

# **Caminho de Dados Simplificado do RISC-V**

## **Trabalho Prático 02 - CCF 252**

**Gabriel Sanches Vinhal de Queiroz - 3748<sup>1</sup>, Vinícius Augusto Assis Ferreira - 4242<sup>2</sup>**

<sup>1</sup>Instituto de Ciências Exatas e Tecnológicas – Universidade Federal de Viçosa(UFV) - Florestal  
Rodovia LMG 818, km 06 – 35.690-000 – Florestal – MG – Brazil

<sup>2</sup>Ciência da Computação – Universidade Federal de Viçosa - UFV  
Florestal, M.G.

`gabriel.sanches@ufv.br, vinicius.assis@ufv.br`

**Abstract.** *This work consists of the implementation in synthesizable verilog language of a simplified version of the RISC-V data path that can execute the ADD, SUB, AND, OR, LD, SD, BEQ, LW, SW, XOR, ADDI, SLL, and BNE instructions. This code will also be used to implement on a DE2-115 FPGA (Cyclone IV, Altera).*

**Resumo.** *Este trabalho consiste na implementação em linguagem verilog sintetizável de uma versão simplificada do caminho de dados do RISC-V que consiga executar as instruções ADD, SUB, AND, OR, LD, SD, BEQ, LW, SW, XOR, ADDI, SLL e BNE. Esse código também será usado para implementar em uma FPGA DE2-115 (Cyclone IV, Altera).*

### **1. Introdução**

A arquitetura RISC-V é uma abordagem moderna e aberta no campo da ciência da computação. Como uma Arquitetura de Conjunto de Instruções (ISA) flexível e simplificada, ela tem ganhado popularidade crescente. Ao seguir os princípios da Computação de Conjunto de Instruções Reduzido (RISC), o RISC-V utiliza um conjunto de instruções simples e um design eficiente, facilitando o desenvolvimento de processadores. Uma das características mais notáveis do RISC-V é sua natureza de código aberto. Com especificações e documentação disponíveis gratuitamente, essa arquitetura promove a colaboração e a inovação.

Em RISC-V, o caminho de dados é uma estrutura composta por unidades lógico-aritméticas responsáveis por executar diversas instruções. Essas unidades realizam operações como adição, subtração, desvios, leitura e escrita de dados, entre outras. Essas funções são essenciais para o processamento e execução de instruções dentro de um sistema computacional. Neste trabalho, foi pedido para utilizar o caminho de dados simplificado, como podemos ver logo abaixo.

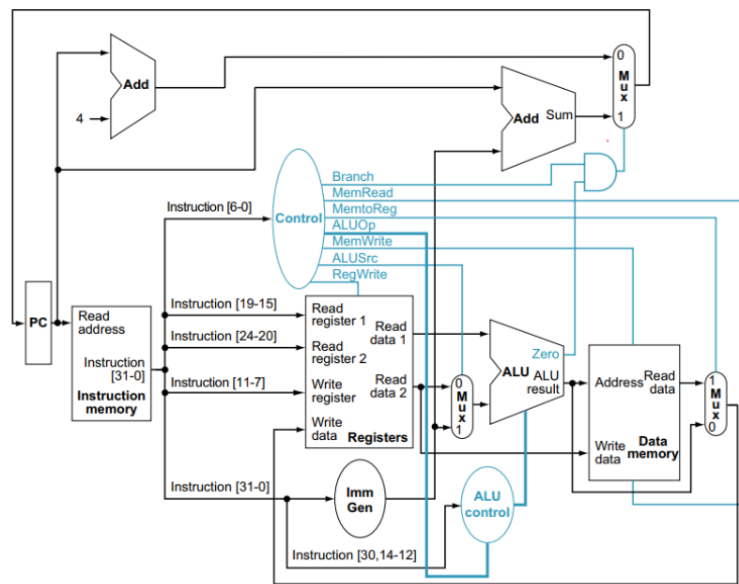
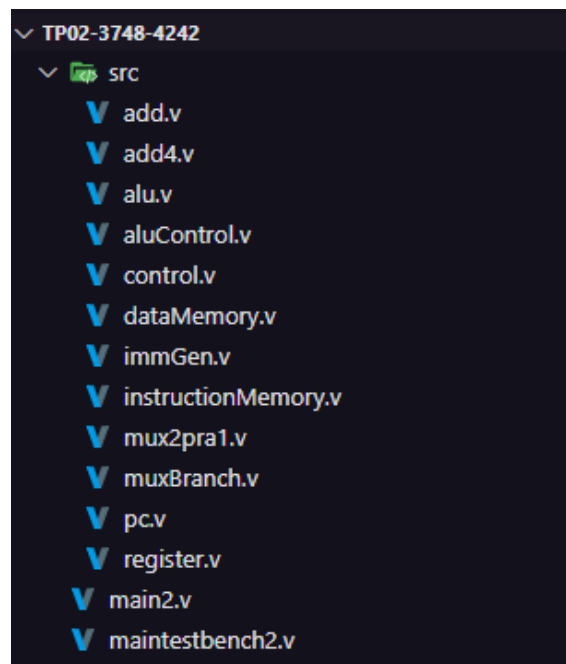


