

SSC0610 – Organização de Computadores Digitais I – 2 semestre 2017

Prof. Dr. Francisco José Monaco

Monitor PEEG: Vitor P. Ribeiro

Monitor: Guilherme Prearo

Trabalho 2

Equipe 6

Nome do Projeto: Bit envolvente

Alexandre Batistella Bellas, 9763168, AlexandreBellas

Felipe Manfio Barbosa, 9771640, felipe-m-barbosa

João Vitor Granzotti Machado, 9393322, JoaoGranzotti

Tiago Lemes Daneluzzi, 8531320, daneluzzitiago

1. Descrição do Problema

O objetivo deste trabalho é implementar em C (gcc/Linux) a CPU MIPS multiciclo de 32 bits vista em sala, e deverá ser baseada no conteúdo do livro de Organização de Patterson & Hennessy, utilizado na disciplina.

É fornecido em conjunto com esta especificação os arquivos `cpu.c` e `cpu.h` contendo o código-fonte inicial necessário para a implementação. Estes códigos devem ser obrigatoriamente seguidos no trabalho, sem alterações. Também é fornecido um arquivo de máscaras (`mask.h`), mas sua utilização é opcional. Para verificar a execução correta do algoritmo desenvolvido é necessário que código supracitado permaneça como está.

Observe que o processador a ser simulado possui várias unidades funcionais que na prática executam em paralelo, ou seja, poderiam ser utilizadas ao mesmo tempo. O seu trabalho deve preservar essa característica de concorrência tanto quanto o possível. Portanto, usando uma programação C sequencial a cada novo ciclo todas as unidades funcionais devem ter a oportunidade de executar (mesmo que sequencialmente). Caberá à unidade de controle (UC) determinar os sinais de controle necessários para permitir ou impedir as possíveis execuções.

A Unidade de Controle (UC) deve ser implementada como uma ROM com sequenciador e tabelas de despacho, conforme discutido em sala. Em caso de dúvidas neste ponto, consultar o Apêndice D (Mapping Control to Hardware) do livro de Patterson e Hennessy utilizado nas aulas.

Implemente neste trabalho, no arquivo `cpu_multi_code.c`, as instruções `add`, `sub`, `slt`, `and`, `or`, `lw`, `sw`, `beq` e `j`.

2. Funcionamento

De acordo com a teoria aprendida em sala podemos identificar algumas características marcantes em uma CPU multi ciclos. Uma máquina de estados é usada para setar os sinais de controle, de modo que, com base no opCode e no estado atual o próximo estado é gerado. A tabela apresenta os diferentes estados necessários para a implementação das funções requeridas com seus respectivos bits de controle.

