Projeto Final
Implementação do Processador Risc-V Pipeline

Leonardo Maximo Silva, 200022172
Wallace Ben Teng Lin Wu, 200028880

1

1. Objetivos

Implementar uma versão simplificada do Processador Risc-V Pipeline em Linguagem de Descrição de Hardware (VHDL) e verificar seu funcionamento por meio de Simulações.

2. Implementação

Implementou-se um Pipeline básico sem as otimizações de adiantamento de dados e do tratamento de exceções. As instruções implementadas foram:

- Lógico-Aritméticas: ADD, ADDi, SUB, AND, ANDi, LUI, SLT, OR, ORi, XOR, XORi, SLLi, SRLi, SRAi, SLTi, SLTu, SLTUi, AUIPC
- Subrotinas: JAL, JALR
- Saltos: BEQ, BNE, BLT, BGE, BGEU, BLTU
- Memória: LW, SW

Para que essas Instruções sejam implementadas, foi necessário alterar o Modelo do Pipeline Básico visto em [1], tendo em vista que essa Arquitetura não possui suporte para as instruções LUI, SLT, AuiPC, JAL, JALR, BNE, BLT, BGE, BGEU e BLTU. Implementou-se, então, a Arquitetura 1, para a qual foram alterados o seguintes parâmetros do Processador Básico:

- Adicionou-se mais uma entrada de dados ao Multiplexador correspondente ao PC, de modo a fornecer suporte para o salto incondicional JALR, que define o PC como a soma do valor de um registrador e o imediato;
- O Sinal MemRead foi descartado, tendo em vista que a Memória de Dados só é lida ou escrita a cada Ciclo, então se pode reaproveitar o sinal de controle MemWrite = 0 para representar o MemRead;
- Adicionaram-se mais três entradas de dados ao Multiplexador correspondente à saída de Dados no Quinto Estágio do Pipeline, de modo a se tornar possível a implementação das instruções JAL, JALR, AUIPC e LUI a partir da Multiplexação das Entradas de Dados;
- Adicionou-se um Bloco PC Control, de modo a gerar os Sinais necessários para que o Registrador PC receba a próxima instrução (PC+4) ou a instrução proveniente de um Salto Condicional ou Incondicional.
- Finalmente, também propagou-se mais sinais pelos registradores auxiliares do pipeline, como o PC+4, PC+IMM e o IMM, extendendo-os até o MEM/WB.