Figure 1. Caminho de dados simplificado

## 2. Desenvolvimento

### 2.1. Organização

Os arquivos ficaram organizados da seguinte forma:



**Figure 2. Estrutura dos arquivos**

Módulos do código:

### **2.1.1. Program Counter (PC)**

O program counter é um registro que contém o endereço da instrução atual que está sendo executada. Após a execução de uma instrução, o PC é atualizado para apontar para o próximo endereço de memória onde a próxima instrução está armazenada.

```

src > V pcv
You, há 23 horas | 1 author (You)
1  module pc (
2      input wire clock,
3      input wire reset,
4      input wire [31:0] enderecoEntrada,
5      output reg [31:0] enderecoSaida);
6
7      always @(posedge clock) begin
8
9          if (reset) begin
10             enderecoSaida = 31'd0;
11         end
12         else begin
13             enderecoSaida = enderecoEntrada;
14         end
15     end
16
17 endmodule

```

Figure 3. Módulo PC

### 2.1.2. add4

O "add4" executa operações aritméticas para incrementar o PC para buscar a próxima instrução.

```

src > V add4.v
You, há 33 minutos | 1 author (You)
1  module add4(
2      input wire clock,
3      input wire reset,
4      input wire [31:0] saidaPC,
5      output reg [31:0] saida
6  );
7
8      always @(posedge clock) begin
9          if (reset) begin
10             saida = 32'b0;
11         end
12         else begin
13             saida = saidaPC + 32'b1;
14         end
15     end
16 endmodule

```

Figure 4. Módulo add4

### 2.1.3. add

O "add" executa operações aritméticas somando dois operandos, o primeiro é endereço do PC e o outro é o imediato gerado.

```

src > V add.v
You, há 2 minutos | 1 author (You)
1  module add(
2      input wire clock,
3      input reset,
4      input wire [31:0] saidaPC,
5      input wire [31:0] saidaImmGen,
6      output reg [31:0] saida
7  );
8      wire [31:0] mudancaIMM;
9
10     assign mudancaIMM = (saidaImmGen <<< 1);
11
12     always @(posedge clock) begin
13         if (reset) begin
14             saida = 32'b0;
15         end
16         else begin
17             saida = saidaPC + (mudancaIMM);
18         end
19     end
20 endmodule

```

Figure 5. Módulo add

#### 2.1.4. Tipos de Multiplexadores implementados

O "mux2pra1" e o "muxBranch" representa um multiplexador que seleciona uma das várias entradas com base em sinais de controle. No caminho de dados, um MUX é usado para selecionar a entrada apropriada para diferentes estágios do caminho de dados, como a seleção entre o PC e o endereço de destino da ramificação.

```

src > V mux2pra1.v
You, há 23 horas | 1 author (You)
1  module mux2pra1(
2      input wire clock,
3      input wire [31:0] entrada1,
4      input wire [31:0] entrada2,
5      input wire controle,
6      output reg [31:0] saida);
7
8      always @(posedge clock) begin
9
10         if (controle == 0) begin
11             saida <= entrada1;
12         end
13
14         if (controle == 1) begin
15             saida <= entrada2;
16         end
17     end
18 end
19
20 endmodule

