

Trabalho Prático 2 - Organização Computacional: Caminho de Dados RISC-V (Versão Simplificada)

MATHEUS JÚNIO DA SILVA¹
YURI ROBERTO²

Discentes de Ciência da Computação do Instituto de Ciências Exatas e Tecnológicas da Universidade Federal de Viçosa - Campus Florestal

¹matheus.junio@ufv.br

²yuri.roberto@ufv.br

1 Resumo

Neste trabalho prático, desenvolvemos uma versão simplificada de um caminho de dados, incluindo um subconjunto de instruções seguindo a arquitetura RISC-V. Nossa documentação foi feita em formato SBC, usando Latex. Desenvolvemos um testbench, disponibilizamos o código no GitHub e nos preparamos para a entrevista.

Palavras-chave: Caminho de dados, RISC-V

2 Abstract

In this practical work, we develop a simplified version of a data path, including a subset of instructions following the RISC-V architecture. Our documentation was done in SBC format, using Latex. We developed a testbench, made the code available on GitHub and prepared for the interview.

Palavras-chave: Caminho de dados, RISC-V

3 Introdução

Este trabalho prático faz parte do curso de Organização de Computadores I e tem como objetivo a implementação de um caminho de dados simplificado para a arquitetura RISC-V. Optamos por desenvolvê-lo utilizando Verilog, uma linguagem de descrição de hardware, que permite a modelagem e simulação de circuitos digitais. O projeto inclui a criação de um testbench para simular a execução das instruções implementadas, que são: ADD, SUB, AND, OR, LD, SD e BEQ, além de fornecer documentação detalhada em formato SBC, com imagens da execução e do desenvolvimento do código.

4 Projeto

A ideia principal do projeto foi implementar um caminho de dados simplificado para a arquitetura RISC-V, subdividindo as instruções em componentes para facilitar a simulação e execução.

Cada instrução é recebida e analisada, onde o nome da instrução é utilizado para determinar o fluxo de controle e as operações a serem realizadas. Para cada tipo de instrução, são necessárias manipulações específicas para gerenciar os dados, incluindo a configuração dos registradores e a memória, utilizando funções auxiliares que tratam as particularidades de cada operação.

5 Explicações do Código

Abaixo estão listadas as funções principais e suas finalidades no código(Dê zoom para ver os códigos com maior nitidez).

- ◊ **mux:** Neste módulo, entrada1 e entrada2 são entradas de 32 bits, out é a saída de 32 bits e controle é um sinal de controle. A saída out é determinada com base no valor de controle: se controle for igual a 0, a saída será igual a entrada1; caso contrário, será igual a entrada2.

```
module modMux2Entrada(
    entrada1,
    entrada2,
    out,
    controle
);

    input [31:0] entrada1,
    entrada2;

    output [31:0] out;

    input controle;

    assign out =
        (controle == 1'b0)
        ? entrada1 : entrada2;

endmodule
```

- ◊ **adicionar:** Adicionar implementa a operação de adição de dois números de 32 bits. Ele possui três portas: x, y e saída. As entradas x e y são números inteiros de 32 bits, e a saída saída também é um número inteiro de 32 bits. Dentro do bloco assign, a operação de adição é realizada: o valor da saída saída é calculado somando os valores de x e y. Em resumo, esse módulo é usado para somar dois números inteiros de 32 bits e produzir o resultado na saída

```
module adicionar(
    x,
    y,
    saida
);

    input[31:0]
    x,
    y;

    output[31:0] saida;

    assign saida = x + y;
endmodule
```

- ◊ **memoriaDeInstrucao**: Implementa uma memória de instrução que lê dados de um arquivo com base no endereço de leitura fornecido. Se o endereço for desconhecido, a instrução é definida como zero e o arquivo é considerado finalizado. Caso contrário, o módulo lê a instrução do arquivo e a armazena. Ou seja, simula uma memória de instrução que busca dados de um arquivo.

```
module memoriaDeInstrucao(
    EnderecoDeLeitura,
    nomeArquivo,
    Instrucao,
    arquivoFinalDelete
);

    input [31:0] EnderecoDeLeitura;
    input [8*25:0] nomeArquivo;
    output reg [31:0] Instrucao;
    output reg arquivoFinalDelete;

    integer arquivo;
    integer error;
    reg [31:0] linha;

    always @(nomeArquivo)

    begin

        arquivo
        = $fopen(nomeArquivo, "r");

    end
```