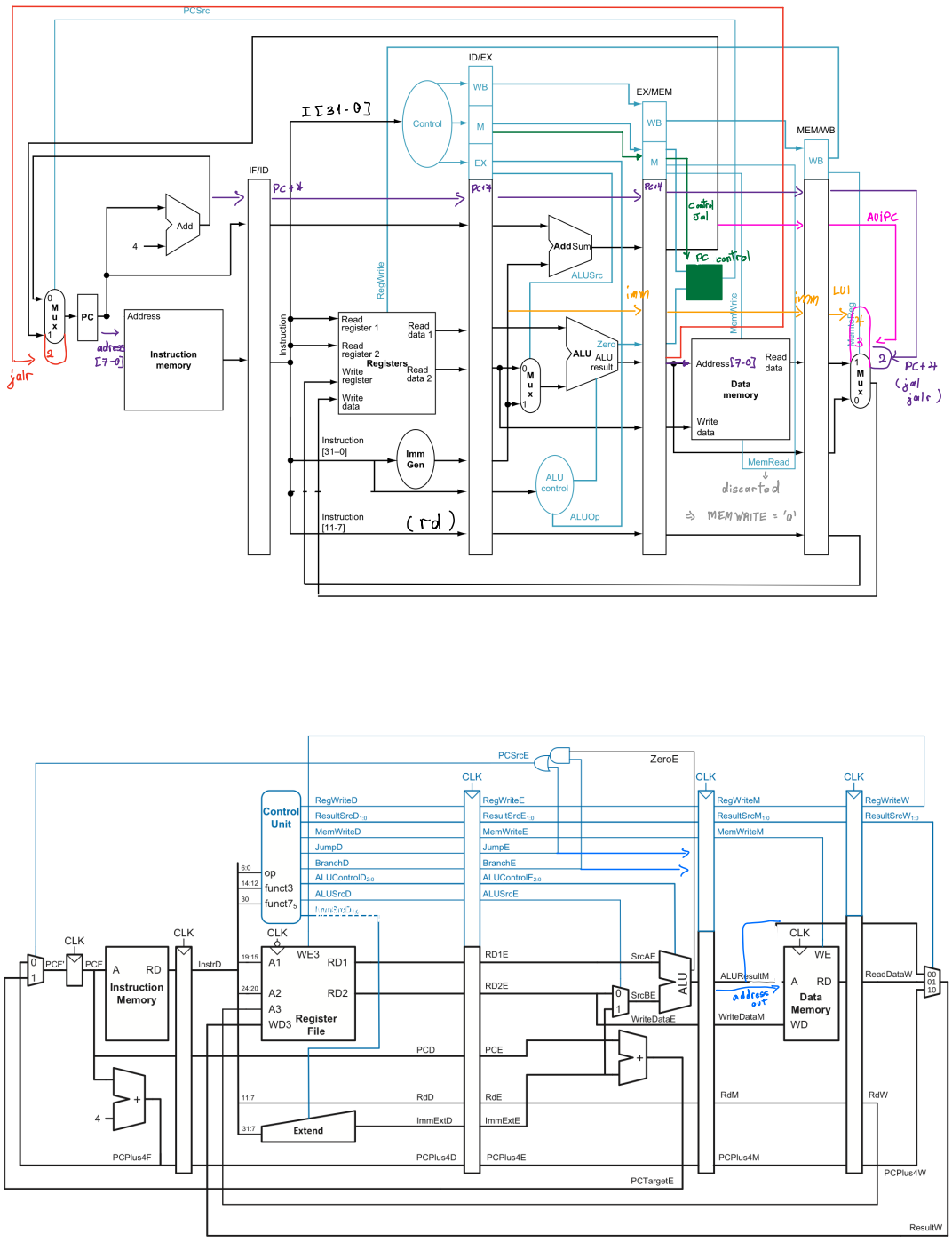

Figure 1. Esquemático do Pipeline Implementado

O Código Implementado, Arquivos de Testes utilizados e demais referências podem ser vistas em `https://github.com/wallacelw/PipelineProcessor.git`.

A seguir, será descrita a implementação de cada Componente pertencente ao Processador:

2.1. Registrador PC

O Registrador PC possui 3 entradas, uma para o Clock, outra para o Reset e uma Entrada de 32 bits correspondente à instrução do Programa. Ele também possui duas saídas, uma correspondente ao número da Instrução de saída e outra de 8 dígitos correspondente ao Endereço da Instrução a ser buscada na memória de instruções. Por ser um Registrador, é apenas atualizado com o próximo valor de entrada durante uma borda de subida do Clock. O Reset implementado é assíncrono, de modo que é independente do Sinal de Clock. A implementação pode ser vista em 2.

```
entity PC is
        port (
                PCReg_clk : in std_logic;
        PCReg_rst : in std_logic;
                PCReg_in : in std_logic_vector(31 downto 0);
        PCReg_out : out std_logic_vector(31 downto 0);

        PCReg_address_out : out std_logic_vector(7 downto 0)
        );
end entity;

architecture df of PC is

begin

    process(PCReg_clk, PCReg_rst) begin
        if (PCReg_rst = '1') then
            PCReg_out <= x"00000000";

        elsif (rising_edge(PCReg_clk)) then
            PCReg_out <= PCReg_in;
            PCReg_address_out <= PCReg_in(7 downto 0);

        end if;
    end process;

end df;
```