```

Figure 6. Módulo mux2pra1

```

src > V muxBranch.v
You, ontem | 1 author (You)
1  module muxBranch(
2      input wire clock,
3      input wire [31:0] entradaADD4,
4      input wire [31:0] entradaADD,
5      input wire branch,
6      input wire saidaZeroALU,
7      output reg [31:0] saida
8  );
9      reg resutado;
10
11     always @(posedge clock) begin
12         resutado = 1'b0; // You, ontem * refator
13         if(branch == 1 && saidaZeroALU == 1)begin
14             resutado = 1'b1;
15         end
16         if (resutado == 0) begin
17             saida = entradaADD4;
18         end
19         if (resutado == 1) begin
20             saida = entradaADD;
21         end
22     end
23 endmodule
24

```

Figure 7. Módulo muxBranch

### 2.1.5. intruccionMemory

O "instructionMemory" é responsável por armazena as instruções do programa. Normalmente, ela é quem fornece as instruções ao processador com base no valor do PC.

```

src > V intruccionMemory.v
You, ontem | 2 authors (vscudermaz001 and others)
1  module intruccionMemory(
2      input wire clock,
3      input wire reset,
4      input wire [31:0] endereco,
5      output reg [31:0] Instrucao);
6
7      reg [31:0] Instrucoes [0:31];
8
9      always @(posedge clock) begin
10         if (reset) begin
11             // Instruções:
12             Instrucoes[0] <- 32'h000000000000100110000000000010011; // ADDI x1, x0, 100
13             Instrucoes[1] <- 32'h000000000000100011000000000010011; // ADDI x4, x0, 200
14             Instrucoes[2] <- 32'h000000000000100011000000000010011; // ADDI x5, x0, 300
15             Instrucoes[3] <- 32'h0000000000000100001100000000000010011; // ADDI x6, x0, 400
16             Instrucoes[4] <- 32'h000000000000000110000000000010011; // ADDI x7, x3, x4
17             Instrucoes[5] <- 32'h010000010000000010000000000010011; // SUB x4, x3, x4
18             Instrucoes[6] <- 32'h000000011001000010000000000010011; // AND x3, x5, x2
19             Instrucoes[7] <- 32'h000000011010000100000000000010011; // OR x10, x5, x2
20             Instrucoes[8] <- 32'h000000000000000010000000000010011; // SW x3, 0(x0)
21             Instrucoes[9] <- 32'h000000000000000010000000000010011; // SW x0, 0(x0)
22             Instrucoes[10] <- 32'h000000000010001100000000000010011; // LW x7, 0(x0)
23             Instrucoes[11] <- 32'h0000000010001100000000000010011; // LW x8, 0(x0)
24             Instrucoes[12] <- 32'h000000000000101100000000000010011; // XOR x6, x3, x5
25             Instrucoes[13] <- 32'h000000000101000010000000000010011; // ADD x5, x5, x2
26             Instrucoes[14] <- 32'h000000000100000100000000000010011; // ADDI x7, x5, x2
27             Instrucoes[15] <- 32'h110000011000000100000000000010011; // BNE x7, x14, 12
28             Instrucoes[16] <- 32'h00000000000000000000000000000000; // Instrução vazia
29             Instrucoes[17] <- 32'h00000000000000000000000000000000; // Instrução vazia
30             Instrucoes[18] <- 32'h00000000000000000000000000000000; // Instrução vazia
31             Instrucoes[19] <- 32'h00000000000000000000000000000000; // Instrução vazia
32             Instrucoes[20] <- 32'h00000000000000000000000000000000; // Instrução vazia
33             Instrucoes[21] <- 32'h00000000000000000000000000000000; // Instrução vazia
34             Instrucoes[22] <- 32'h00000000000000000000000000000000; // Instrução vazia
35             Instrucoes[23] <- 32'h00000000000000000000000000000000; // Instrução vazia
36             Instrucoes[24] <- 32'h00000000000000000000000000000000; // Instrução vazia
37             Instrucoes[25] <- 32'h00000000000000000000000000000000; // Instrução vazia
38             Instrucoes[26] <- 32'h00000000000000000000000000000000; // Instrução vazia
39             Instrucoes[27] <- 32'h00000000000000000000000000000000; // Instrução vazia
40             Instrucoes[28] <- 32'h00000000000000000000000000000000; // Instrução vazia
41             Instrucoes[29] <- 32'h00000000000000000000000000000000; // Instrução vazia
42             Instrucoes[30] <- 32'h00000000000000000000000000000000; // Instrução vazia
43             Instrucoes[31] <- 32'h00000000000000000000000000000000; // Instrução vazia
44
45         end else begin
46             Instrucao = Instrucoes[endereco];
47         end
48     end
49 endmodule
50

