Trabalho Prático 1 - Montador

Fernanda de Oliveira Ramalho Gustavo Henrique Oliveira Costa
Gabriel Henrique Souto Pires Raydan Elias Gaspar

1 Introdução

Nesse trabalho prático foi implementado um montador para a CPU Swombat. Um montador, que também pode ser chamado de assembler, consiste de um programa que cria o código objeto traduzindo as instruções de um programa em assembly para linguagem de máquina, ou seja, código binário que o computador entenda. O tipo de montador desenvolvido foi o montador de dois passos.

No primeiro dos passos, é feita uma varredura pelo programa inteiro em busca de labels definidas, que são armazenadas em uma tabela juntamente da sua posição no código. Essa tabela é usada no segundo passo para resolver o problema de referencia antecipada, que consiste em referenciar uma parte do código que ainda não se sabe onde está.

No passo dois também é usada uma tabela de tipos que contém todas as funções com formato semelhante agrupadas em tipos, o que facilita na hora de traduzi-las para código de máquina, uma vez que funções com mesmo formato são construídas do mesmo jeito, com exceção do opcode. A máquina utilizada contém apenas instruções de 16 bits, sendo elas escritas byte a byte em uma memória de 256 bytes, isso significa que cada instrução ocupa duas posições na memória (PC e PC+1 por exemplo) então as instruções são buscadas incrementando o PC de 2 em 2.

2 Desenvolvimento

O montador foi desenvolvido em C++ usando como saída o formato *.mif* utilizado pelo CPUSim. O formato *.mif* tem um cabeçalho que indica o tamanho da memória, quantos bits tem cada posição de memória, a base dos endereços e a base dos dados na memória. Para o Swombat, foi especificado que a memória tem 256 posições que contém 8 bits cada de dados, os endereços são exibidos em hexadecimal e os dados em binário. No início do programa é chamada a função *escreve_cabecalho_mif()* que imprime esse cabeçalho no arquivo de saída. O cabeçalho é igual para todos os programas.

O montador propriamente dito começa a ser executado com a função *preenche_tabela_tipos(),* que cria uma lista do tipo *Tabela_tipos* que é uma struct composta por uma string com o nome da operação (add, load, etc), e um inteiro que identifica a qual tipo essa instrução pertence. Essa função preenche a lista de tipos levando em consideração o número de registradores/operandos que

cada instrução usa, ou seja, o formato de cada uma. Instruções com o mesmo formato são escritas da mesma forma na memória, com exceção do opcode, então é útil usar uma mesma função para escrever todas as funções de um mesmo tipo. Essa função também não depende de nenhuma entrada, uma vez que todas as instruções do Swombat foram previamente definidas.

A primeira coisa que temos que tratar no código é o problema de referência antecipada, que consiste em chamar um procedimento que ainda não sabemos onde está. Isso é resolvido na função *preenche_lista_labels*() que lê o código todo, linha por linha, em busca de cada label. Ao encontrar uma label, a função guarda o seu nome e a posição em que a label se encontra na lista, que fica disponível para consulta posterior. Optamos aqui por criar uma lista de structs assim como na tabela de tipos, dessa vez do tipo *Label*, sendo que cada struct possui uma string com o nome da label e um inteiro que guarda o endereço da label encontrada.

A passagem dois começa na função *traduz_programa_fonte()* que usa as listas de tipo e de labels criadas anteriormente. Nessa função vamos ler as instruções e traduzi-las para código de máquina. Cada instrução tem um formato específico, então a cada operando lido uma consulta é feita à tabela de tipos para identificar qual o formato da instrução. Essa consulta é feita através da função *busca_tipo()* que recebe como parâmetros o nome do campo lido e a lista de tipos previamente preenchida. Dependendo do tipo da instrução lida, ela é construída de uma maneira diferente, por exemplo, a função *add* usa dois registradores e 5 bits não são utilizados, já a função *return* usa apenas o seu opcode, já que apenas retorna para o endereço contido no topo da pilha, e por esse motivo 11 bits não são utilizados. Após separar cada operando da instrução, ela é escrita na memória, sendo os 8 primeiros bits em *memoria[pc]* e os 8 últimos bits em *memória[pc+1]*. O nome da variável *pc* não é tecnicamente correto uma vez que na verdade estamos referenciando o valor do Intruction Location Counter (ILC), mas usamos esse nome para facilitar a compreensão da estrutura da memória. Existe uma variável chamada ILC nessa função, mas ela é usada apenas para saber em qual posição da memória será escrita a ultima instrução, para que os bytes alocados pela função *.data* sejam alocados a partir dessa posição.