Figure 2. Esquemático do Registrador PC Implementado

2.2. Memória de Instruções (MI)

A Memória de Instruções é constituída por um Array de 256 linhas, de modo que cada linha corresponda a uma instrução de 32 bits. Sua entrada é um endereço de 8 bits da Instrução provinda do Registrador PC e sua saída é uma Instrução de 32 bits. As instruções ficam contidas em um arquivo .txt de 256 Linhas fixas, de modo que programas menores possuem o resto das instruções preenchidas como NOPs, implementados

como addi x0,x0,0. Como o Arquivo .txt correspondente à memória possui 256 linhas e o Código realizado é sensível ao número de linhas do Arquivo e o PC é incrementado de 4 para representar um salto de instruções de 32 bits, é necessário, durante a atribuição da saída da memória, dividir o endereço desejado por 4 de modo a se obter a saída desejada. A Memória de Instruções é, logo, uma memória apenas de leitura. A Implementação pode ser vista em 3

```
entity MEM_INSTR is
    port (
        MI_address : in std_logic_vector(7 downto 0);
        MI_Instr_out : out std_logic_vector(31 downto 0)
    );
end entity;

architecture df of MEM_INSTR is

    Type rom_type is array (0 to (2**MI_address'length)-1) of std_logic_vector(MI_Instr_out'range);

    impure function init_rom_hex return rom_type is

        file text_file : text open read_mode is "ROM_data.txt";
        variable text_line : line;
        variable rom_content : rom_type;

    begin
        for i in 0 to (2**MI_address'length)-1 loop
            readline(text_file, text_line);
            hread(text_line, rom_content(i));
        end loop;

        return rom_content;
    end function;

    signal mem : rom_type := init_rom_hex;

begin

    MI_Instr_out <= mem(to_integer(unsigned(MI_address))/4);

end df;
```

Figure 3. Esquemático da Memória de Instruções Implementada

2.3. Memória de Dados (MD)

A Memória de Dados é implementada de forma análoga à Memória de Instruções, porém, sem a leitura de um Arquivo, de modo que não necessita da leitura de um Arquivo e a memória é inicialmente tomada como estando vazia, ou seja, todos os seus valores são 0. Também é necessário que seja possível escrever e ler da Memória de Dados. Foi necessário, logo, que a Leitura e Escrita na Memória esteja sincronizada com o Clock do Processador para que a leitura e escrita sejam feitas durante a borda de subida do clock. Também é necessário um Sinal de WriteEnable, para garantir que não se escrevam valores incorretos no Componente e também uma entrada de Dados para se escreverem novos valores na Memória. Da Implementação, é notável o uso de um Sinal auxiliar read_address para garantir que o valor retornado pela Memória seja o novo valor logo que é atualizado. A implementação pode ser vista em 4.

```
entity MEM_DADOS is
    port (
        MD_clk : in std_logic;
        MD_we : in std_logic;
        MD_address : in std_logic_vector(7 downto 0);
        MD_datain : in std_logic_vector(31 downto 0);
        MD_dataout : out std_logic_vector(31 downto 0)
    );
end entity;

architecture df of MEM_DADOS is

    Type ram_type is array (0 to (2**MD_address'length)-1) of std_logic_vector(MD_dataout'range);
    signal mem : ram_type;
    signal read_address : std_logic_vector(MD_address'range);

begin

    Write: process (MD_clk) begin
        if (rising_edge(MD_clk) and MD_we = '1') then
            mem(to_integer(unsigned(MD_address))/4) <= MD_datain;
        end if;
        read_address <= MD_address; -- Adds 1 cycle
    end process;

    MD_dataout <= mem(to_integer(unsigned(MD_address))/4);

end df;
```