```

Figure 8. Módulo instructionMemory

### 2.1.6. ImmGen

O modulo "immGen" extrai o valor imediato da instrução. Normalmente, ele é usado em instruções como ramificação, carregamento e armazenamento.

```

src > V immGen.v
You, ontem | 1 author (You)
1  module immGen (
2      input wire clock,
3      input wire [31:0] entrada, // erro
4      output reg signed [31:0] saida
5  );
6      always @(posedge clock) begin
7          if (entrada[6:0] == 7'b0000011) begin // lw
8              saida = {{21{entrada[31]}}, entrada[31:20]}; // You, ontem • refatoração do
9              saida = saida / 4;
10         end
11         if (entrada[6:0] == 7'b0100011) begin // sw
12             saida = {{21{entrada[31]}}, entrada[31:25], entrada[11:7]};
13             saida = saida / 4;
14         end
15         if (entrada[6:0] == 7'b0010011) begin // addi
16             saida = {{20{entrada[31]}}, entrada[31:20]};
17         end
18         if (entrada[6:0] == 7'b1100011) begin // beq
19             saida = {{21{entrada[31]}}, entrada[7], entrada[30:25], entrada[11:8]};
20         end
21     end
22 endmodule
23

```

Figure 9. Módulo ImmGen

### 2.1.7. control

O módulo "control" é responsável por gerar sinais que controlam o funcionamento dos componentes do caminho de dados de um processador. Ele recebe como entrada o código de operação (opcode) da instrução atual e produz sinais de controle que ativam ou desativam componentes específicos no caminho de dados.

Esses sinais de controle são essenciais para coordenar as operações dentro do processador. Eles indicam quais componentes devem estar ativos ou inativos em cada ciclo de clock, garantindo que a instrução correta seja executada e que os recursos necessários sejam utilizados adequadamente.

O módulo "control" desempenha um papel fundamental na sincronização e coordenação do processador, permitindo que as instruções sejam executadas corretamente e controlando o fluxo de dados através dos diferentes componentes do caminho de dados.

```

1 module control(
2     input wire clock,
3     input [6:0] opcodeDaInstrucao,
4     output reg ALUSrc,
5     output reg MemToReg,
6     output reg RegWrite,
7     output reg MemRead,
8     output reg MemWrite,
9     output reg Branch,
10    output reg [1:0] ALUOp);
11
12    //LD, SD, LW, SW, BNE
13    always @(posedge clock) begin
14        case (opcodeDaInstrucao)
15            7'b0110011: begin //R-type - ADD, SUB
16                ALUSrc <= 0;
17                MemToReg <= 0;
18                RegWrite <= 1;
19                MemRead <= 0;
20                MemWrite <= 0;
21                Branch <= 0;
22                ALUOp <= 2'b10;
23            end
24
25            7'b0000011: begin //LOAD
26                ALUSrc <= 1;
27                MemToReg <= 1;
28                RegWrite <= 1;
29                MemRead <= 1;
30                MemWrite <= 0;
31                Branch <= 0;
32                ALUOp <= 2'b00;
33            end
34
35            7'b0100011: begin //STORE
36                ALUSrc <= 1;
37                MemToReg <= 0;
38                RegWrite <= 0;
39                MemRead <= 0;
40                MemWrite <= 1;
41                Branch <= 0;
42                ALUOp <= 2'b00;
43            end
44
45            7'b1100011: begin //JALR
46                ALUSrc <= 0;
47                MemToReg <= 0;
48                RegWrite <= 0;
49                MemRead <= 0;
50                MemWrite <= 0;
51                Branch <= 1;
52                ALUOp <= 2'b01;
53            end
54
55            7'b0010011: begin //Tipo-1 - ADDI
56                ALUSrc <= 1;
57                MemToReg <= 0;
58                RegWrite <= 1;
59                MemRead <= 0;
60                MemWrite <= 0;
61                Branch <= 0;
62                ALUOp <= 2'b11;
63            end
64
65            default: begin
66                ALUSrc <= -1;
67                MemToReg <= -1;
68                RegWrite <= 0;
69                MemRead <= 0;
70                MemWrite <= 0;
71                Branch <= 1;
72                ALUOp <= 2'b11;
73            end
74        endcase
75    end
76 endmodule