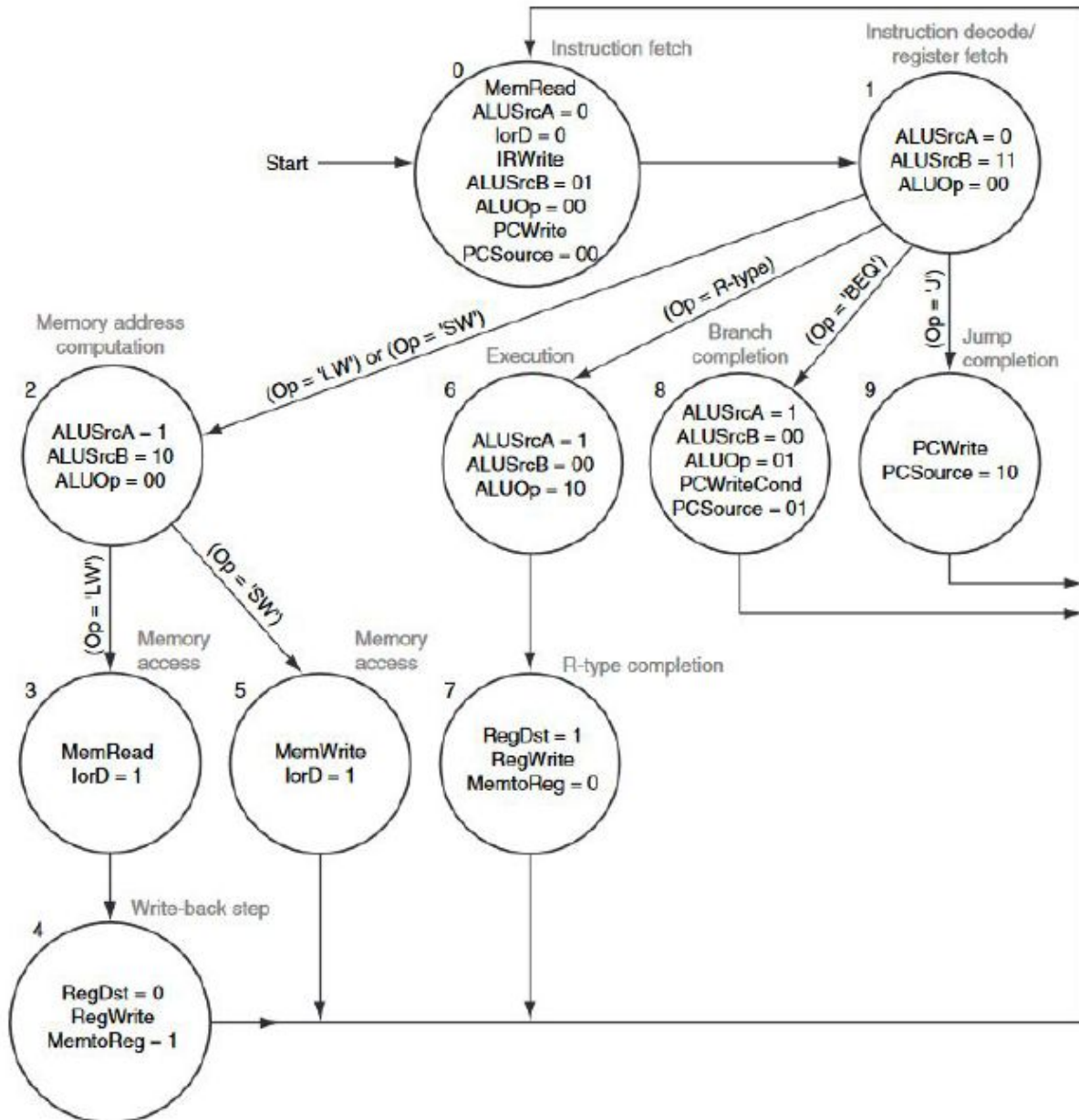


Figura 1: máquina de estados para a implementação em questão.

Para realizar as diferentes instruções existem basicamente 5 passos, sendo cada um deles responsável por uma função dentro dos tipos de instruções, São eles:

- A busca por instrução (Instruction Fetch) - IF.
- Decodificação e Busca de Operandos (instruction Decode / Register Read) - ID.
- Execução (Execute) - EX.
- Acesso à memória (Memory Access) - MEM.
- Escrita vinda da memória (Write Back) - WB.

Esses passos ocorrem de maneira específica de acordo com a instrução executada, como podemos verificar na tabela abaixo:

	LW	SW	Tipo-R	BEQ	Jump
IF	$IR = Mem[PC]$ $PC = PC + 4$				
ID	$A = Reg[IR[25-21]]$ $B = Reg[IR[20-16]]$ $ALUOut = PC + ext(IR[15-0] \ll 2)$				
EX	$ALUOut = A + ext(IR[15-0])$		$ALUOut = A \text{ op } B$	Se $(A==B)$ $PC = ALUOut$	$PC = PC[31-28]$ $ (IR[25-0] \ll 2)$
MEM	$MDR = Mem[ALUOut]$	$Mem[ALUOut] = B$	$Reg[IR[15-11]] = ALUOut$	X	X
WB	$Reg[IR[20-16]] = MDR$	X	X	X	X

Tabela 1: passos para as diferentes instruções propostas.

