Memória do RISCV

Kálley Wilkerson - 170038050 - Turma A

Resumo

Este arquivo documenta o trabalho 2 da disciplina o qual consiste em fazer uma simulação das instruções de acesso a memória do RISCV em linguagem C.

1 Descrição do problema

A ideia do projeto é criar funções que tratam um certo conjunto de dados da mesma forma que as instruções do RISCV tratam a memória de um processador. A memória de um processador é simulada através de um vetor de inteiros de *32* bits onde cada posição no vetor representa uma palavra que é um conjunto de *4* bytes sendo que os *8* bits menos significativos da palavra são o primeiro byte da palavra e os *8* bits mais significativos da palavra são o último byte. As funções sempre podem ter acesso a memória, o que faz com que quase todas elas recebam a memória como argumento.

2 Informações úteis sobre o programa

Este programa foi feito no sitema operacional **Ubuntu 18.04** e compilado utilizando o compilador **gcc 7.3.0**. Ele foi feito no editor de texto **GNU Emacs** e portanto não está atrelado a uma **IDE** específica.

Para compilar o programa vá para a pasta **src** no terminal e digite **make** que o programa irá compilar normalmente. Depois, execute o programa digitando **./main** em seu terminal.

3 Funções implementadas

Uma das decisões de projeto tomadas foi de que a memória de dados será passada como parâmetro das funções e dessa forma, a posição do byte mandado como argumento será sempre em relação à posição 0 da memória.

3.1 lw()

Esta função recebe o endereço da memória simulada e o offset de bytes a partir do endereço inicial do vetor de memória. Ela não tem segredos

e quando recebe algum valor inválido de posição (um valor não múltiplo de quatro) ela simplesmente trunca para a maior posição válida menor que o valor fornecido. Ela simula a instrução **lw** do *RISCV* carregando da memória uma palavra de 32 bits na posição dada retornando o valor lido.

3.2 lbu()

Esta função carrega da memória um byte sem sinal, ou seja, quando ela for retornar um valor em um formato inteiro ela não estenderá o bit de sinal caso o byte lido pudesse ser lido como um valor negativo em complemento a dois. Ela também recebe o endereço do vetor que simula a memória e o offset de bytes para o destino e retorna o valor do byte lido sem considerar o sinal no byte.

O que a função faz é simplesmente acessar a palavra onde está o byte desejado, arrastar os bits da palavra até chegar nele e então utiliza uma máscara para deixar sobrando apenas o byte desejado.

3.3 lhu()

Esta função se parece muito com a apresentada na Subseção 3.2, os tipos dos argumentos e saída da função são os mesmos, a diferença é que agora o valor de retorno é uma meia-palavra de 16 bits em que sua posição é dada pelo usuário e espera-se que este coloque um valor par para a posição, mas caso isto não aconteça, a função irá retornar o valor lido na maior posição válida menor que a posição fornecida pelo usuário.

Da mesma forma que a função da Subseção 3.2, ela acessa a posição da palavra onde está a meia-palavra procurada, arrasta o valor e separa a meia-palavra procurada pelo usuário utilizando uma máscara.

3.4 lb()

Esta função faz a leitura de um byte na memória, mas, ao contrário da função **lbu** ela faz a leitura considerando um byte com sinal, ou seja quando ela for expandir o valor lido para **int32_t** o sinal será considerado e passado para o restante da palavra. Ela recebe como argumento o endereço inicial do vetor que representa a memória e o offset do byte procurado.

A função primeiramente encontra a posição na memória da palavra onde está o byte procurado, depois separa esse byte, faz um *cast* do valor encontrado para **int8_t** e depois refaz o cast para **int32_t** o que estende o bit de sinal para o restante da palavra.

3.5 lh()

Esta função carrega uma meia-palavra da memória considerando que é uma meia-palavra com sinal da mesma forma que a função da Subseção 3.4

faz para bytes. Ela recebe como argumento o endereço da memória e o offset do byte que indica o início da meia palavra, sendo que esse offset deve ser um valor par, mas caso não seja, a função trunca o valor para o maior valor válido menor que o valor fornecido.

Esta função começa encontrando a posição na memória da palavra onde está a meia-palavra que se quer encontrar, depois separa essa meia-palavra através de operações com bits. Então a função faz um cast do valor encontrado para o tipo **int16_t** e logo depois outro cast para o tipo **int32_t** o que faz com que o bit de sinal da meia-palavra seja estendido para os outros bits restantes.

3.6 sw()

Esta é a mais simples das funções que guardam valores na memória. Ela tem três argumentos, o primeiro deles é o endereço da memória, o segundo o offset para o byte inicial onde se quer guardar a palavra e o terceiro é o valor que se quer guardar.

O que a função faz é simplesmente alterar o valor da posição de memória fornecida diretamente.

