

Sistemas Embarcados - Trabalho Prático 2

Montador

Arthur Gomes de Lima (2017023650)

`arthur.lima@dcc.ufmg.br`

João Gabriel de Oliveira Bicalho (2017015134)

`jpgobi@ufmg.br`

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

2021

1. Introdução

Esta documentação descreve o desenvolvimento de um montador, um programa que traduz a linguagem de montagem em um código legível por máquina. O montador e a linguagem de montagem são parte dos níveis mais baixos de abstração entre o código de máquina e as linguagens de programação que utilizamos atualmente. Por isso, os montadores e linguagens de *Assembly* são desenvolvidos para uma arquitetura específica de CPUs.

Assim, neste trabalho, foi implementado um programa montador para a arquitetura *Swombat R3.0*, uma arquitetura simples que permite explorar os conceitos básicos da montagem de programas e que possui um simulador pronto para que os programas gerados sejam testados - o *CPU Sim*.

2. Entrada e saída

O montador desenvolvido recebe um arquivo **.a** como entrada na linha de comando e gera um arquivo **.mif** (Memory Initialization File) como saída.

O arquivo de entrada contém um programa escrito na linguagem *assembly* da CPU.

No arquivo de saída, além de um cabeçalho definido na especificação do formato, cada linha contém um endereço ou intervalo de endereços de memória e o seu conteúdo (1 byte).

Após a compilação com **make**, deve-se executar o montador com o comando:
`./montador arquivo_entrada arquivo_saida`

Caso o arquivo de saída não seja informado, a saída padrão será usada em seu lugar.

3. Considerações gerais de implementação

- O montador foi desenvolvido em C++11
- Para facilitar a tradução do código *assembly*, o conjunto de instruções do *Swombat R3.0* foi internamente dividido em 6 categorias de *opcode*, baseadas no tipo e na quantidade de operandos.

R = registrador, M = referência de memória, I = imediato, _ = não usado

Nome	Configuração dos 11 bits após opcode
no_operation	-----
register_memory	RRMMMMMMMMM
one_register	-----RR
two_registers	RR-----RR
memory_address	--MMMMMMMMM
register_immediate	RRIIIIIIIII

4. Resumo da implementação

O montador desenvolvido é conhecido tradicionalmente como **montador de duas passadas**, pois o arquivo de entrada será varrido duas vezes para gerar o arquivo de saída. A execução do montador segue este roteiro:

a. Passada 1

O arquivo de entrada é processado linha por linha, gerando duas tabelas: uma tabela de símbolos e uma tabela de dados, ambas implementadas usando **std::map**. A tabela de símbolos contém o nome de todas as *labels* encontradas no código *assembly* e a sua posição na memória (o valor do ILC). Já a tabela de dados registra as *labels* do programa que antecedem a pseudo-instrução **.data**; nesta tabela são gravados o tamanho em bytes da memória que será reservada e valor que será escrito nesse endereço.

b. Alocação da memória (variáveis)

Terminada a leitura do arquivo, o contador do ILC marcará o fim do programa e o início dos endereços de memória livres. Em nossa implementação, o montador utiliza os endereços diretamente após o fim do programa para escrever a memória de dados

Os dados são alocados na memória em ordem alfabética de *labels*, uma consequência do uso do **std::map** para representar a tabela de dados. Feita essa alocação, o programa atualiza a tabela de símbolos com a localização correta das variáveis.

c. Passada 2

É na segunda passada que o montador traduz as instruções e gera o código de máquina. Assim como no primeiro passo, o arquivo é lido completamente e a tradução ocorre linha por linha. A análise de cada linha gera zero ou uma instrução de 16 bits do *Swombat R3.0*, conforme o seguinte processo:

Caso a pseudo-instrução **.data** seja encontrada, a linha é ignorada.

Caso seja encontrado um opcode, ele é lido, determinando assim os 5 primeiros bits da saída e o tipo de instrução a ser gerada. Depois, o tipo da instrução lida determina a sua quantidade de operandos e como a tradução será feita. Sempre que uma *label* for encontrada, a tabela de símbolos é acessada para realizar a substituição dela por sua posição na memória associada. Por meio de uma sequência de *bit-shift* e *bitwise-OR*, é gerado um número de 16 bits contendo a instrução traduzida.

Ao final do processo de tradução, a memória de dados é escrita no arquivo de saída, seguindo a ordem mencionada anteriormente.

d. Geração do arquivo .mif

Salvo cabeçalho e rodapé, a escrita do arquivo de saída pelo montador é feita durante o segundo passo da tradução, escrevendo as instruções conforme elas são traduzidas. Este arquivo contém uma representação dos 128 bytes de memória disponíveis na máquina alvo, sendo que as posições não utilizadas são preenchidas com zeros.

5. Testes

A fim de comprovar o correto funcionamento do programa montador, desenvolvemos alguns programas de teste. O objetivo dos programas é utilizar as diversas instruções da arquitetura *Swombat R3.0* e, por isso, não se preocupam em otimizar o uso de memória. No total foram testadas 17 instruções, incluindo pelo menos uma de cada categoria de opcode definida anteriormente.

