Trabalho Prático 2 - Caminho de Dados do RISC-V

Guilherme Guimarães Pianetti - 5360, Dalmo Nolasco Dantas Rainer - 5361

[1]

1. Introdução

O RISC-V é uma arquitetura de conjunto de instruções (ISA) aberta e baseada no princípio de reduzir a complexidade das instruções, conhecida como Reduced Instruction Set Computer (RISC). Desenvolvida inicialmente na Universidade da Califórnia, Berkeley, o RISC-V tem ganhado popularidade devido à sua flexibilidade, modularidade e a ausência de restrições de propriedade intelectual, permitindo que qualquer pessoa ou organização possa implementá-la e adaptá-la conforme suas necessidades. Essa arquitetura se destaca por sua simplicidade e eficiência, facilitando o design de processadores de alta performance e baixo consumo de energia, o que é essencial para diversas aplicações, desde dispositivos embarcados até supercomputadores.

O trabalho proposto envolve a implementação de uma versão simplificada do caminho de dados do RISC-V, focando em um conjunto específico de instruções: ADD, SUB, AND, OR, LD, SD e BEQ. Cada grupo de estudantes será responsável por implementar um subconjunto diferente dessas instruções, partindo de uma base comum. Este exercício visa aprofundar o entendimento dos alunos sobre a arquitetura RISC-V e o funcionamento interno de um processador, abordando aspectos como a execução de instruções, manipulação de dados e controle de fluxo. Ao final, espera-se que os alunos tenham uma compreensão prática e teórica mais sólida sobre o design de processadores e a importância da ISA no desempenho e eficiência de sistemas computacionais.

2. Desenvolvimento

No processo de implementação do caminho de dados do RISC-V, a dupla selecionou a linguagem de descrição de hardware verilog como sua principal ferramenta. O primeiro passo foi a criação de um roteiro detalhado para nortear o desenvolvimento do projeto, que foi estrategicamente dividido em várias etapas:

1. Criação das principais componentes do caminho de dados (PC, memória de instruções, registrador, ALU, memória de dados, multiplexador, ImmGEN e controle);
2. Realizar o testebench de cada um desses componentes do camino de dados separadamente;
3. Realizar a junção de todos os componentes do caminho de dados em apenas um arquivo verilog comum;
4. Realizar o testebench desse arquivo comum para garantir a sua execução correta, com o auxilio da ferramenta GTKWave;
5. No nosso RISC-V serão implementados apenas os opcode´s das instruções especificadas na tabela de divisão dos grupo (Ficamos responsáveis pela linha 25); Por isso, algumas particularidades não foram tratadas.

2.1. Arquivo principal: RISK-V.v

O arquivo `RISC_V.v` descreve a implementação de um processador RISC-V simplificado em Verilog. O módulo principal conecta diversos componentes essenciais, como o contador de programa (`PC`), a memória de instruções (`instruction_mem`), o banco de registradores (`register_bank`), a unidade lógica e aritmética (`alu`), e a memória de dados (`data_mem`). Esses componentes trabalham em conjunto para buscar, decodificar e executar instruções armazenadas na memória, com o objetivo de realizar operações aritméticas, lógicas e de controle de fluxo.

No desenvolvimento do processador, são utilizados vários sinais de controle, gerados pela unidade de controle (`control`), que determinam o comportamento de cada módulo. Por exemplo, os multiplexadores (`mux`) são usados para selecionar quais dados devem ser enviados para a ALU e para decidir o próximo valor do contador de programa, dependendo das condições de desvio. A unidade de controle também coordena operações como leitura e escrita de dados na memória, ativando esses processos conforme necessário.

Em conclusão, o módulo `RISC_V` integra diversos blocos funcionais para formar um processador básico que consegue executar um conjunto limitado de instruções do conjunto de instruções RISC-V. Através da interação coordenada entre esses componentes, o processador realiza operações aritméticas e lógicas, controla o fluxo de execução e manipula dados em memória, cumprindo as funções essenciais de um processador simples.

