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 ou Python.

DESCRIÇÃO

Tipos de dados

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

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

Python: Utilizar o pacote numpy, com tipos de dados int8, uint8, int32.

Memória

A memória é simulada como um arranjo de *bytes* de 32 bits.

C:
```
#define MEM_SIZE 16384
int8_t mem[MEM_SIZE];
```

Python:
```
import numpy as np
mem = np.zeros(16384, dtype=np.uint8)
```

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

Funções de acesso à Memória

As funções a serem implementadas são:

- *lb(reg, kte)*: lê um byte da memória e o converte para um inteiro de 32 bits estendendo o sinal do byte. Retorna o inteiro de 32 bits.
- *lbu(reg, kte)*: lê um byte da memória e o converte para um inteiro de 32 bits sem sinal (valor positivo). Retorna o inteiro de 32 bits.
- *lw(reg, kte)*: lê uma palavra de 32 bits da memória e retorna o seu valor.
- s*b(reg, kte, byte)*: escreve o byte passado como parâmetro na memória.
- sw*(reg, kte, word)*: escreve os 4 bytes de word na memória, colocando o menos significativo no endereço especificado e os outros nos endereços de *byte* seguintes.

Endereçamento da memória: todas as funções recebem dois parâmetros, *reg* e *kte*. O endereço de byte é dado pela soma dos dois:

address = reg + kte

No caso das funções *lw* e *sw*, os endereços devem ser múltiplos de 4.

Sugestões: tanto em C quanto em Python é possível utilizar operações de mascaramento e deslocamento para processar as palavras.

Ex: extração dos bytes de uma palavra de 32 bits *word*:

seja *word* = 0x01020304 temos que:

byte0 = *word* & 0xFF retorna o primeiro byte, 0x04.

byte1 = (*word* >> 8) & 0xFF produz o segundo byte de word, 0x03.

byte2 = (*word* >> 16) & 0xFF produz o terceiro byte de word, 0x02.

byte3 = (*word* >> 24) & 0xFF produz o quarto byte de word, 0x01.

Ex: construção de uma palavra de 32 bits a partir de 4 bytes:

word = (byte3 << 24) | (byte2 << 16) | (byte1 << 8) | byte0;

Em Python, a extensão de sinal de um byte para uma palavra pode ser feita com o seguinte código:

```
byte = np.int8(mem[address])
if byte < 0:
    word = (1 << 32) + byte
```

Por exemplo, a sequência de código abaixo:

```
sb(7, 0, 0xaa)
word = lb(7, 0)
print("word_10 = ",  word)
print("word_x = ", hex(word))
```

imprime na tela:

```
word_10 = 4294967210
word_x =  0xffffffaa
```

Verificação

1. Escrever a sequência de dados na memória:
   a. *sw(0, 0, 0xABACADEF)*
   b. *sb(4, 0, 1)*
   c. *sb(4, 1, 2)*
   d. *sb(4, 2, 3)*
   e. *sb(4, 3, 4)*

2. Ler as informações escritas na memória e imprimir na tela (em hexadecimal):

a. lw(0, 0)
b. lb(0, 0)
c. lb(0, 1)
d. lb(0, 2)
e. lb(0, 3)
f. lbu(0, 0)
g. lbu(0, 1)
h. lbu(0, 2)
i. lbu(0, 3)

3. Respostas esperadas:

```
a. 0xabacadef
b. 0xffffffef
c. 0xffffffad
d. 0xffffffac
e. 0xffffffab
f. 0xef
g. 0xad
h. 0xac
i. 0xab
```