TRABALHO PRÁTICO 3:
Expansor de Macros

Pedro Lopes Miranda Junior

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

plmj@dcc.ufmg.br

Resumo. *Este documento tem por objetivo descrever um expansor de macros que será usado durante a diciplina de Software Básico a fim de realizar experimentos em uma máquina virtual com base nos conceitos teóricos vistos.*

1. Introdução

Ao longo do curso serão estudados diversos conceitos sobre liguagem de máquina. Perante isso, serão dados diversos trabalhos que testarão alguns dos inúmeros conceitos abordados.

Após a implementação de uma máquina virtual e de um montador é necessário um expansor de macros que expanda e substitua as chamadas de macros pelo código inserido entre as pseudo-instruções BEGINMACRO e ENDMACRO. Uma macro deve receber um nome que é usado para identifica-la.

Para isso é necessário ler o arquivo de entrada, que contem as macros, duas vezes. Na primeira há a construção da tabela de macros. Na segunda a substituição da macro pelo trecho de código é feita.

Abaixo será apresentada a solução proposta bem como os testes e decisões de implementação.

2. Solução Proposta

Para solucionar o problema, foi criada uma estrutura de dados chamada macro_t a tabela de macros. Esta tabela guarda o nome, o código e o parametro da macro, caso ela apresente.

O programa expansor foi criado em duas funções que percorrem o arquivo de entrada com o código fonte duas vezes. As funções estão explicadas abaixo.

2.1. Algoritmos

Foram implementadas algumas funções que auxiliam no funcionamento do expansor, portanto todas as funções operam sobre o tipo macro_t.

2.1.1. Funções

1: Funções do expansor
firs_tstep; *Faz a primeira passagem sobre o código fonte criando a tabela de macros.* **second_step;** *Faz a segunda passagem sobre o código fonte substituindo as chamdas de macro pelo código inserido na mesma. Além disso faz o tratamento de macros com parâmetro e de labels internas a macro.*

3. Implementação

Na implementação do problema proposto foram tomadas várias decisões, dentre elas criar um tipo de dados para o expansor, dividir o código em funções de modo que na função principal fique o menor conteúdo possível ajudando no encapsulamento do código e em futuras manutenções e melhorias, além do fato de usar a linguagem C++ e seus recursos extras em relação a linguagem C, utilizada nos trabalhos anteriores.

Foi decidido criar um tipo de dados para a macro pois esse tipo de disposição facilita a adição futura de demais estruturas caso necessário. Além disso o acesso ao TAD é feito de maneira melhor estruturada e o encapsulamento é melhor feito.

A divisão do código em funções ajuda no encapsulamento do TAD, e na melhor modularização do mesmo.

3.1. Código

O código foi dividido em arquivos *.cpp* e *.h* que estão listados abaixo

3.1.1. Arquivos .cpp

- **main.cpp:** Contém a função principal do expansor;
- **expansor.cpp:** Contém as funções do tipo macro;

3.1.2. Arquivos .h

- **expansor.h:** Contém as definições das funções do tipo macro

3.2. Compilação

O programa deve ser compilado através de um makefile, chamando *expansor* ou através do compilador GCC chamando:

```
gpp -Wall main.cpp expansor.cpp -o expansor
```

3.3. Execução

Para a execução do programa deverão ser recebidos,impreterívelmente, o nome do arquivo com o código assembly a ser executado e o arquivo de saída.

O comando para execução do programa é da forma:

```
./expansor <entrada> <saída>
```

3.3.1. Formato da entrada

O arquivo de entrada citado deverá ser um programa em linguagem assembly, onde cada linha conterá uma instrução, podendo ou não ter uma macro.

3.3.2. Formato da saída

O programa imprimirá no arquivo de saída o código também no assembly característico, porém com as macros expandidas.

4. Avaliação Experimental

O programa foi executado e testado numa máquina rodando o sistema baseado em debian Linux Mint. A máquina em questão tem uma memória de 8GB e um processador Core I7 de 2.2GHz.

Foram feitos vários testes de execução do programa,onde todas as funções testadas funcionaram de maneira rápida e sem erro. Abaixo apresentamos imagens da execução do programa e sua saída.

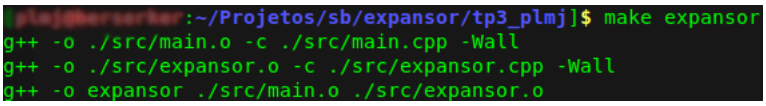

Figura 1. Comando Make

Figura 2. Comando de execução do programa

```
MULTIPLICA: BEGINMACRO RESULT
  LOAD RC V0
  COMP RA RC
  BEZ FIM
  COMP RB RC
  BEZ MULTFIM
  MULT: ADD RC RA
    LOAD RD V1
    SUB RB RD
    LOAD RD V0
    COMP RB RD
    BNZ MULT
  MULTFIM: STORE RESULT RC
ENDMACRO

READ BASE
READ EXPOENTE
EXPONENCIACAO: LOAD RA BASE
  LOAD RB EXPOENTE
  LOAD RC V0
  LOAD RD V1
  COMP RB RC
  MOVE RC RD
  BEZ EXPFIM
  LOAD RC V0
  COMP RA RC
  BEZ EXPFIM
  SUB RB RD
  MOVE RC RA
  EXP: PUSH RB ;SALVA RB NA PILHA
    MOVE RB RC ;RB = RC
    MULTIPLICA RMULT
    LOAD RMULT RC; RC = RMULT
    POP RB; PEGA RB NA PILHA
    LOAD RD V1
    SUB RB RD ;RB = RB-1
    LOAD RD V0
    COMP RB RD
    BNZ EXP
EXPFIM: STORE POTENCIA RC
WRITE POTENCIA
END
```

```
READ BASE
READ EXPOENTE
EXPONENCIACAO: LOAD RA BASE
  LOAD RB EXPOENTE
  LOAD RC V0
  LOAD RD V1
  COMP RB RC
  MOVE RC RD
  BEZ EXPFIM
  LOAD RC V0
  COMP RA RC
  BEZ EXPFIM
  SUB RB RD
  MOVE RC RA
  EXP: PUSH RB ;SALVA RB NA PILHA
    MOVE RB RC ;RB = RC
  LOAD RC V0
  COMP RA RC
  BEZ FIM
  COMP RB RC
  BEZ MULTFIM1
  MULT1: ADD RC RA
    LOAD RD V1
    SUB RB RD
    LOAD RD V0
    COMP RB RD
    BNZ MULT1
  MULTFIM1: STORE  RC
    LOAD RMULT RC; RC = RMULT
    POP RB; PEGA RB NA PILHA
    LOAD RD V1
    SUB RB RD ;RB = RB-1
    LOAD RD V0
    COMP RB RD
    BNZ EXP
EXPFIM: STORE POTENCIA RC
WRITE POTENCIA
END
V0: WORD 0; CONSTANTE 0
V1: WORD 1; CONSTANTE 1
BASE: WORD 0; BASE
EXPOENTE: WORD 0
```

Figura 3. Saída exemplos de programas normal e já expandido

5. Conclusão

O trabalho correu sem grandes problemas, sendo a parte mais difícil a criação dos progrmas de testes, pois montar alguns daqueles códigos em asembly é extremamente difícil.

O programa atendeu a diversos valores de entrada e creio que a solução proposta atenda ao especificado