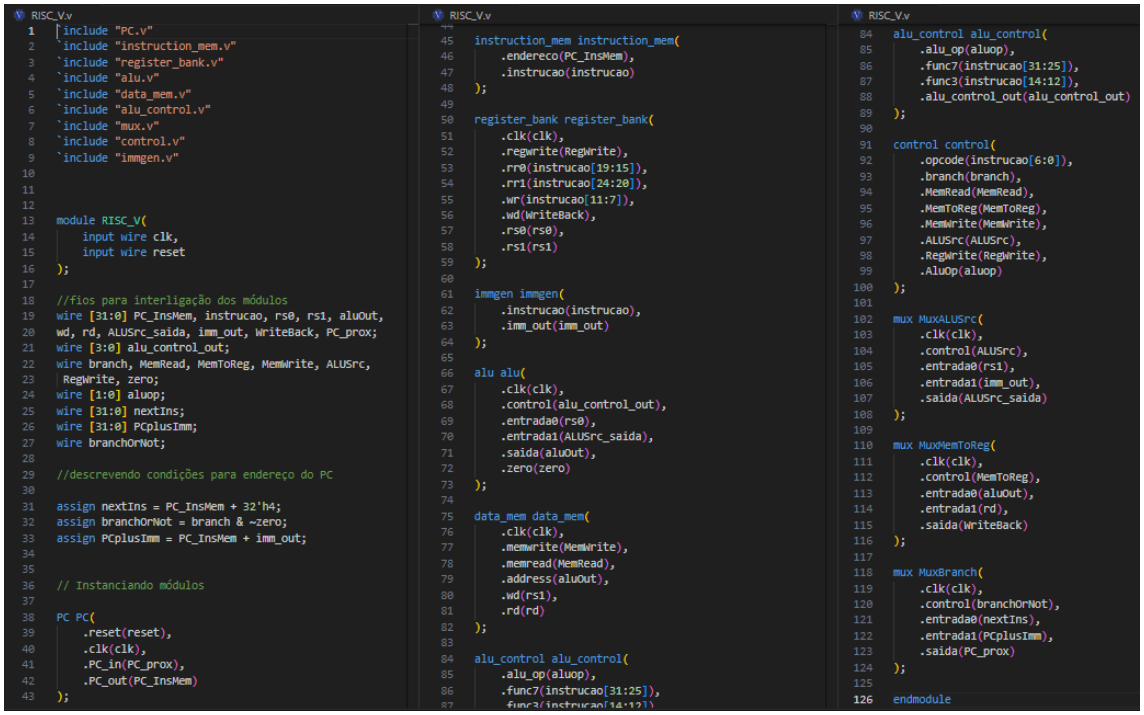

Figura 1. Arquivo principal RISC-V.v para a junção de todos os módulos do caminho de dados.

2.2. Arquivo: PC.v

O módulo `PC.v` define o comportamento de um contador de programa (PC) em um processador. Ele possui entradas para o sinal de reset (`reset`), o sinal de relógio (`clk`) e o valor de entrada do PC (`PC_in`), além de uma saída para o valor atual do PC (`PC_out`). O contador de programa é responsável por armazenar e atualizar o endereço da próxima instrução a ser executada pelo processador.

Dentro do bloco `always`, que é sensível à borda de subida do sinal de clock (`posedge clk`), o módulo verifica o sinal de reset. Se o reset estiver ativo, o `PC_out` é definido para `0x00000000`, reiniciando o contador de programa. Caso contrário, o `PC_out` é atualizado com o valor de `PC_in`, que sempre representa o endereço da próxima instrução a ser executada. Esse comportamento permite que o processador avance corretamente na execução das instruções, ou reinicie a execução em caso de reset.

Obs.: O módulo `PC.v` é o único que possui um reset interno.

PC.v

```
1   module PC(
2       input wire reset, clk,
3       input wire [31:0]PC_in,
4       output reg [31:0]PC_out
5   );
6
7   always @(posedge clk) begin
8       if (reset) begin
9           PC_out <= 32'h00000000;
10
11      end
12      else PC_out <= PC_in;
13
14  end
15
16  endmodule
```

Figura 2. PC

2.3. Arquivo: instruction-mem.v

O arquivo `instruction_mem.v` define um módulo Verilog que representa uma memória de instruções para um processador. Ele recebe um endereço de 32 bits como entrada e fornece uma instrução de 32 bits como saída. A memória de instruções é implementada como um array de registradores (`mem_instruction`) com 64 posições, onde cada posição armazena uma instrução de 32 bits. O array pode conter até 64 instruções, o que corresponde a um espaço de memória de 256 bytes, com cada instrução ocupando 4 bytes.

O módulo funciona de maneira simples: sempre que o endereço de entrada é atualizado, a instrução correspondente ao endereço é recuperada da memória e atribuída à saída `instrucao`. O endereço é dividido por 4 antes de acessar o array de instruções (`mem_instruction[endereco/4]`), pois cada instrução ocupa 4 bytes, e o endereço fornecido representa bytes. Portanto, dividir o endereço por 4 converte o endereço de byte em um índice de instrução. Esse módulo é fundamental para fornecer as instruções que o processador deve executar em cada ciclo de clock.

Obs.: Esse módulo é inicializado pelo comoando do arquivo mem-intruction.mem, onde ficarão as instruções desejadas pelo usuário em binário.

instruction_mem.v

```
1   module instruction_mem(
2       input wire [31:0] endereco,
3       output reg [31:0] instrucao
4   );
5
6       reg [31:0] mem_instruction [0:63];
7
8       always @(*) begin
9           instrucao = mem_instruction[endereco/4];
10      end
11
12  endmodule
```

Figura 3. Memória de instrução

2.4. Arquivo: register-bank.v

O arquivo `register_bank.v` define um módulo Verilog que implementa um banco de registradores para um processador. Este módulo possui 32 registradores de 32 bits, que são acessíveis para leitura e escrita. Os sinais de entrada incluem o relógio (`clk`), um sinal de escrita (`regwrite`), os identificadores dos registradores de leitura (`rr0` e `rr1`), o identificador do registrador de escrita (`wr`), e o dado a ser escrito (`wd`). As saídas do módulo são os valores dos registradores de leitura (`rs0` e `rs1`), que são fornecidos com base nos identificadores de leitura.

O comportamento do módulo é dividido em três partes principais. Primeiro, o registrador de índice 0 é sempre forçado a zero, assegurando que o registrador de número 0 seja sempre zero, como especificado pela arquitetura RISC-V. Em segundo lugar, o módulo escreve o dado `wd` no registrador especificado por `wr` apenas quando `regwrite` está ativo e na borda de subida do clock. Finalmente, o módulo atualiza as saídas `rs0` e `rs1` com os valores dos registradores identificados por `rr0` e `rr1`, respectivamente, sempre que os identificadores ou os valores dos registradores são alterados. Isso permite que o banco de registradores forneça corretamente os valores dos registradores solicitados para operações subsequentes no processador.

