Simulação de processador em verilog
Documentação

Caíque G. Leite[1], **Tharsos Gabriel C. Fernandes**[2]

[1]Instituto de Ciências Exatas e Aplicadas (ICEA) – Universidade Federal de Ouro Preto (UFOP)
Caixa Postal 24 – 35.931-008 – João Monlevade – MG – Brazil

Abstract. *This document aims to describe and instruct the use of the Verilog implementation of a single-cycle, non-pipelined RISC-V processor. Its focus is to demonstrate how the basic concepts covered in theoretical classes work and connect in practice. The testbench file is responsible for generating all the visual information we receive (the output), while the modules file is the heart of the project, largely executing the main functions of each module of the RISC-V processor.*

Resumo. *O presente documento visa descrever e instruir a utilização da implementação, usando Verilog, de um processador RISC-V sem pipeline e de ciclo único, com foco em mostrar como a parte básica, vista nas aulas teóricas, funciona e se conecta na prática, com o arquivo de testbench ficando responsável por gerar toda a informação visual que recebemos, a saída, enquanto o arquivo de módulos é o coração do projeto, executando, em grande parte, as funções principais de cada módulo do processador RISC-V.*

1. Introdução

O código do projeto está dividido em 2 arquivos principais, um de testbench que é responsável por mostrar a saida no terminal, enquanto o coração se encontra no arquivo de módulos, através do presente documento será feita a explicação de como o código foi escrito e qual a sua função, separado por módulos, pois cada módulo representa um bloco do processador diferente, ao final será feito um teste com o código sugerido do professor, em assembly, adaptado para os comandos que recebemos e convertido para binário pelo anterior trabalho. Por fim apresentaremos a conclusão em relação ao projeto.

2. Desenvolvimento

O desenvolvimento se sucederá da forma mencionada, onde explicaremos cada um dos dois arquivos módulo a módulo, de forma mais geral.

2.1. ”modulos.v”

2.1.1. PC

O ”Program Counter” é uma das partes mais simples do projeto, ele consiste em um pequeno módulo com 3 entradas, o clock, representado por ”clk”, o reset, representado por ”rst”, e o endereço da próxima instrução, representada por ”next_pc”, e 1 saída que é o endereço da instrução atual, que vai ser passada ao restante dos módulos, sendo esses dois últimos do tipo ”reg” de 32 bits.

Após as ”declarações de entradas e saídas”, o programa se inicia com a condição de executar sempre que encontrar uma borda de subida em clk ou rst, a seguir é feito um teste simples para saber se o sinal foi de clk ou de rst, onde se for rst a instrução atual recebe o endereço 0, ou seja, é reiniciado, já se o sinal recebido for de clk, o endereço atual recebe o próximo endereço, funcionando exatamente como o PC visto em aula, servindo apenas como um ”contador e ponteiro de instruções”.

```
module pc (
    input wire clk,
    input wire rst,
    input wire [31:0] next_pc,
    output reg [31:0] current_pc
);

    always @(posedge clk or posedge rst) begin
        if (rst)
            current_pc <= 32'b0;
        else
            current_pc <= next_pc;
    end
endmodule
```

Figure 1. ”pc”

2.1.2. Memória de Instrução

A ”Instruction Memory” possui uma constante para designar o tamanho da memória de instrução, nesse caso deixamos em 2048, uma entrada, um cabo conectado ao PC com o tamanho de 32 bits transmitindo o endereço da instrução, representado por ”read_address”, e uma saída, que vai conter a instrução de 32 bits lida na memória. Seguindo o código, alocamos a memória, onde cada compartimento terá o tamanho de uma instrução, e o número de blocos dependerá do tamanho da constante citada anteriormente, alocamos também espaço para o nome do arquivo, o suficiente para 127 caracteres.