A instrução *.data* é na verdade uma pseudo instrução, e é construída diferente de todas as outras. Não existe *opcode* nessa instrução, na verdade a única informação gravada por *.data* na memória é o valor a ser alocado. Para gravar de forma correta na memória o valor alocado, o número de bytes a ser alocado é lido e guardado numa variável, assim como o valor a ser alocado. O valor lido é então convertido para binário e jogado em uma string. Essa string inicialmente tem o tamanho da memória, ou seja, 256*8 bits. Isso teve de ser feito, uma vez que o tipo de dados *bitset* usado para representar os números binários não pode ser alocado com tamanho dinâmico, então optamos por alocar o tamanho máximo (espero que ninguém tente alocar alguma coisa maior que a memória) e depois ler apenas os bits relevantes. Depois de ter os dados no formato correto a memória é gravada a partir da posição indicada pelo ILC, ou seja, a posição imediatamente depois da ultima instrução. Os bits a serem gravados são lidos da esquerda para a direita em blocos de 8 bits e gravados sequencialmente na memória enquanto o valor de ILC é incrementado.

3 Programas testados

3.1 Programa 1

3.1.1 Descrição

O programa de teste chama-se diversas_instruções.a. Nesse programa são testadas alumas instruções, por exemplo, add, subtract, loadi, storei, move, multiply, clear e loadc. Primeiramente, são recebidos dois valores digitados pelo usuário; esses valores são recebidos pelo instruções loadi. Os dois valores são somados e o resultado é imprimido na tela. Em seguida, o resultado da soma é subtraído do segundo valor digitado pelo usuário e o valor da subtração é imprimido na tela. O resultado da subtração é carregado para o registrador A2 usando a instrução move, é carregado a constante -3 para o registrador A3 e é realizada a multiplicação dos registradores A2 e A3. O resultado da multiplicação é imprimido na tela. Por fim, é carregado para o registrador A4 um dado de 2 bytes inicializado com o valor 3, é limpado o registrador A0, é feita a soma de A0 com A4 e o resultado é imprimido na tela.

3.1.2 Entrada de dados

Para realizar o teste desse programa, foi digitado como entrada os valores 5 e -10.

3.1.3 Saída esperada

Para os valores 5 e -10, a registrador A0 recebe 5 e o registrador A1 recebe -10. Logo a soma será -5. A0 passará a ter o valor -5. A subtração de A0 e A1 dará 5. Esse valor será carregado para A2 onde fará a multiplicação com o registrador A3, que terá o valor -3. A multiplicação dará -15. Por fim, a soma de A0, que teve seu valor limpo, com o registrador A4, que recebeu o valor 3 alocado na memória, será 3.

3.2 Programa 2

3.2.1 Descrição

O seguinte programa chama-se push_pop_teste.a. O objetivo desse programa é fazer o uso da pilha utilizando as funções push e pop. São lidos dois valores de entrada e esses valores são armazenados na pilha. Em seguida, os valores são desempilhados e é feita a divisão desses valores. Imprime-se o resultado da divisão na tela. Se a divisão for menor que zero, o programa chama uma label que encerra o programa; caso contrário, o valor da divisão é invertido e esse novo valor é imprimido na tela. Vale ressaltar que o resultado da divisão é sempre arredondado para baixo.

3.2.2 Entrada de dados

Como nesse programa existem dois tipos de saída possíveis, será fornecida dois tipos de entradas diferentes. A primeira entrada terá os valores 10 e 5. A segunda entrada será com os valores -10 e 5.

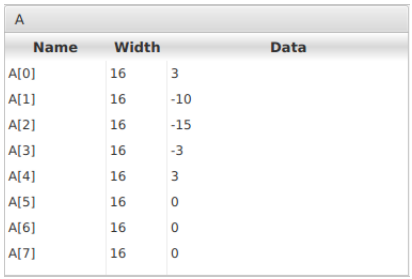
3.2.3 Saída esperada

Para a entrada com os valores 10 e 5, esses números serão empilhados e, posteriormente, serão desempilhados para, enfim, ser feita a divisão. Como o resultado da divisão é maior que zero, o valor da divisão, que é 2, será invertido e esse novo valor será impresso na tela. Logo, teremos dois valores impressos na tela, nesse caso 2 e -2.

