

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 integrado (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:

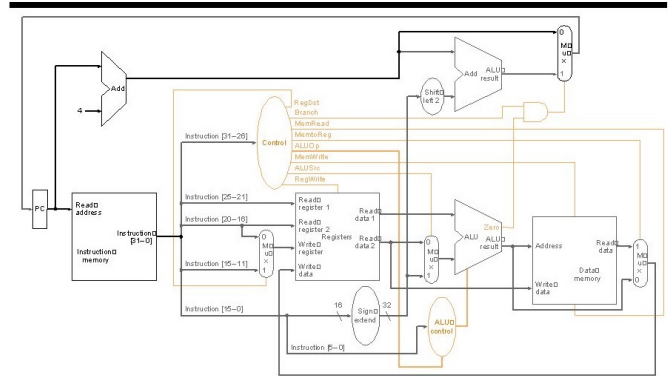


Figura 1: Caminho de Dados - Caminho de Dados Simplificado

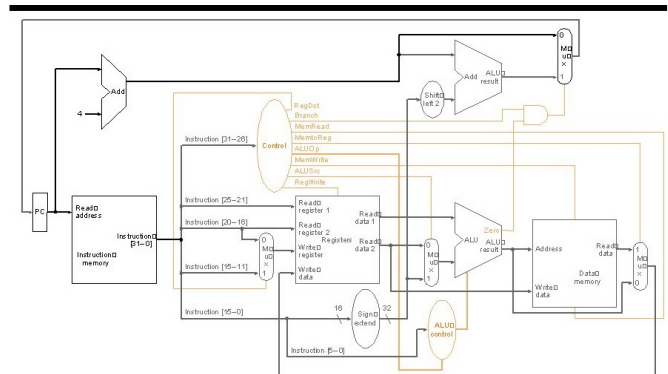


Figura 2: Caminho de Dados Samuel/Vinicius - Caminho de Dados Quartus

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   initial begin
7     endereco_pc = 0 ;
8   end
9   always@(posedge clk ) begin
10    endereco_pc = prox_endereco;
11  end
12 endmodule
13
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:

```
15 assign Shamt = instrucao[10:6];
16 assign Funct = instrucao[5:0];
17 assign Endereco = instrucao[15:0];
18 endmodule
19
```

Listing 2: Somador Próximo Progam Counter! em verilog

```
1 module somador_pc (endereco_pc,clk,proximo_endereco);
2   input [31:0] endereco_pc;
3   input clk;
4   output reg [31:0] proximo_endereco;
5   always@(posedge clk) begin
6     proximo_endereco = endereco_pc + 4;
7   end
8 endmodule
9
```

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

```
1 module memoria_de_instrucao(endereco, instrucao);
2   input [31:0] endereco;
3   output reg [31:0] instrucao;
4
5   reg [31:0] memoria [0:31];
6   initial begin
7     memoria[0] = 32'h000000010000101000000000100000;
8   end
9
10  always @(endereco) begin
11    instrucao = memoria[endereco];
12    if(endereco > 31) begin
13      instrucao = memoria[31];
14    end
15  end
16 endmodule
17
```

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 bit 31:26, RegRs bit 25:21, RegRt bit 20:16, RegRd bit 15:11 e Endereço bit 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

```
1 module monta_instrucao(instrucao, Op_code, Funct, Register_rs, Register_rt, Register_rd, Shamt, Endereco);
2   input [31:0] instrucao;
3   output [5:0] Op_code,
4     Funct;
5   output [4:0] Register_rs,
6     Register_rt,
7     Register_rd,
8     Shamt;
9   output [15:0] Endereco;
10
11  assign Op_code = instrucao[31:26];
12  assign Register_rs = instrucao[25:21];
13  assign Register_rt = instrucao[20:16];
14  assign Register_rd = instrucao[15:11];
```

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 5: 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