A execução da memória de instrução acontece apenas uma vez, ao início da simulação, a princípio temos uma função que vai ler os argumentos passados e procurar o nome do arquivo, caso não haja, o código é finalizado ali e é passada uma instrução por terminal indicando o uso correto, caso seja encontrado o nome do arquivo, o programa o lerá e guardará suas instruções na ”memória rom”.

```
module instruction_memory #(
    parameter MEM_DEPTH = 2048
)(
    input wire [31:0] read_address,
    output reg [31:0] instruction
);

    reg [31:0] rom [0:MEM_DEPTH-1];
    reg [1023:0] nome_do_arquivo;

    initial begin

        if ($value$plusargs("%s", nome_do_arquivo)) begin
            $display("Carregando instrucoes do arquivo: %s", nome_do_arquivo);
            $readmemb(nome_do_arquivo, rom);
        end else begin
            $display("-----------------------------------------------------------------");
            $display("ERRO: Arquivo de instrucoes nao especificado.");
            $display("Uso correto: vvp <executavel> +<caminho_para_o_arquivo>");
            $display("Exemplo: vvp ford +saida.asm");
            $display("-----------------------------------------------------------------");
            $finish;
        end
    end


    always @(*) begin
        instruction = rom[read_address[11:2]];
    end
```

Figure 2. ”memória de instrução”

2.1.3. Banco de Registradores

O banco de registradores recebe diversas entradas, sendo elas o clock, a ”permissão” do controle para escrever em registradores, o endereço dos dois registradores que serão lidos, ambos de 5 bits de tamanho, o endereço do registrador que vamos escrever em cima, com 5 bits de tamanho também, e o dado que vamos escrever no registrador. A saída serão os dados dos dois registradores lidos, cada um de 32 bits. Após essa parte inicial, declaramos fisicamente os registradores em um ”vetor”.

Após essa parte inicial, fazemos de forma assíncrona com que se o endereço do registrador a ser lido for 0, então a saída será 0, caso contrário, a saída será o valor que está em no registrador, para garantir o funcionamento do registrador ”x0”. A seguir, sempre na borda de subida, acontecerá a escrita de registrador, caso tenha ”permissão” do bloco de controle para isso e caso não tente escrever no registrador ”x0”. Seguindo vemos um ”initial begin” que representa a execução única ao iniciar, dentro dele todos os registradores são inicializados com o valor ”0”, evitando problemas com resíduos na memória.

```
module register_file (
    input wire clk,
    input wire reg_write,
    input wire [4:0] read_reg1,
    input wire [4:0] read_reg2,
    input wire [4:0] write_reg,
    input wire [31:0] write_data,
    output wire [31:0] read_data1,
    output wire [31:0] read_data2
);

    reg [31:0] registers [0:31];

    assign read_data1 = (read_reg1 == 5'b0) ? 32'b0 : registers[read_reg1];
    assign read_data2 = (read_reg2 == 5'b0) ? 32'b0 : registers[read_reg2];

    always @(posedge clk) begin
        if (reg_write && (write_reg != 5'b0)) begin
            registers[write_reg] <= write_data;
        end
    end
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            registers[i] = 32'b0;
        end
    end
endmodule
```

Figure 3. ”banco de registradores”

2.1.4. Controle da ALU

A princípio, o bloco de controle da ALU recebe uma ordem do bloco de controle principal dizendo qual o tipo geral da instrução, se é de branch ou de memória, etc. Após isso, também recebe o funct3, que é responsável por ajudar a especificar qual operação será executada, a última entrada é o ”funct7_5” que serve para selecionar diferentes operações com o mesmo funct3, por fim há a saída, que é um código de 4bits que diz à ALU qual operação executar, sem qualquer ambiguidade.

