Estevam Galvão Albuquerque – 16/0005663 – Turma C

PROJETO 2 – SIMULADOR ASSEMBLY: ACESSO À MEMÓRIA

OBJETIVO

Este trabalho consiste na simulação das instruções de acesso à memória do RISCV RV32I em linguagem C.

DESCRIÇÃO

Para simular o acesso a memória do RISCV utilizando e *loads* e *stores*, foram implementadas funções que se relacionavam com um *array* de inteiros de 32 bits. Dessa forma, escrevíamos e líamos as informações do *array* variando apenas a quantidade de *bits* que seriam lidos/escritos.

Compilador: MinGW

Sistema Operacional: Ubuntu 18.04

IDE: CLion

TESTES E RESULTADOS:

Foram realizados os testes exigidos, tanto os sugeridos pelo professor, quanto os testes extras.

```
void userTest() {

    volatile int i;

    for (i = 0; i < 40; i += 4) {
        sw(i, 0, 0);
    }

    dump_mem(0, 10);

    sb(0, 4, 0x10);
    sb(0, 5, 0x0B);
    sb(0, 6, 0xAC);
    sb(0, 7, 0x09);

    sh(0, 10, 0xFF00);
    sh(0, 14, 0x00FF);
    sb(10, 7, 0xFF);
    sb(10, 10, 0xFF);

    sw(36, 0, 0x1234CAFE);

    sh(20, 6, 0xC1C0);
    sb(20, 9, 0xAC);
    sb(32, 0, 0xEC);

    printf("\n\n");
    dump_mem(0, 10);
    printf("\n\n");

    lw(36, 0);
    lh(36, 0);
    lh(10, 0);
    lhu(10, 0);
    lb(28, 4);
    lb(20, 0);
    lbu(10, 10);

}
```

```
mem[0] = 00000000
mem[1] = 00000000
mem[2] = 00000000
mem[3] = 00000000
mem[4] = 00000000
mem[5] = 00000000
mem[6] = 00000000
mem[7] = 00000000
mem[8] = 00000000
mem[9] = 00000000


mem[0] = 00000000
mem[1] = 09AC0B10
mem[2] = FF000000
mem[3] = 00FF0000
mem[4] = 0000FF00
mem[5] = 000000FF
mem[6] = C1C00000
mem[7] = 0000AC00
mem[8] = 000000EC
mem[9] = 1234CAFE


load_word(36, 0)
    HEXADECIMAL -> 0x1234CAFE
    DECIMAL -> 305449726

load_half(36, 0)
    HEXADECIMAL -> 0xCAFE
    DECIMAL -> -13570

load_half(10, 0)
    HEXADECIMAL -> 0xFF00
    DECIMAL -> -256

load_uhalf(10, 0)
    HEXADECIMAL -> 0xFF00
    DECIMAL -> 65280

load_byte(28, 4)
    HEXADECIMAL -> 0xEC
    DECIMAL -> -20

load_byte(20, 0)
    HEXADECIMAL -> 0xFF
    DECIMAL -> -1

load_ubyte(10, 10)
    HEXADECIMAL -> 0xFF
    DECIMAL -> 255
```

```
void test() {

    sb(0, 0, 0x04);
    sb(0, 1, 0x03);
    sb(0, 2, 0x02);
    sb(0, 3, 0x01);
    sb(4, 0, 0xFF);
    sb(4, 1, 0xFE);
    sb(4, 2, 0xFD);
    sb(4, 3, 0xFC);
    sh(8, 0, 0xFFF0);
    sh(8, 2, 0x8C);
    sw(12, 0, 0xFF);
    sw(16, 0, 0xFFFF);
    sw(20, 0, 0xFFFFFFFF);
    sw(24, 0, 0x80000000);

    dump_mem(0, 7);

    printf("\n\nLOADS\n\n");
    lb(0,0);
    lb(0,1);
    lb(0,2);
    lb(0,3);
    lb(4,0);
    lb(4,1);
    lb(4,2);
    lb(4,3);

    printf("\n\nLOADS U\n\n");
    lbu(4,0);
    lbu(4,1);
    lbu(4,2);
    lbu(4,3);

    printf("\n\nLOADS HALF\n\n");
    lh(8,0);
    lh(8,2);

    printf("\n\nLOAD WORD\n\n");
    lw(12,0);
    lw(16,0);
    lw(20,0);

}
```

```
mem[0] = 01020304
mem[1] = FCFDFEFF
mem[2] = 008CFFF0
mem[3] = 000000FF
mem[4] = 0000FFFF
mem[5] = FFFFFFFF
mem[6] = 80000000


LOADS

load_byte(0, 0)
    HEXADECIMAL -> 0x04
    DECIMAL -> 4

load_byte(0, 1)
    HEXADECIMAL -> 0x03
    DECIMAL -> 3

load_byte(0, 2)
    HEXADECIMAL -> 0x02
    DECIMAL -> 2

load_byte(0, 3)
    HEXADECIMAL -> 0x01
    DECIMAL -> 1

load_byte(4, 0)
    HEXADECIMAL -> 0xFF
    DECIMAL -> -1

load_byte(4, 1)
    HEXADECIMAL -> 0xFE
    DECIMAL -> -2

load_byte(4, 2)
    HEXADECIMAL -> 0xFD
    DECIMAL -> -3

load_byte(4, 3)
    HEXADECIMAL -> 0xFC
    DECIMAL -> -4



LOADS U

load_ubyte(4, 0)
    HEXADECIMAL -> 0xFF
    DECIMAL -> 255

load_ubyte(4, 1)
    HEXADECIMAL -> 0xFE
    DECIMAL -> 254

load_ubyte(4, 2)
    HEXADECIMAL -> 0xFD
```

