

# Implementação de uma Versão simplificada do caminho de dados do RISC-V.

Letícia Cristina A. Silva<sup>1</sup>, Melissa Alanis S. Oliveira<sup>1</sup>

<sup>1</sup>Ciência da Computação – Universidade Federal de Viçosa (UFV)  
Florestal – MG – Brazil

leticia.c.silva@ufv.br, melissa.alanis@ufv.br

## 1. Introdução

A presente documentação aborda questões acerca da **implementação simplificada do caminho de dados do processador RISC-V**, sendo à equipe designado um subconjunto de instruções, as quais serão implementadas ao longo deste Trabalho Prático - tanto em simulação, quanto físico. O principal objetivo deste documento é relatar o percurso das instruções ao perpassarem o caminho de dados, implementado em **Verilog**, e serem, assim, sintetizados no **FPGA DE2-115 (Cyclone IV, Altera)**.

## 2. Desenvolvimento dos Módulos

Para a implementação do caminho de dados, a equipe baseou-se no modelo fornecido na especificação do trabalho, com os respectivos **módulos** e o **controle**. A partir do recebimento de um **arquivo.asm (instrucoes.asm)**, tem-se a leitura das instruções, já transformadas em binário com o auxílio do **primeiro Trabalho Prático**, e o processamento destas no caminho de dados. Além disso, também foram utilizados **arquivos.asm** para inicializar o banco de registradores (**registradores.asm**) e a memória de dados (**dados.asm**).

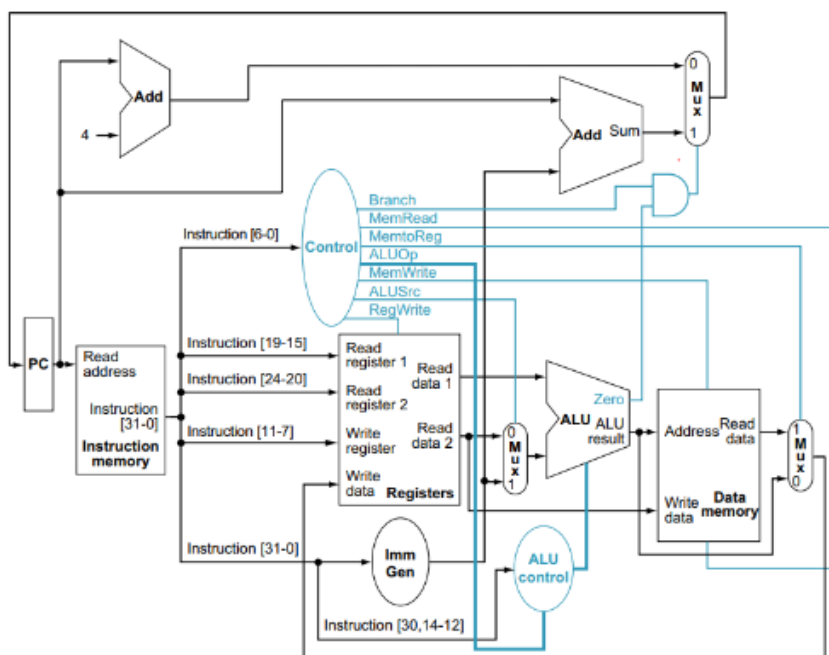


Figure 1. Modelo do Caminho de Dados utilizado

Com relação às instruções, por mais que as padronizadas pelo modelo da especificação sejam add, sub, and, or, lw, sw e beq; a equipe recebeu o seguinte conjunto para ser trabalhado: **lh, sh, add, or, andi, sll, bne**. Vale salientar que o fato de algumas das instruções recebidas pelo grupo (**Half-words de 16 bits**) serem diferentes das "padrões" de 32 bits influencia diretamente no tratamento destas na **Memória de Dados**. A seguir, tem-se a imagem de uma tabela que contém a descrição das respectivas funções de acordo com os seus formatos - que variam entre os **formatos R, I e S**.

