Projeto e Simulação do processador RISC-V Uniciclo

Gabriel Lucas França do Nascimento
190107111
nascimento.franca@aluno.unb.br

17/02/2025

1 Descrição do Trabalho

O objetivo deste projeto foi desenvolver um processador capaz de executar um subconjunto de instruções RISC-V, incluindo operações aritméticas, lógicas, de deslocamento, de memória e de controle de fluxo.

2 Introdução

O principal objetivo do projeto foi implementar um processador RISC-V Uniciclo em VHDL, capaz de executar instruções básicas como AND, ADD, SUB, ADDI, LW, SW, BEQ, JAL, AUIPC e LUI. Além disso, o processador foi projetado para ser simulado em ferramentas como ModelSim ou EdaPlayground, garantindo o funcionamento correto das instruções implementadas.

Foi utilizado VHDL para o desenvolvimento, através de uma abordagem modular - os módulos foram desenvolvidos ao longo do semestre. Cada componente do processador (PC, memória de instruções, banco de registradores, ULA, memória de dados, unidade de controle, etc.) foi implementado separadamente e interconectado através de sinais. A simulação foi realizada para verificar o funcionamento correto de cada módulo e do processador como um todo.

3 Componentes implementados

- PC (Program Counter): Responsável por armazenar o endereço da próxima instrução a ser executada.

- Memória de Instruções (ROM): Armazena o código do programa a ser executado.
- Banco de Registradores (XREGS): Contém 32 registradores de 32 bits, sendo o registrador x0 hardwired para zero.
- ULA (Unidade Lógico-Aritmética): Realiza operações aritméticas, lógicas e de deslocamento.
- Memória de Dados (RAM): Armazena dados do programa.
- Unidade de Controle: Decodifica as instruções e gera os sinais de controle para os demais módulos.
- Gerador de Imediato (genImm32): Extrai e estende valores imediatos das instruções.
- MUX 2-to-1: Realiza a seleção da entrada em diversos momentos do código - seleção de operadores da ULA, seleção de dado da memória ou imediato etc.
- Unidade de controle da ULA: Define o tipo de instrução conforme os campos funct3 e funct7.

4 Destaques da Implementação

- Instruções Aritméticas e Lógicas: ADD, SUB, AND, OR, XOR, ADDI, ANDI, ORI, XORI.
- Instruções de Deslocamento: SLL, SRL, SRA, SLLI, SRLI, SRAI.
- Instruções de Memória: LW, SW, LB, LBU, SB.
- Instruções de Controle de Fluxo: BEQ, BNE, BLT, BGE, BLTU, BGEU, JAL, JALR.
- Instruções de Geração de Constantes: AUIPC e LUI.

Toda a parte aritmética foi implementada, sinais de controle, lógica de escrita e leitura, seleção de sinais utilizando MUX, além da leitura de instruções em hexadecimal, a partir de arquivo de texto. Foram criados testbenches para cada módulo, garantindo o funcionamento correto individualmente, contudo, não realizei os testes de JAL, JALR, mesmo estando com toda a parte lógica implementada.

