Sistemas Embarcados - Trabalho Prático 2

Montador

Arthur Gomes de Lima (2017023650)
arthur.lima@dcc.ufmg.br

João Gabriel de Oliveira Bicalho (2017015134)
jgobi@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`	`RRIIIIIIII`

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 S*wombat 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 termino da função chamada
- O retorno da função é guardado no registrador A0
- Todos os registradores relevantes devem ser salvos pela função chamadora

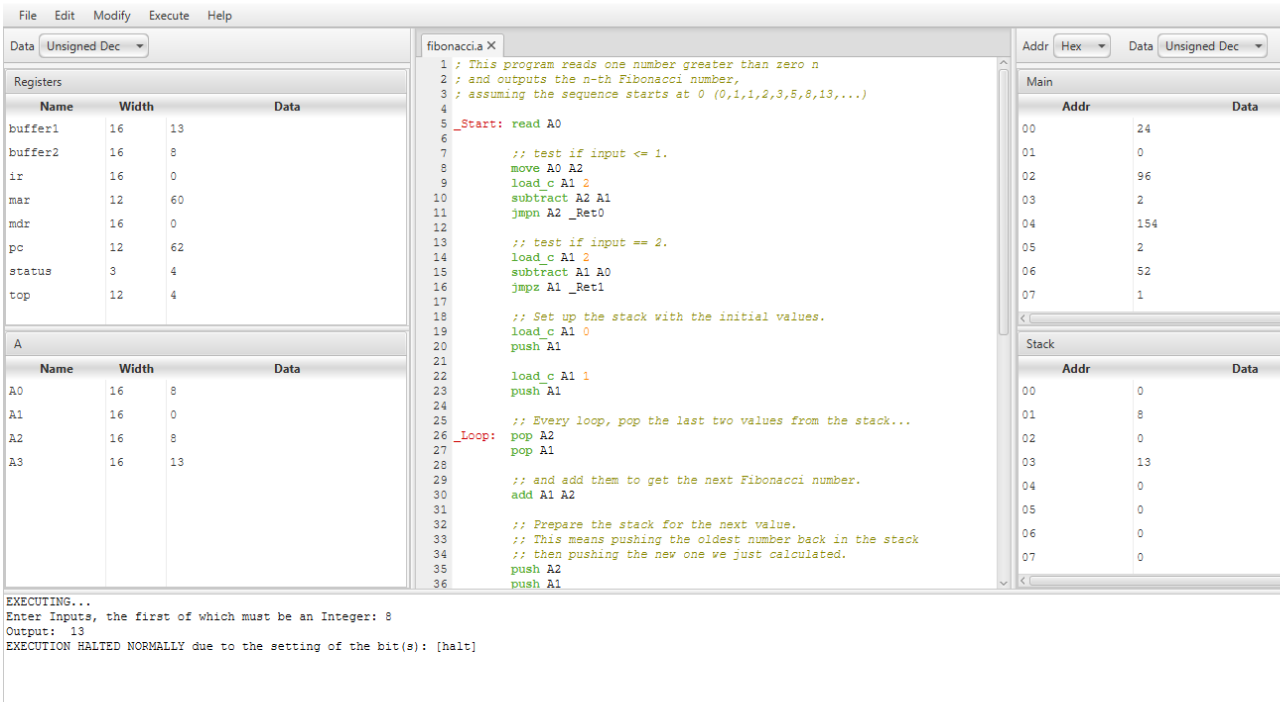
A exclusividade da passagem de parâmetros pela pilha foi motivada pela pequena quantidade de registradores disponíveis na arquitetura.

a. fibonacci.a

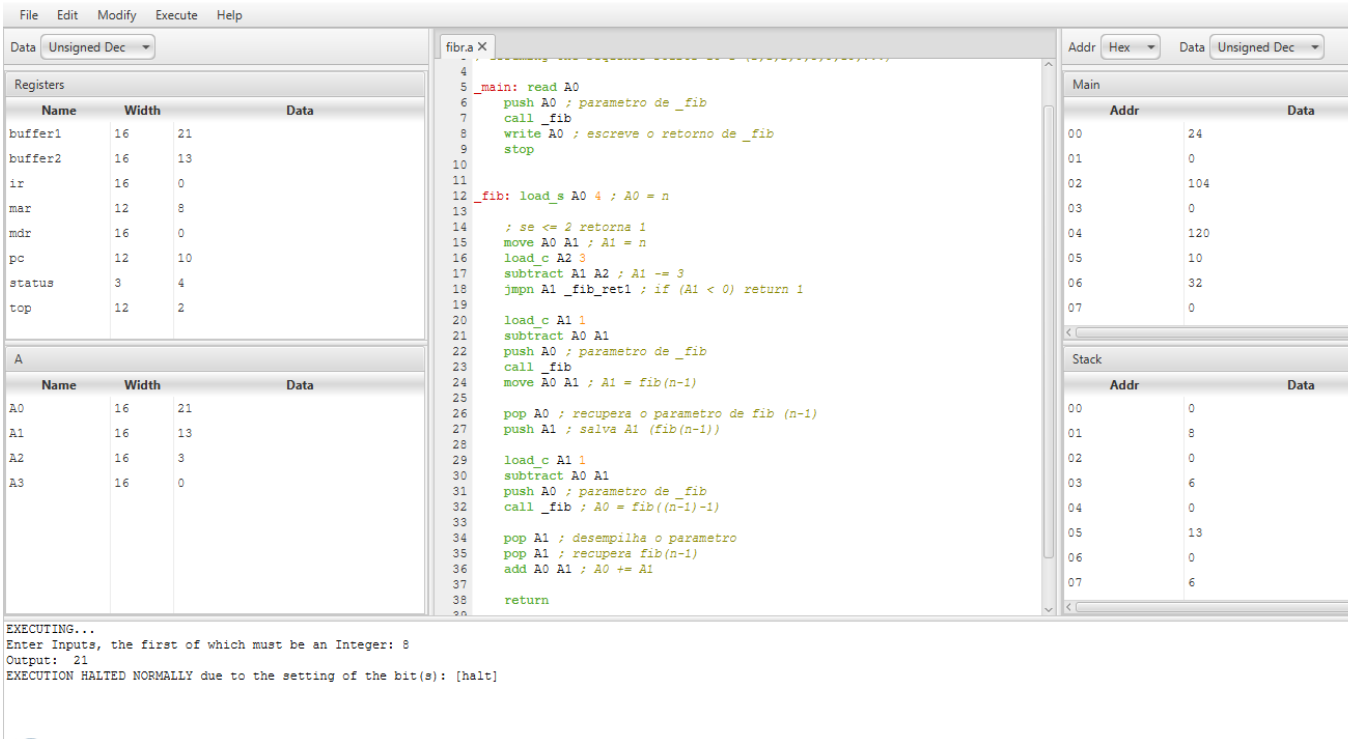
Este programa recebe um número inteiro **n** do usuário e escreve na saída o n-ésimo número da sequência de Fibonacci iniciada por 0 (0,1,1,2,3,5,8,13...). Internamente, o programa utiliza instruções de soma e subtração, desvios condicionais e incondicionais, manipulação de variáveis na memória e manipulação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



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



fibr.a: 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*.