Formato S:

SH	imm[11:5]	rs2	rs1	funct3: 001	imm[4:0]	opcode: 0100011
BNE	imm[11:5]	rs2	rs1	funct3: 001	imm[4:0]	opcode: 1100011

Formato I:

LH	imm[11:0]	rs1	funct3: 001	rd	opcode: 0000011
ANDI	imm[11:0]	rs1	funct3: 111	rd	opcode: 0010011

Formato R:

ADD	funct7: 0000000	rs2	rs1	funct3: 000	rd	opcode: 0110011
OR	funct7: 0000000	rs2	rs1	funct3: 110	rd	opcode: 0110011
SLL	funct7: 0000000	rs2	rs1	funct3: 001	rd	opcode: 0110011

Figure 2. Instruções recebidas pela equipe

## 2.1. Módulo Program Counter (PC)

O PC constitui-se como um dos elementos cruciais do processador RISC-V, sendo este o responsável pelo armazenamento do endereço da próxima instrução que será buscada na memória de instrução e, ao ser executada, cruzará o caminho de dados. Neste módulo, o PC realiza um tipo de "retroalimentação", em que o endereço de saída, após passar pelo cálculo da próxima instrução, sempre retorna como o endereço do pc, tornando o fluxo do caminho de dados, assim, contínuo.

```

1 module PC(
2     input clock,
3     input reset,
4     input wire [31:0] endereco_pc, // Endereco que vem do PC
5     output reg [31:0] endereco_saida // Vem do PC+4 ou PC+imm
6 );
7
8 always @(posedge clock) begin

```

```

9     if (reset) begin
10         endereco_saida <= 32'b0;
11     end
12     else begin
13         endereco_saida <= endereco_pc;
14     end
15 end
16 endmodule

```

## 2.2. Módulo Memória de Instrução

A memória de instrução armazena todo o **conjunto de instruções**, uma em cada posição da memória, que cruzarão o caminho de dados, uma a uma. É nela que o endereço vindo Program Counter (PC) procurará pela instrução correspondente, sendo esta estará pronta para ser executada. Partindo para o seu módulo abaixo, observa-se que, a depender do seu tipo, cada instrução é armazenada na memória de maneira distinta, de maneira a obedecer os padrões do RISC-V para que os demais módulos operem corretamente ao manipular estes endereços.

```

1  module MemInstrucoes (
2      input wire [31:0] endereco,    // Endereco da instrucao a ser lida
3      output reg [31:0] instrucao_saida, // Instrucao lida
4      output reg [4:0] rs1,           // Registrador de origem 1
5      output reg [4:0] rs2,           // Registrador de origem 2
6      output reg [4:0] rd,            // Registrador de destino
7      output reg [6:0] opcode,        // OpCode da instrucao
8      output reg [2:0] funct3,        // Campo funct3 (3 bits)
9      output reg [6:0] funct7         // Campo funct7 (7 bits)
10 );
11
12     reg [31:0] memoria_instrucoes [0:50];
13
14     // Atribuicao da instrucao ao endereco calculado do pc
15     always @(endereco) begin
16         instrucao_saida <= memoria_instrucoes[endereco/4];
17
18         opcode <= instrucao_saida[6:0];    // Campos opcode
19         funct3 <= instrucao_saida[14:12];  // Campos funct3 (3 bits)
20
21         // Decodificando as instrucoes
22         case (opcode)
23             7'b0110011: begin // Tipo R: ADD, OR e SLL
24                 funct7 <= instrucao_saida[31:25]; // Funct7 (7 bits)
25                 rs1 <= instrucao_saida[19:15]; // Reg 1 (5 bits)
26                 rs2 <= instrucao_saida[24:20]; // Reg 2 (5 bits)
27                 rd <= instrucao_saida[11:7]; // Reg de destino (5
28                     bits)
29             end
30             7'b0010011: begin // Tipo I: ANDI
31                 rs1 <= instrucao_saida[19:15]; // 5 bits para o
32                     registrador fonte
33                 rd <= instrucao_saida[11:7]; // 5 bits para o
34                     registrador de destino
35             end
36         endcase
37     end

```

```

34
35
36         7'b0000011: begin // Tipo I: LH
37             rs1    <= instrucao_saida[19:15]; // Reg 1 (5 bits)
38             rd     <= instrucao_saida[11:7];  // Reg de destino (5
39                 bits)
40         end
41
42         7'b0100011: begin // Tipo S: SH
43             rs1 <= instrucao_saida[19:15]; // Reg 1 (5 bits)
44             rs2 <= instrucao_saida[11:7]; // Reg 2 (5 bits)
45         end
46
47         7'b1100011: begin // Tipo B: BNE
48             rs1    <= instrucao_saida[19:15]; // Reg 1 (5 bits)
49             rs2    <= instrucao_saida[24:20]; // Reg 2 (5 bits)
50         end
51     endcase
52 end
53
54 endmodule

