Organização e Arquitetura de Computadores
Trabalho I: Memória do MIPS

Gabriel Correia de Vasconcelos
Departamento de Ciência da Computação
Universidade de Brasília
Campus Universitário Darcy Ribeiro, DF
gcvasconcelos98@gmail.com

***Abstract*— Primeiro trabalho de Organização e Arquitetura de Computadores com o objetivo de implementar, em C, uma simulação de instruções de manipulação da memória na arquitetura de um processador MIPS.**

I. DESCRIÇÃO DO PROBLEMA

O trabalho consiste na simulação de instruções de uma arquitetura de microprocessadores RISC, baseada em registradores. A simulação utiliza máscaras de bits e operações lógicas bit a bit do C para simular como o microprocessador de fato funciona.

II. DESCRIÇÃO DAS INSTRUÇÕES

As instruções implementadas apenas manipulam um vetor de inteiros de 32 bits (palavras), que simula uma memória de 16 KBytes.

Nas funções abaixo os offsets são referentes a posição do byte na palavra da memória. Por exemplo, dado um offset 0, ele presentaria o primeiro byte da palavra na memória de endereço address, um offset 1 representaria o segundo byte e assim por diante.

Para a verificação da divisibilidade foi utilizada o operador % (módulo) do C e para a aplicação da máscara foram utilizados os operadores & (and bitwise) e | (or bitwise). Para o deslocamento dos bytes foram utilizados os operadores » (shift right) e « (shift left), sempre multiplicados por 1 byte (8 bits) vezes a constante.

A função de leitura percorre todos os bytes de memória que estão entre o endereço do argumento e o endereço somado ao tamanho. As funções de retorno de bytes inicialmente checam se os endereços dos argumentos estão alinhados, caso negativo, printam uma mensagem de erro. A posição na memória é encontrada pela soma dos argumentos de endereço e a constante, divida por 4. A função retira da memória o byte requisitado zerando todos os outros utilizando uma máscara, com um AND bit a bit e guardando numa variável. O byte contido nessa variável é deslocado até a posição menos significativa e retornado. Na leitura de palavra isso não é necessário pois ela já ocupa a memória toda.

As funções implementadas foram:

```
void dump_mem(uint32_t addr, uint32_t size);
```

A função *dump_mem* serve para printar na tela todos os bytes da memória que iniciam no valor da variável *addr* e tem tamanho definido pela variável *size*.

```
int32_t lw(uint32_t address, int16_t kte);
```

A função *lw* significa "load word", do inglês, carregar palavra, e tem a função de retornar uma palavra, com 32 bits, no formato com sinal, dado um endereço de início e um offset, ambos múltiplos de 4 (32 bits = 4 bytes), pois os endereços não podem estar desalinhados.

```
int32_t lh(uint32_t address, int16_t kte);
```

A função *lh* significa "load half word", do inglês, carregar meia palavra, e tem a função de retornar uma meia palavra, com 16 bits, no formato com sinal, dado um endereço de início e um offset, ambos múltiplos de 2 (16 bits = 2 bytes), pois os endereços não podem estar desalinhados.

```
int32_t lhu(uint32_t address, int16_t kte);
```

A função *lhu* significa "load half word unsigned", do inglês, carregar meia palavra sem sinal, e tem a função de retornar uma meia palavra, com 16 bits, no formato sem sinal, dado um endereço de início e um offset, ambos múltiplos de 2 (16 bits = 2 bytes), pois os endereços não podem estar desalinhados.

```
int32_t lb(uint32_t address, int16_t kte);
```

A função *lb* significa "load byte", do inglês, carregar byte, e tem a função de retornar um byte, com 8 bits, no formato com sinal, dado um endereço de início e um offset.

```
int32_t lbu(uint32_t address, int16_t kte);
```

A função *lbu* significa "load byte unsigned", do inglês, carregar byte sem sinal, e tem a função de retornar um byte, com 8 bits, no formato sem sinal, dado um endereço de início e um offset.

As funções de escrita de bytes na memória sempre checam se os endereços dos argumentos estão alinhados, caso negativo, printam uma mensagem de erro. A posição na memória é facilmente encontrada pela soma dos argumentos de endereço e a constante, divida por 4. Inicialmente o dado é

movido para sua posição na palavra, tem suas outras posições zeradas pela máscara, com um AND bit a bit. Após esse tratamento do dado, ele é inserido na memória com um OR bit a bit com as informações anteriores da memória. Na escrita de palavra isso não é necessário pois ela já ocupa a memória toda.

As funções desenvolvidas foram:

```
void sw(uint32_t address, int16_t kte,
    int32_t dado);
```

A função *sw* significa "store word", do inglês, guardar palavra, e tem a função de escrever na memória uma palavra, com 32 bits, dado um endereço de início e um offset, ambos múltiplos de 4 (32 bits = 4 bytes), pois o dado não pode ser escrito de forma desalinhada.

```
void sh(uint32_t address, int16_t kte,
    int16_t dado);
```

A função *sh* significa "store half word", do inglês, guardar meia palavra, e tem a função de escrever na memória uma meia palavra, com 16 bits, dado um endereço de início e um offset, ambos múltiplos de 2 (16 bits = 2 bytes), pois o dado não pode ser escrito de forma desalinhada.

```
void sb(uint32_t address, int16_t kte,
    int8_t dado);
```

A função *sb* significa "store byte", do inglês, guardar byte, e tem a função de escrever na memória um byte, com 8 bits, dado um endereço de início e um offset.

III. TESTES E RESULTADOS

Para o teste das funções implementadas foram utilizados inicialmente os dados fornecidos no relatório de descrição do trabalho. Após todos estes testes estarem passando, foram elaborados outros para testar as exceções. Foi utilizada a biblioteca <*assert.h*> do C para a execução dos testes.

Os resultados foram de acordo com o que era esperado das funções.