```

Figure 10. Módulo control

### 2.1.8. Register

O módulo register vai armazenar os dados durante a execução das instruções dentro dos registradores. Eles fornecem acesso rápido aos operandos para operações aritméticas e lógicas. No caminho de dados, os registradores são normalmente implementados como uma matriz e inicializada com zeros.



```

1 module register(
2     input wire clock,
3     input wire regWrite,
4     input wire [4:0] rs1,
5     input wire [4:0] rs2,
6     input wire [4:0] writeRegister,
7     input wire [31:0] writeData,
8     output reg [31:0] readData1,
9     output reg [31:0] readData2);
10
11     integer i;
12     reg [31:0] registradores [0:31];
13
14     initial begin
15
16         for (i = 0; i < 32; i = i + 1) begin
17             registradores[i] <= 32'h0;
18         end
19     end
20
21     always @(posedge clock) begin
22
23         begin
24             readData1 = registradores[rs1];
25             readData2 = registradores[rs2];
26
27             if (regWrite == 1'b1 && writeRegister != 0) begin
28                 registradores[writeRegister] = writeData;
29             end
30         end
31     end
32 endmodule

```

Figure 11. Módulo register

### 2.1.9. aluControl

O módulo "aluControl" é responsável por gerar sinais de controle que determinam a operação a ser realizada pela Unidade de Lógica Aritmética (ALU). Ele recebe como entrada o código de função da instrução e produz sinais de controle que especificam qual operação aritmética ou lógica deve ser executada, como adição, subtração, operações lógicas E (AND), OU (OR), entre outras.

Esses sinais de controle são cruciais para configurar corretamente a ALU e direcionar suas operações de acordo com a instrução atual. Eles indicam à ALU qual operação específica deve ser realizada, garantindo que a instrução seja executada corretamente e produza o resultado desejado.

O módulo "aluControl" desempenha um papel importante na coordenação e no controle das operações aritméticas e lógicas dentro do processador, permitindo que diferentes instruções sejam executadas de acordo com suas respectivas operações específicas.

```

1 module aluControl(
2     input wire clock,
3     input wire [1:0] ALUOp,
4     input wire [6:0] funct7,
5     input wire [2:0] funct3,
6     output reg [3:0] saida);
7
8     always @(posedge clock) begin
9         case ({ALUOp, funct7, funct3})
10             12'b0000000000: saida <= 4'b0010; //add - load and Store - ld, lw, sd, sw
11             12'b1000000000: saida <= 4'b0010; //add
12             12'b1100000000: saida <= 4'b0010; //add
13             12'b1001000000: saida <= 4'b0110; //sub
14             12'b1100000000: saida <= 4'b0110; //beq
15             12'b1000000111: saida <= 4'b0000; //and
16             12'b1000000110: saida <= 4'b0001; //or
17             12'b1000000001: saida <= 4'b0111; //bne
18             12'b1000000000: saida <= 4'b0111; //addi
19             12'b1000000000: saida <= 4'b0111; //addi
20             12'b0000000001: saida <= 4'b0000; //sll
21         endcase
22     end
23 endmodule

```

Figure 12. Módulo aluControl

### 2.1.10. ALU

A ALU é responsável pela execução de operações aritméticas e lógicas nos operandos. Ela recebe a entrada do arquivo de register ou do ImmGen e executa a operação especificada com base nos sinais de controle gerados pelo controle da ALU.

```
src > V alu.v
You, ontem | 1 author (You)
1  module alu(
2      input wire clock,
3      input wire [3:0] ALUcontrol,
4      input wire [31:0] entrada1,
5      input wire [31:0] entrada2,
6      output reg [31:0] saida,
7      output reg Zero);
8
9
10 always @(posedge clock) begin
11
12     case (ALUcontrol)
13
14         4'b0000: begin
15             saida <= entrada1 & entrada2;
16             Zero = 1'b0;
17         end
18
19         4'b0001: begin
20             saida <= entrada1 | entrada2;
21             Zero = 1'b0;
22         end
23
24         4'b0010: begin
25             saida <= entrada1 + entrada2;
26             Zero = 1'b0;
27         end
28
29         4'b0110: begin
30             saida <= entrada1 - entrada2;
31
32             if (saida == 0) begin
33                 Zero = 1'b1;
34             end else begin
35                 Zero = 1'b0;
36             end
37         end
38     endcase
39 end
40
41 endmodule
```