```

### 2.3. Módulo Banco de Registradores

É neste módulo que estão armazenados todos os **32 registradores** que compõem as instruções, com os seus respectivos valores, vindos do arquivo registradores.asm, que serão manipulados ao longo dos percursos no caminho de dados. A cada instrução lida, o registrador recebe respectivamente o seus valores de **leitura e escrita**, a depender da operação, e retorna, após buscar no banco, os registradores que serão manipulados na **ALU**. Vale salientar que o dado de escrita será um input do módulo quando o sinal de escrita (que vem do controle e sinaliza quando um dado não é de leitura) estiver ativo (1) e o registrador de destino (rd) for diferente de 0. Portanto, o banco de registradores é um vetor de 32 posições que armazena os registradores, com seus respectivos 32 bits também.

```

1  module BRegistradores(
2      input clock,
3      input reg_escrita,           // Sinal de controle para escrita
4      input [4:0] endereco_regd,  // Registrador de destino
5      input [4:0] endereco_reg1,  // Registrador de origem 1
6      input [4:0] endereco_reg2,  // Registrador de origem 2
7      input [31:0] dado_escrita,  // Dado a ser escrito
8      output reg [31:0] valor_reg1, // Valor do registrador de origem 1
9      output reg [31:0] valor_reg2 // Valor do registrador de origem 2
10 );
11
12     reg [31:0] registradores[0:31]; // Vetor com 32 registradores de 32
13         bits
14
15     always @(posedge clock) begin
16         //$display("regd: %b", endereco_regd);
17
18         if (reg_escrita && endereco_regd != 0) begin //Registrador 0 eh
19             somente de leitura, portanto nao eh possivel escrever nele

```

```

18     registradores[endereco_regd] = dado_escrita;
19     end
20     // $display("dado: %b", dado_escrita);
21 end
22
23 always @(endereco_reg1, registradores[endereco_reg1]) begin
24     valor_reg1 = registradores[endereco_reg1];
25     // $display("leitura: %b dado:", endereco_reg1, valor_reg1);
26 end
27 always @(endereco_reg2, registradores[endereco_reg2]) begin
28     valor_reg2 = registradores[endereco_reg2];
29     // $display("leitura: %b dado:", endereco_reg2, valor_reg2);
30 end
31
32 endmodule

```

## 2.4. Módulo ImmGen

Neste módulo importante para determinadas funções, faz-se necessária a **extensão de sinal** no valor do imediato para que este possa ser utilizado corretamente com os respectivos **32 bits** nas operações da ALU ou, no caso do **BNE**, para que este seja somado com a instrução e o desvio possa ir para o Program Counter (PC) com o endereço certo do rótulo. Como o imediato tem **12 bits**, o trecho de código abaixo o lê, de acordo com a sua respectiva instrução, e estende o **bit mais significativo** do número binário em 20 bits, ou seja, o imediato passa a ter **32 bits**, sem perder o seu valor e, principalmente, o seu sinal.

```

1 module ImmGen (
2     input [31:0] instrucao,
3     input [6:0] opcode,
4     output reg [31:0] imm_estendido
5 );
6     always @(*) begin
7         case(opcode)
8             7'b0010011: // Tipo I: ANDI
9                 imm_estendido = {{20{instrucao[31]}}, instrucao
10                    [31:20]};
11             7'b0000011: // Tipo I: LH
12                 imm_estendido = {{20{instrucao[31]}}, instrucao
13                    [31:20]};
14             7'b0100011: // Tipo S: SH
15                 imm_estendido = {{20{instrucao[31]}}, instrucao[31:25],
16                    instrucao[11:7]};
17             7'b1100011: // Tipo B: BNE
18                 imm_estendido = {{19{instrucao[31]}}, instrucao[7],
19                    instrucao[30:25], instrucao[11:8], 1'b0};
20             default: imm_estendido = 32'b0; // Valor padrao para
21                 casos desconhecidos
22         endcase
23     end
24 endmodule