3.7 sb()

Esta função guarda um byte em uma dada posição na memória. Ela recebe como parâmetros o endereço da memória, o offset de onde se quer guardar o byte e o próprio valor de byte que se quer guardar.

A função primeiramente cria a máscara que será usada para limpar a posição do byte na palavra em que se quer apagar, depois encontra a posição na memória dessa palavra, então ela define o valor do shift que será usado para "shiftar" o dado/máscara, depois a máscara é "shiftada" e negada e é feita uma operação **and** bit a bit com a palavra para zerar a posição onde se quer escrever o byte. Depois é criado um segundo valor de dado que é do tipo **int32_t** que depois de receber o dado original é "shiftado" e então faz-se uma operação de **set** fazendo um **ou** bit a bit para definir apenas aqueles bits na posição onde se quer guardar o byte.

3.8 sh()

Esta função é muito parecida com a mostrada na Subseção 3.7 mudando apenas que agora as operações consideram que estamos lidando com uma meia-palavra. Ela recebe o endereço da memória, o offset para a palavra e o dado que se quer guardar.

3.9 dump_mem()

Esta função simplesmente imprime os valores guardados na memória na tela do computador. Ela recebe o endereço inicial da memória e a quantidade de palavras que o usuário quer que o programa imprima.

4 Testes e resultados

Foi utilizado um desenvolvimento orientado a testes bem simples para este projeto. Os testes feitos podem ser encontrados no arquivo **test_mem.c** dentro da pasta **src**. Nestes testes foram inicializados alguns vetores para testar as funções nos bytes desses vetores. As subseções seguintes mostram alguns dos testes feitos.

Observe que nem todos os testes feitos foram colocados nas próximas subseções, mas a grande maioria deles sim.

4.1 test_lw()

A seguinte configuração de memória foi inicializada:

```
mem[0] = 0xCAFE03C0; /*  0 */
mem[1] = 0xF3F2F1F0; /*  4 */
mem[2] = 0xC3C2C1C0; /*  8 */
mem[3] = 0xA3A2A1A0; /* 12 */
```

Alguns testes e resultados para esta função foram:

```
lw(mem,  0)  = 0xCAFE03C0
lw(mem,  4)  = 0xF3F2F1F0
lw(mem, 12) = 0xA3A2A1A0
```

4.2 test_lbu()

A configuração de memória inicializada foi:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Alguns dos testes feitos e seus resultados foram:

```
lbu(mem, 13) = 0x23
lbu(mem, 15) = 0xCA
lbu(mem,  6)  = 0x54
```

4.3 test_lhu()

Memória de teste inicialidada:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Testes e resultados feitos obtidos, os valores ímpares são traduzidos para o maior valor par menor que o valor fornecido.

```
lhu(mem,  9)  = 0x247F
lhu(mem,  6)  = 0x7654
lhu(mem,  15) = 0xCAFE
```

4.4 test_lb()

Memória inicializada:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Testes e resultados obtidos:

```
lb(mem,  6)  = 0x54
lb(mem,  10) = 0xFFFFFFFD
lb(mem,  15) = 0xFFFFFFCA
```

4.5 test_lh()

Memória teste inicialidada:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Teste e resultados:

```
lh(mem,  13) = 0x2312
lh(mem,  6)  = 0x7654
lh(mem,  12) = 0xFFFFABCD
```

4.6 test_sw()

A memória foi inicializada com zeros. As chamadas seguintes foram feitas para a função:

```
sw(mem,  3,  0xABCDEF98);
sw(mem,  12,  0xCCAAFFEE);
sw(mem,  8,  0xFACA);
```

Depois disso, o resultado foi a memória assim:

```
mem[0] = 0xABCDEF98
mem[3] = 0xCCAAFFEE
mem[2] = 0xFACA
```

4.7 test_sb()

A memória foi inicializada seguinte maneira:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Algumas das chamadas feitas foram:

```
sb(mem, 2, 0x23);
sb(mem, 15, 0xFE);
sb(mem, 8, 0x07);
```

Depois disso, as posições de iteresse da memória ficaram:

```
mem[0] = 0xAB23EF98
mem[2] = 0xCEFD2407
mem[3] = 0xFEFE2312
```

4.8 test_sh()

A memória foi inicializada da seguinte maneira:

```
mem[0] = 0xABCDEF98; /*  0 */
mem[1] = 0x76543210; /*  4 */
mem[2] = 0xCEFD247F; /*  8 */
mem[3] = 0xCAFE2312; /* 12 */
```

Algumas das chamadas da função foram:

```
sh(mem, 3, 0xCCCC);
sh(mem, 4, 0xCACA);
sh(mem, 14, 0x2222);
```

Os resultados na memória nas posições de interesse foram:

```
mem[0] = 0xCCCCEF98
mem[1] = 0x7654CACA
mem[3] = 0x22222312
```