Figure 4. Esquemático da Memória de Dados Implementada

2.4. Banco de Registradores (XREGS)

O Banco de Registradores é composto por um Array de 32 Registradores de 32 bits, os quais são utilizados para o armazenamento temporário de dados, de modo que seja possível ler e escrever nesses Registradores. É necessário, logo, três entradas de 8 bits contendo o Endereço dos Registradores, sendo 2 de leitura (rs1 e rs2) e 1 de escrita (rd); uma entrada de dados para se escrever o valor desejado em rd e duas saídas de dados contendo os valores de 32 bits dentro dos Registradores rs1 e rs2. É importante notar que, por ser um banco de Registradores, toda as operações realizadas devem ocorrer dentro da borda de subida de clock, de modo que é comum que se leia em um ciclo e se escreva no outro, o que torna possível instruções do tipo ”t0 ¡= t0+1”. É necessário, também, Sinais de Controle, em especial, o Sinal de Write Enable, para permitir a escrita em rd. A implementação pode ser vista em 5 e em 6.

```
entity XREGS is
    port (
        XRegs_clk, XRegs_wren, XRegs_rst : in std_logic;
        XRegs_rs1, Xregs_rs2, XRegs_rd : in std_logic_vector(4 downto 0);
        XRegs_data : in std_logic_vector(31 downto 0);
        XRegs_ro1, XRegs_ro2 : out std_logic_vector(31 downto 0)
        );
end entity;

architecture df of XREGS is
```

Figure 5. Esquemático do Banco de Registradores Implementador (Primeira Parte)

```
architecture df of XREGS is

    type regArray is array(natural range <>) of std_logic_vector(31 downto 0);
    signal breg: regArray(31 downto 0);

begin

        --Leitura
    XRegs_ro1 <= (others => '0') when (XRegs_rs1 = "00000")
                 else breg(to_integer(unsigned(XRegs_rs1)));

    XRegs_ro2 <= (others => '0') when (Xregs_rs2 = "00000")
                 else breg(to_integer(unsigned(Xregs_rs2)));

    process(XRegs_clk, XRegs_rst, XRegs_wren, breg,XRegs_data) begin

        if (rising_edge(XRegs_clk)) then

            --Reset
            if (XRegs_rst = '1') then
                for i in 31 downto 0 loop
                    breg(i) <= x"00000000";
                end loop;

            --Escrita
            elsif (XRegs_wren = '1') then
                breg(to_integer(unsigned(XRegs_rd))) <= XRegs_data;

            end if;

        end if;

    end process;

end df;
```

Figure 6. Esquemático do Banco de Registradores Implementador (Segunda Parte)

2.5. Unidade Lógico-Aritmética(ULA)

A Unidade Lógico Aritmética (ULA) é implementada de modo a selecionar, por meio da função ”if”, ”elsif” e ”else”, a correta operação correspondente a cada Instrução. É necessário, dessa forma, duas entradas de dados de 32 bits, as quais podem corresponder a rs1,rs2 ou a um dado imediato. Sua saída é composta pela saída de 32 bits correspondente à operaçcão realizada pela ULA com suas entradas. A ULA também possui uma saída ”zero”, a qual verifica se os dados contidos em rs1 são iguais aos contidos em rs2. Como a ULA deve selecionar entre diversas operações, as quais podem usar os campos opcode, funct3 e funct7 de uma instrução, é necessário um bloco para o Controle da ULA, o qual gera o Sinal ALUOp, que controla a Operação desejada.

As Operações Implementadas pela ULA são: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, STL, STLU, SGE, SGEU, SEQ, SNE. A implementação da ULA pode ser vista em 7, 8 e em 9.

```
entity ALU is
    port (
        ALU_opcode : in std_logic_vector(3 downto 0);
        ALU_A, ALU_B : in std_logic_vector(31 downto 0);
        ALU_Z : out std_logic_vector(31 downto 0);
        ALU_zero : out std_logic
    );
end entity ALU;

architecture df of ALU is

    signal a32 : std_logic_vector(31 downto 0);
```

