Edinburgh Napier UNIVERSITY

Coursework Report

Cool Student
4008000@napier.ac.uk
Edinburgh Napier University - Module Title (SET00000)

1 Introdução

Com o avanço da tecnologia os termos hardware e software se tornaram substantivos muito utilizados em nosso cotidiano, o hardware pode ser definido como equipamentos físicos que se aplica à dispositivos de entrada e saída, memórias, unidade central de processamento e entre outras. O software é a parte digital e basicamente a parte lógica (conjunto de instruções), que utilizam dos circuitos eletrônicos contidos nos hardware para dar fim a uma utilidade dita pelo ser humano. Como esses conceitos tecnológicos são de suma importância para estudantes de computação, no decorrer de sua formação acadêmica há um estudo bem aprofundado sobre esses e diversos outros. A disciplina Organização de Computadores I é uma das matérias cursadas no curso de ciência da computação que traz bem a fundo explicações em geral sobre a importância do hardware. O caminho de dados simplificado é um dos artefatos explicitados na disciplina e com intuito de nos proporcionar um melhor entendimento desses é que nosso Prof. José Nacif aplicou um trabalho pratico onde o objetivo principal foi implementar um processador MIPS utilizando os conhecimentos adquiridos em sala e estudos mais aprofundados extraclasse.

2 Desenvolvimento

Para a realização desse trabalho foi utilizado da linguagem de programação especifica para hardware verilog, de um circuito intregrado (FPGA) disposto pela universidade para uso acadêmico, de um embasamento teórico aprofundado na matéria "Caminho de Dados" e de um exemplo do caminho de dados (Figura 1) a ser seguido, exemplo esse que estava contido nas especificações do trabalho.

Por decisão de projeto, a implementação da nossa arquitetura MIPS foi realizada programando cada modulo do caminho de dados em arquivos separados e ao final um arquivo principal responsável por unir todos os arquivos em um sô. Logo após está fixada a imagem da visão em alto nível da nossa versão do MIPS (Figura 2).

2.1 Progam Counter (PC)

O modulo Progam Counter (PC) é responsável por receber como entrada o endereço da próxima instrução a ser executada e repassá-lo como saída, saída essa que será utilizada como entrada dos próximos módulos. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

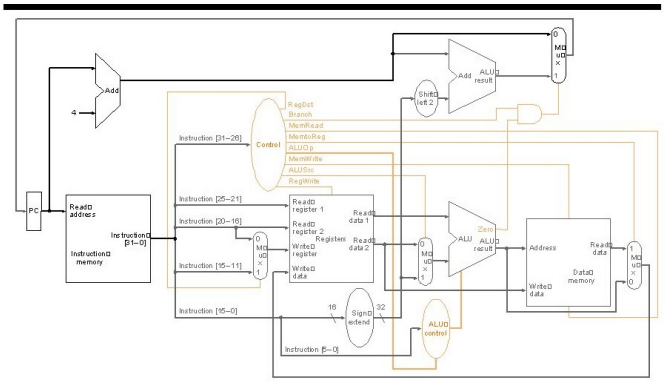

Figura 1: **Caminho de Dados** - Caminho de Dados Simplificado

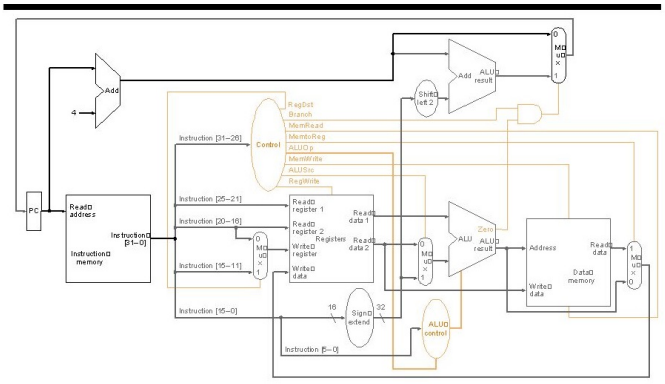

Figura 2: **Caminho de Dados Samuel/Vinicius** - Caminho de Dados Quartus

Listing 1: Progam Counter! em verilog

```
1   module pc(clk,endereco_pc,prox_endereco);
2     input clk;
3     input [31:0] prox_endereco;
4     output reg [31:0] endereco_pc;
5
6
7     initial begin
8       endereco_pc = 0 ;
9     end
10    always@(posedge clk ) begin
11      endereco_pc = prox_endereco;
12    end
13  endmodule
14
```