A seguir, a lógica da ALU control é basicamente um switch case, onde se o comando for ”ADD”, por exemplo, a saída será o número do add e, quando for o referente a um tipo específico, inicia-se outro ”switch case”, testando todos os tipos de funct3 e, quando necessário, funct7_5 também

```
module alu_control (
    input wire [1:0] alu_op,
    input wire [2:0] funct3,
    input wire funct7_5,
    output reg [3:0] alu_control_out
);

    always @(*) begin
        case (alu_op)
            2'b00: begin
                alu_control_out = 4'b0010;
            end
            2'b01: begin
                alu_control_out = 4'b0110;
            end
            2'b10: begin
                case (funct3)
                    3'b000: alu_control_out = funct7_5 ? 4'b0110 : 4'b0010;
                    3'b001: alu_control_out = 4'b0100;
                    3'b100: alu_control_out = 4'b0011;
                    3'b101: alu_control_out = 4'b0101;
                    3'b110: alu_control_out = 4'b0001;
                    3'b111: alu_control_out = 4'b0000;
                    default: alu_control_out = 4'bxxxx;
                endcase
            end
            2'b11: begin
                case (funct3)
                    3'b000: alu_control_out = 4'b0010;
                    3'b001: alu_control_out = 4'b0100;
                    3'b101: alu_control_out = 4'b0101;
                    3'b110: alu_control_out = 4'b0001;
                    3'b111: alu_control_out = 4'b0000;
                    default: alu_control_out = 4'bxxxx;
                endcase
            end
```

Figure 4. ”Controle da ALU”

2.1.5. ALU

A ALU, que é onde as operações lógico-aritméticas acontecerão, se inicia recebendo 3 entradas, 2 delas são os valores que serão usados na operação e a terceiro é a operação oriunda da ALU control, e possui 2 saídas, uma, de 32 bits, é o resultado da operação, enquanto a outra é um especial de 1 bit, onde se o resultado da operação for 0, ele será 1 e se for qualquer outro, ele será 0.

Semelhantemente à ALU control, é iniciado um switch case e, para cada comando possível, é realizada a sua respectiva operação, ao final do código, a saída especial de 1 bit recebe o resultado do teste para saber se o resultado da operação foi 0 ou não.

```
module alu (
    input wire [31:0] a,
    input wire [31:0] b,
    input wire [3:0] alu_control,
    output reg [31:0] result,
    output wire zero
);

    always @(*) begin
        case (alu_control)
            4'b0000: result = a & b;
            4'b0001: result = a | b;
            4'b0010: result = a + b;
            4'b0011: result = a ^ b;
            4'b0101: result = a >> b[4:0];
            4'b0100: result = a << b[4:0];
            4'b0110: result = a - b;
            default: result = 32'b0;
        endcase
    end

    assign zero = (result == 32'b0);

endmodule
```

Figure 5. ”ALU”

2.1.6. Gerador de Imediato

O gerador de imediatos recebe uma instrução inteira e devolve o imediato montado de 32 bits. O código se segue com um switch case que sempre é executado a cada instrução, caso a instrução seja de um tipo que utiliza imediatos, é selecionado cada bit e montado na ordem correta para uso.

```
module imm_gen (
    input wire [31:0] instruction,
    output reg [31:0] immediate
);
    always @(*) begin
        case (instruction[6:0])
            7'b0010011, 7'b0000011:
                immediate = {{20{instruction[31]}}, instruction[31:20]};
            7'b0100011:
                immediate = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]};
            7'b1100011:
                immediate = {{19{instruction[31]}}, instruction[31], instruction[7], instruction[30:25], instruction[11:8], 1'b0};
            default:
                immediate = 32'hxxxxxxxx;
        endcase
    end
endmodule
```

Figure 6. ”Gerador de imediatos”

2.1.7. Memória de Dados

A memória de dados tem diversas entradas, sendo elas o clock, já préviamente explicado, as ”autorizações” do control para ler e escrever na memória, o endereço a ser acessado, o dado que é para escrever e, por fim, o funct3, que é essencial para saber o tamanho do dado a ser utilizado, há também uma saída, que é o dado lido. Ao início do código é alocado o espaço da memória ram e uma palavra de 12 bits para se adereçar a todos os 4096 espaços que existem alocados.

```
module data_memory (
    input wire clk,
    input wire mem_write,
    input wire mem_read,
    input wire [31:0] address,
    input wire [31:0] write_data,
    input wire [2:0] funct3,
    output reg [31:0] read_data
);
    reg [7:0] ram [0:4095];
    wire [11:0] word_address = address[11:0];
    always @(*) begin
        if (mem_read) begin
            case (funct3)
                3'b010: read_data = {ram[word_address+3], ram[word_address+2], ram[word_address+1], ram[word_address]};
                3'b001: read_data = {{16{ram[word_address+1][7]}}, ram[word_address+1], ram[word_address]};
                3'b101: read_data = {{16{1'b0}}, ram[word_address+1], ram[word_address]};
                3'b000: read_data = {{24{ram[word_address][7]}}, ram[word_address]};
                3'b100: read_data = {{24{1'b0}}, ram[word_address]};
                default: read_data = 32'hxxxxxxxx;
            endcase
        end else begin
            read_data = 32'b0;
        end
    end
```

