Software Básico - TP1

Ana Luisa Lima Rodrigues
Carlos Alberto de Carvalho Antônio
Luiz Otávio Teixeira Caldonazo

Maio 2018

1 Introdução

Um montador é um programa que cria um código objeto traduzindo instruções da linguagem de máquina (Assembly) para o código de máquina. O montador substitui instruções, variáveis pelos códigos binários ou hexadecimais e endereços de memória correspondentes. Os montadores mais usados em compiladores de linguagens de alto nível executam a tradução em duas leituras do código fonte:

- A primeira consiste na definição de uma tabela de endereçamento (tabela de símbolos) para rótulos e pré-processamento caso necessário (o que não é o caso deste trabalho);
- A segunda na tradução efetiva de cada instrução para o formato de linguagem de máquina.

O objetivo deste trabalho é desenvolver em linguagem C/C++ um montador para a máquina *Swombat R3.0* e testar os resultados no simulador CPUSim. O planejamento e desenvolvimento tiveram como base o livro Organização Estruturada de Computadores, dos autores Tanenbaum e Austin.

2 Implementação

Inicialmente são definidas constantes, que representam o código das operações, e a definição dos registradores que serão utilizados.

Além disso, são declarados os seguintes atributos: a tabela de símbolos, que tem como finalidade armazenar rótulos e endereços; um vetor auxiliar, para manusear a função .data; um vetor que possuirá todas as instruções hexadecimais; e outro vetor para manipulação de instruções.

Outras variáveis como contador do programa (PC) e tamanho de memória também são declaradas.

2.1 Funções Auxiliares

Como complemento para as duas passadas, são necessárias algumas funções facilitadoras:

- hexToDecimal: Recebe uma string em hexadecimal e converte para inteiro decimal.
- decimalToHex: Recebe um inteiro em decimal e converte para string hexadecimal.
- checkSum: Gera byte de validação para código hexadecimal.
- GetHexFromBin: Recebe uma string binária e converte para string hexadecimal.
- DecToBin: Recebe um inteiro decimal e converte para string binário.
- criaHex: Gera código em hexadecimal usando como parâmetros o endereço de instrução em hexadecimal e dado binário.
- formataAddress: Coloca o inteiro que representa a posição da memória no formato hexadecimal e adiciona 0s à esquerda até o tamanho ser igual a 4, para concatenar com o resto da informação especificada no formato Intel HEX.

2.2 Primeira Passada

Conforme citado na introdução, a primeira passada consiste na criação de uma tabela dos rótulos do programa com base em seus ILCs (Instruction Location Counters).

A passada começa com a leitura sequencial do arquivo, buscando por rótulos, que começam com o caractere "_". Ao serem detectados, são tratados e adicionados à tabela de símbolos, juntamente aos seus respectivos ILCs.

É importante mencionar que a primeira passada ignora comentários, que são strings seguidas de ";"e que a utilização de espaçamentos não especificados como 'TAB' e quebras de linha afetam negativamente esta passada resultando em um código de maquina errado.

2.3 Segunda Passada

A segunda passada também percorre o código sequencialmente. No entanto, ela é responsável por efetuar a tradução de cada instrução.

Para alguns tipos de instruções, devido ao fato de seus parâmetros serem estruturalmente iguais, o processo de conversão é o mesmo. Já para funções com parâmetros diferentes, a conversão é um pouco diferente. A seguir serão explicados os tipos de conversões utilizadas nas instruções:

- STOP: Converte endereço atual em hexadecimal, preenchendo o início da instrução por meio da função FormataAddress com esse endereço. E posteriormente, gera campo de dados com 16 bits zerados a fim de encerrar o programa.

- LOAD, STORE, JMPZ, JMPN, LOAD_S, STORE_S, LOADC: Todas essas instruções possuem 2 bits referentes ao registrador a ser utilizado e 9 bits de endereçamento. Desta forma, a passada gera os primeiros 8 bits da instrução a serem preenchidos no campo dados do código traduzido, sendo os 5 primeiros para o código da operação em hex juntamente com 2 bits

referentes ao registrador em hexadecimal + 1 bit nulo para completar os 8 bits. E logo em seguida, converte o endereço a ser utilizado pela operação, o adicionando à um novo endereço de código traduzido de 8 bits.