```

## 2.5. Módulo ALU

No módulo da **Unidade Lógica Aritmética** (ALU), por sua vez, tem-se a realização das operações aritméticas e lógicas relacionadas às instruções - como soma, subtração, and,

or, entre outros. Dessa maneira, a partir do recebimento de dois valores, que podem ser estes dois registradores ou apenas um registrador e um imediato, a ALU realiza uma operação. Tal comando depende do seu módulo de controle, que neste trabalho é o **ALU-Control**, pois este interpreta e combina o sinal **ALUop** (baseado no opcode) - que vem do controle com base na instrução que está sendo executada - e a **Funct3**, gerando um sinal que especifica exatamente à Unidade Lógica Aritmética a operação que ela deve fazer.

- Módulo ALU

```
1  module ALU (  
2      input clock,  
3      input wire [3:0] resultado_alu_control,  
4      input wire [31:0] valor1,  
5      input wire [31:0] valor2,  
6      output reg [31:0] resultado_alu,  
7      output reg resultado_desvio //Retorna 1 se os numeros forem  
          diferentes  
8  );  
9  
10     always @(negedge clock) begin  
11         resultado_desvio <= 0;  
12         case(resultado_alu_control)  
13             4'b0000: begin  
14                 resultado_alu <= valor1 & valor2; // andi  
15             end  
16             4'b0001: resultado_alu <= valor1 | valor2; // or  
17             4'b0010: resultado_alu <= valor1 + valor2; // add, sh e lh  
18             4'b0011: resultado_alu <= valor1 << valor2[4:0]; // sll  
19             4'b0110: begin  
20                 resultado_alu <= valor1 - valor2; // bne  
21             end  
22             if (resultado_alu != 0) //Retorna 1 se os numeros forem  
                diferentes  
23                 resultado_desvio <= 1;  
24             else  
25                 resultado_desvio <= 0;  
26         end  
27         default: resultado_alu <= 32'b0;  
28     endcase  
29 end  
30 endmodule
```

- Módulo ALUControl

```
1  /*ALUop:  
2  00 - para acesso a memoria  
3  01 - bne  
4  10 - para tipo R  
5  11 - ANDI  
6  
7  ALU Control  
8  0000 - AND  
9  0001 - OR  
10 0010 - SOMA  
11 0011 - DESLOCAMENTO
```

```

12 0110 - SUBTRACAO
13 */
14
15 module ALUControl (
16     input wire [1:0] ALUOp, //Vem do controle, ajuda a descobrir qual
        tipo de instrucao sera executada
17     input wire [2:0] funct3, // Campo funct3 (3 bits)
18     output reg [3:0] operacao_selecionada // Operacao que a ALU vai
        realizar
19 );
20
21 always @(*) begin
22     case (ALUOp)
23         2'b00: operacao_selecionada = 4'b0010; // lh e sh (SOMA)
24         2'b01: operacao_selecionada = 4'b0110; // bne (SUBTRACAO)
25         2'b10: begin
26             case(funct3)
27                 3'b110: operacao_selecionada = 4'b0001; // or (OR)
28                 3'b000: operacao_selecionada = 4'b0010; //add (SOMA)
29                 3'b001: operacao_selecionada = 4'b0011; //sll (
                    DESLOCAMENTO)
30             endcase
31         end
32         2'b11: operacao_selecionada = 4'b0000; // andi (AND)
33         default: operacao_selecionada = 4'b1111; // Valor desconhecido
34     endcase
35 end
36 endmodule

```

Caso a instrução seja um desvio, que no conjunto fornecido à equipe é um **BNE** (branch if not equal), a ALU realiza uma operação de subtração em que, se esta resultar em um valor diferente de 0 (pois o zero refletiria na igualdade dos dois valores), tem-se uma alteração no caminho de execução do programa por meio do cálculo do desvio, sendo este um **AND** entre a **branch** (1) e o **resultado desvio** (1 quando os valores forem diferentes), o resultado dessa operação determina qual endereço será enviado para o PC.