Figure 7. ”Memória de dados p1”

A princípio, ao início das execuções, é testado se há autorização para ler, caso haja, inicia-se um switch case com o funct3 para saber como deve ser feita a leitura do dado. Caso haja autorização para a escrita na memória, novamente se inicia um switch case com o funct3 e o dado é salvo no endereço e da forma correta com base no tipo de instrução. Ambas as funções são síncronas e acontecem na borda de subida do clock.

```
    always @(posedge clk) begin
        if (mem_write) begin
            case (funct3)
                3'b010: begin
                    ram[word_address]   <= write_data[7:0];
                    ram[word_address+1] <= write_data[15:8];
                    ram[word_address+2] <= write_data[23:16];
                    ram[word_address+3] <= write_data[31:24];
                end
                3'b001: begin
                    ram[word_address]   <= write_data[7:0];
                    ram[word_address+1] <= write_data[15:8];
                end
                3'b000: begin
                    ram[word_address] <= write_data[7:0];
                end
            endcase
        end
    end

endmodule
```

Figure 8. ”Memória de dados p2”

2.1.8. Controle

O bloco de controle se inicia somente com o opcode de entrada, porém com diversas saídas, sendo elas cada uma das autorizações e instruções que serão usadas pelos demais blocos, branch, caso seja um desvio, mem_read e mem_write, caso sejam operações relacionadas à memória de dados, mem_to_reg para decidir se o dado escrito no registrador vem da memória ou da ALU, alu_op, que ajuda nas decisões do controle da ALU, alu_src, que decide se o segundo operando é um imediato e o reg_write que é a permissão para escrever em registradores.

o código do bloco de controle sempre é executado a cada instrução, ele inicia zerando todas as ”variáveis” de saída e inicia um switch case com o opcode e, em cada tipo, são designados os corretos valores para cada uma das operações, sendo que em algumas os valores permanecem intocados.

```
module main_control (
    input wire [6:0] opcode,
    output reg branch,
    output reg mem_read,
    output reg mem_to_reg,
    output reg [1:0] alu_op,
    output reg mem_write,
    output reg alu_src,
    output reg reg_write
);

    always @(*) begin
        branch      = 1'b0;
        mem_read    = 1'b0;
        mem_to_reg  = 1'b0;
        alu_op      = 2'b00;
        mem_write   = 1'b0;
        alu_src     = 1'b0;
        reg_write   = 1'b0;

        case (opcode)
            7'b0110011: begin
                alu_op      = 2'b10;
                reg_write   = 1'b1;
            end
            7'b0010011: begin
                alu_src     = 1'b1;
                reg_write   = 1'b1;
                alu_op      = 2'b11;
            end
```

Figure 9. ”Controle”

2.1.9. MUX

Diferentemente dos demais blocos, o MUX não é um bloco único, mas sim um bloco genérico usado para tomadas de decisão 2 para 1, e é exatamente assim que será usado. Ele recebe 4 entradas, sendo três delas com tamanhos adaptáveis, por causa da utilização

do ”WIDTH”, onde ”a” e ”b” seriam as entradas e ”y” o resultado, já o último input é a chave seletora, que mudará dependendo da ocasião. O que o MUX faz é simplesmente escolher a ou b dependendo do valor da chave seletora e atribuir esse valor a y.

```
module mux2_1 #(parameter WIDTH = 32) (
    input wire [WIDTH-1:0] a,
    input wire [WIDTH-1:0] b,
    input wire sel,
    output wire [WIDTH-1:0] y
);
    assign y = sel ? b : a;
endmodule
```

Figure 10. ”MUX genérico”