Figure 7. Esquemático da ULA implementada (Primeira Parte)

```
begin

    ALU_Z <= a32;

    proc_ula: process (ALU_A, ALU_B, ALU_opcode, a32) begin
        if (a32 = x"00000000") then ALU_zero <= '1'; else ALU_zero <= '0'; end if;

        case ALU_opcode is
            when "0000" => -- ADD
                a32 <= std_logic_vector(signed(ALU_A) + signed(ALU_B));
            when "0001" => -- SUB
                a32 <= std_logic_vector(signed(ALU_A) - signed(ALU_B));
            when "0010" => -- AND
                a32 <= ALU_A and ALU_B;
            when "0011" => -- OR
                a32 <= ALU_A or ALU_B;
            when "0100" => -- XOR
                a32 <= ALU_A xor ALU_B;
            when "0101" => -- SLL
                a32 <= std_logic_vector( unsigned(ALU_A) sll To_integer( signed(ALU_B) ));
            when "0110" => -- SRL
                a32 <= std_logic_vector( unsigned(ALU_A) srl To_integer( signed(ALU_B) ));
            when "0111" => -- SRA
                a32 <= std_logic_vector( signed(ALU_A) sra To_integer( signed(ALU_B) ) );
            when "1000" => -- SLT
                if (signed(ALU_A) < signed(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1001" => -- SLTU
                if (unsigned(ALU_A) < unsigned(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1010" => -- SGE
                if (signed(ALU_A) >= signed(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
```

Figure 8. Esquemático da ULA implementada (Segunda Parte)

```
            when "1001" => -- SLTU
                if (unsigned(ALU_A) < unsigned(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1010" => -- SGE
                if (signed(ALU_A) >= signed(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1011" => -- SGEU
                if (unsigned(ALU_A) >= unsigned(ALU_B)) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1100" => -- SEQ
                if (ALU_A = ALU_B) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when "1101" => -- SNE
                if (ALU_A /= ALU_B) then a32 <= x"00000001";
                else a32 <= x"00000000";
                end if;
            when others => -- Default
                a32 <= x"00000000";
        end case;
    end process;

end df;
```

Figure 9. Esquemático da ULA implementada (Terceira Parte)

2.6. Multiplexadores

O Primeiro Multiplexador implementado corresponde à Entrada do PC e seleciona entre os Sinais de Entrada PC+4, um Salto Condicional dado por uma instrução condicional ou a um salto incondicional dado por uma instrução JAL ou JALR. Ele é controlado pelo Bloco PC Control, o qual é um desmembramento do Bloco Principal de Controle.

O Segundo Multiplexador corresponde à entrada da ULA e escolhe entre os Sinal de dados de rs2 ou de um dado imediato, o qual é gerado por um Gerador

de Imediatos que, de acordo com a instrução selecionada, utiliza campos diferentes da instrução para compô-lo ou o zera por completo. Esse Sinal é Controlado pelo sinal ALUSrc gerado pelo Bloco de Controle. Sua implementação pode ser vista em

O Terceiro Multiplexador corresponde à ”saída” do Processador e corresponde ao Sinal de Entrada de Dados para Escrita do Banco de Registradores. Ele possui 4 entradas de dados, de modo a selecionar entre as entradas correspondentes a uma instrução LUI, AUIPC, resultado da ULA ou leitura da Memória de Dados. Os Sinais de LUI e AUIPC são calculados de forma diferente do da ULA por ser mais simples carregar o resultado do LUI, ou seja, o imediato 20 bits à esquerda ou do AUIPC, ou seja, o imediato 20 bits à esquerda somado com o valor do Registrador PC, e carregá-lo pelos Registradores de Pipeline que gerar um novo Sinal de Controle da ULA e uma nova Operação para realizá-los.

2.7. Somadores

Foram utilizados dois Somadores. O primeiro corresponde à Soma do Atual valor de PC com 4 o Segundo corresponde à Soma de PC com o valor de Imediato 20 bits à Esquerda, ou seja, à instrução AUIPC. Sua implementação pode ser vista em 10 e em 11.

```
---- Soma PC com IMM
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ADDER_PC_IMM is
        port (
                PC_IMM_Adder_PC : in std_logic_vector(31 downto 0);
        PC_IMM_Adder_Imm : in std_logic_vector(31 downto 0);
        PC_IMM_Adder_out : out std_logic_vector(31 downto 0)
        );
end entity;

architecture df of ADDER_PC_IMM is

begin

    PC_IMM_Adder_out <= std_logic_vector(signed(PC_IMM_Adder_PC) + signed(PC_IMM_Adder_Imm));

end df;
```