No arquivo code.c algumas máscaras foram criadas para facilitar o entendimento e a visualização do código:

```

//mascara para definir os diferentes estados da maquinas de estados
#define estado_0      0x9408      // 1001 0100 0000 1000
#define estado_1      0x0018      // 0000 0000 0001 1000
#define estado_2      0x0014      // 0000 0000 0001 0100
#define estado_3      0x1800      // 0001 1000 0000 0000
#define estado_4      0x4002      // 0100 0000 0000 0010
#define estado_5      0x0802      // 0000 1000 0000 0010
#define estado_6      0x0044      // 0000 0000 0100 0100
#define estado_7      0x0003      // 0000 0000 0000 0011
#define estado_8      0x02A4      // 0000 0010 1010 0100
#define estado_9      0x0480      // 0000 0100 1000 0000

//mascaras para opcodes das operações
#define opCode_Tipo_R  0x00      // 000000
#define opCode_LW      0x23      // 100011
#define opCode_SW      0x2b      // 101011
#define opCode_Beq     0x04      // 000100
#define opCode_J       0x02      // 000010

```

Figura 2: máscaras criadas para facilitar o entendimento

A unidade de controle tenta imitar a função de uma ROM onde os bits do opCode e o estado atual (definido como o grupo de sinais de controles ativos para cada estado) decidem qual será o próximo estado. Podemos verificar a logica utilizada na imagem abaixo, a qual pode ser vista em detalhes no próprio arquivo.

```

void control_unit(int IR, short int *sc)
{
    //Conferindo Se é o inicio do programa
    if(IR == -1){
    }
    else{
        //Separando o opCode para identificar a operação que pode ser tipo R, LW, SW e etc
        char opCode = (IR & separa_cop) >> 26;

        //O instruction Fetch acontece pra qualquer opCode então vamos verificar se está nesse estado
        if(*sc == ((short int)estado_0)){
            *sc = estado_1; // jogamos para o estado dois então, que também acontece em todas as operações.
        }
        else{
            //com o opCode em mão vamos verificar qual o peração será feita
            switch(opCode){
            }
        }
    }
    //not_implemented();
}

```

Figura 3: código compactado da função da unidade de controle.

Uma unidade de controle para escolher a operação da ULA de acordo com a necessidade também foi implementada. Podemos verificar seu codigo (compactado) abaixo.

```

//Função que recebe o function (5-0 iR) e os bits de sc ALUOp0 e ALUOp1 e calcula o valor de alu_op
void alu_control(int IR, int sc, char *alu_op){
    //vamos agora verificar os diferentes valores
    switch(((sc & separa_ALUOp0) | (sc & separa_ALUOp1)) >> 5){
        //Essa combinação indica LW e SW
        case 0x0:
            //independe de function
            *alu_op = 0b0010;
            break;
        //Essa combinação indica Branch
        case 0x1:
            //independe de function
            *alu_op = 0b0110;
            break;
        //Essa combinação indica Tipo-R
        case 0x2:
            //Tipo-R depende de function
            //Vamos realizar as diferentes combinações de function
            //function são os últimos 4 bits de IR
            switch(IR & 0x0f){
                break;
            }
        //Essa combinação indica Tipo-R
        case 0x3:
            //Tipo-R depende de function
            //Vamos realizar as diferentes combinações de function
            switch(IR & 0x0f){
                break;
            }
        default:
            return;
            break;
    }
}

```

Figura 4: código compactado da função da unidade de controle da ULA.

As funções para administrar as diferentes etapas de uma instrução específica encontram-se implementadas, e podem ser verificadas abrindo o arquivo code.c, elas fazem basicamente o que é descrito como sua funcionalidade, porém usando a lógica do C. Segue abaixo o exemplo da Instruction Fetch.

```

void instruction_fetch(short int sc, int PC, int ALUOUT, int IR, int* PCnew, int* IRnew, int* MDRnew)
{
    if(sc == ((short int)estado_0)){
        //coloca em IR o que tem no endereço de memória em PC
        *IRnew = memory[PC];

        //vamos primeiro pegar a operação da ula
        char alu_op;
        alu_control(IR, sc, &alu_op);

        //incrementa o PC usando a ula
        alu(PC, 1, alu_op, &ALUOUT, &zero, &overflow);
        *PCnew = ALUOUT;
        *MDRnew = memory[PC];
        if(*IRnew == 0){
            loop = 0;
            return;
        }
    }
    else{
        return;
    }
}

```

Figura 5: código completo da função de Instruction Fetch (IF).