2.1.10. Processador

O módulo do processador é aquele que vai instanciar e conectar todos os outros, ele recebe apenas duas entradas, o clock e o reset. A princípio são declarados todos os fios que serão usados para conectar as saídas de um bloco até a entrada de outro, eles agirão como ponteiros em C, sendo passados como referência para os módulos executarem suas funções, depois se inicia, de fato, o funcionamento do processador.

```
module risc_v_processor (
    input wire clk,
    input wire rst
);

    wire [31:0] pc_current, pc_next, pc_plus_4, pc_branch;
    wire [31:0] instruction;
    wire [31:0] read_data1, read_data2;
    wire [31:0] immediate;
    wire [31:0] alu_in_b;
    wire [31:0] alu_result;
    wire [31:0] mem_read_data;
    wire [31:0] write_back_data;

    wire branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write;
    wire [1:0] alu_op;
    wire [3:0] alu_control_out;
    wire beq_cond = (instruction[14:12] == 3'b000) & alu_zero;
    wire bne_cond = (instruction[14:12] == 3'b001) & ~alu_zero;
    wire alu_zero;
    wire branch_control;
```

Figure 11. ”Processador p1”

Primeiramente o bloco do pc é instanciado, ele retornará o endereço da instrução atual e depois é calculado o valor do endereço + 4, que será usado caso não haja uma operação de branch, logo em seguida se instancia o bloco da memória de instrução, que por sua vez, retorna a instrução de 32 bits que está no endereço.

```
pc u_pc (
    .clk(clk),
    .rst(rst),
    .next_pc(pc_next),
    .current_pc(pc_current)
);

assign pc_plus_4 = pc_current + 4;

instruction_memory u_imem (
    .read_address(pc_current),
    .instruction(instruction)
);
```

Figure 12. ”Processador p2”

O opcode da instrução é enviado ao bloco de controle principal e retorna todos os sinais de controle, em sequência o banco de registradores é instanciado, com os registradores da instrução sendo passados como referência e devolvendo os dados que estão armazenados no endereço daqueles registradores e, enquanto isso, a instrução inteira é enviada ao gerador de imediatos, que extrai o valor imediato, caso haja.

```
main_control u_ctrl (
    .opcode(instruction[6:0]),
    .branch(branch),
    .mem_read(mem_read),
    .mem_to_reg(mem_to_reg),
    .alu_op(alu_op),
    .mem_write(mem_write),
    .alu_src(alu_src),
    .reg_write(reg_write)
);

register_file u_regfile (
    .clk(clk),
    .reg_write(reg_write),
    .read_reg1(instruction[19:15]),
    .read_reg2(instruction[24:20]),
    .write_reg(instruction[11:7]),
    .write_data(write_back_data),
    .read_data1(read_data1),
    .read_data2(read_data2)
);

imm_gen u_immgen (
    .instruction(instruction),
    .immediate(immediate)
);
```

Figure 13. ”Processador p3”

O MUX da ALU é usado logo em seguida, utilizando o sinal enviado pelo bloco de controle para selecionar se o segundo valor será um imediato ou não, para depois ser ativo o controle da ALU, que enviará a operação exata a ser feita. Ao unir os últimos dois sinais, a ALU executa sua operação e devolve seu resultado, logo à frente no código são feitos os cálculos para saber o destino do desvio e se haverá ou não desvio, com o MUX do pc ficando encarregado de tomar a decisão final.

```
mux2_1 #(32) alu_mux (
    .a(read_data2),
    .b(immediate),
    .sel(alu_src),
    .y(alu_in_b)
);

alu_control u_alu_ctrl (
    .alu_op(alu_op),
    .funct3(instruction[14:12]),
    .funct7_5(instruction[30]),
    .alu_control_out(alu_control_out)
);

alu u_alu (
    .a(read_data1),
    .b(alu_in_b),
    .alu_control(alu_control_out),
    .result(alu_result),
    .zero(alu_zero)
);

assign pc_branch = pc_current + immediate;
assign branch_control = branch & (beq_cond | bne_cond);

mux2_1 #(32) pc_mux (
    .a(pc_plus_4),
    .b(pc_branch),
    .sel(branch_control),
    .y(pc_next)
);
```

Figure 14. ”Processador p4”

Por fim vem o bloco da memória de dados que ficará encarregada de armazenar ou ler, a depender dos sinais recebidos dos blocos anteriores, dados dentro dela, com o MUX ao final que ”falará” para o banco de registradores se algum valor será salvo lá.