FUNÇÕES IMPLEMENTADAS

lw(uint32_t address, int32_t kte):

i. É uma função que irá ler uma *word*, conjunto de 32 *bits*, e retornará 32 *bits*.
ii. Identifico se o usuário digitou um intervalo múltiplo de 4, para caso não seja, imprimo uma notificação e não realizo a operação de leitura.
iii. Inicializo um ponteiro para *words*, o qual servirá para andar pelo meu *array.*
iv. Como o usuário entrega um "endereço + constante" andando de 8 em 8 *bits*, divido esse valor por 4 porque meu ponteiro anda de 32 em 32 *bits*.
v. Caso tudo tenha ocorrido corretamente, imprimo o valor do dado lido em Hexadecimal e em Decimal e o retorno.

lh(uint32_t address, int32_t kte):

i. É uma função que irá ler uma *half word*, conjunto de 16 *bits*, e retornará 32 *bits*.
ii. Identifico se o usuário digitou um intervalo múltiplo de 2, para caso não seja, imprimo uma notificação e não realizo a operação de leitura.
iii. Inicializo um ponteiro para *half words*, o qual servirá para andar pelo meu *array.*
iv. Como o usuário entrega um "endereço + constante" andando de 8 em 8 *bits*, divido esse valor por 2 porque meu ponteiro anda de 16 em 16 *bits*.

lhu(uint32_t address, int32_t kte):

i. É uma função que irá ler uma *unsigned half word*, conjunto de 16 *bits* sem sinal, e retornará 16 *bits*.
ii. Identifico se o usuário digitou um intervalo múltiplo de 2, para caso não seja, imprimo uma notificação e não realizo a operação de leitura.
iii. Inicializo um ponteiro para *unsigned half word*, o qual servirá para andar pelo meu *array.*
iv. Como o usuário entrega um "endereço + constante" andando de 8 em 8 *bits*, divido esse valor por 2 porque meu ponteiro anda de 16 em 16 *bits*.
v. Caso tudo tenha ocorrido corretamente, imprimo o valor do dado lido em Hexadecimal e em Decimal e o retorno.

lb(uint32_t address, int32_t kte):

i. É uma função que irá ler um *byte*, conjunto de 8 *bits*, e retornará 32 *bits*.
ii. Inicializo um ponteiro para *bytes*, o qual servirá para andar pelo meu *array.*
iii. Caso tudo tenha ocorrido corretamente, imprimo o valor do dado lido em Hexadecimal e em Decimal e o retorno.

lb(uint32_t address, int32_t kte):

i. É uma função que irá ler um *unsigned byte*, conjunto de 8 *bits* sem sinal, e retornará 32 *bits*.
ii. Inicializo um ponteiro para unsigned *bytes*, o qual servirá para andar pelo meu *array.*
iii. Caso tudo tenha ocorrido corretamente, imprimo o valor do dado lido em Hexadecimal e em Decimal e o retorno.

sw(uint32_t address, int32_t kte, int32_t data):

i. É uma função que irá escrever uma *word*, conjunto de 32 *bits* no *array* de memória.
ii. Identifico se o usuário digitou um intervalo múltiplo de 4, para caso não seja, imprimo uma notificação e não realizo a operação de escrita.
iii. Inicializo um ponteiro para *word*, o qual servirá para andar pelo meu *array.*
iv. Como o usuário entrega um "endereço + constante" andando de 8 em 8 *bits*, divido esse valor por 4 porque meu ponteiro anda de 32 em 32 *bits*.
v. Caso tudo tenha ocorrido corretamente, escrevo o dado no *array*.

sh(uint32_t address, int32_t kte, int32_t data):

vi. É uma função que irá escrever uma half *word*, conjunto de 16 *bits* no *array* de memória.
vii. Identifico se o usuário digitou um intervalo múltiplo de 2, para caso não seja, imprimo uma notificação e não realizo a operação de escrita.
viii. Inicializo um ponteiro para half *word*, o qual servirá para andar pelo meu *array.*
ix. Como o usuário entrega um "endereço + constante" andando de 8 em 8 *bits*, divido esse valor por 2 porque meu ponteiro anda de 16 em 16 *bits*.
x. Caso tudo tenha ocorrido corretamente, escrevo o dado no *array*.

sb(uint32_t address, int32_t kte, int32_t data):

xi. É uma função que irá escrever uma *word*, conjunto de 32 *bits* no *array* de memória.
xii. Inicializo um ponteiro para *byte*, o qual servirá para andar pelo meu *array.*
xiii. Caso tudo tenha ocorrido corretamente, escrevo o dado no *array*.