Obs.: Caso o registrador zero seja escrito, automaticamente ele será zerado, pois se trata de um padrão do RISC-V.

register_bank.v

```
module register_bank(
    input wire clk,
    input wire regwrite,
    input wire [4:0] rr0, //entrada 1
    input wire [4:0] rr1, //entrada 2
    input wire [4:0] wr, // registrador de onde ser es
    input wire [31:0] wd, // writeBack (conteudo a ser escrito)
    output reg [31:0] rs0,
    output reg [31:0] rs1
);

    reg [31:0] register [0:31];

    always @(register[0]) begin//trata o zero
        register[0] <= 0;
    end

    always @(posedge clk) begin
        if (regwrite == 1) begin
            register[wr] <= wd;
        end
    end

    always @(register[rr0] or rr0) begin
       rs0 <= register[rr0];
    end

    always @(register[rr1] or rr1) begin
       rs1 <= register[rr1];
    end

endmodule
```

Figura 4. Registrador (register-bank)

2.5. Arquivo: alu.v

O arquivo `alu.v` descreve um módulo Verilog que implementa uma Unidade Lógica e Aritmética (ALU) para um processador. A ALU realiza várias operações aritméticas e lógicas com base em um sinal de controle de 4 bits (`control`). As operações suportadas incluem AND, OR, ADD, SUB e SHIFT LEFT. As entradas para a ALU são duas palavras de 32 bits (`entrada0` e `entrada1`), e a ALU produz uma saída de 32 bits (`saida`) que é o resultado da operação selecionada. Além disso, a ALU também gera um sinal `zero`, que indica se o resultado da subtração é zero, caso isso ocorra, podemos utilizar desse método com o intuito de realizar operações de desvio.

A ALU é projetada para executar operações diferentes dependendo do valor do sinal de controle. O código utiliza uma estrutura `case` para determinar qual operação realizar com base no valor de `control`. Por exemplo, `control` igual a `4'b0000` executa uma operação AND, enquanto `4'b0010` realiza uma adição. A operação SUB (subtração) também é suportada, e o sinal `zero` é definido como 1 se o resultado da subtração for zero. Para operações de deslocamento à esquerda, o sinal `control` `4'b0100` é utilizado. O módulo garante que a operação correta seja realizada e o sinal `zero` seja corretamente atualizado conforme necessário.

alu.v

```
module alu(
    input clk,
    input [3:0] control,
    input [31:0] entrada0,
    input [31:0] entrada1,
    output reg [31:0] saida,
    output reg zero
);

    always @(*) begin
        case (control)
            4'b0000: begin // AND
                saida <= entrada0 & entrada1;
                zero <= 0;
            end
            4'b0001: begin // OR
                saida <= entrada0 | entrada1;
                zero <= 0;
            end
            4'b0010: begin // ADD
                saida <= entrada0 + entrada1;
                zero <= 0;
            end
            4'b0110: begin // SUB
                saida <= entrada0 - entrada1;
                zero <= ((entrada0 - entrada1) == 32'b0) ? 1 : 0; // Define zero se saida for 0
            end
            4'b0100: begin // ADD
                saida <= entrada0 << entrada1;
                zero <= 0;
            end
            default: begin // Default
                saida <= entrada0;
                zero <= 0;
            end
        endcase
    end

endmodule
```

Figura 5. ALU

2.6. Arquivo: data-mem.v

O arquivo `data_mem.v` define um módulo Verilog para uma memória de dados que suporta operações de leitura e escrita. O módulo tem um array de 64 palavras de 32 bits, e é projetado para trabalhar com operações que envolvem bytes, como instruções `LB` (Load Byte) e `SB` (Store Byte). Quando a leitura de memória (`memread`) está ativa, a

memória lê o byte apropriado com base nos dois bits menos significativos do endereço (`address[1:0]`). Dependendo do valor desses bits, a ALU retorna o byte correto, preenchendo os bits restantes com o sinal do byte lido para formar um valor de 32 bits.

Para operações de escrita (`memwrite`), a memória escreve o byte de dados (`wd`) no local apropriado da palavra de 32 bits, de acordo com os dois bits menos significativos do endereço. A operação de escrita é realizada na borda de descida do clock (`negedge clk`), garantindo que a escrita ocorra de forma sincronizada. O endereço é dividido por 4 (`address >> 2`) para determinar o índice correto na array de memória, e os bytes são armazenados no quarto correto da palavra, conforme indicado pelos bits menos significativos do endereço. Assim, o módulo manipula adequadamente as operações de leitura e escrita byte a byte dentro de uma palavra de 32 bits.

Obs.: Esse módulo só escreve em borda de decida do clock, será atualizado sempre que houver mudanças nas entradas e na leitura realizamos extenção de sinal com o byte desejado.

data_mem.v

```
module data_mem(
    input wire clk,
    input wire memwrite,
    input wire memread,
    input wire [31:0] address,
    input wire [31:0] wd,
    output reg [31:0] rd
);

output reg [31:0] memoria [0:63];

//adaptado apenas para instruções LB e SB (sempre operara com um byte)

always @(*) begin
    if (memread == 1) begin
        case (address[1:0])  // Verifica os dois bits menos significativos do endereço para identificar a parte da palavra
            2'b00: rd <= {{24{memoria[address >> 2][31]}}, memoria[address >> 2][31:24]};  // 1º quarto
            2'b01: rd <= {{24{memoria[address >> 2][23]}}, memoria[address >> 2][23:16]};  // 2º quarto
            2'b10: rd <= {{24{memoria[address >> 2][15]}}, memoria[address >> 2][15:8]};   // 3º quarto
            2'b11: rd <= {{24{memoria[address >> 2][7]}}},  memoria[address >> 2][7:0]};    // 4º quarto
        endcase
    end
end

always @(negedge clk) begin
    if (memwrite == 1) begin
        case (address[1:0])  // Verifica os dois bits menos significativos do endereço para identificar a parte da palavra
            2'b00: memoria[address >> 2][31:24] <= wd[31:24];  // 1º quarto da palavra
            2'b01: memoria[address >> 2][23:16] <= wd[23:16];  // 2º quarto da palavra
            2'b10: memoria[address >> 2][15:8]  <= wd[15:8];  // 3º quarto da palavra
            2'b11: memoria[address >> 2][7:0]   <= wd[7:0];  // 4º quarto da palavra
        endcase
    end
end

endmodule
```

