Organização e Arquitetura de Computadores
Trabalho I: Simulador MIPS

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

***Abstract*— Segundo trabalho de Organização e Arquitetura de Computadores com o objetivo de implementar, em C, uma simulação de funções de de busca da instrução, decodificação da instrução e execução da instrução na arquitetura de um processador MIPS, a partir de um binário gerado no montador MARS.**

I. DESCRIÇÃO DO PROBLEMA

O trabalho consiste na simulação do ciclo de busca e execução que acontece em um programa armazenado na memória. O ciclo consiste inicialmente na busca por instruções na memória e no armazenamento delas em um registrador especial chamado IR (*instruction register*). Os 32 bits desse registrador contém informações sobre o tipo de instrução e seus argumentos e metadados que precisam ser codificados para a execução. A última etapa, de fato executa essa instrução e busca a próxima.

O código do programa desenvolvido está disponibilizado na plataforma github e pode ser acessado pelo link `https://github.com/gcvasconcelos/OAC_trabalhos`

II. DESCRIÇÃO DAS FUNÇÕES

As instruções implementadas apenas manipulam um vetor de inteiros de 32 bits (palavras), que simula uma memória de 16 KBytes. Para simular os registradores foi utilizado um vetor de palavras. Para as instruções de escrita e leitura na memória foi reutilizado o código desenvolvido no primeiro trabalho da disciplina.

A fim de facilitar o entendimento de como aconteceria o acesso a registadores no vetor e a correspondência dos códigos de operação foi utilizado o recurso do C chamado *enum*.

As três principais funções implementadas configuram as partes do ciclo e são as seguintes:

```
void fetch();
```

A função *fetch* é a mais simples e basicamente lê uma palavra da memória que indica o PC (*program counter*) e a salva no RI; e incrementa o PC de forma a apontar para a próxima palavra. Anteriormente a ela é executada uma série de funções para inicializar o ambiente, que leem os arquivos binários de código e dados, exportados pelo montador MARS, e passam essas informações para a memória simulada. Também são zerados os valores dos registradores e inicializados os valores do GP(*global pointer*) e SP(*stack pointer*).

Foram definidos macros para facilitar a inserção correta os endereços de começo e fim do código e parte de dados do binário para a memória simulada.

```
void decode();
```

A função *decode* serve para extrair as informações da instrução, independente de seu formato. Essas informações são extraídas rotacionando o RI para a direita com base na sua posição e passando uma máscara para zerar os valores a esquerda, com base no seu tamanho, que pode ser de 5, 6, 16 ou 26 bits.

Esses valores correspondem aos campos da instrução de 32 bits e representam:

- OPCODE: código da operação
- RS: índice do primeiro registrador fonte
- RT: índice do segundo registrador fonte
- RD: índice do registrador destino, que recebe o resultado da operação
- SHAMNT: quantidade de deslocamento em instruções shift e rotate
- FUNCT: código auxiliar para determinar a instrução a ser executada
- K16: constante de 16 bits, valor imediato em instruções tipo I
- K26: constante de 26 bits, para instruções tipo J

Esses valores são salvos em variáveis inteiras com sinal com 8 bits, para as de 5 e 6 bits; 16 bits para a constante de 16 bits; e 32 bits para a constante de 26 bits (essa sem sinal).

```
void execute();
```

A função *execute* é onde as informações extraídas da etapa de decodificação são interpretadas. Inicialmente é identificado o que a operação faz pelo seu código. Instruções de transferência de dados, branches, operações immediatas e saltos são executados diretamente. Caso o OPCODE seja 0, isso significa que a função é determinada pelos primeiros 6 bits, que são novamente analisados e por sua vez executados. Caso a função seja uma chamada de sistema, ainda há uma seleção de qual tipo de chamada foi escolhido, que no caso

pode ser o print de um inteiro, uma string ou uma interrupção no programa.

As operações de transferência de dados implementadas foram:

- LW: Load Word
- LB: Load Byte
- LBU: Load Byte Unsigned
- LH: Load Halfword
- LHU: Load Halfword Unsigned
- LUI: Load Upper Immediate
- SW: Store Word
- SB: Store Byte
- SH: Store Half

As operações de branch implementadas foram:

- BEQ: Branch on EQual
- BNE: Branch on Not Equal
- BLEZ: Branch Less or Equal than Zero
- BGTZ: Branch Greater Than Zero

As operações imediatas implementadas foram:

- ADDI: ADD Immediate
- ADDIU: ADD Immediate Unsigned
- SLTI: Set to 1 if Less Than Immediate
- SLTIU: Set to 1 if Less Than Immediate Unsigned
- ANDI: bitwise AND Immediate
- ORI: bitwise OR Immediate
- XORI: bitwise XOR Immediate

As operações de salto implementadas foram:

- J: Jump
- JAL: Jump And Link

Para operações de função (OPCODE = 0x00), foram implementadas as funções:

- ADD: ADD
- ADDU: ADD Unsigned
- SUB: SUBtract
- MULT: MULTiply
- DIV: DIVide
- AND: bitwise AND
- OR: bitwise OR
- XOR: bitwise XOR
- NOR: bitwise NOR
- SLT: Set to 1 if Less Than
- SLTU: Set to 1 if Less Than Unsigned
- JR: Jump to address in Register
- SLL: Shift Left Logical
- SRL: Shift Right Logical (0-extended)
- SRA: Shift Right Arithmetic (sign-extended)
- MFHI: Move From HI register
- MFLO: Move From LO register

E por último, para as chamadas de sistema (FUNCT = 0x0C), a valor do registrador v0 pode ser:

- 1: para printar inteiros
- 4: para printar strings
- 10: para interromper o programa

III. TESTES E RESULTADOS

Para o teste das funções implementadas foram utilizados inicialmente os dados fornecidos no relatório de descrição do trabalho. A verificação do corretude do sistema foi feita utilizando as funções *dump_mem()* e *dump_regs()* para verificar se os valores de memória e dos registradores estavam iguais ao do MARS.

Após todos estes testes estarem passando,