- ◊ **geradorImediato**: Descreve dois módulos Verilog: G_imm_tb e G_imm. O módulo G_imm calcula o campo GIimm com base no valor da entrada instr, seguindo regras específicas para diferentes tipos de instruções (como lw, sw, addi e beq).

```
module G_imm(
    instr,
    GIimm
);

    input [31:0] instr;
    output reg [31:0] GIimm;

    always @(instr)
    begin
        case (instr[6:0])
            7'b0000011: GIimm
            <= {{2{instr[31]}},
                  instr[30:20]}; // lw - I

            7'b0100011: GIimm
            <= {{2{instr[31]}},
                  instr[30:25],
                  instr[11:7]}; // sw - S

            7'b0010011: GIimm
            <= {{2{instr[31]}},
                  instr[30:20]}; // addi - I

            7'b1000011: GIimm
            <= {{2{instr[31]}},
                  instr[7],
                  instr[30:25],
                  instr[11:8],
                  {1{1'b0}}}; // beq - SB

            default: GIimm <= 32'bx;
        endcase
    end
endmodule
```

- ◊ **control**: Control gera sinais de controle com base na instrução fornecida, seguindo regras específicas para diferentes tipos de operações (como lw, sw, addi, etc.).

```

module control(
    Instrucaocontrole,
    Branchcontrole,
    lermemControle,
    MemToRegC,
    controlealuop,
    memoriaDeEscritac,
    ALUSrc,
    registradorDeEscritaC
);

    input [6:0] Instrucaocontrole;

    output Branchcontrole,
    lermemControle,
    MemToRegC,
    memoriaDeEscritac,
    ALUSrc,
    registradorDeEscritaC;

    output [1:0] controlealuop;

```

- ◊ **controleDaAlu**: Descreve dois módulos Verilog: test_tb e controleDaAlu. O módulo controleDaAlu gera sinais de controle para a unidade de lógica aritmética (ALU) com base nas instruções fornecidas. Diferentes valores de aluOpAcontrole e funct3Acontrole correspondem a operações específicas, como adição, subtração, xor e srl.

```

module controleDaAlu(
    f3f7Acontrole,
    aluOpAcontrole,
    alucontroleAcontrole
);

    output reg [3:0] alucontroleAcontrole;

    input [1:0] aluOpAcontrole;

    input [3:0] f3f7Acontrole;

    wire funct7Acontrole;

    wire [2:0] funct3Acontrole;

    assign funct7Acontrole = f3f7Acontrole[3];

    assign funct3Acontrole = f3f7Acontrole[2:0];

    always @(*)
    begin
        case (aluOpAcontrole)
            2'b00: alucontroleAcontrole
            <= 4'b0010; // 00 - adicionar (addi, lw, sw)

```

- ◊ **alu**: Descreve dois módulos Verilog: alu_tb e alu. O módulo alu implementa uma unidade lógico-aritmética (ALU) de 32 bits. Ele realiza operações como adição, subtração, xor

e shift right (srl) com base no valor de ControleDeAlu. O resultado é armazenado em resultadoDaAlu, e o sinal zeroo indica se o resultado é zero.

```
module alu(
    aluentrada1,
    aluentrada2,
    ControleDeAlu,
    resultadoDaAlu,
    zeroo
);

    input [31:0]
    aluentrada1,
    aluentrada2;

    input [3:0] ControleDeAlu;

    output reg [31:0] resultadoDaAlu;

    output zeroo;

    assign zeroo = (resultadoDaAlu == 0) ? 1 : 0;

    always @(*)
    begin
        case (ControleDeAlu)
            4'b0010: resultadoDaAlu
            <= aluentrada1 + aluentrada2;// adicionar
```

- ◊ **memoriaDeDados:** Descreve dois módulos Verilog: teste_tb e memoriaDeDados. O módulo memoriaDeDados simula uma memória que pode ser lida ou escrita, com endereços e dados especificados

```
module memoriaDeDados(
    clock,
    lerMemoria,
    escreverNaMemoria,
    endereco,
    dadosEscrita,
    lerDados
);

    input clock,
    lerMemoria,
    escreverNaMemoria;

    input [31:0] endereco,
    dadosEscrita;

    output reg [31:0] lerDados;

    reg [31:0] memoriaDeDados [0:512];
```

- ◊ pc: Representa um contador de programa (PC). Ele possui uma entrada (entradaDoPc) que define o próximo valor do PC e uma saída (saídaDoPc) que contém o valor atual do PC. O sinal de relógio (clock) sincroniza as operações, e o sinal de reset (pcReset) zera o contador. Ou seja, esse módulo implementa um PC que avança ou é redefinido com base nos sinais de controle.

```
module pc(
    entradaDoPc,
    saídaDoPc,
    clock,
    pcReset
);

    input [31:0] entradaDoPc;
    input clock, pcReset;
    output [31:0] saídaDoPc;
    reg [31:0] saídaDoPc;

    always @(posedge clock)
        if (!pcReset) saídaDoPc <= 0;
        else saídaDoPc <= entradaDoPc;
endmodule
```