Figura 6. Memória de dados

2.7. Arquivo: immgen.v

O arquivo `immgen.v` define um módulo Verilog responsável pela geração de valores imediatos para um processador RISC-V com base na instrução de entrada. O módulo recebe uma instrução de 32 bits como entrada (`instrucao`) e gera um valor imediato de 32 bits (`imm_out`) necessário para a execução de instruções que utilizam

valores imediatos. A geração do valor imediato depende do tipo de instrução, que é determinado pelos primeiros 7 bits da instrução (`instrucao[6:0]`).

O módulo utiliza uma estrutura `case` para identificar o tipo de instrução e gerar o valor imediato correspondente. Para instruções do tipo I, como `lb` e `ori`, o valor imediato é extraído diretamente dos campos da instrução, com a extensão de sinal apropriada. Para instruções do tipo S, como `sb`, o valor imediato é formado a partir de uma combinação de campos da instrução com a extensão de sinal. Para instruções do tipo SB, como `bne`, o valor imediato é gerado a partir de uma combinação mais complexa de bits da instrução. Se a instrução não corresponder a nenhum dos tipos conhecidos, o módulo retorna um valor indefinido (`32'bx`) como condição básica. Esse processo garante que o valor imediato correto seja gerado para instruções que o exigem.

Obs.: No defaut não será gerado zeros e sim 'x'.

immgen.v

```
1  module immgen(
2      input wire [31:0] instrucao,
3      output reg [31:0] imm_out
4  );
5
6  always @(instrucao) begin
7      case({instrucao[6:0]})
8          7'b0000011: imm_out <= {{21{instrucao[31]}}, instrucao[30:20]}; // Tipo I (lb)
9          7'b0100011: imm_out <= {{21{instrucao[31]}}, instrucao[30:25], instrucao[11:7]}; // Tipo S (sb)
10         7'b0010011: imm_out <= {{21{instrucao[31]}}, instrucao[30:20]}; // Tipo I (ori)
11         7'b1100011: imm_out <= {{20{instrucao[31]}}, instrucao[7], instrucao[30:25], instrucao[11:8], {1{1'b0}}}; // Tipo SB (bne)
12         default: imm_out <= 32'bx; // condição básica
13     endcase
14 end
15
16 endmodule
```

Figura 7. ImmGen

2.8. Arquivo: mux.v

O arquivo `mux.v` define um módulo Verilog para um multiplexador (MUX) que seleciona entre duas entradas de 32 bits com base em um sinal de controle. O módulo possui quatro portas: `clk` (sinal de clock), `control` (sinal de controle), `entrada0` e `entrada1` (entradas de 32 bits), e `saida` (saída de 32 bits). Embora o sinal de clock esteja presente, ele não é utilizado diretamente no funcionamento do multiplexador; sua presença pode ser relevante para integração com outros módulos que requerem sincronização de clock.

O funcionamento do multiplexador é determinado pelo sinal de controle (`control`). Se o valor de `control` for `0`, o multiplexador seleciona `entrada0` como a saída (`saida`). Se `control` for `1`, a saída será `entrada1`. A escolha entre `entrada0` e `entrada1` é feita através de uma estrutura condicional simples (`ternary operator`), que assegura que a saída reflita uma das duas entradas com base no valor do sinal de controle. Esse módulo é útil para roteamento de dados dentro de um processador, permitindo a seleção entre diferentes fontes de dados conforme necessário para a execução das instruções.

mux.v

```v
module mux (
    input wire clk,             // Declaração do sinal de clock como entrada
    input wire control,         // Declaração do sinal de controle como entrada
    input wire [31:0] entrada0, // Declaração da entrada0 como entrada de 32 bits
    input wire [31:0] entrada1, // Declaração da entrada1 como entrada de 32 bits
    output reg [31:0] saida     // Declaração da saída como saída de 32 bits
);

always @(*) begin
    saida = (control == 1'b0) ? entrada0 : entrada1;
end

endmodule
```

Figura 8. Multiplexador

2.9. Arquivo: control.v

O arquivo `control.v` define um módulo Verilog que implementa a unidade de controle de um processador RISC-V. Este módulo é responsável por gerar sinais de controle para coordenar as operações de diferentes componentes do processador com base no código de operação (opcode) da instrução. A entrada principal do módulo é o opcode de 7 bits, que determina o tipo de instrução a ser executada. Com base nesse opcode, o módulo gera vários sinais de controle, incluindo `branch`, `MemRead`, `MemToReg`, `MemWrite`, `ALUSrc`, `RegWrite`, e `AluOp`.

Durante o processo de desenvolvimento, o módulo utiliza uma estrutura `case` para definir o comportamento de controle para diferentes tipos de instruções. Para cada opcode específico, ele configura os sinais de controle de acordo com a necessidade da operação. Por exemplo, para o opcode `7'b0110011` (instruções do tipo R), a unidade de controle define `RegWrite` como 1 e todos os outros sinais de controle relacionados a memória e ramificação como 0, enquanto para o opcode `7'b0000011` (instruções `lb`), ela ativa a leitura de memória (`MemRead`) e a escrita no registrador (`MemToReg`), mas desativa a escrita na memória e a ramificação.

