International Journal of Computer Science
and Engineering (IJCSE)
ISSN(P): 2278-9960; ISSN(E): 2278-9979
Vol. 4, Issue 5, Aug - Sep 2015, 1-6
© IASET

International Academy of Science, Engineering and Technology
IASET
Connecting Researchers; Nurturing Innovations

IMPLEMENATION OF MACRO PROCESSOR USING FILE OPERATIONS

NIRAV SHAH[1] & SAUMYA LAHERA[2]

[1]Department of Computer Engineering, Rajiv Gandhi Institute of Technology, Mumbai, Maharashtra, India

[2]Department of Computer Engineering, Shah & Anchor Kutchhi Engineering College, Mumbai, Maharashtra, India

ABSTRACT

The design objective for this macro processor is to be as powerful as possible and yet remain simple to use and implement. It was developed primarily to manipulate computer programs where the processor takes a symbol table plus a program template containing macros and produces a specific program. This approach is applied to the macro processor itself. The macro processor template may be run through the portable macro processor to produce a version tailored to the local computing environment. Example: In particular, it is easy to produce a IBM 360 version of the macro processor. We consider macro processors to be a particular kind of translator writing system rather than simply a mechanism for performing textual replacement.

KEYWORDS: Macro, Macro Expansion

INTRODUCTION

Assembly language programmer often finds it necessary to repeat some block of code many times in course of a program the block may consist of code to save or exchanges set of registers. In such situations programmer will find macroinstructions useful. Macroinstructions (macros) are single line abbreviations for group of instructions. In employing a block of code, the programmer essentially defines a single "instruction" to represent a block of code. For every occurrence of this one-line macroinstruction in program, macro-processing assembler will substitute entire block.

A macroinstruction is simply a notational convenience for the programmer. A macro consists of a name, a set of parameters and a body of code. Macroinstructions are usually considered as extension of basic assembler language, and macro processor is viewed as extension of basic assembler algorithm. Example macro processors of higher level languages-PL/I, C, Ada and C++. Macro can be defined as following.

```
MACRO
macro name
[arguments]
-----
sequeneve to be abbreviated
 -----
 MEND
```

Figure 1: Macroinstruction Definition

MACRO is pseudo op and it is at first line of definition and identified following line as macroinstruction name can have optional arguments. Following the name line is the sequence of instructions being abbreviated-the instruction you want macro processor to replace when you call macro. The definition is terminated with MEND pseudo -op. Figure 2 shows the example of a macro.

```
macro
add
add1 r1,5
add1 r2,5
add1 r3,5
mend
```

Figure 2: Example of Macro

The macro processor replaces each macroinstruction with the corresponding group of source language statements. This is called macro expansion or expanding the macros. Note that macro definition itself does not appear in expanded source code. The definition is saved but macro processor. And its is expanded at each macro call in source program.

There are two kinds of macro expansion:

- Lexical expansion
- Semantic expansion

Lexical Expansion implies replacement of a character string by another character string during program generation.

Semantic Expansion implies generation of instructions tailored to the requirements of a specific usage. We have used to lexical expansion in our implementation

Variation in Macro:

- Macro can gave arguments
- Conditional macro expansion
- Macro calls within macros
- Macro instructions defining macros

As this is the topic to discuss, we do not elaborate it, however reader can read about it more here [3].

DESIGN

Macro processor can be implemented in 1 or 2 pass. 2 pass is more generally used and we also have used that in our implementation. So we will discuss here 2-pass algorithm for macro processor. There are 4 basic tasks that any macro processor must perform.

- Recognize macro definitions: Macro processor recognizes macro definitions identified by MACRO and MEND pseudo-op. This task can be complicated if we are using nested macro.
- Save the definitions: processor must store macroinstruction definition, which it will need for expanding macro calls.
- Recognize calls: the processor must recognize macro calls.
- Expand calls and substitute arguments: the processor must substitute for dummy arguments with corresponding arguments in macro call. Note that having argument in macro is optional.

Now after defining basic task we must decide which all tables or database we will use in macro processor implementation. Over the two passes we can use 3 tables.

- **Table:** macro definition table (MDT)-- store body of macro
- **Table:** macro name table (MNT)-used to store names of defined macros.
- **Table:** argument List Array (ALA) -to store arguments and substitute later with actual ones. In our implementation we are not using arguments so we will discuss more about 1 and 2 and their roles in both pass. Now lets see what both pass do.

Pass 1: **MACRO DEFINITION:** in thus program test each input line. If macro pseudo-op entire macro definitions that flows is saved in next available location MDT. The name is entered in MNT along with pointer that points to first location of MDT entry of definition. Process is repeated until END pseudo is found means end of program.

Pass 2: MACRO CALLS AND EXPANSION: scanning the program, when call is found, call processor sets a pointer(MDTP) for corresponding definition stored in MDT. Starting entry is obtained from MNT table .as we are not using arguments it will directly take mdt index that we got from mnt table and starts to replace call by set of instruction from MDT table until MEND pseudo-op is reached i.e. Reading MEND line in MDT terminated expansion of macro.

This is how we can get expanded program after replacing each macro call.