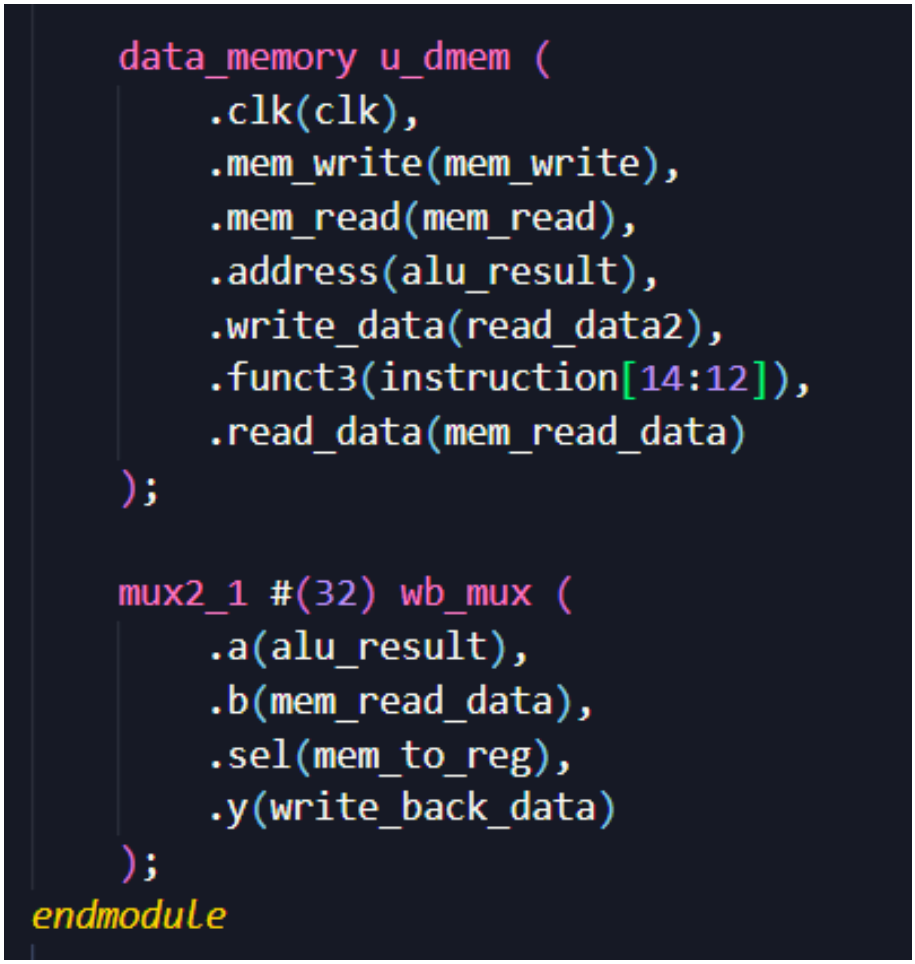


Figure 15. ”Processador p5”

2.2. saída

2.2.1. ”testbench.v”

O arquivo de testbench tem somente um módulo, que leva o mesmo nome, e esse módulo é o ”universo de simulação”, ele se inicia definindo o tempo de cada clock e declarando a variável que vai guardar o nome da instrução, para mostrar no terminal, não necessariamente de uso do processador.

```
`timescale 1ns / 1ps

module testbench;

    reg clk;
    reg rst;

    risc_v_processor dut (
        .clk(clk),
        .rst(rst)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    reg [8*7-1:0] instruction_mnemonic;
```

Figure 16. ”TB1”