## 2.6. Módulo Memória de Dados

Este módulo se constitui como essencial, principalmente para o funcionamento das instruções **LH (Load Halfword)** e **SH (Store Halfword)**, que precisam acessá-lo para armazenar ou carregar os dados. Ao receber o endereço resultante da ALU, ou o dado de escrita - caso uma operação de escrita esteja sendo realizada - o módulo toma uma decisão baseada nos sinais de controle que foram conjuntamente enviados à memória de dados. Caso o sinal de escrita seja 1 (verdadeiro), o dado de escrita é armazenado no endereço indicado (Store); já o sinal de leitura, quando 1, retorna o dado presente no endereço fornecido para que seja lido (Load). Destaca-se que um multiplexador, executado posteriormente à memória de dados, decidirá, com base no sinal de controle **MemToReg**, se o que será escrito no banco de registradores é o resultado da ALU ou o dado lido da memória.

```

1     module MemDados (
2         input wire clock,
3         input wire [31:0] endereco,          // Endereco de memoria

```

```

4   input wire [31:0] valor_reg2,      // Dado a ser escrito
5   input wire sinal_escrita,         // Sinal de controle para escrita
6   input wire sinal_leitura,         // Sinal de controle para leitura
7   output reg [31:0] dado_saida      // Dado lido da memoria
8 );
9   reg [31:0] memoria_dados [0:63]; // 64 enderecos de 32 bits
10  reg [5:0] temp;                   // Endereco de memoria ajustado (
    dividido por 4)
11  reg byte;                          // Determina o byte a ser escrito/
    lido
12
13  // Ajuste do endereco e determinacao do byte a ser acessado
14  always @(*) begin
15      temp = endereco >> 2;         // Endereco de memoria (dividido por 4
    para alinhamento de palavras)
16      byte = endereco[1];           // Determina o byte de 0 ou 1
17  end
18
19  // Leitura da memoria
20  always @(*) begin
21      if (sinal_leitura) begin
22
23          if (byte == 1'b0) begin
24              dado_saida <= {{16{memoria_dados[temp][15]}},
    memoria_dados[temp][15:0]}; // Leitura dos bytes 0
    e 1
25          end else begin
26              dado_saida <= {{16{memoria_dados[temp][31]}},
    memoria_dados[temp][31:16]}; // Leitura dos bytes 2
    e 3
27          end
28      end
29  end
30
31  // Escrita na memoria
32  always @(negedge clock) begin
33
34      if (sinal_escrita) begin
35          if (byte == 1'b0) begin
36              memoria_dados[temp][15:0] <= valor_reg2[15:0]; //
    Escrita nos bytes 0 e 1
37          end else begin
38              memoria_dados[temp][15:0] <= valor_reg2[15:0]; //
    Escrita nos bytes 2 e 3
39          end
40      end
41  end
42 endmodule

```

## 2.7. Módulo Controle

O módulo de controle é o grande responsável pelo gerenciamento dos sinais que estão sendo enviados no momento em que as instruções perpassam o caminho de dados. Dessa forma, a partir do opcode delas, é possível definir todas as operações as quais a instrução deve passar, a fim de que ela seja executada da maneira correta. Percebe-se, no código