5 Telas da Simulação

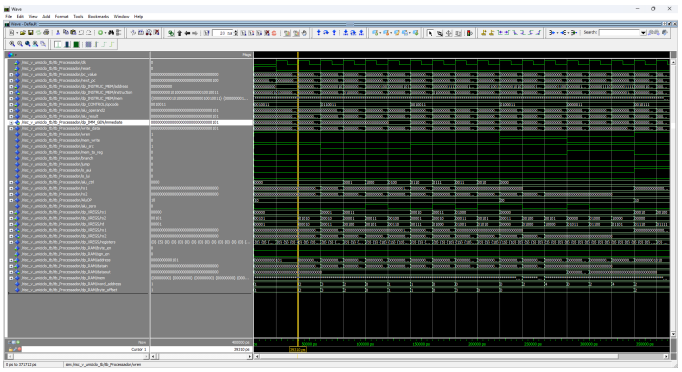

Figure 1: Visão geral do funcionamento

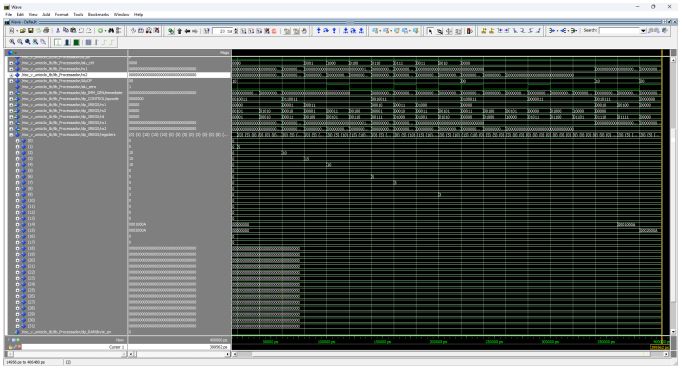

Figure 2: Visão detalhada dos registradores

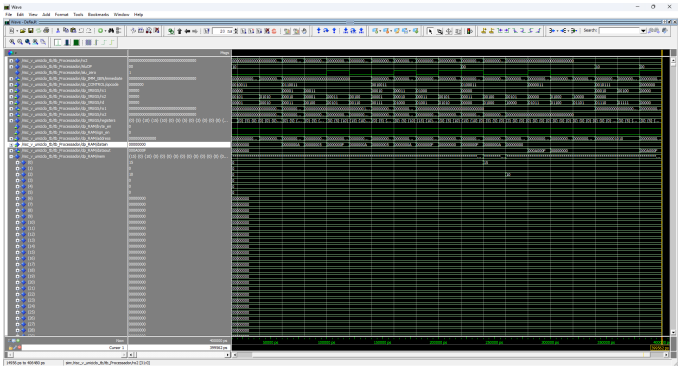

Figure 3: Visão geral da memória

6 Código e Testbench

6.1 Código geral do processador (RISC_V_Uniciclo.vhd)

Listing 1: Código da ULA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RISC_V_Uniciclo is
    Port (
        clk : in STD_LOGIC; -- Sinal de clock
        reset : in STD_LOGIC -- Sinal de reset
    );
end RISC_V_Uniciclo;

architecture Behavioral of RISC_V_Uniciclo is

    -- Sinais PC
    signal pc_value, next_pc : STD_LOGIC_VECTOR(31 downto 0);

        -- Sinais ROM
    --signal instruction_rv : STD_LOGIC_VECTOR(31 downto 0);
    signal instruction : STD_LOGIC_VECTOR(31 downto 0);

        -- Sinais ULA
    signal alu_result, alu_operand2 : STD_LOGIC_VECTOR(31 downto 0);

        -- Sinais gerador de imediato
    signal imm_value : STD_LOGIC_VECTOR(31 downto 0);

        -- Sinais de controle
    signal wren, mem_write, alu_src,
    mem_to_reg, branch, jump, is_aui, is_lui : STD_LOGIC;

        -- Sinais Controle ULA
    signal alu_ctrl : STD_LOGIC_VECTOR(3 downto 0);

        -- Sinais RAM
    signal byte_en, sgn_en : STD_LOGIC; -- Sinais adicionais para a RAM
```

```
signal data_from_memory : STD_LOGIC_VECTOR(31 downto 0);

     -- Sinais registradores
signal ro1, ro2 : STD_LOGIC_VECTOR(31 downto 0);
signal write_data : STD_LOGIC_VECTOR(31 downto 0);

-- Sinais operacao ULA
    signal AluOP : STD_LOGIC_VECTOR(1 downto 0);
signal alu_zero : STD_LOGIC;

-- Componentes
component PC is
    Port (
        clk   : in STD_LOGIC;
        reset  : in STD_LOGIC;
        next_pc  : in STD_LOGIC_VECTOR(31 downto 0);
        pc_value : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component rom_rv is
    Port (
        address : in STD_LOGIC_VECTOR(9 downto 0);
        instruction : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component ControlUnit is
    Port (
        opcode : in STD_LOGIC_VECTOR(6 downto 0);
        wren : out STD_LOGIC;
        mem_write : out STD_LOGIC;
        alu_src : out STD_LOGIC;
        mem_to_reg : out STD_LOGIC;
        branch : out STD_LOGIC;
        jump : out STD_LOGIC;
        AluOP : out STD_LOGIC_VECTOR(1 downto 0);
        is_aui : out STD_LOGIC;
        is_lui : out STD_LOGIC
    );
end component;
```

```
component XREGS is
    Port (
        clk   : in STD_LOGIC;
        reset    : in STD_LOGIC;
        wren    : in STD_LOGIC;
        rs1 : in STD_LOGIC_VECTOR(4 downto 0);
        rs2 : in STD_LOGIC_VECTOR(4 downto 0);
        rd    : in STD_LOGIC_VECTOR(4 downto 0);
        --dado a ser escrito no reg
        write_data  : in STD_LOGIC_VECTOR(31 downto 0);
        ro1, ro2   : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component genImm32 is
    Port (
        instruction : in STD_LOGIC_VECTOR(31 downto 0);
        immediate : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component ULA is
    Port (
        A, B : in STD_LOGIC_VECTOR(31 downto 0);
        control : in STD_LOGIC_VECTOR(3 downto 0);
        is_aui : in STD_LOGIC;
        is_lui : in STD_LOGIC;
        pc : in STD_LOGIC_VECTOR(9 downto 0);
        result : out STD_LOGIC_VECTOR(31 downto 0);
        zero : out STD_LOGIC
    );
end component;

component ram_rv is
    Port (
        -- Clock
        clk : in std_logic;
        -- Sinal de escrita (write enable)
        wren : in std_logic;
        -- Acesso a byte (1) ou palavra (0)
```

