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

3.1.2 Entrada de dados

3.1.3 Saída esperada

3.2 Programa 2

3.2.1 Descrição

3.2.2 Entrada de dados

3.2.3 Saída esperada

4 Testes e simulações

“Deve conter elementos que comprovem que o montador foi testado (Ex.: imagens das telas de montagem e execução no CPUSim). Quaisquer arquivos relativos a testes devem ser enviados no pacote do trabalho, como mencionado na Seção 2. A documentação deve conter referências a esses arquivos, explicação do que eles fazem e dos resultados obtidos.“ <– apaga isso

5 Conclusão