3. Exemplo de entrada

Ao compilar e executar o arquivo nenhuma entrada do usuário é solicitada, o programa basicamente simula a funcionalidade de um MIPS, uma memória já pré definida e alguns registradores e executa as instruções baseado nas definições.

```
/* Memory. */
memory[0] = 0x8c480000; // opCode RS RT RD lw $t0, 0($v0) *5*
memory[1] = 0x010c182a; // 000000 01000 01100 00011 00000 101010 slt $v1, $t0, $t4 *4*
memory[2] = 0x106d0004; // 000100 00011 01101 0000000000000100 beq $v1, $t5, fim(4 palavras abaixo de PC+4) *3*
memory[3] = 0x01084020; // 000000 01000 01000 01000 00000 100000 add $t0, $t0, $t0 *4*
memory[4] = 0xac480000; // 101011 00010 01000 0000 0000 0000 0000 sw $t0, 0($v0) *4*
memory[5] = 0x004b1020; // 000000 00010 01011 0001 0000 0010 0000 add $v0, $t3, $v0 *4*
memory[6] = 0x08000000; // 000010 00000 00000 0000 0000 0000 0000 j inicio (palavra 0) *3*
memory[7] = 0; // fim (critério de parada do programa) (27*6)+(5+4+3)+1
memory[8] = 0;
memory[9] = 0;

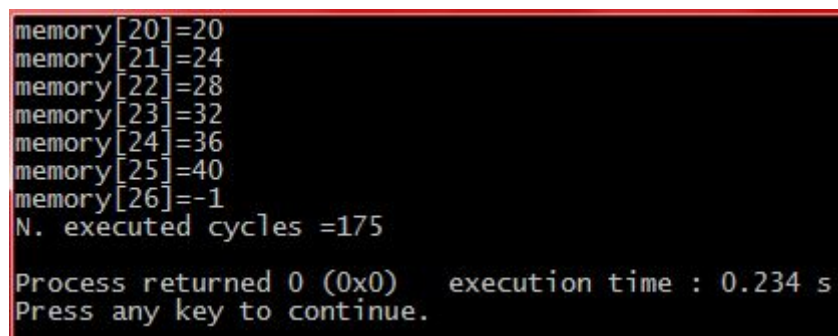
/* Data. */
memory[20] = 10;
memory[21] = 12;
memory[22] = 14;
memory[23] = 16;
memory[24] = 18;
memory[25] = 20;
memory[26] = -1;

/* Registers. */
reg[2] = 20; // $v0
reg[11] = 1; // $t3
reg[12] = 0; // $t4
reg[13] = 1; // $t5
```

Figura 6: configuração dos registradores e da memória no início do programa.

Note que os valores do reg[2] e reg[11] foram alterados, pois estavam incorretos no arquivo fornecido pelo professor.

Ao final da execução a seguinte informação é gravada no prompt.



```
memory[20]=20
memory[21]=24
memory[22]=28
memory[23]=32
memory[24]=36
memory[25]=40
memory[26]=-1
N. executed cycles =175
Process returned 0 (0x0)   execution time : 0.234 s
Press any key to continue.
```

Figura 7: tela com a saída final do programa.

4. Instruções de execução e compilação

Para rodar o programa, basta compilar-lo através de qualquer compilador e depois executá-lo, o programa é compatível tanto com Windows quanto com Linux.

5. Observações e conclusão

Dentro do proposto, é possível verificar que as exigências foram cumpridas, além disso, o intuito maior do trabalho, o qual era o aprendizado, foi de máximo aproveitamento, já que a prática ajuda a entender melhor os conceitos estudados em sala. É interessante frisar a mudança feita no arquivo `cpu.c`, nos valores dos registradores, os quais fazem mais sentido que os valores fornecidos no arquivo original, e permitem o correto funcionamento do projeto.