UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

CIC0099 ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

Trabalho I: Memória do RISCV

OBJETIVO

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

DESCRIÇÃO

Tipos de dados

Utilizar os tipos de dados definidos em <stdint.h>:

uint8_t : inteiro sem sinal de 8 bits.
int32_t, **int16_t**, **int8_t** : inteiro com sinal de 32, 16 e 8 bits, respectivamente.

Memória

A memória é simulada como um arranjo de inteiros de 32 bits.

```
#define MEM_SIZE 4096
int32_t mem[MEM_SIZE];
```

Ou seja, a memória é um arranjo de 4KWords, ou 16KBytes.

Funções de acesso à Memória

Cálculo do endereço do dado na memória:

Todas as funções calculam o endereço do dado na memória por meio de dois parâmetros: *address* e *kte*. Estes parâmetros representam os parâmetros utilizados pelas instruções de acesso à memória do RISCV:

- `lw reg, kte(address)`
- `sw reg, kte(address)`

Assim, o endereço do dado é um endereço de *byte*, dado pela soma dos parâmetros:

- `endereço = address + kte`

O que significa "endereço de byte": o endereço obtido pela soma acima localiza um *byte* na memória.

- Uma função que lê um *byte* (*lb - load byte*) vai retornar o valor do byte endereçado.
- Uma função que lê uma meia-palavra (*lh - load half*) vai retornar dois *bytes*, aquele endereçado e o *byte* que está no próximo endereço.
- Uma função que lê uma palavra (*lw - load word*) irá retornar 4 *bytes*, sendo o endereço fornecido para a função o do *byte* menos significativo.

Ex: supor que *int32_t mem[4] = 0x0A0B0C0D*. Como cada entrada do vetor é um inteiro de 32 bits, temos que o endereço do *byte* menos significativo da palavra (0x0D) é 16 (4x4).

Assim:

- lb(16, 0) = lb(15, 1) = lb (14, 2) = 0x0D
- lb(17, 0) = lb(16, 1) = 0x0C
- lh(16, 0) = lh(15, 1) = 0x0C0D
- lh(16, 1) => erro, pois o endereço é ímpar. Lh só aceita endereços pares.
- lw(16, 0) = lw(13, 3) = 0x0A0B0C0D
- lw(16, 2) => erro, pois os endereço de lw tem que ser múltiplos de 4

Note que o endereço é sempre um valor positivo. Não existem endereços negativos. Assim, o parâmetro *address* é inteiro sem signal (positivo). Por outro lado, o parâmetro *kte* pode ser positivo ou negativo, representando um deslocamento para frente ou para trás com relação ao endereço especificado em *address*.

Desenvolver as seguintes funções:

```
int32_t  lw(uint32_t address, int32_t kte);
```

Lê um inteiro alinhado - endereços múltiplos de 4.

A função deve checar se o endereço é um múltiplo de 4 (%4 == 0).

Se não for, deve escrever uma mensagem de erro e retornar zero.

Se o endereço estiver correto, a função deve:

- Dividi-lo por 4 para obter o *índice* do vetor memória
- Retornar o o valor lido da memória

```
int32_t  lb(uint32_t address, int32_t kte);
```

Lê um byte do vetor memória e retorna-o, estendendo o sinal para 32 bits.

Lembrando que as palavras da memória tem 4 bytes cada, para acessar um byte dentro da palavra pode-se:

- Ler a palavra que contém o byte e, por operações de mascaramento, extrair byte endereçado, ou
- Criar um ponteiro para *byte* e fazer um *type cast* (coerção de tipo) do endereço do vetor memória (int32_t *) para byte (int8_t *). Usando o ponteiro para *byte* pode-se acessar diretamente o dado desejado.

```
int32_t  lbu(uint32_t address, int32_t kte);
```

Lê um *byte* do vetor memória e retorna-o como um número positivo, ou seja, todos os bits superiores devem ser zerados.

```
void  sw(uint32_t address, int32_t kte, int32_t dado);
```

Escreve um inteiro alinhado na memória - endereços múltiplos de 4. O cálculo do endereço é realizado da mesma forma que na operação *lw().*

```
void  sb(uint32_t address, int32_t kte, int8_t dado);
```

Escreve um *byte* na memória. Caso utilize operações de mascaramento, a palavra que contém o *byte* deve ser lida da memória, o *byte* deve ser posicionado corretamente através de deslocamentos e a escrita ocorre utilizando máscaras. Alternativamente pode-se utilizar a coerção para (int8_t *) e escrever diretamente no vetor memória.

Verificação

A verificação do código será feita de forma automática pelo *Coderunner*. O aluno deve entrar com o código da solução no questionário do Moodle. A linguagem deve ser C. Para simplificar o desenvolvimento, colocar todo o código em um arquivo único.

A entrada de dados deve ser no formato especificado abaixo. Os dados estão especificados nesta ordem e formato no *Coderunner*.

Código de teste:

```
int main (int argc, const char * argv[]) {
    int8_t b0, b1, b2, b3;
    int32_t w;

    scanf("%hhx, %hhx, %hhx, %hhx", &b0, &b1, &b2, &b3);
    sb(0, 0, b0);
    sb(0, 1, b1);
    sb(0, 2, b2);
    sb(0, 3, b3);
    scanf("%hhx, %hhx, %hhx, %hhx", &b0, &b1, &b2, &b3);
    sb(4, 0, b0);
    sb(4, 1, b1);
    sb(4, 2, b2);
    sb(4, 3, b3);
    scanf("%x", &w);
    sw(8, 0, w);
    scanf("%x", &w);
    sw(12, 0, w);
    scanf("%x", &w);
    sw(16, 0, w);
    scanf("%x", &w);
    sw(20, 0, w);

    for (int i = 0; i < 6; i++)
        printf("%08x\n", mem[i]);

    printf("%08x\n", lb(4,0));
    printf("%08x\n", lbu(4,0));
    printf("%08x\n", lw(12, 0));
    printf("%08x\n", lw(16, 0));
    printf("%08x\n", lw(20, 0));

    return 0;
}
```