```
            byte_en : in std_logic;
            -- Acesso com sinal (1) ou sem sinal (0)
            sgn_en : in std_logic;
             -- Endereco de 13 bits
            address : in std_logic_vector(12 downto 0);
             -- Dado de entrada (32 bits)
            datain : in std_logic_vector(31 downto 0);
            dataout : out std_logic_vector(31 downto 0)
        );
    end component;

         component ALU_Control is
               Port (
                      AluOP : in STD_LOGIC_VECTOR(1 downto 0);
                      funct3 : in STD_LOGIC_VECTOR(2 downto 0);
                      funct7 : in STD_LOGIC_VECTOR(6 downto 0);
                      alu_control : out STD_LOGIC_VECTOR(3 downto 0)
               );
         end component;

         component mux_2to1 is
        Port (
            sel : in STD_LOGIC;
            a : in STD_LOGIC_VECTOR(31 downto 0);
            b : in STD_LOGIC_VECTOR(31 downto 0);
            y : out STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;

begin
    -- Instanciando modulos
    dp_PC : PC
        port map(
            clk,
            reset,
            next_pc,
            pc_value
        );

    dp_INSTRUC_MEM : rom_rv
        port map(
```

```
        pc_value(11 downto 2), -- Conecta os 10 bits do PC
        instruction -- A instrucao lida eh armazenada em instruction
    );

dp_CONTROL : ControlUnit
    port map(
        instruction(6 downto 0),
        wren,
        mem_write,
        alu_src,
        mem_to_reg,
        branch,
        jump,
        AluOP,
        is_aui,
        is_lui
    );

dp_IMM_GEN : genImm32
    port map(
        instruction,
        imm_value
    );

dp_ALU_CONTROL : ALU_Control
    port map(
          AluOP,
          instruction(14 downto 12), --funct3
          instruction(31 downto 25), --funct7
          alu_control => alu_ctrl
    );

-- MUX define se vai usar o dado da memoria ou o dado da ULA
dp_MUX_MEM_ALU : mux_2to1
    port map (
        sel => mem_to_reg, -- Sinal de selecao
        a => alu_result, -- Entrada A (dado da ULA)
        b => data_from_memory, -- Entrada B (dado da memoria)
        y => write_data -- Saida (dado a ser escrito no registrador)
    );
```

```
dp_ULA : ULA
    port map(
        ro1,
        alu_operand2,
        alu_ctrl,
        is_aui => is_aui,
        is_lui => is_lui,
        pc => pc_value(11 downto 2),
        result => alu_result,
        zero => alu_zero
    );

dp_XREGS : XREGS
    port map(
        clk,
        reset,
        wren,
        instruction(19 downto 15), -- rs1
        instruction(24 downto 20), -- rs2
        instruction(11 downto 7), -- rd
        write_data,
        ro1,
        ro2
    );

-- Instanciacao da RAM
dp_RAM : ram_rv
    port map(
        clk,
        mem_write, -- Sinal de escrita (wren)
        byte_en, -- byte_en (0 para acesso a palavra)
        sgn_en, -- sgn_en (0 para acesso sem sinal)
        alu_result(12 downto 0), -- Endereco de 13 bits
        ro2, -- Dado de entrada (datain)
        data_from_memory -- Dado de saida (dataout)
    );

-- MUX define se a ULA vai usar o registrador 2 ou o imediato
dp_MUX_ULA_IMM : mux_2to1
    port map (
        sel => alu_src, -- Sinal de selecao
```

```
        a => ro2, -- Entrada A (dado do registrador 2)
        b => imm_value, -- Entrada B (dado imediato)
        y => alu_operand2 -- Saida (dado a ser escrito no registrador)
    );

-- Logica do PC (falta otimizaao)
process(clk, reset ,pc_value, branch,
jump, alu_zero, imm_value, instruction)

begin
    if branch = '1' and alu_zero = '1' then
        -- Instrucao BEQ: salta se a condicao for verdadeira
        next_pc <=
        std_logic_vector(unsigned(pc_value) + unsigned(imm_value));

    elsif jump = '1' then
        -- Instrucao JAL: salto incondicional
        next_pc <=
        std_logic_vector(unsigned(pc_value) + unsigned(imm_value));

    elsif reset = '1' then
        -- Reset: volta para o endereco 0
        next_pc <= (others => '0');

    else
        next_pc <= std_logic_vector(unsigned(pc_value) + 4);
    end if;
end process;