abaixo, que os sinais são enviados da maneira supracitada.

```
1  module Controle (
2      input wire [6:0] opcode,
3      input wire [6:0] funct7,
4      input wire [2:0] funct3,
5      output reg [1:0] ALUop,           // Sinal de controle da ALU
6      output reg sinal_leitura,        // Sinal de leitura da memoria
7      output reg sinal_escrita,        // Sinal de escrita na memoria
8      output reg reg_escrita,          // Sinal de escrita no
          registrador
9      output reg ALUSrc, // Se eh imediato ou reg2 que vai entrar na alu
10     output reg branch, // Se a instrucao eh um desvio
11     output reg MemToReg // Se os dados sao da memoria ou da alu
12 );
13
14     always @(opcode) begin
15         case (opcode)
16             7'b0000011: begin // Load (lh)
17                 ALUop <= 2'b00; // Acesso a memoria
18                 sinal_leitura <= 1;
19                 sinal_escrita <= 0;
20                 reg_escrita <= 1;
21                 ALUSrc <= 0; //AVALIAR
22                 branch <= 0;
23                 MemToReg <= 1;
24             end
25             7'b0100011: begin // Store (sh)
26                 ALUop <= 2'b00; // Acesso a memoria
27                 sinal_leitura <= 0;
28                 sinal_escrita <= 1;
29                 reg_escrita <= 0;
30                 ALUSrc <= 1;
31                 branch <= 0;
32                 MemToReg <= 0;
33             end
34             7'b0110011: begin // Tipo R
35                 ALUop <= 2'b10; // OR, ADD e SLL
36                 sinal_leitura <= 0;
37                 sinal_escrita <= 0;
38                 reg_escrita <= 1;
39                 ALUSrc <= 0;
40                 branch <= 0;
41                 MemToReg <= 0;
42             end
43             7'b0010011: begin // Tipo I
44                 ALUop <= 2'b11; // ANDI
45                 sinal_leitura <= 0;
46                 sinal_escrita <= 0;
47                 reg_escrita <= 1;
48                 ALUSrc <= 1;
49                 branch <= 0;
50                 MemToReg <= 0;
51             end
52             7'b1100011: begin // Resultado de desvio (bne)
53                 ALUop <= 2'b01; // bne (subtracao para comparacao)
```

```

54         sinal_leitura <= 0;
55         sinal_escrita <= 0;
56         reg_escrita <= 0;
57         ALUSrc <= 0;
58         branch <= 1;
59         MemToReg <= 0;
60     end
61     default: begin //Caso entre um valor indefinido, tudo sera
62         0
63         ALUop <= 2'b00;
64         sinal_leitura <= 0;
65         sinal_escrita <= 0;
66         reg_escrita <= 0;
67         ALUSrc <= 0;
68         branch <= 0;
69         MemToReg <= 0;
70     end
71 endcase
72 end
endmodule

```

## 2.8. Módulo Processador

Por fim, o módulo do Processador realiza a integração de todos os demais módulos do trabalho prático, bem como a declaração dos sinais a serem utilizados nas instâncias, o clock e o reset.

```

1  module Processador(
2      input clock,
3      input reset
4  );
5
6
7      // Declara o de sinais
8      wire [31:0] endereco_pc;
9      wire [31:0] endereco_soma4;
10     wire [31:0] endereco_desvio;
11     wire [31:0] valor_regd;
12     wire [31:0] valor_reg1;
13     wire [31:0] valor_reg2;
14     wire [31:0] valor2;
15     wire [31:0] resultado_alu;
16     wire [31:0] dado_saida;
17     wire [31:0] imm_estendido;
18     wire [31:0] endereco;
19     wire [31:0] dado_escrita;
20     wire [31:0] endereco_saida;
21     wire [31:0] instrucao_saida;
22     wire [4:0] rd;
23     wire [3:0] operacao_selecionada;
24     wire [4:0] endereco_reg1;
25     wire [4:0] endereco_reg2;
26     wire [4:0] endereco_regd;
27     wire [6:0] opcode;
28     wire [2:0] funct3;
29     wire [6:0] funct7;

```

```

30 //wire [11:0] imediato;
31 wire [1:0] ALUOp;
32 wire sinal_leitura;
33 wire branch;
34 wire sinal_escrita;
35 wire reg_escrita;
36 wire ALUSrc;
37 wire resultado_desvio;
38 wire MemToReg;
39 wire sinal_mux;
40
41
42
43 assign sinal_mux = resultado_desvio & branch;
44 assign endereco_desvio = endereco_saida + imm_estendido;
45 assign endereco_soma4 = endereco_saida + 32'h4;
46
47 // Instancia do modulo PC
48 PC pc_inst (
49     .clock(clock),
50     .reset(reset),
51     .endereco_pc(endereco_pc),
52     .endereco_saida(endereco_saida)
53 );
54
55 // Instancia do modulo MemInstrucoes
56 MemInstrucoes mem_inst (
57     .endereco(endereco_saida),
58     .instrucao_saida(instrucao_saida),
59     .rs1(endereco_reg1),
60     .rs2(endereco_reg2),
61     .rd(endereco_regd),
62     .opcode(opcode),
63     .funct3(funct3),
64     .funct7(funct7)
65 );
66
67 // Instancia do modulo BRegistradores
68 BRegistradores reg_inst (
69     .clock(clock),
70     .reg_escrita(reg_escrita),
71     .endereco_regd(endereco_regd),
72     .endereco_reg1(endereco_reg1),
73     .endereco_reg2(endereco_reg2),
74     .dado_escrita(dado_escrita),
75     .valor_reg1(valor_reg1),
76     .valor_reg2(valor_reg2)
77 );
78
79 ImmGen immgen_inst (
80     .instrucao(instrucao_saida
81     ),
82     .imm_estendido(imm_estendido),
83     .opcode(opcode)
84 );
85