- ◊ memoriaRegistro: O bloco registerMem representa um banco de registradores. Ele possui 32 registradores de 32 bits cada. Durante a borda de subida do sinal de relógio, o registrador especificado por idRegistradorEscrita é atualizado com o valor de dadoWr. Os valores dos registradores especificados por lerRegistrador1 e lerRegistrador2 são disponibilizados nas saídas dadoLeitura e dadoLeitura2. Ou seja, esse módulo gerencia a leitura e escrita nos registradores.

```
module registerMem_tb;
    reg _clock, _registradorDeEscrita;
    reg [4:0] _lerRegistrador1,
    _lerRegistrador2,
    _idRegistradorEscrita;
    reg [31:0] _dadoWr;
    wire [31:0] _dadoLeitura,
    _dadoLeitura2;
    registerMem rm(
        _clock,
        _registradorDeEscrita,
        _lerRegistrador1,
        _lerRegistrador2,
        _idRegistradorEscrita,
        _dadoWr,
        _dadoLeitura,
        _dadoLeitura2
    );
    initial begin
```

- ◊ **caminhoDeDados**: Gerencia um banco de registradores, lê instruções de um arquivo, controla unidades funcionais como a ALU e implementa multiplexadores. Os sinais de controle coordenam as operações do processador. Ou seja, caminhoDeDados representa o fluxo de dados e controle para um processador RISC-V.

```
`timescale 1ns/1ps

module datapath_tb;
    reg CDclock_,
        CDreset_;

    reg [8*25:0] nomeArquivoInstDP_;

    integer error;

    integer i;

    reg signed [31:0] valorReg;

    wire fimArquivo;

    caminhoDeDadosRiscV dpR5(
        CDclock_,
        CDreset_,
        nomeArquivoInstDP_,
        fimArquivo
    );

    initial begin
```

- ◊ **testeCaminho**: Esse arquivo ".vcd" realiza um teste no Caminho de dados desenvolvido pela dupla.

```
$date
Tue Aug 06 18:12:42 2024
$end
$version
    Icarus Verilog
$end
$timescale
    1ps
$end
$scope module datapath_tb $end
$var wire 1 ! fimArquivo $end
$var reg 1 " CDclock_ $end
$var reg 1 # CDreset_ $end
$var reg 201 $ nomeArquivoInstDP_ [200:0] $end
$var reg 32 % valorReg [31:0] $end
$var integer 32 & i [31:0] $end
$scope module dpR5 $end
$var wire 1 " CDclock $end
$var wire 201 ' CDnomeArquivo [200:0] $end
$var wire 1 # CDreset $end
$var wire 1 ( CDsaidaE $end
$var wire 1 ! fimArquivo $end
$var wire 32 ) CDSomaDesvio [31:0] $end
$var wire 32 * CDSoma4Saida [31:0] $end
$var wire 32 + CDsaídaPc [31:0] $end
$var wire 32 , CDmuxAluSaída [31:0] $end
$var wire 1 - CDmemóriaParaReg $end
```

As imagens de código apresentadas nessa documentação não representam os códigos em sua totalidade, sendo apenas parte deles. Para uma maior precisão consulte os códigos diretamente no GitHub do trabalho.

Observação: Esse TP foi feito no Sistema Operacional Windows 11, utilizando Visual Studio Code. Por algum motivo, se for preciso cirar arquivos de texto para novos inputs, opte por criar fora da IDE Visual Studio Code, pois a mesma apresenta uma formatação padrão que atrapalha a abertura do arquivo, sendo assim, arquivos devem ser criados arquivos no modelo ".txt" externos a IDE.