Em conclusão, o módulo `control` é fundamental para a operação do processador, pois traduz o opcode da instrução em sinais de controle que determinam como a ALU e a memória devem operar. Ele garante que as operações de leitura e escrita, bem como o controle de fluxo, sejam realizados corretamente, conforme especificado pelo tipo de instrução. Este módulo atua como o cérebro da lógica de controle, coordenando as ações dos diversos componentes do processador para executar as instruções corretamente.

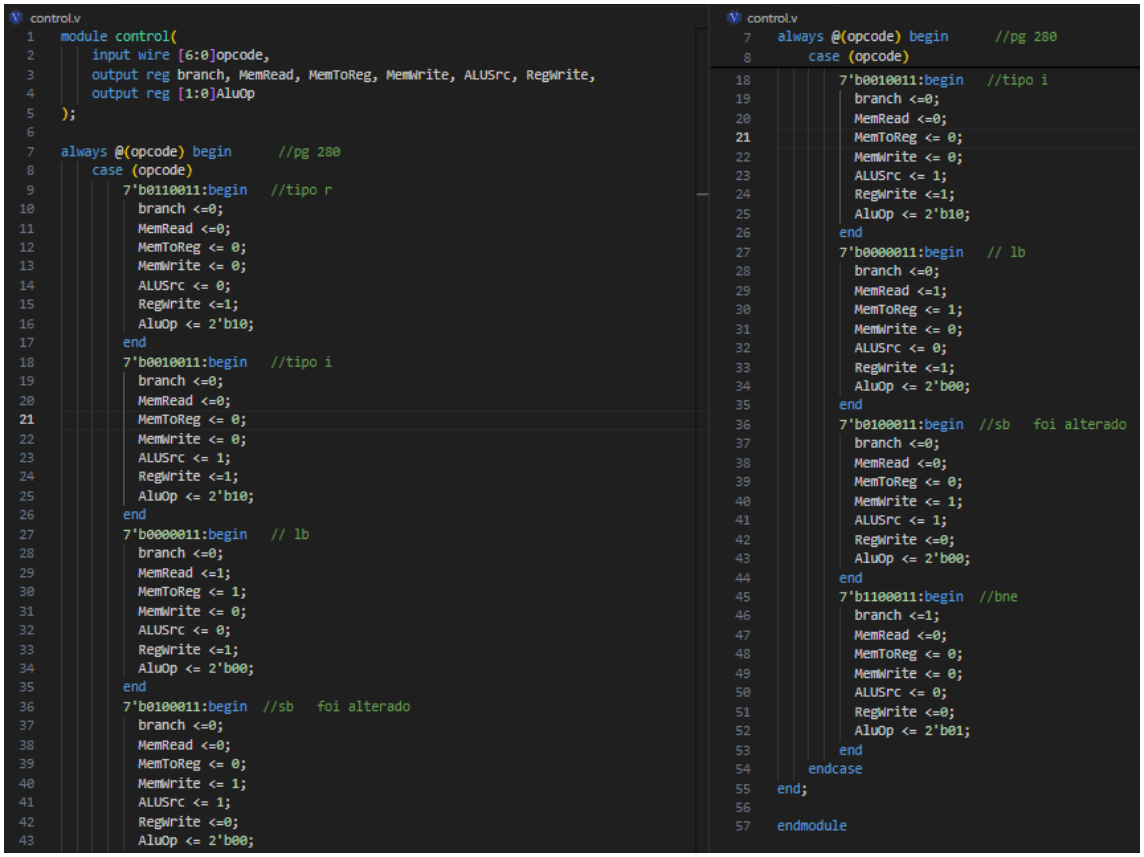

Figura 9. Controle

2.10. Arquivo: alu-control.v

O arquivo `alu_control.v` define um módulo Verilog responsável pela geração dos sinais de controle para a Unidade Lógica e Aritmética (ALU) de um processador RISC-V. Esse módulo determina a operação específica que a ALU deve realizar com base nos sinais de controle recebidos da unidade de controle principal (`alu_op`), bem como os campos adicionais da instrução (`func7` e `func3`). O resultado é um código de controle de 4 bits (`alu_control_out`) que indica a operação aritmética ou lógica que a ALU deve executar, como adição, subtração, AND, OR, ou deslocamento.

Durante o desenvolvimento, o módulo utiliza uma estrutura `casez` para mapear combinações dos sinais de controle (`alu_op`, `func7`, `func3`) para o código de operação da ALU. A estrutura `casez` é empregada para permitir que o `case` lide com valores indefinidos ('z' ou 'x'), facilitando a comparação de diferentes combinações de sinais. Por exemplo, quando o sinal `alu_op` é `00`, o módulo configura a ALU para realizar uma adição (`4'b0010`), que é adequada para instruções de carregamento e armazenamento (`LB` e `SB`). Para operações de subtração, o sinal `alu_op` é `01`, enquanto os tipos R com `alu_op` igual a `10` especificam operações aritméticas ou lógicas mais complexas, como ADD, SUB, AND, OR e SLL.