Figure 10. Esquemático do Adder de Imediato implementado

```
entity ADDER_4 is
        port (
                PCAdder_in : in std_logic_vector(31 downto 0);
        PCAdder_out : out std_logic_vector(31 downto 0)
        );
end entity;

architecture df of ADDER_4 is

begin

    PCAdder_out <= std_logic_vector(signed(PCAdder_in) + 4);

end df;
```

Figure 11. Esquemático do Adder do PC implementado

2.8. Blocos de Controle

Foram utilizados três blocos de Controle. O Primeiro Corresponde ao Bloco Central (Control), que gera os Sinais correspondentes a todos os Estágios do Pipeline. Ele é gerado durante a Etapa de Fetch e é carregado pelas outras Etapas pelos Registradores de Pipeline, sendo responsável pelos Sinais de Controle do Multiplexador da ULA (ALUSrc), Operação da ULA (ALUOp), correspondentes à Etapa de Execução; Saltos Condicionais (Branch), Saltos Incondicionais (Jal), Escrita na Memória (MemWrite), correspondentes à Etapa de Acesso à Memória e Escrita no Registrador rd (RegWrite) e resultado do Multiplexador que leva para a entrada de Dados rd (ResultSrc) correspondentes à Etapa de Write-Back. Sua implementação pode ser vista em 12, 13, 14, 15 e em 16.

```
entity CONTROL_MODULE is
        port (
        CONTROL_instr : in std_logic_vector(31 downto 0);

        --EX
        CONTROL_ALUSrc : out std_logic;
        CONTROL_ALUOp : out std_logic_vector(1 downto 0);

        --MEM
        CONTROL_Branch : out std_logic;
        CONTROL_Jal : out std_logic_vector(1 downto 0); -- indica se a instrução é do tipo Jal (== 01) or Jalr (== 10)
        CONTROL_MemWrite : out std_logic; -- memWrite = 1; memRead = 0

        --WB
        CONTROL_RegWrite : out std_logic;
        CONTROL_ResultSrc : out std_logic_vector(2 downto 0) -- Mem2Reg extendido
        );
end entity;
```

Figure 12. Esquemático do Bloco de Controle implementada (Primeira Parte)

```
architecture df of CONTROL_MODULE is

    signal opcode : std_logic_vector(7 downto 0);
    signal funct3 : std_logic_vector(3 downto 0);
    signal funct7 : std_logic_vector(7 downto 0);

begin

    opcode <= '0' & CONTROL_instr(6 downto 0);
    funct3 <= '0' & CONTROL_instr(14 downto 12);
    funct7 <= '0' & CONTROL_instr(31 downto 25);

    process (CONTROL_instr, opcode, funct3, funct7) begin

    ---- Lógico-Aritméticas

    -- Type R : ADD, SUB, AND, SLT, OR, XOR, SLTU
    if (opcode = x"33") then
        CONTROL_ALUSrc <= '0'; -- Use Reg Value for ALU
        CONTROL_ALUOp <= "10"; -- R type or I type
        CONTROL_Branch <= '0'; -- Don't update PC due to Branch
        CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
        CONTROL_MemWrite <= '0'; -- Don't Write in Memory
        CONTROL_RegWrite <= '1'; -- Write Back to Register
        CONTROL_ResultSrc <= "000"; -- Register receives ALU output
```