6 Como executar o Código

A execução do Código é feita basicamente através de dois comandos. O comando "make all" é usado para compilar o código e o comando "make run", executa o programa. Para realizar a troca dos inputs, basta alterar o "nomeArquivoInstDP_", que esta localizado no testbench do arquivo "caminhoDeDados.v"

```
initial begin
    nomeArquivoInstDP_ = "inputs/input1.txt";
    $dumpfile("saidas/testeCaminho.vcd");
    $dumpvars(3, datapath_tb);
end
```

Figure 1: nome do arquivo

7 Resultados

Abaixo estão três exemplos de entrada para o programa e as saídas que eles geram:

```
addi x1, x2, 42
sub x3, x4, x5
xor x6, x7, x8
sw x6, 0(x10)
lw x9, 0(x10)
sw x11, 4(x12)
addi x4, x2, 7
addi x13, x14, -10
sub x15, x16, x17
xor x18, x19, x20
sw x1, 0(x22)
lw x21, 0(x22)
sub x3, x1, x4
```

Figure 2: Instruções de entrada

```
00000010101000010000000010010011  
010000000101001000000011010011  
0000000010000011110000110010011  
00000000011001010010000000100011  
00000000000001010010010010000011  
00001000101101100010000000100011  
000000000111000010000001000010011  
11111111011001110000011010010011  
01000001000110000000011101010011  
0000000101001001110010010010011  
00000000000110110010000000100011  
00000000000010110010101010000011  
0100000001000000100000011010011
```

Figure 3: Instruções em Binário

```
VCD info: dumpfile saidas/testeCaminho.vcd opened for output.
regidor | 0|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 1|: 42 <decimal> 00000000000000000000000000000000 <binario>
regidor | 2|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 3|: 35 <decimal> 00000000000000000000000000000000 <binario>
regidor | 4|: 7 <decimal> 00000000000000000000000000000011 <binario>
regidor | 5|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 6|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 7|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 8|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 9|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 10|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 11|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 12|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 13|: -10 <decimal> 11111111111111111111111111111110110 <binario>
regidor | 14|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 15|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 16|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 17|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 18|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 19|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 20|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 21|: 42 <decimal> 000000000000000000000000000000101010 <binario>
regidor | 22|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 23|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 24|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 25|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 26|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 27|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 28|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 29|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 30|: 0 <decimal> 00000000000000000000000000000000 <binario>
regidor | 31|: 0 <decimal> 00000000000000000000000000000000 <binario>
caminhoDeDados.v:42: $finish called at 32000 (1ps)
```

Figure 4: Output no Terminal

Figure 5: Confirmação de Resultado

```

addi x1, x2, 10
addi x3, x4, -5
addi x4, x2, 7
srl x5, x6, x7
sw x1, 0(x9)
lw x8, 0(x9)
sw x3, 0(x11)
lw x7, 0[x11]
xor x12, x13, x14
sw x15, 8(x16)
beq x19, x20, 8
sub x21, x22, x23

```

Figure 6: Instruções de entrada 2

```

00000000101000010000000010010011
1111111101100100000000110010011
00000000011100010000001000010011
00000000011100110101001010110011
000000000000101001010000000100011
0000000000000001001010010000000011
00000000000000001101011010000000100011
0000000000000000001011010001110000011
00000000000000000000111001101100011000110011
00010000111110000010000000100011
00001001001111010000000001100011
01000001011110110000101010110011

```

Figure 7: Instruções em Binário 2

```

VCD info: dumpfile saidas/testeCaminho.vcd opened for output.
regidor | 0|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 1|: 10 <decimal>| 000000000000000000000000000000001010 <binario>
regidor | 2|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 3|: -5 <decimal>| 11111111111111111111111111111111011 <binario>
regidor | 4|: 7 <decimal>| 0000000000000000000000000000000000111 <binario>
regidor | 5|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 6|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 7|: -5 <decimal>| 11111111111111111111111111111111011 <binario>
regidor | 8|: 10 <decimal>| 00000000000000000000000000000000001010 <binario>
regidor | 9|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 10|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 11|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 12|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 13|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 14|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 15|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 16|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 17|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 18|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 19|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 20|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 21|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 22|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 23|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 24|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 25|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 26|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 27|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 28|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 29|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 30|: 0 <decimal>| 00000000000000000000000000000000 <binario>
regidor | 31|: 0 <decimal>| 00000000000000000000000000000000 <binario>
caminhoDeDados.v:42: $finish called at 28000 (ips)

```

Figure 8: Output no Terminal 2

Insira seu código RISC-V aqui:

		Valor Inicial	Registras	Decimal	Hex	Binário
1	addi x1, x2, 10	0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
2	addi x3, x4, -5	10	x1 (s)	10	0x0000000a	0b000000000000000000000000000000001010
3	addi x4, x2, 7	0	x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
4	sr1 x5, x6, x7	-5	x3 (pa)	-5	0xfffffff0	0b111111111111111111111111111111110111
5	sw x1, 0(x9)	0	x4 (t2)	7	0x00000007	0b00000000000000000000000000000000111
6	lw x8, 0(x9)	0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
7	sw x3, 0(x11)	0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
8	lw x7, 0(x11)	0	x7 (t2)	-5	0xfffffff0	0b111111111111111111111111111111110111
9	xor x12, x13, x14	0	x8 (s0/quadro)	10	0x0000000a	0b000000000000000000000000000000001010
10	sw x15, 8(x16)	0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
11	beq x19, x20, 8	0	x10 (s0)	0	0x00000000	0b00000000000000000000000000000000
12	sub x21, x22, x23	0	x11 (s1)	0	0x00000000	0b00000000000000000000000000000000
13		0	x12 (s2)	0	0x00000000	0b00000000000000000000000000000000
14		0				
--						

Figure 9: Confirmação de Resultado 2

```

addi x1, x2, 42
sub x3, x4, x5
xor x6, x7, x8
sw x6, 0(x10)
addi x4, x1, 3
sw x11, 4(x12)
addi x4, x2, 7
addi x13, x14, -10
sub x15, x16, x17
xor x18, x19, x20
sw x1, 0(x22)
lw x21, 0(x22)
addi x1, x2, 1
addi x1, x2, -1
xor x11, x9, x10
addi x13, x12, -3
sw x14, 20(x15)
sw x3, 0(x22)
lw x23, 0(x22)

```

Figure 10: Instruções de entrada 3

```

0000001010100010000000010010011
01000000010100100000000110110011
00000000100000111100001100110011
00000000011000101001000000100011
00000000001100001000001000010011
0000100010110110001000000100011
000000000011100010000001000010011
000000000011000110000001000010011
11111111011001110000011010010011
0100000100011000000011110110011
00000001010010011100100100110011
0000000000011011001000000100011
0000000000001011001010101000011
000000000000100010000000010010011
11111111110001000000010010011
00000000101001001100010110110011
11111111110101100000011010010011
0010100011100111101000000100011
000000000001110110010000000100011
00000000000010110010110000011

```

Figure 11: Instruções em Binário 3

```

VCD info: dumpfile saidas/testeCaminho.vcd opened for output.
registrador | 0]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 1]:      -1 <decimal> 11111111111111111111111111111111 <binario>
registrador | 2]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 3]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 4]:      7 <decimal> 00000000000000000000000000000111 <binario>
registrador | 5]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 6]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 7]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 8]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 9]:      0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 10]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 11]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 12]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 13]:    -3 <decimal> 11111111111111111111111111111101 <binario>
registrador | 14]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 15]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 16]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 17]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 18]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 19]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 20]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 21]:    42 <decimal> 0000000000000000000000000000101010 <binario>
registrador | 22]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 23]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 24]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 25]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 26]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 27]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 28]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 29]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 30]:     0 <decimal> 00000000000000000000000000000000 <binario>
registrador | 31]:     0 <decimal> 00000000000000000000000000000000 <binario>
caminhoDeDados.v:42: $finish called at 44000 (1ps)

```

Figure 12: Output no Terminal 3

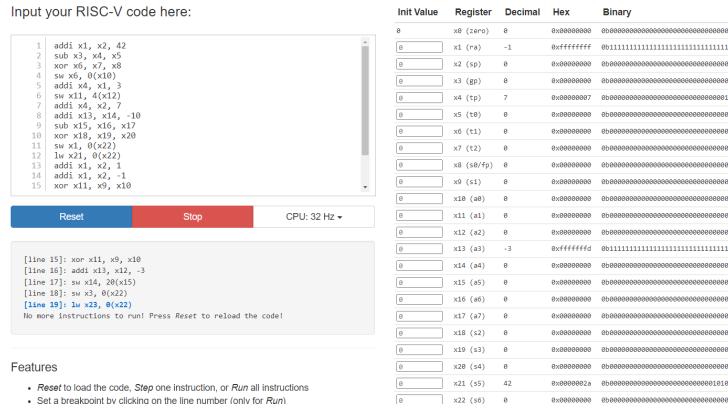


Figure 13: Confirmação de Resultado 3

8 Conclusão

O caminho de dados do RISC-V foi implementado com sucesso, atendendo às especificações propostas e incluindo as instruções requeridas. A realização deste trabalho prático contribuiu significativamente para a compreensão dos conceitos de organização de computadores discutidos em aula. A escolha de utilizar Verilog foi adequada, pois permite uma modelagem precisa do hardware; no entanto, uma linguagem como VHDL poderia oferecer uma sintaxe mais robusta para a descrição de circuitos digitais.

References

- [1] PATTERSON, David A; HENNESSY, John L.“Computer Organization and Design. RISC-V,” Edition, 2018.