Em conclusão, o módulo `alu_control` é essencial para a operação correta da ALU, traduzindo os sinais de controle em operações específicas. Ele garante que a ALU

execute a operação correta com base no tipo de instrução e nos parâmetros fornecidos. Ao gerar os sinais de controle apropriados, o módulo permite que a ALU realize cálculos aritméticos e operações lógicas conforme requerido pelas instruções do processador, assegurando o correto funcionamento do processador no processamento de tarefas.

alu_control.v

```
module alu_control(
    input wire [1:0] alu_op,
    input wire [6:0] func7,  // Correcao: [31:25] eh 7 bits, entao precisa ser [6:0]
    input wire [2:0] func3,  // Correcao: [14:12] eh 3 bits, entao precisa ser [2:0]
    output reg [3:0] alu_control_out
);

always @(*) begin
    casez ({alu_op, func7, func3}) // usa casez para permitir 'z' ou 'x'
        12'b00_???????_???: alu_control_out = 4'b0010;  // ADD (LB/SB)
        12'b01_???????_???: alu_control_out = 4'b0110;  // SUB (Branch)
        12'b10_0000000_000: alu_control_out = 4'b0010;  // ADD (R-type)
        12'b10_0100000_000: alu_control_out = 4'b0110;  // SUB (R-type)
        12'b10_0000000_111: alu_control_out = 4'b0000;  // AND (R-type)
        12'b10_???????_110: alu_control_out = 4'b0001;  // OR (R-type)
        12'b10_0000000_001: alu_control_out = 4'b0100;  // SLL (R-type)
        default: alu_control_out = 4'b0000;  // default
    endcase
end

endmodule
```

Figura 10. Controle da ALU

2.11. Arquivo: testbench.v

O arquivo `testbench.v` define um ambiente de teste para o módulo `RISC_V` usando o Verilog. O objetivo principal desse arquivo é simular o comportamento do processador RISC-V e verificar se ele funciona corretamente ao executar as instruções fornecidas. O testbench configura e controla o ambiente de simulação, fornecendo os sinais de clock e reset necessários, e também inicializa a memória do processador com dados de teste. Além disso, ele gera um arquivo de resultados (`saida.vcd`) para visualização e análise da simulação.

No desenvolvimento, o testbench instancia o módulo `RISC_V` e inicializa seus componentes com arquivos de memória. Os arquivos `mem_instrucoes.mem`, `mem_reg.mem`, e `mem_data.mem` são usados para carregar as instruções, os registros e a memória de dados, respectivamente. O sinal de reset é ativado por um curto período e depois desativado para iniciar a simulação. O gerador de clock cria um sinal de clock periódico necessário para a operação do processador, alternando o valor do clock a cada 1 nanosegundo.

Na conclusão, o testbench monitora as instruções executadas pelo processador e verifica se há algum valor indefinido (`32'bx`) na instrução. Caso encontre tal valor, o testbench imprime o estado atual dos registradores e da memória de dados, e então encerra a simulação com `$finish()`. Esse processo permite analisar o estado interno do

processador após a execução das instruções e verificar se ele está funcionando conforme esperado. O uso de um arquivo de dump (`saida.vcd`) possibilita a análise detalhada das mudanças de estado ao longo da simulação, facilitando a depuração e validação do design.

Obs.: Não pode haver o preenchimento de toda a memória de instrução, deve sempre manter um espaço em branco.

testbench.v

```v
`include "RISC_V.v"
`timescale 1ns/100ps
module testbench();
    reg clk, reset;
    // Instanciação do módulo RISC_V
    RISC_V RISC_V_inst (
        .clk(clk),
        .reset(reset)
    );
    // Inicialização dos arquivos de memória
    initial begin
        $readmemb("mem_instrucoes.mem", RISC_V_inst.instruction_mem.mem_instruction);
        $readmemb("mem_reg.mem", RISC_V_inst.register_bank.register);
        $readmemb("mem_data.mem", RISC_V_inst.data_mem.memoria);
        $dumpfile("saida.vcd");
        $dumpvars(3, RISC_V_inst);

        reset = 1;
        clk = 0;
        #10 reset = 0;
    end
    // Gerador de clock
    always begin
        #1 clk = ~clk;
    end
    // Monitoramento das instruções e resete do sistema
 always @(RISC_V_inst.instrucao) begin
    if (RISC_V_inst.instrucao === 32'bx) begin
        for (integer i = 0; i < 32; i++) begin
            $display("Registrador %d = %b", i, RISC_V_inst.register_bank.register[i]);
        end
        for (integer i = 0; i < 32; i++) begin
            $display("Data_mem %d = %b", i, RISC_V_inst.data_mem.memoria[i]);
        end
        $finish();
    end
end
endmodule
```

Figura 11. testbench

2.12. Arquivos com a extenção .mem, .vcd ou .vvp

- `mem-data.mem`: Arquivo responsável por armazenar uma matriz de 64 linhas por 32 colunas. Utilizado para inicializar o `data-mem.v`.
- `mem-instrucoes.mem`: Arquivo responsável por armazenar as instruções, no formato de binário, que o usuário deseja testar no código (precisam ter 32 bits).
- `mem-reg.mem`: Arquivo responsável por armazenar uma matriz de 32 linhas por 32 colunas. Onde serão armazenados os valores numéricos desejados pelo usuário

em cada 'variável'. Ex.: x1 = 3, já que no caso do nosso código o x1 começa com o valor numérico igual a 3.

- `testbench.vvp`: Arquivo compilado do `testbench.v`
- `mem-reg.mem`: Arquivo de ondas que utilizamos no GTKWave para conferir as entradas e saídas dos módulos, com o intuito de corrigir bugs e analisar os resultados esperados.