A seguir é extraído o Opcode da instrução e é feito um switch case com ele para descobrir qual o comando da instrução, para posteriormente exibir no terminal.

```
always @(*) begin
    instruction_mnemonic = "???";

    case (dut.instruction[6:0])

        7'b0110011:
            case (dut.instruction[14:12])
                3'b000: instruction_mnemonic = (dut.instruction[30]) ? "SUB" : "ADD";
                3'b101: instruction_mnemonic = (dut.instruction[30]) ? "SRA" : "SRL";
                3'b001: instruction_mnemonic = "SLL";
                3'b100: instruction_mnemonic = "XOR";
                3'b110: instruction_mnemonic = "OR";
                3'b111: instruction_mnemonic = "AND";
                default: instruction_mnemonic = "R-Type?";
            endcase
```

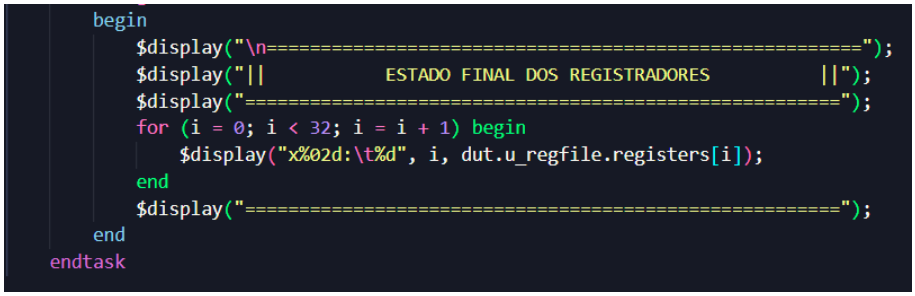
Figure 17. ”TB2”

Seguindo, se ativa o sinal de reset por um tempo e depois desativa, para iniciar a simulação e rodar ela por um tempo. Após essa parte, é iniciado o módulo que vigiará a execução do programa para imprimir na tela as instruções que estão sendo executadas.

```
module testbench;
    initial begin
        $display("--- INICIO DA SIMULACAO ---");

        #1000;

        print_final_state;
        $display("--- FIM DA SIMULACAO ---");
        $finish;
    end

    always @(posedge clk) begin
        if (!rst && dut.instruction != 0) begin
            if (dut.u_ctrl.reg_write) begin
                $display("PC: %d | %s \t-> Registrador x%0d recebe o valor %d (0x%h)",
                         dut.pc_current, instruction_mnemonic, dut.instruction[11:7], dut.write_back_data, dut.write_back_data);
            end else if (dut.u_ctrl.branch) begin
                $display("PC: %d | %s \t-> (Comparando registradores, resultado para desvio: %b)",
                         dut.pc_current, instruction_mnemonic, dut.branch_control);
            end else if (dut.u_ctrl.mem_write) begin
                $display("PC: %d | %s \t-> (Salvando o valor %d no endereco de memoria %h)",
                         dut.pc_current, instruction_mnemonic, dut.u_regfile.read_data2, dut.alu_result);
            end
        end
    end

    task print_final_state;
        integer i;
```

Figure 18. ”TB3”

Por fim é executado o bloco que mostra na tela o estado final de todos os registradores.



```
        begin
            $display("\n=======================================================");
            $display("||           ESTADO FINAL DOS REGISTRADORES           ||");
            $display("=======================================================");
            for (i = 0; i < 32; i = i + 1) begin
                $display("x%02d:\t%d", i, dut.u_regfile.registers[i]);
            end
            $display("=======================================================");
        end
    endtask
```

Figure 19. ”TB4”

2.2.2. Entrada, execução e saída

A entrada, como sugerido pelo professor, será o código adaptado aos nossos comandos do primeiro trabalho, porém em binário, como mostra a imagem

```
ori x1, x0, 7
sub x2, x1, x0
beq x1, x2, SAIDA

# Caso o fluxo venha para cá, seu processador está errado
sub x1, x1, x1
sb x1, 0(x0)

SAIDA:
and x1, x1, x2
ori x1, x1, 0
sb x1, 0(x0)
```