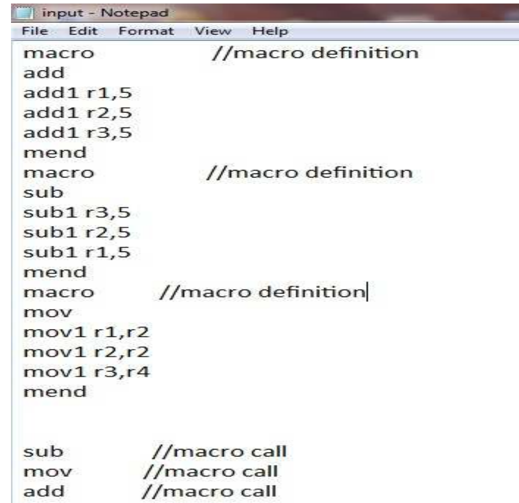
When END is reached the expanded program is given to assembler. Flowchart shows working of both pass for better understanding.

IMPLEMENTATION

We have developed macro processor in C language. Entire logic is based on string matching and we are using file operations [5]. Our program takes file named "input.txt" as input although it can be changed via program. Program outputs 3 files, they are:

- MDT.txt=in pass 1 it is created and all macro definition present in input.txt. While in pass 2 it is used to expand macro calls.
- MNT.txt=in pass 1 it is created and stores macro and index field that has value as starting index of that macro in MDT. In pass 2, when call is made processor checks MNT.txt and takes MDT index value and goes to MDT.txt and expands call until MEND is found.
- OUT.txt=this file contain output.

Screenshots are provided for better understanding and shows our implementation part.

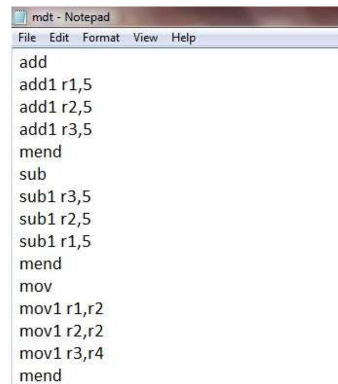


```
input - Notepad
File Edit Format View Help
macro            //macro definition
add
add1 r1,5
add1 r2,5
add1 r3,5
mend
macro            //macro definition
sub
sub1 r3,5
sub1 r2,5
sub1 r1,5
mend
macro        //macro definition
mov
mov1 r1,r2
mov1 r2,r2
mov1 r3,r4
mend


sub          //macro call
mov         //macro call
add         //macro call
```

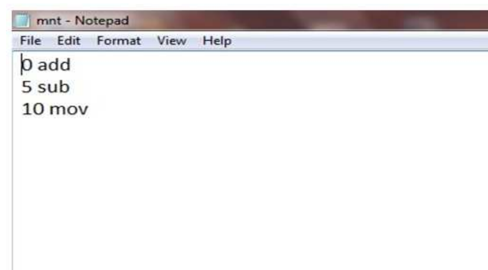
Figure 3: Input File

This image shows what will be the input for program, comments must be removed before running program, they are just provided for better understanding.



```
mdt - Notepad
File Edit Format View Help
add
add1 r1,5
add1 r2,5
add1 r3,5
mend
sub
sub1 r3,5
sub1 r2,5
sub1 r1,5
mend
mov
mov1 r1,r2
mov1 r2,r2
mov1 r3,r4
mend
```

Figure 4: MDT File



```
mnt - Notepad
File Edit Format View Help
0 add
5 sub
10 mov
```

Figure 5: MNT File

0,5,10 in MNT.txt file are respective starting index of that macro in MDT while add, sub, mov are macro created by programmer. So as example when sub is called, program goes to MNT.txt and searches sub in it, and when found it takes its MDT index i.e. found at 2 record, so it extracts 5 from MNT and goes to now MDT file's line no 5, and starts to expand sub call until mend is found in MDT.txt. We hope its clear now.

The output file would be like this.

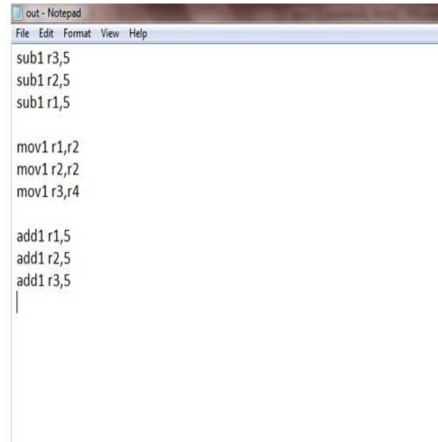


Figure 6: Output File

ADVANTAGES

By defining the appropriate macroinstruction, an assembly language programmer can tailor his own higher-level facility in convenient manner, at no cost in control over structure of program.

He can achieve the conciseness and ease in coding of high-level languages without losing advantage of assembly language programming.

No need to care for different facilities for different languages.

Once developed (even cost more), can be used for any language, and also for a long time.

CONCLUSIONS

We conclude that macros helps programmer to make shorter code and that codes are easy to understand and debug. Also gives programmer flexibility to change only macro definition and changes will be automatically reflected wherever macro is called.

FUTURE SCOPE

In future we might see same macros can be expanded in different languages leaving complexity for designers.

We showed you how could you implement macro without arguments, so macro with arguments can be implemented similarly.

REFERENCES

1. Donovan, System Programming, Tata McGraw-Hill Education,1991
2. https://www.classle.net/book/macros-and-system-software
3. http://web.thu.edu.tw/ctyang/www/files/sp_chap4.pdf
4. http://elearning.vtu.ac.in/P7/enotes/CS51/MacroProcessor-SS.pdf
5. http://www.studytonight.com/c/file-input-output.php