- READ, PUSH, POP, COPYTOP: Já para essas instruções, o processo é ligeiramente diferente. Para os 8 primeiros bits, é inserido somente o valor referente ao código das operações, e 3 bits nulos para completar os 8 bits. Nos 8 bits seguintes os primeiros 6 bits são '0' e os ultimos 2 bits são referentes ao registrador utilizado pela instrução.
- WRITE, JUMP, CALL: Nestas instruções, os 5 primeiros bits são para código da operação, 3 bits nulos para completar o primeiro byte dos dados. E os outros 8 bits são preenchidos com o endereço a ser utilizado pelas instruções.
- ADD, SUBTRACT, MULTIPLY, DIVIDE: Para essas instruções, os primeiros 8 bits são preenchidos com o código da operação e com o primeiro registrador acrescido de um bit '0' ao final. Para o segundo byte, são preenchidos 8 bits de '0' e é traduzido e adicionado o segundo registrador.
- MOVE, LOADI, STOREI: Analogamente, nos primeiros 8 bits, insere-se código da operação, primeiro registrador e 1 bit '0'. No segundo byte, são preenchidos 6 bits '0' e o segundo registrador.
- RETURN: Neste caso, os 16 bits são a própria instrução, que diz para o CPUsim que deve realizar a operação return.
- .DATA: Esta instrução implementa a pseudo-instrução .data, ela realiza o armazenamento dos valores cujo os rótulos utilizam esta instrução, os dados tem tamanho de 2 bytes e são armazenados do fim da memoria pra cima, ou seja, o primeiro valor é inserido nos ultimos dois endereços da memoria e os seguintes nas ultimas duas posições que a memória estiver livre.

3 Testes

3.1 Arquivos de entrada

- entrada_teste1.a

```
1     _start: read A0
2     _L0:
3         load_c A2 2     ;load_c A2 2
4          load A3 _b
5          load A1 _a
6          multiply A1 A3
7          add A0 A1
8          multiply A0 A2     ;comentario
9          multiply A0 A2     ;multiply A0 A2
10         subtract A0 A2
11         divide A0 A2
12         write A0
13         jmpn A0 _done2     ;jmpn A0 _done2
```

```
    _done:
         load_c A0 1
         move A0 A1
         write A1
         stop
    _done2:
         load_c A0 -1
         move A0 A1
         write A1
         stop
    _a: .data 2 20
    _b: .data 2 10
```

Essa entrada tem o objetivo de implementar diferentes operações matemáticas para explorar ao máximo a utilização de instruções e registradores. Ela pode ser representada pelo seguinte código em C, possibilitando a compreensão de seu funcionamento:

```
    resultado = (A0 + (_a * _b)) * 4 - 2) / 2;
    if( resultado >= 0 ) printf("%d\n1\n", resultado);
    else printf("%d\n-1\n", resultado);
```

Onde A0 representa o registrador A0 e _a e _b representam variáveis guardadas na memória pelo .data. Portanto, a entrada deverá ser um valor qualquer e a saída será o resultado das operações aplicadas nos valores de A1, A2 e A3.

- entrada_teste2.a

```
    _start: read A0
    read A1
    move A0 A3
    load_c A2 1
    jmpz A1 _done2
    subtract A1 A2
    call _pow
    _done:
    write A0
    stop
    _done2:
    write A2
    stop
    _pow:
    jmpz A1 _return
    multiply A0 A3
    subtract A1 A2
    jump _pow
    _return: return
```

Já essa entrada tem o objetivo de elevar o valor do registrador A0 pelo valor do registrador A1, que equivalentemente em C seria a operação **pow(A0,A1)**. Portanto, as entradas deverão ser dois valores, sendo o primeiro a base e o segundo o expoente. O resultado dessa operação será o valor de saída.

3.2 Instruções utilizadas

As instruções utilizadas na primeira entrada foram 12, de um total de 24, se for incluída a instrução *.data*, representando um total de 50%. Caso contrário, se o *.data* não for considerado, foram utilizadas 11 instruções de um total de 23, representando 47,82%. São elas:

1. read
2. load_c
3. load
4. multiply
5. add
6. subtract
7. divide
8. write
9. jmpn
10. move
11. stop
12. .data

Já na segunda entrada, foram utilizadas 11 instruções de um total de 24 (se for considerada a *.data*, contabilizando 45,83%. Se não for considerada, seriam 10 instruções de um total de 23, ou seja, 43,47%. São elas:

1. read
2. move
3. load_c
4. jmpz
5. subtract
6. multiply
7. call
8. write
9. stop
10. return
11. .data

Portanto, somando as duas entradas foram utilizadas 66,66% das instruções (contando a *.data*) ou 65,21% se não for considerada.

4 Conclusão

Com esse trabalho pudemos adquirir maior conhecimento sobre o processo de montagem de uma máquina e ver na prática seu funcionamento. Anteriormente nas aulas estudamos seu funcionamento na teoria e agora, com maior conhecimento, pudemos colocá-lo em prática.

Esse aprendizado é de extrema importância já que as linguagens de alto nível que tanto utilizamos são lidas e decodificadas pelo montador para assim serem reconhecidas pelo sistema operacional e diversos sistemas embarcados e drivers são programados diretamente em linguagem de montagem.