Figure 13. Esquemático do Bloco de Controle implementada (Segunda Parte)

```
-- Type I : ADDi, ANDi, ORi, XORi, SLLi, SRLi, SRAi, SLTi, SLTUi
elsif (opcode = x"13") and (funct3 = x"0") then
    CONTROL_ALUSrc <= '1'; -- Use IMM Value for ALU
    CONTROL_ALUOp <= "10"; -- R type or I type
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "000"; -- Register receives ALU output

-- TYPE U
elsif (opcode = x"17") then -- AUIPC
    CONTROL_ALUSrc <= '-'; -- (Don't care)
    CONTROL_ALUOp <= "--"; -- (Don't care)
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "011"; -- Register receives IMM + PC

elsif (opcode = x"37") then -- LUI
    CONTROL_ALUSrc <= '-'; -- (Don't care)
    CONTROL_ALUOp <= "--"; -- (Don't care)
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "100"; -- Register receives IMM
```

Figure 14. Esquemático do Bloco de Controle implementada (Terceira Parte)

```
---- Subrotinas

-- JAL
elsif (opcode = x"6F") then
    CONTROL_ALUSrc <= '-'; -- (Don't Care)
    CONTROL_ALUOp <= "--"; -- (Don't Care)
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "01"; -- Update PC due to Jal
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "010"; -- Register RD receives PC+4

-- JALR
elsif (opcode = x"67") and (funct3 = x"0") then
    CONTROL_ALUSrc <= '1'; -- Use IMM Value for ALU
    CONTROL_ALUOp <= "10"; -- R type or I type
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "10"; -- Update PC due to Jalr
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "010"; -- Register RD receives PC+4
```

Figure 15. Esquemático do Bloco de Controle implementada (Quarta Parte)

```
---- Saltos
-- Type SB (BRANCH):  BEQ, BNE, BLT, BGE, BLTU, BGEU
elsif (opcode = x"63") then
    CONTROL_ALUSrc <= '0'; -- Use Reg Value for ALU
    CONTROL_ALUOp <= "01"; -- Branch Type
    CONTROL_Branch <= '1'; -- update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '0'; -- Don't Write in Memory
    CONTROL_RegWrite <= '0'; -- Don't Write Back to Register
    CONTROL_ResultSrc <= "---"; -- (Don't Care)


---- Memória

-- LW (I)
elsif (opcode = x"03") and (funct3 = x"2") then
    CONTROL_ALUSrc <= '1'; -- Use Imm Value for ALU
    CONTROL_ALUOp <= "00"; -- Memory Type
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '0'; -- Read Mem
    CONTROL_RegWrite <= '1'; -- Write Back to Register
    CONTROL_ResultSrc <= "001"; -- Register receives Mem output

-- SW (S)
elsif (opcode = x"23") and (funct3 = x"2") then
    CONTROL_ALUSrc <= '1'; -- Use Imm Value for ALU
    CONTROL_ALUOp <= "00"; -- Memory Type
    CONTROL_Branch <= '0'; -- Don't update PC due to Branch
    CONTROL_Jal <= "00"; -- Don't update PC due to Jal(r)
    CONTROL_MemWrite <= '1'; -- Write in Mem
    CONTROL_RegWrite <= '0'; -- Write Back to Register
    CONTROL_ResultSrc <= "---"; -- (Don't Care)
```

Figure 16. Esquemático do Bloco de Controle implementada (Quinta Parte)

O Bloco Alu Control é responsável por, a partir do Comando AluOp, gerar uma instrução de 4 bits correspondente a uma instrução da ULA considerando também, quando necessário, os Segmentos funct3 e funct7 da Instrução original gerada. Sua implementação pode ser vista em 17, 18, 19 e em 20.

```
entity ALU_Control is
        port (
                ALU_Control_instr : in std_logic_vector(31 downto 0);
        ALU_Control_alu_op : in std_logic_vector(1 downto 0);
        ALU_Control_out : out std_logic_vector(3 downto 0)
        );
end entity;

architecture df of ALU_Control is

    signal funct3 : std_logic_vector(3 downto 0);
    signal funct7 : std_logic_vector(7 downto 0);

begin

    funct3 <= ('0' & (ALU_Control_instr(14 downto 12)));
    funct7 <= ('0' & (ALU_Control_instr(31 downto 25)));

    process(ALU_Control_alu_op, ALU_Control_instr, funct3, funct7) begin

        case ALU_Control_alu_op is

        -- for LW, SW
        when "00" =>
            ALU_Control_out <= "0000"; -- ADD
```