Para a entrada com os valores -10 e 5, esses número serão empilhados e depois, desempilhados para ser feita a divisão. O resultado da divisão é impresso na tela. Como o resultado é menor que zero, nesse caso -2, o programa chamará a label que encerrará a execução do programa. Logo teremos apenas um número impresso na tela, nesse caso, o número -2.

4 Testes e simulações

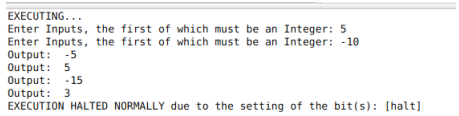
As duas figuras a seguir estão relacionadas com a execução do programa diversas_instruções.a. A primeira figura mostra os valores em cada registrador usado no programa e a segunda figura mostra a saída para entrada 5 e -10.



A

Name	Width	Data
A[0]	16	3
A[1]	16	-10
A[2]	16	-15
A[3]	16	-3
A[4]	16	3
A[5]	16	0
A[6]	16	0
A[7]	16	0

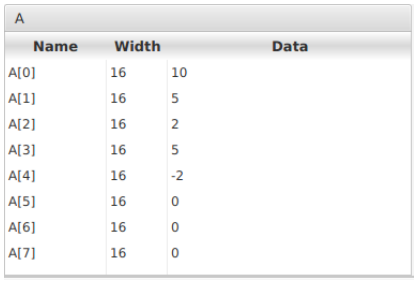
Figure 1: Registradores diversas_instruções.a



```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 5
Enter Inputs, the first of which must be an Integer: -10
Output:  -5
Output:  5
Output:  -15
Output:  3
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
```

Figure 2: Saída para diversas_instruções.a

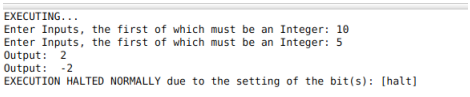
As próximas figuras mostrarão a execução do programa push_pop_teste.a. A primeira figura mostra o conteúdo dos registradores usados no programa e a segunda figura mostra o resultado para a entrada 10 e 5.



A

Name	Width	Data
A[0]	16	10
A[1]	16	5
A[2]	16	2
A[3]	16	5
A[4]	16	-2
A[5]	16	0
A[6]	16	0
A[7]	16	0

Figure 3: Registradores push_pop_teste.a



```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 10
Enter Inputs, the first of which must be an Integer: 5
Output:  2
Output:  -2
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
```

Figure 4: Saída push_pop_teste.a

A seguir, teremos mais duas figuras para mostrar o estado dos registradores e a saída para a entrada -10 e 5.

Name	Width	Data
A[0]	16	-10
A[1]	16	5
A[2]	16	-2
A[3]	16	5
A[4]	16	0
A[5]	16	0
A[6]	16	0
A[7]	16	0

Figure 5: Registradores push_pop_teste.a

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: -10
Enter Inputs, the first of which must be an Integer: 5
Output:  -2
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt]
```

Figure 6: Saída push_pop_teste.a

5 Conclusão

Em geral o desenvolvimento do montador não foi muito difícil, o que mais gerou dúvidas foi o formato dos programas de entrada. A escolha por C++ ao invés de C facilitou muito o desenvolvimento, uma vez que optamos por trabalhar com números de tipo binário e vetores de tamanho dinâmico para implementar as tabelas, em C isso teria de ser feito de forma muito manual, o que criaria muitos problemas desnecessários. Traduzir algumas funções como call e return foi muito mais simples do que achávamos, já que muito do trabalho é feito exclusivamente pela máquina, como manipular a pilha por exemplo. Outro problema que encontramos na implementação foi a falta de padronização, como a forma de alocar os bytes na função *.data* por exemplo, por esse motivo tentamos seguir o método que o CPUSim utiliza o mais fielmente possível. Em todos os testes, nossa memória foi escrita corretamente, sendo aceita pelo CPUSim em cada um dos casos.