Para criar os programas de teste, elaboramos uma pequena **ABI** a ser seguida durante as chamadas de procedimento:

- Todos os parâmetros são passados para a função através da pilha
- A função chamadora é responsável por empilhar os parâmetros e desempilhá-los após o término da função chamada
- O retorno da função é guardado no registrador A0
- Todos os registradores relevantes devem ser salvos pela função chamadora

A exclusividade da passagem de parâmetros pela pilha foi motivada pela pequena quantidade de registradores disponíveis na arquitetura.

a. fibonacci.a

Este programa recebe um número inteiro **n** do usuário e escreve na saída o **n**-ésimo número da sequência de Fibonacci iniciada por 0 (0,1,1,2,3,5,8,13...). Internamente, o programa utiliza instruções de soma e subtração, desvios condicionais e incondicionais, manipulação de variáveis na memória e manipulação simples da pilha.

b. fibr.a

Este programa recebe um número inteiro **n** do usuário e escreve na saída o **n**-ésimo número da sequência de Fibonacci iniciada por 1 (1,1,2,3,5,8,13,21...). Este programa foi implementado de forma recursiva, com o objetivo de testar chamadas de função e utilização mais exigente da pilha.

File Edit Modify Execute Help

Data Unsigned Dec

Name	Width	Data
buffer1	16	13
buffer2	16	8
ir	16	0
mar	12	60
mdr	16	0
pc	12	62
status	3	4
top	12	4

A

Name	Width	Data
A0	16	8
A1	16	0
A2	16	8
A3	16	13

fibonacci X

```

1 ; This program reads one number greater than zero n
2 ; and outputs the n-th Fibonacci number,
3 ; assuming the sequence starts at 0 (0,1,1,2,3,5,8,13,...)
4
5 _Start: read A0
6
7 ;; test if input <= 1.
8 move A0 A2
9 load_c A1 2
10 subtract A2 A1
11 jmpn A2 _Ret0
12
13 ;; test if input == 2.
14 load_c A1 2
15 subtract A1 A0
16 jmpz A1 _Ret1
17
18 ;; Set up the stack with the initial values.
19 load_c A1 0
20 push A1
21
22 load_c A1 1
23 push A1
24
25 ;; Every loop, pop the last two values from the stack...
26 _Loop: pop A2
27 pop A1
28
29 ;; and add them to get the next Fibonacci number.
30 add A1 A2
31
32 ;; Prepare the stack for the next value.
33 ;; This means pushing the oldest number back in the stack
34 ;; then pushing the new one we just calculated.
35 push A2
36 push A1

```

Addr Hex Data Unsigned Dec

Main

Addr	Data
00	24
01	0
02	96
03	2
04	154
05	2
06	52
07	1

Stack

Addr	Data
00	0
01	8
02	0
03	13
04	0
05	0
06	0
07	0

EXECUTING...

Enter Inputs, the first of which must be an Integer: 8

Output: 13

EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]

fibonacci.a: Estado da máquina após execução. Neste programa, a pilha é utilizada para guardar os últimos valores da sequência assim que são calculados.

File Edit Modify Execute Help

Data Unsigned Dec

Name	Width	Data
buffer1	16	21
buffer2	16	13
ir	16	0
mar	12	8
mdr	16	0
pc	12	10
status	3	4
top	12	2

A

Name	Width	Data
A0	16	21
A1	16	13
A2	16	3
A3	16	0

fibra X

```

4
5 _main: read A0
6 push A0 ; parametro de _fib
7 call _fib
8 write A0 ; escreve o retorno de _fib
9 stop
10
11
12 _fib: load_s A0 4 ; A0 = n
13
14 ; se <= 2 retorna 1
15 move A0 A1 ; A1 = n
16 load_c A2 3
17 subtract A1 A2 ; A1 -= 3
18 jmpn A1 _fib_ret1 ; if (A1 < 0) return 1
19
20 load_c A1 1
21 subtract A0 A1
22 push A0 ; parametro de _fib
23 call _fib
24 move A0 A1 ; A1 = fib(n-1)
25
26 pop A0 ; recupera o parametro de fib (n-1)
27 push A1 ; salva A1 (fib(n-1))
28
29 load_c A1 1
30 subtract A0 A1
31 push A0 ; parametro de _fib
32 call _fib ; A0 = fib((n-1)-1)
33
34 pop A1 ; desempilha o parametro
35 pop A1 ; recupera fib(n-1)
36 add A0 A1 ; A0 += A1
37
38 return
39

```

Addr Hex Data Unsigned Dec

Main

Addr	Data
00	24
01	0
02	104
03	0
04	120
05	10
06	32
07	0

Stack

Addr	Data
00	0
01	8
02	0
03	6
04	0
05	13
06	0
07	6

EXECUTING...

Enter Inputs, the first of which must be an Integer: 8

Output: 21

EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]

fibra: Estado da máquina após execução. Na pilha, é possível ver vestígios das chamadas de função e do uso da ABI para passagem de parâmetros. O primeiro inteiro da pilha, por exemplo, é o parâmetro da primeira chamada, seguido do endereço de retorno para a função principal após a chamada inicial para *_fib*.