3. Resultados

Nesta seção, apresentaremos os resultados obtidos a partir de um exemplo teste.

```
12    add x3, x1, x0
13    sb  x1, 3(x0)
14    lb  x5, 3(x0)
15    ori x6, x1, 4
16    and x2, x6, x1
17    sll x7, x6, x1
18    bne x1, x7, 8
19    add x10, x1, x0
20    add x11, x1, x0
```

Figura 12. Instruções usadas como exemplo

3.1. Execução dos módulos em conjunto feito por um testbench

Para executar o testbench, abra o terminal e siga os seguintes passos:

1. Digitar no terminal: `iverilog -o testbench.vvp testbench.v` e dê enter para que o código seja compilado e esteja pronto para a execução;
2. Caso deseje printar um teste no terminal digite: `vvp testbench.vvp` e dê enter;
3. Para analisar o GTKWave dê o comando: `gtkwave .\saida.vcd`
    - Obs.:Neste RISC-V foram implementados apenas os opcode´s das instruções especificadas para o grupo 25. Só executará se no `mem-instrucoes.mem` tiver a entrada em binário. E, lembrando, que x1 começa com o valor numérico de 3.

mem_instrucoes.mem

```
1    00000000000000001000000110110011
2    00000000000100000000000110100011
3    00000000001100000000000100000011
4    00000000010000001110001100010011
5    00000000000100110111001010110011
6    00000000000100110001001110110011
7    00000000011100001001010001100011
8    00000000000000001000010100110011
9    00000000000000001000010110110011
```

Figura 13. Exemplo de entrada transformado em binário

3.2. Saída no terminal

De acordo com o teste realizado pelo tópico anterior obtivemos essas saídas no terminal:

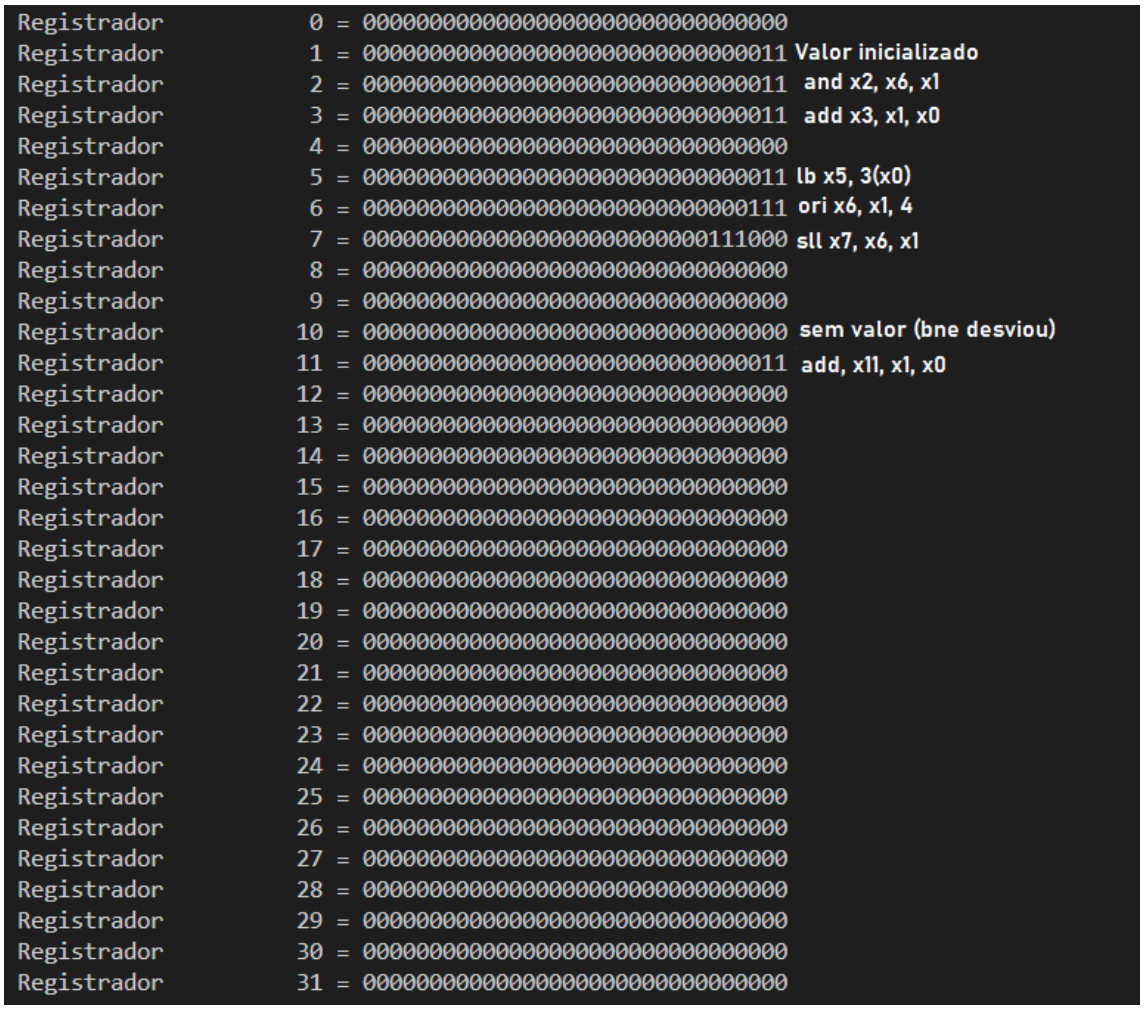

Figura 14. Saída no terminal do registrador

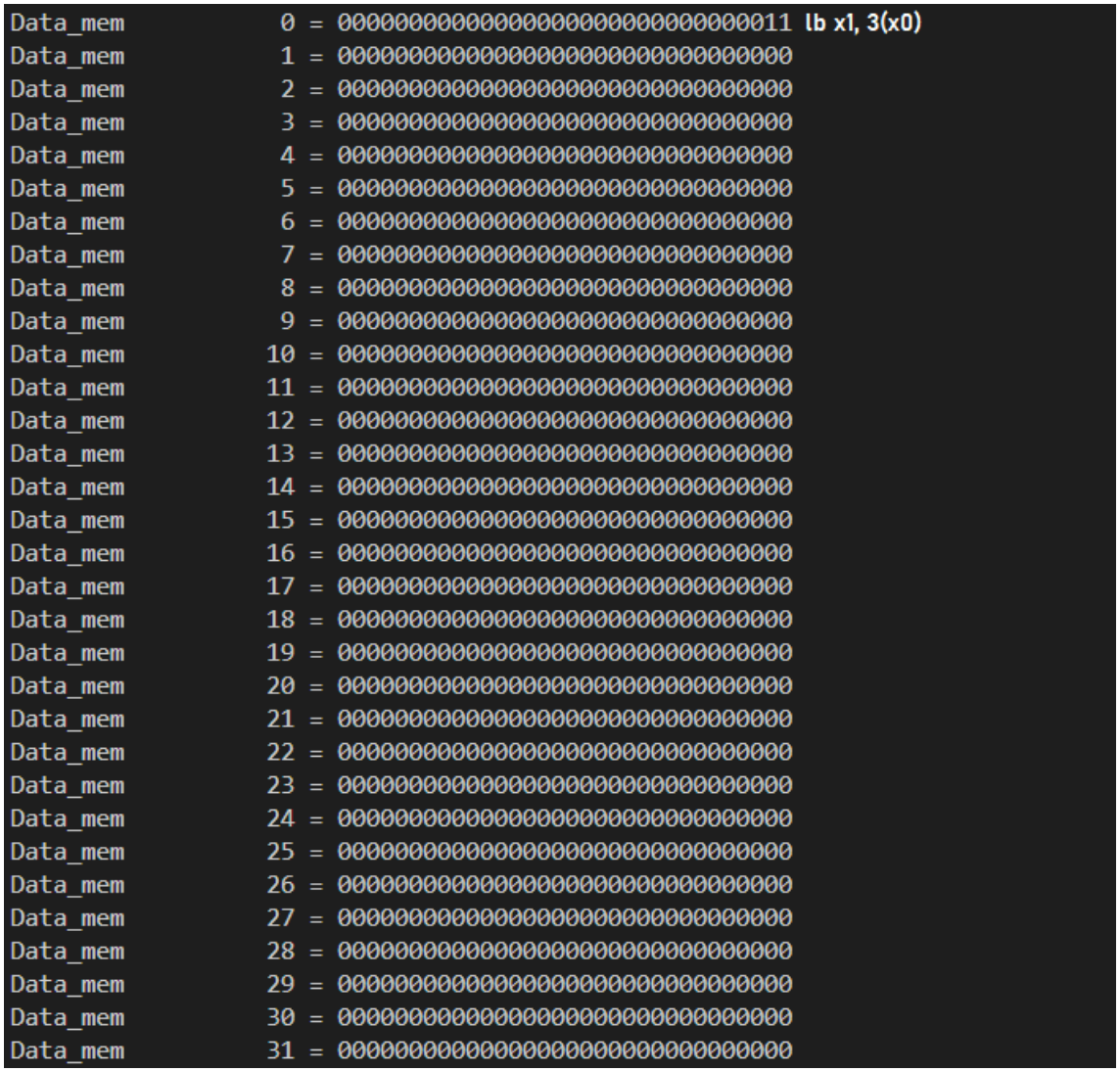

Figura 15. Saída no terminal da memória de dados

4. Conclusão

Neste projeto, exploramos a implementação de um processador RISC-V simplificado utilizando a linguagem Verilog. Através da criação e integração de diversos módulos, como o banco de registradores, a unidade de controle, a ALU, e outros componentes fundamentais, tivemos a oportunidade de entender na prática como cada parte de um processador funciona e interage com as demais. Essa experiência nos proporcionou uma visão clara do funcionamento interno de um processador, um conhecimento essencial para qualquer programador ou engenheiro de computação.

Durante o desenvolvimento deste trabalho, aprendemos a importância da modularidade e da organização no design de sistemas digitais complexos. Ao dividir o processador em módulos distintos e interconectá-los de forma lógica, foi possível trabalhar de maneira mais eficiente e com maior clareza. Além disso, a prática com Verilog nos ajudou a consolidar conceitos-chave de hardware, como a sincronização com o clock, o uso de sinais de controle, e a manipulação de dados binários em diferentes contextos. Esses conhecimentos são valiosos não apenas para o design de hardware, mas também para a programação em geral, onde a eficiência e a clareza do código são sempre prioridades.

Concluímos que a experiência adquirida ao projetar e implementar um processador em Verilog será de grande valor em nossa carreira futura como programadores. Compreender os fundamentos do hardware nos torna profissionais mais completos e preparados para enfrentar os desafios que surgem na interface entre software e hardware. Além disso, a capacidade de pensar de forma modular e de abordar problemas complexos com uma estratégia bem estruturada são habilidades que levaremos conosco para qualquer área de desenvolvimento de sistemas, seja em software, hardware ou em projetos multidisciplinares. Este trabalho reforçou a importância de uma base sólida em conceitos de arquitetura de computadores e design digital, que serão essenciais em nossa trajetória profissional.

5. Referências

- Material/slides utilizados em sala de aula pelo professor;
- Livro: Patterson, D.A.; Hennessy J.L. Computer Organization and Design: RISC-V Edition, 2ª Edição, Editora Morgnan Kaufman, 2021;
- Material extra de Verilog: `https://docs.google.com/presentation/d/1JOFFnvNb5eB2nUv1TNRIbNGpcxBhNR0QgvWHinsLBiQ/edit#slide=id.p`