```

```

86
87 // Instancia do modulo ALU
88 ALU alu_inst (
89     .clock(clock),
90     .resultado_alu_control(operacao_selecionada),
91     .valor1(valor_reg1),
92     .valor2(valor2),
93     .resultado_alu(resultado_alu),
94     .resultado_desvio(resultado_desvio)
95 );
96
97 // Instancia do modulo MemDados
98 MemDados memdados_inst (
99     .clock(clock),
100     .endereco(resultado_alu),
101     .valor_reg2(valor_reg2),
102     .sinal_escrita(sinal_escrita),
103     .sinal_leitura(sinal_leitura),
104     .dado_saida(dado_saida)
105 );
106
107 // Instancia do modulo ALUControl
108 ALUControl alucontrol_inst (
109     .ALUop(ALUop),
110     .funct3(funct3),
111     .operacao_selecionada(operacao_selecionada)
112 );
113
114 // Instancia do modulo Controle
115 Controle controle_inst (
116     .opcode(opcode),
117     .funct7(funct7),
118     .funct3(funct3),
119     .ALUop(ALUop),
120     .sinal_leitura(sinal_leitura),
121     .sinal_escrita(sinal_escrita),
122     .reg_escrita(reg_escrita),
123     .ALUSrc(ALUSrc),
124     .branch(branch),
125     .MemToReg(MemToReg)
126 );
127
128
129 // Instancia do mux
130 mux muxpc_inst (
131     .valor1(endereco_soma4),
132     .valor2(endereco_desvio),
133     .sinal_mux(sinal_mux),
134     .endereco_saida(endereco_pc)
135 );
136
137 // Instancia do mux
138 mux muxalu_inst (
139     .valor1(valor_reg2),
140     .valor2(imm_estendido),
141     .sinal_mux(ALUSrc),

```

```

142     .endereco_saida(valor2)
143 );
144 // Instancia do mux
145 mux muxmemreg_inst (
146     .valor1(resultado_alu),
147     .valor2(dado_saida),
148     .sinal_mux(MemToReg),
149     .endereco_saida(dado_escrita)
150 );
151 );
152 endmodule

```

### 3. Resultados e Conclusão

O caminho de dados está funcionando exatamente como esperado para todas as instruções atribuídas ao grupo (17), gerando as saídas esperadas tanto para o banco de registradores, quanto para a memória de dados. Abaixo realizamos alguns teste com todas as instruções implementadas:

- Teste 1: Foram testas as instruções add, or, sll e andi, sendo que inicialmente os registradores utilizados possuíam os seguintes valores:

```

add x3, x2, x2
sll x4, x1, x2
or x5, x3, x1
andi x6, x1, 3

```

Figure 3. Instruções executadas no teste 1

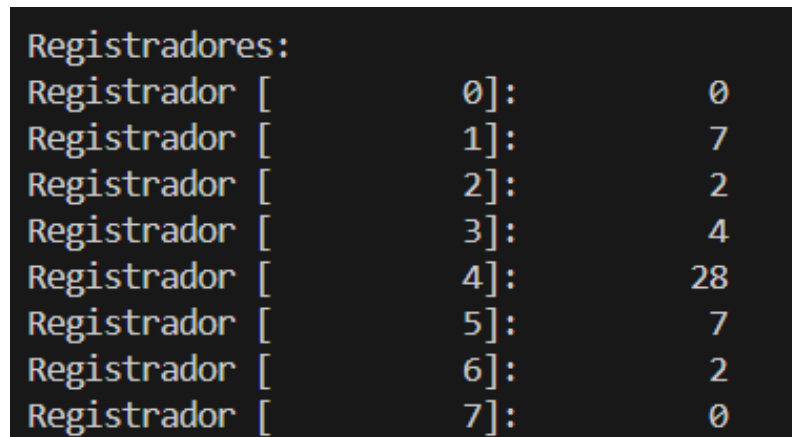
```

0. 00000000000000000000000000000000
1. 00000000000000000000000000000011
2. 00000000000000000000000000000010
3. 00000000000000000000000000000000
4. 00000000000000000000000000000000
5. 00000000000000000000000000000000
6. 00000000000000000000000000000000
7. 00000000000000000000000000000000