2.2 Somador Próximo Progam Counter

O modulo Somador Próximo Progam Counter é responsável por receber como entrada a saída do PC e somar +4 à essa entrada, sendo assim o resultado dessa soma é armazenado

na saída do modulo, saída essa que já é o endereço da próxima instrução. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

Listing 2: Somador Próximo Progam Counter! em verilog

```
module somador_pc (endereco_pc,clk,proximo_endereco);
    input [31:0] endereco_pc;
    input clk;
    output reg [31:0] proximo_endereco;
    always@(posedge clk) begin
        proximo_endereco = endereco_pc + 4;
    end
endmodule

```

2.3 Memória de Instrução

O modulo Memória de Instrução é responsável por receber como entrada a saída do PC e através dela selecionar qual instrução será carregada, após o carregamento a instrução é armazenada na saída de 32 bits do modulo, saída essa que será utilizada como entrada do modulo que monta a instrução em Opcode, RegRs, RegRt e etc. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

Listing 3: Memória de Instrução! em verilog

```
module memoria_de_instrucao(endereco, instrucao);
    input [31:0] endereco;
    output reg [31:0] instrucao;

    reg [31:0] memoria [0:31];
    initial begin
        memoria[0] = 32'b00000010000100010100000000100000;
    end

    always @(endereco) begin
        instrucao = memoria[endereco];
        if(endereco > 31) begin
            instrucao = memoria[31];
        end
    end
endmodule

```

2.4 Montador de Instrução

O modulo Montador de Instrução é responsável por receber como entrada a saída da memória de instrução e através dela separar a instrução em seus respectivos bits, Opcode bits 31:26, RegRs bits 25:21, RegRt bits 20:16, RegRd bits 15:11 e Endereço bits 15:0. Após a separação cada pedaço da instrução foram armazenada em variáveis de saída cada uma com seu respectivo tamanho. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

Listing 4: Montador de Instrução! em verilog

```
module monta_instrucao(instrucao, Op_code, Funct, Register_rs, Register_rt, Register_rd, Shamt, Endereco);
    input [31:0] instrucao;
    output [5:0] Op_code,
                 Funct;
    output [4:0] Register_rs,
                 Register_rt,
                 Register_rd,
                 Shamt;
    output [15:0] Endereco;

    assign  Op_code = instrucao[31:26];
    assign  Register_rs = instrucao[25:21];
    assign  Register_rt = instrucao[20:16];
    assign  Register_rd = instrucao[15:11];
    assign Shamt = instrucao[10:6];
    assign Funct = instrucao[5:0];
    assign Endereco = instrucao[15:0];
endmodule

```

2.5 Banco de Registradores

O modulo Banco de Registradores possui quatro entradas onde três delas são provindas das saídas do montador de instrução e a outra receberá a saída do multiplexador quando o sinal de controle RegDst for equivalente ao valor 1. O banco de registradores é responsável por atribuir valores aos registradores, valores esses que são buscados na memoria do banco a partir do valor das entradas, após obtido o valor de cada registrador eles são armazenados nas duas saídas do modulo. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

