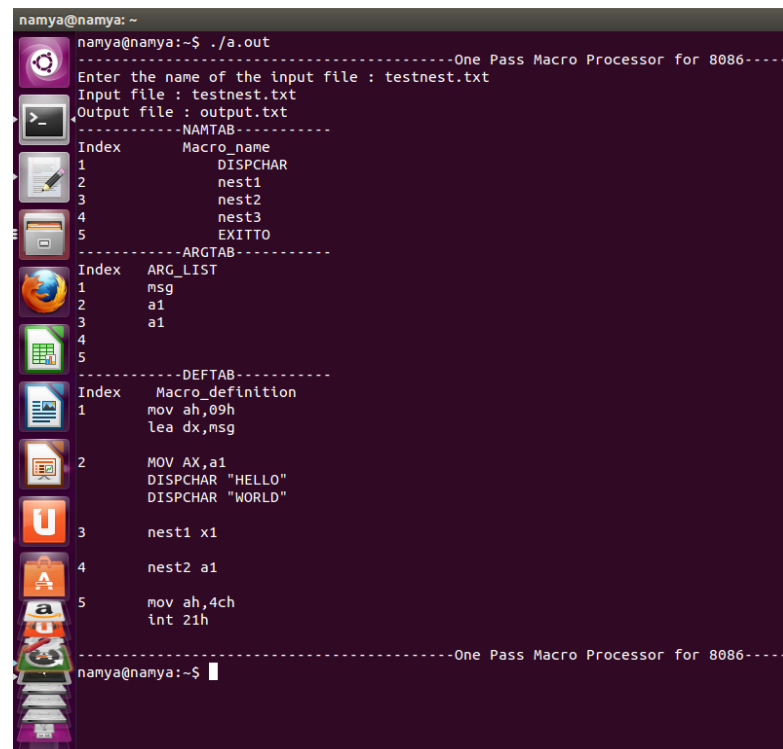


Figure 6a – Output on running program with input file containing nested macro definitions in 6b (NAMTAB, ARG TAB, DEFTAB Displayed)

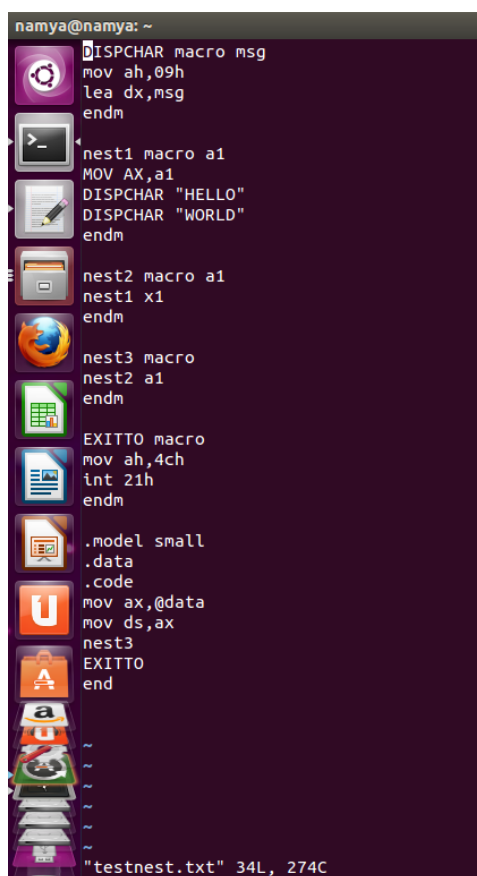


Figure 6b – Input file with macro definitions

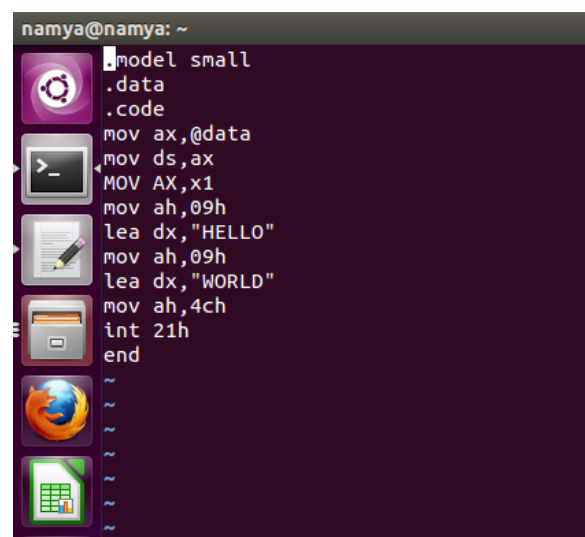


Figure 6c – Output file (No macro definitions)