Figure 13. Módulo alu

### 2.1.11. DataMemory

O módulo "dataMemory" é responsável por armazenar os dados que são acessados por instruções de leitura e escrita. Ele oferece operações de leitura e escrita para transferir dados entre o processador e a memória. O endereço utilizado para acessar a memória de dados geralmente é fornecido pela ALU (Unidade Lógica Aritmética) ou pelo gerador imediato.

Esse módulo atua como uma interface entre o processador e a memória, permitindo que o processador leia dados da memória ou escreva dados na memória conforme necessário. Quando um endereço de memória e um sinal de controle são recebidos,

o módulo "dataMemory" realiza as operações adequadas para recuperar ou armazenar os dados na memória.

```
src > V dataMemory.v
You, há 2 horas | 1 author (You)
1  module dataMemory(
2      input wire clock,
3      input reset,
4      input wire MemWrite,
5      input wire MemRead,
6      input wire [31:0] enderecoDeEntrada,
7      input wire [31:0] writeData,
8      output reg [31:0] readData;
9
10     reg [31:0] dadosMemoria [0:31];
11
12     always @(posedge clock) begin
13
14         if(reset) begin
15             You, há 2 horas * ajuste main ...
16             for (integer i = 0; i < 32; i = i + 1) begin
17                 dadosMemoria[i] <= 32'd0;
18             end
19         end else begin
20
21             if(MemWrite) begin
22                 dadosMemoria[enderecoDeEntrada] = writeData;
23             end
24
25             if(MemRead) begin
26                 readData = dadosMemoria[enderecoDeEntrada];
27             end
28         end
29     end
30 end
31 endmodule
```

Figure 14. Módulo dataMemory

Esses componentes trabalham em conjunto no caminho de dados de um processador RISC-V para executar instruções. O PC (Program Counter) mantém o endereço da próxima instrução a ser buscada na memória de instruções. A Unidade de Controle decodifica a instrução buscada, gerando sinais de controle que são enviados para os outros componentes. O gerador imediato extrai o valor imediato presente na instrução, enquanto a ALU (Unidade Lógica Aritmética) executa operações aritméticas ou lógicas com base nos sinais de controle fornecidos pela Unidade de Controle da ALU. Os registradores armazenam operandos e resultados intermediários, e a memória de dados é utilizada para armazenar e recuperar dados conforme necessário.

De forma geral, o caminho de dados forma o núcleo central para a execução de instruções em um processador RISC-V, fornecendo os componentes essenciais para buscar, decodificar, executar e armazenar os resultados das instruções. Cada componente desempenha um papel específico, contribuindo para o funcionamento geral do processador.

### 3. Instruções de execução

Para a execução do programa, foi criado um makefile para facilitar, sendo importante ressaltar que é pré-requisito ter o "Icarus Verilog" instalado na máquina. Para rodar o programa, basta digitar o comando no terminal: *"make all"* e, logo em seguida digitar o comando *"make run"*.

#### **4. Conclusão**

Em conclusão, a arquitetura RISC-V é uma arquitetura de conjunto de instruções moderna e aberta que oferece simplicidade, flexibilidade e personalização. Sua natureza de código aberto e escalabilidade a tornam uma opção atraente para estudantes e profissionais interessados em arquitetura de computadores. Esse trabalho também permitiu conhecimento prático dos estudos dados em aula.

#### **5. Referências**

Nesse trabalho prático foi usado como suporte bibliográfico o livro base da disciplina de Organização de Computadores I, o [Patterson and Hennessy 2021].

#### **References**

Patterson, D. A. and Hennessy, J. L. (2021). *Computer Organization and Design - RISC-V edition*. Patterson-Hennessy, 2<sup>a</sup> edição edition.