Listing 5: Banco de Registradores! em verilog

```
module banco_registrador(RegWrite ,Numero_Reg1, Numero_Reg2, Numero_Reg_Escrita, Dado_escrita, clk, Valor_Reg1, Valor_Reg2);
    input clk, RegWrite;
    input [4:0] Numero_Reg1, Numero_Reg2, Numero_Reg_Escrita;
    input [31:0] Dado_escrita;
    output reg [31:0]Valor_Reg1;
    output reg [31:0]Valor_Reg2;

    reg [31:0] registradores [0:31];
    initial begin
        registradores[0] = 0;
        registradores[1] = 5;
        registradores[2] = 15;
        registradores[3] = 20;
        registradores[4] = 25;
        registradores[5] = 10;
        registradores[6] = 2;
        registradores[7] = 12;
        registradores[8] = 16;
        registradores[9] = 30;
        registradores[10] = 24;
        registradores[11] = 22;
        registradores[12] = 32;
        registradores[13] = 35;
        registradores[14] = 40;
        registradores[15] = 3;
        registradores[16] = 4;
        registradores[17] = 5;
        registradores[18] = 6;
        registradores[19] = 46;
        registradores[20] = 44;
        registradores[21] = 45;
        registradores[22] = 50;
        registradores[23] = 52;
        registradores[24] = 54;
        registradores[25] = 56;
        registradores[26] = 58;
        registradores[27] = 60;
        registradores[28] = 62;
        registradores[29] = 64;
        registradores[30] = 66;
        registradores[31] = 68;
    end

    always @ (posedge clk)begin
        if(RegWrite == 1)begin
            registradores[Numero_Reg_Escrita] = Dado_escrita;
        end
    end

    always@(Numero_Reg1 or Numero_Reg2 or registradores[Numero_Reg1] or registradores[Numero_Reg2])begin
        Valor_Reg1 = registradores[Numero_Reg1];
        Valor_Reg2 = registradores[Numero_Reg2];
    end

```

```
55    endmodule
56
```

2.6 ALU

O modulo ALU possui três entradas onde duas delas são provindas da saída do banco de registradores e a outra resultante da saída da ALUcontrol. A ALU é responsável por realizar as operações necessárias a que foi programada, nesse caso operações de soma, subtração, and, or e entre outras. A entrada que recebe a saída da ALUcontrol é a responsável por dizer qual tipo de operação a ALU irá executar enquanto as outras duas entradas resultantes da saída do banco são os operandos. Exemplos do código produzido em verilog, pseudocódigo e alto nível logo a baixo:

Listing 6: ALU! em verilog

```
1   module alu (data1,saida_mux_registrador,saida_alu_control,zero,alu_resultado);
2       input[31:0] data1 ;
3       input [31:0]saida_mux_registrador;
4       input [3:0]saida_alu_control;
5       output reg [31:0] alu_resultado;
6       output reg zero;
7       always@(data1 or saida_mux_registrador or saida_alu_control) begin
8           if( saida_alu_control == 'b0000) begin
9               zero = 0;
10              alu_resultado = (data1 & saida_mux_registrador);
11          end
12          if(saida_alu_control == 'b0001) begin
13              zero = 0;
14              alu_resultado = (data1 | saida_mux_registrador);
15          end
16          if(saida_alu_control == 'b0010) begin
17              zero = 0;
18              alu_resultado = (data1 + saida_mux_registrador);
19          end
20          if(saida_alu_control == 'b0110) begin
21              alu_resultado = (data1 - saida_mux_registrador);
22              if (alu_resultado == 0 ) begin
23                  zero = 1;
24              end
25          end
26          if(saida_alu_control == 'b0111) begin
27              alu_resultado = 1;
28              if(data1<saida_mux_registrador) begin
29              alu_resultado = 1;
30              zero = 0;
31              end
32          end
33          if(saida_alu_control == 'b1100) begin
34              alu_resultado = ~(data1 | saida_mux_registrador);
35              zero = 0;
36          end
37      end
38  endmodule
39
```

3 Formatting

Some common formatting you may need uses these commands for **Bold Text**, *Italics*, and underlined.

3.1 LineBreaks

Here is a line

Here is a line followed by a double line break. This line is only one line break down from the above, Notice that latex can ignore this

We can force a break
with the break operator.

3.2 Maths

Embedding Maths is Latex's bread and butter

$$J = \left[\frac{\delta e}{\delta \theta_0}\frac{\delta e}{\delta \theta_1}\frac{\delta e}{\delta \theta_2}\right] = e_{current} - e_{target}$$

3.3 Code Listing

You can load segments of code from a file, or embed them directly.

Listing 7: Hello World! in c++

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello World!" << std::endl;
5     std::cin.get();
6     return 0;
7 }
```

3.4 PseudoCode

for $i = 0$ **to** 100 **do**
    print_number = true;
    if *i is divisible by 3* **then**
        print "Fizz";
        print_number = false;
    end
    if *i is divisible by 5* **then**
        print "Buzz";
        print_number = false;
    end
    if *print_number* **then**
        print i;
    end
    print a newline;
end

Algorithm 1: FizzBuzz

4 Conclusion

Referências