Trabalho Prático Software Básico - Montador

Rafael Rubbioli Ferreira - Ana Luisa Rodrigues - Luiz Otavio R. V.

October 10, 2016

Introdução

Este trabalho tem o objetivo de implementar um montador de dois passos para a máquina Wombat2, como foi visto em sala de aula. A partir de um arquivo texto como entrada .a, a função do montador será transformar esse programa escrito na linguagem Assembly para um programa em linguagem de máquina. Foi escolhido o tipo .mif.

Para isso, ele deve ser capaz de interpretar o conjunto de 27 instruções fornecidas na documentação, de forma eficiente. O trabalho foi implementado na linguagem C++.

Todo o processo de desenvolvimento pode ser conferido no repositório no GitHub:

https://github.com/Luiz0tavio/wombat2

(alterado de privado para público na data de entrega) na guia "commits"

Desenvolvimento

O montador será capaz de interpretar estruções com o seguinte formato:

_[rótulo:] operador [operando(s)] [;comentário]

A instrução deve ter um operador e a quantidade de operandos depende da instrução, podendo ter nenhum, um ou dois.

A instrução pode conter um rótulo (label) que deve ser inicializado com com um underscore "_" que será responsável por identificar a posição de memória da instrução em questão.

Os comentários são opcionais e devem começar com ";". A seguir um exemplo de instrução:

_loop: add R0 R1 ;teste

Para o tratamento de referências antecipadas, o montador deve realizar duas passadas pelo código, podendo assim realizar a tradução do programa.

Na primeira passada, são tratados dois eventos:

1. O primeiro evento a ser tratado são os desvios (mapeamento dos labels). Quando ocorre um desvio no código o montador deve transformar os labels dos desvios em endereços de PC. Para resolver isso, é criada uma tabela para mapear os desvios com seus devidos valores de PC.

2. O segundo evento a ser tratado são as alocações de memória no final do código. Para resolver isso, foi criada mais uma tabela. Essa tabela irá alocar na memória um número de bytes e dar valor à eles. Dessa forma, quando a segunda passada começar, os valores de todos os símbolos já serão conhecidos, não restando nenhuma referência antecipada e cada declaração poderá ser lida, montada e produzida com seu respectivo valor e endereço de memória.

A pseudo-instrução .data tem a função de reservar uma região da memória da máquina. Seu formato será:

label: .data num_bytes valor_inicial

portanto, ela alocará uma região de tamanho num_bytes com um valor_inicial. Essa região poderá ser identificada ao longo do código pelo rótulo label, podendo ser acessada a qualquer momento durante sua execução. Abaixo temos um exemplo da utilização dessa pseudo-instrução:

wombat: .data 2 0

Neste caso, o nome da instrução é "wombat", tem um tamanho de 2 bytes e começa com valor inicial 0.

Na implemetação deste trabalho prático, as variáveis são alocadas na posição mais baixa da memória (0 em diante). A cada definição de variável encontrada é inserida uma entrada na tabela de variáveis com o nome e a posição referente à essa variável. A seguir é mostrado um programa escrito em linguagem de montagem e a tabela de variáveis criada:

```
...
wombat .data 2 0 ;pseudo-instrucao de 2 bytes com valor inicial 0
...
move R0 R1
wombat2 .data 3 1 ;pseudo-instrucao de 3 bytes com valor inicial 1
_loop: add R0 R1
...
exit
```

Variável	Endereço
wombat	0
wombat2	1

Com as duas tabelas mapeadas, iniciamos a segunda passada.

Percorremos então novamente o arquivo de entrada .a, e, para cada instrução, escrevemos em um arquivo temporário(<name.a>.mif.temp) seu respectivo código binário. Cada instrução tem 16 bits, escritos em duas linhas de 8.

Os 5 primeiros bits são sempre o opcode da instrução. O formato dos outros 11 irá depender do opcode.

Após a inserção de todos os 16 bits de cada instrução, nosso arquivo temporário está pronto.

Abrimos um novo arquivo (definitivo <name.a>.mif), adiciomos o 'header' do tipo .mif, o qual, contém informações do tipo tamanho da instrução, formato do índice, etc.

Colocamos então um índice em hexadecimal (de 00 até FF) crescente no começo da linha e seguido de uma parte de instrução (8bits) copiada do arquivo temporário até seu EOF.

Então o arquivo temporário é deletado e temos o arquivo final '.mif'

Tomada de decisão

Escolhemos implementar da maneira mais próxima às escolhas apresentadas no output do CPUSim:

- Todas as instruções alocadas de maneira crescente na memória
- Posição de memória escolhida para alocar os labels, são os próprios pc's da instruções (em binário)
- As .data são declaradas, no teste apresentado e nos nossos, após as instruções, e então, alocadas logo após elas

Código

Como dito na introdução, foi desenvolvido em C++11.

Para compilá-lo, basta executar *make* na pasta assembler que o arquivo *montador* será gerado (para limpar o executável, utilize make clear)

Para executar, utilize o comando

$ ./montador ../tst/W2-X.a

onde X = 2, 3 ou 4 (cada teste é descrito na próxima seção)

Será então gerado o arquivo de mesmo nome *.mif* pronto para ser executado no *Wombat2*

Testes

Desenvolvemos os seguintes testes:

1. *Fibonacci Iterativo (W2-2.a)* :

   Recebe um número 'n' como entrada e retorna o F(n) (fibonacci de n).

   Ex: F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3 ...

   ```
   EXECUTING...
   Enter Inputs, the first of which must be an Integer: 6
   Output:  8
   EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
   ```

   F(6) = 8

   ```
   EXECUTING...
   Enter Inputs, the first of which must be an Integer: 7
   Output:  13
   EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
   ```

   F(7) = F(6) + F(5) = 8 + 5 = 13

2. *Cálculo de RSG (W2-3.a) :*

   O RSG é dado pela média ponderada:

   $\frac{\sum(ValorConceito'n' x CréditosConceito'n')}{\sum CréditosConceito'n'}$ onde 'n' é uma disciplina.

   O programa pedirá duas entradas:

   (a) valor do conceito

   (b) créditos da disciplina

Então, pedirá novamente o *valor do conceito* da próx disciplina.

Irá parar quando receber um número negativo no *valor do conceito*

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Enter Inputs, the first of which must be an Integer: 5
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 4
Enter Inputs, the first of which must be an Integer: -1
Output:  2
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
```

Nesse exemplo, foram inseridas 2 disciplinas:

(a) VC = 2 e CD = 5

(b) VC = 3 e CD = 4

Logo, $\frac{(2*5)+(3*4)}{5+4} = 2.44 \sim 2$

3. *Cálculo de Potência utilizando Stack Pointer (W2-4.a):*

   Recebe M e N e calcula M^N utilzando operações de Stack Pointer:

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 9
Enter Inputs, the first of which must be an Integer: 2
Output:  81
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
```

   $9^2 = 81$

Conclusão

Desenvolver o montador para o Wombat2 auxiliou na compreensão e fixação das etapas de montagem de 2 etapas (conteúdo visto no capítulo 7). A maior dificuldade encontrada foi na elaboração de testes, visto que, é o primeiro contato com a linguagem do Wombat2 e trouxe várias dúvidas. Após sanadas, o desenvolvimento fluiu bem.