```

Figure 4. Registradores antes do teste 1

Após executar as instruções os resultados obtidos foram exatamente os esperados:

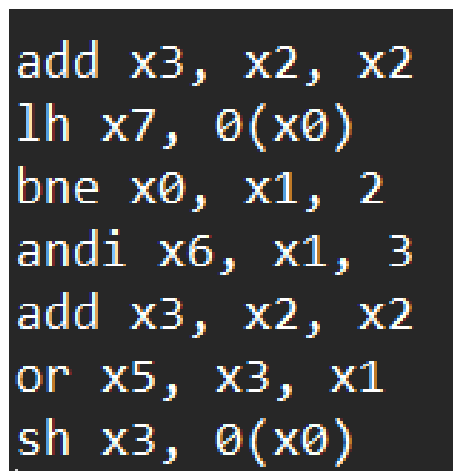


```
Registradores:
Registrador [    0]:    0
Registrador [    1]:    7
Registrador [    2]:    2
Registrador [    3]:    4
Registrador [    4]:   28
Registrador [    5]:    7
Registrador [    6]:    2
Registrador [    7]:    0
```

A screenshot of a terminal window with a dark background and light-colored text. It displays the state of eight registers, labeled 'Registrador' followed by an index in brackets, and their corresponding values. The values are: 0, 7, 2, 4, 28, 7, 2, and 0.

**Figure 5. Registradores após o teste 1**

- Teste 2: Foram testadas as instruções add, or, lh, sh, bne e andi, sendo que inicialmente os registradores utilizados possuíam os mesmos valores do teste anterior, e o valor localizado na posição 0 da memória de dados é 1111 (15):



```
add x3, x2, x2
lh x7, 0(x0)
bne x0, x1, 2
andi x6, x1, 3
add x3, x2, x2
or x5, x3, x1
sh x3, 0(x0)
```

A screenshot of assembly code on a dark background. The code consists of seven instructions: 'add x3, x2, x2', 'lh x7, 0(x0)', 'bne x0, x1, 2', 'andi x6, x1, 3', 'add x3, x2, x2', 'or x5, x3, x1', and 'sh x3, 0(x0)'.

**Figure 6. Instruções executadas no teste 2**

Após executar as instruções os resultados obtidos foram exatamente os esperados, tanto no banco de registrador, quanto na memória de dados:

Registradores:		
Registrador [	0]:	0
Registrador [	1]:	7
Registrador [	2]:	4
Registrador [	3]:	4
Registrador [	4]:	0
Registrador [	5]:	0
Registrador [	6]:	0
Registrador [	7]:	15
Registrador [	8]:	0
Registrador [	9]:	0
Registrador [	10]:	0

Figure 7. Registradores após o teste 2

Memória de dados:		
Dados [	0] =	4
Dados [	1] =	0
Dados [	2] =	0
Dados [	3] =	0
Dados [	4] =	0
Dados [	5] =	0

Figure 8. Memória de dados após o teste 2

Por fim, conclui-se que a implementação do caminho de dados do RISC-V foi um sucesso e os objetivos estabelecidos na especificação do trabalho prático foram compreendidos e atendidos corretamente pela equipe.

#### 4. Referências

1. cristinaleticia/**TPII-OC: Implementação Caminho de Dados RISC-V**. Disponível em: <https://github.com/cristinaleticia/TPII-OCI> Acesso em: 11 ago. 2024.
2. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2ª Edição, 2020.
3. Sathira Basnayake (18 may 2020). **Simple 8-bit Processor Design and Verilog implementation (Part 1)**. <https://studentsxstudents.com/simple-8-bit-processor-design-and-verilog-implementation-part-1-8735fac284b>, [Acesso em 11 ago].
4. Francisquini, R. (22 jun 2018). **Projetando um Processador Simples em Verilog** - Rodrigo Francisquini - Medium. <https://medium.com/@francisquini/projetando-um-processador-simples-em-verilog-ea1b67f36da2>, [Acesso em 11 ago].