Figure 20. ”Entrada em assembly”

```
00000000011100000110000010010011
01000000000000001000000100110011
00000000000100000100001100100011
01000000000100001000000010110011
00000000000010000000000000100011
00000000000100000111100010110011
00000000000000000111000010010011
00000000000010000000000000100011
```

Figure 21. ”Entrada em binário”

Como explicado no bloco da testbench, a saída pode ser dividida em 2 partes, as instruções que foram executadas no processador e, mais ao final, os valores que ficaram armazenados nos registradores. Para executar o código é preciso utilizar no terminal o comando ”vvp ford +saida.asm”.

```
PS C:\Users\Admin\Documents\GitHub\Assembler_Em_OA-C> vvp ford +saida.asm
Carregando instrucoes do arquivo:
sm
WARNING: modulos.v:33: $readmemb(saida.asm): Not enough words in the file for the requested range [0:2047].
--- INICIO DA SIMULACAO ---
PC:          0 |     ORI        -> Registrador x1 recebe o valor          7 (0x00000007)
PC:          4 |     SUB        -> Registrador x2 recebe o valor          7 (0x00000007)
PC:          8 |     BEQ        -> (Comparando registradores, resultado para desvio: 1)
PC:         20 |     AND        -> Registrador x1 recebe o valor          7 (0x00000007)
PC:         24 |     ORI        -> Registrador x1 recebe o valor          7 (0x00000007)
PC:         28 |      SB        -> (Salvando o valor          7 no endereco de memoria 00000000)
```

Figure 22. ”Saída: instruções executadas”

```
====================================================
||           ESTADO FINAL DOS REGISTRADORES         ||
====================================================
x00:            0
x01:            7
x02:            7
x03:            0
x04:            0
x05:            0
x06:            0
x07:            0
x08:            0
x09:            0
x10:            0
x11:            0
x12:            0
x13:            0
x14:            0
x15:            0
x16:            0
x17:            0
x18:            0
x19:            0
x20:            0
x21:            0
x22:            0
x23:            0
x24:            0
x25:            0
x26:            0
x27:            0
x28:            0
x29:            0
x30:            0
x31:            0
====================================================
--- FIM DA SIMULACAO ---
```

Figure 23. ”Saída: valor dos registradores”

Como é possível ver na imagem do site sugerido pelo professor abaixo, a saída dos registradores é a esperada.

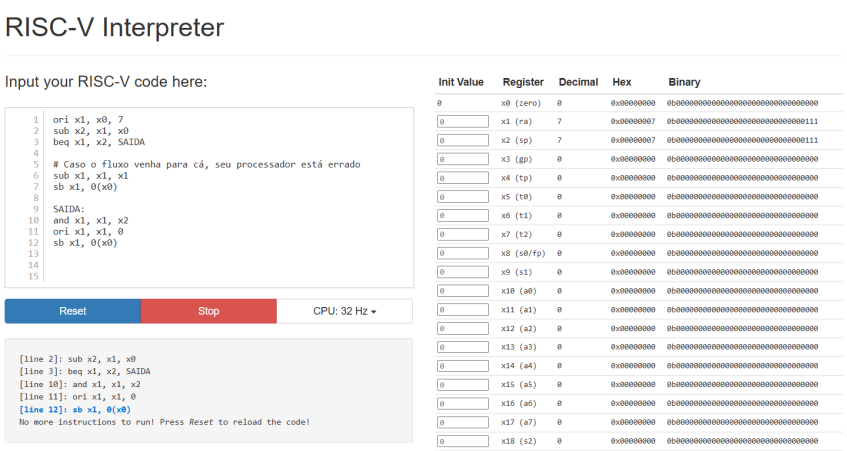


Figure 24. ”Saída esperada”

3. Conclusão

A partir da implementação do código, foi possível observar na prática a interação entre os diferentes componentes que formam o caminho de dados de um processador RISC-V simpless. Este trabalho prático permitiu solidificar os conceitos teóricos abordados em sala de aula, traduzindo os diagramas e a teoria de funcionamento em uma simulação. Conclui-se que o funcionamento de um processador é muito mais coordenado e minucioso do que podemos enxergar, mas não foge do que foi estudado