-- Lgica para gerar byte_en e sgn_en
process(instruction)
    variable byte_en_var : STD_LOGIC := '0';
    variable sgn_en_var : STD_LOGIC := '0';
begin
    -- Valores padro
    byte_en_var := '0';
    sgn_en_var := '0';

    -- Lgica para gerar byte_en e sgn_en
    case instruction(6 downto 0) is -- Opcode
        when "0000011" => -- Load instructions
```

```
                case instruction(14 downto 12) is -- funct3
                    when "000" => -- LB
                        byte_en_var := '1';
                        sgn_en_var := '1';
                    when "001" => -- LH
                        byte_en_var := '1';
                        sgn_en_var := '1';
                    when "010" => -- LW
                        byte_en_var := '0';
                        sgn_en_var := '0';
                    when "100" => -- LBU
                        byte_en_var := '1';
                        sgn_en_var := '0';
                    when "101" => -- LHU
                        byte_en_var := '1';
                        sgn_en_var := '0';
                    when others =>
                        byte_en_var := '0';
                        sgn_en_var := '0';
                end case;
            when others => -- Outras instrucoes
                byte_en_var := '0';
                sgn_en_var := '0';
        end case;

        -- Atribui as variveis aos sinais de sada
        byte_en <= byte_en_var;
        sgn_en <= sgn_en_var;
    end process;

end Behavioral;
```

6.2 Código do Testbench

Listing 2: Código do Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--use std.env.stop;
```

```
entity RISC_V_Uniciclo_TB is
end RISC_V_Uniciclo_TB;

architecture Behavioral of RISC_V_Uniciclo_TB is
    component RISC_V_Uniciclo
        Port (
            clk  : in STD_LOGIC; -- Sinal de clock
            reset  : in STD_LOGIC -- Sinal de reset
        );
    end component;

    -- Sinais de entrada
    signal clk : STD_LOGIC := '0';
    signal reset : STD_LOGIC := '0';

    -- Periodo do clock
    constant clk_period : time := 20 ns;

begin
    -- Instanciacao do processador
    tb_Processador: entity work.RISC_V_Uniciclo
        port map (
            clk => clk,
            reset => reset
        );

    -- Geracao do clock
    process
    begin

        wait for clk_period;
        for i in 0 to 500 loop
            clk <= not clk;
            wait for clk_period / 2;
        end loop;

        wait;

    end process;

    -- Processo de teste
```

```
    test_process: process
    begin
        -- Reset inicial
        reset <= '1';
        wait for clk_period;
        reset <= '0';

        -- Finalizacao do teste
        report "Finalizado com sucesso!" severity note;
        wait; -- Encerra a simulacao

    end process;
end Behavioral;
```

7 Lista de instruções testadas

- 00500093 // ADDi x1, x0, 5 — x1 = 5
- 00A00113 // ADDi x2, x0, 10 — x2 = 10
- 002081B3 // ADD x3, x1, x2 — x3 = 15
- 40118233 // SUB x4, x3, x1 — x4 = 10
- 0031A2B3 // SLT x5, x4, x3 — x5 = 1
- 0041C333 // XOR x6, x3, x4 — x6 = 5
- 00115393 // SRLi x7, x2, 1 — x7 = 5
- 4021D413 // SRAi x8, x3, 2 — x8 = 3
- 00346493 // ORi x9, x8, 3 — x9 = 3
- 00547513 // ANDi x10, x8, 5 — x10 = 1
- 00302023 // SW x3, 0(x0) — Mem[0] = 15
- 00402423 // SW x4, 8(x0) — Mem[8] = 10
- 00502823 // SW x5, 16(x0) — Mem[16] = 1
- 00002583 // LW x11, 0(x0) — x11 = 15

- 00802603 // LW x12, 8(x0) — x12 = 10
- 01002683 // LW x13, 16(x0) — x13 = 1
- 00010717 // AUIPC x14, 0x1000 — x14 = PC + 0x1000000
- 00020797 // LUI x15, 0x2000 — x15 = 0x2000000

8 Dificuldades Encontradas

Durante o desenvolvimento do projeto, encontrei algumas dificuldades:

A primeira delas foi o fato da execução concorrente. Ainda não tinha experiência com esse tipo de execução, então precisei aprender e entender esse paradigma de programação. Além disso, tive alguns bugs durante a execução do código, como os conceitos de atualização de registradores durante a segunda subida de clock.

Outro ponto foi lidar com a complexidade de seguir um fluxo de execução correto de cada instrução, demandando bastante atenção aos sinais e tentando entender como cada coisa se encaixava.

9 Conclusão

Foi um projeto bem desafiador para mim, até porque escolhi fazer sozinho e não tive muito tempo disponível. Foram diversos detalhes para serem observados, mas no final foi excelente entender melhor como funciona essa parte de hardware.