Figure 17. Esquemático do Controle da ULA implementado (Primeira Parte)

```
    process(ALU_Control_alu_op, ALU_Control_instr, funct3, funct7) begin

        case ALU_Control_alu_op is

        -- for LW, SW
        when "00" =>
            ALU_Control_out <= "0000"; -- ADD

        -- for BRANCHES (SB)
        when "01" =>
            if    (funct3 = x"0") then -- BEQ
                ALU_Control_out <= "1100"; -- SEQ

            elsif (funct3 = x"1") then -- BNE
                ALU_Control_out <= "1101"; -- SNE

            elsif (funct3 = x"4") then -- BLT
                ALU_Control_out <= "1000"; -- SLT

            elsif (funct3 = x"5") then -- BGE
                ALU_Control_out <= "1010"; -- SGE

            elsif (funct3 = x"6") then -- BLTU
                ALU_Control_out <= "1001"; -- SLTU

            elsif (funct3 = x"7") then -- BGEU
                ALU_Control_out <= "1011"; -- SGEU

            end if;
```

Figure 18. Esquemático do Controle da ULA implementado (Segunda Parte)

```
-- for R type and I type
when "10" =>
    if    (funct3 = x"0" and funct7 = x"00") then
       ALU_Control_out <= "0000"; -- ADD

    elsif (funct3 = x"0" and funct7 = x"20") then
        ALU_Control_out <= "0001"; -- SUB

    elsif (funct3 = x"7" and funct7 = x"00") then
        ALU_Control_out <= "0010"; -- AND

    elsif (funct3 = x"6" and funct7 = x"00") then
        ALU_Control_out <= "0011"; -- OR

    elsif (funct3 = x"4" and funct7 = x"00") then
        ALU_Control_out <= "0100"; -- XOR

    elsif (funct3 = x"1" and funct7 = x"00") then
        ALU_Control_out <= "0101"; -- SLL

    elsif (funct3 = x"5" and funct7 = x"00") then
        ALU_Control_out <= "0110"; -- SRL

    elsif (funct3 = x"5" and funct7 = x"20") then
        ALU_Control_out <= "0111"; -- SRA

    elsif (funct3 = x"2"                   ) then
        ALU_Control_out <= "1000"; -- SLT

    elsif (funct3 = x"3"                   ) then
        ALU_Control_out <= "1001"; -- SLTU

    elsif (funct3 = x"0"                   ) then --JALR
        ALU_Control_out <= "0000"; -- ADD

    end if;

 when others => ALU_Control_out <= "1111";
```

Figure 19. Esquemático do Controle da ULA implementado (Terceira Parte)

```
entity PC_Control is
        port (
                PC_Control_Branch : in std_logic;
        PC_Control_Zero : in std_logic;
        PC_Control_Jal : in std_logic_vector(1 downto 0);

        PC_Control_PCSRC : out std_logic_vector(1 downto 0)
        );
end entity;

architecture df of PC_Control is

begin

    process(PC_Control_Branch, PC_Control_Zero, PC_Control_Jal) begin

        if (PC_Control_Branch and (not PC_Control_Zero)) then -- Branch
            PC_Control_PCSRC <= "01";

        elsif (PC_Control_Jal = "01") then -- JAL
            PC_Control_PCSRC <= "01";

        elsif (PC_Control_Jal = "10") then -- JALR
            PC_Control_PCSRC <= "10";

        else -- PC + 4
            PC_Control_PCSRC <= "00";

        end if;

    end process;
end df;
```

Figure 20. Esquemático do Controle da ULA implementado (Quarta Parte)

O Bloco PC Control é responsável por gerar o valor de 2 bits utilizados para selecionar entre as entradas possíveis do Multiplexador conectado ao Registrador PC. Sua implementação pode ser vista em 21.

```
entity PC_Control is
        port (
                PC_Control_Branch : in std_logic;
        PC_Control_Zero : in std_logic;
        PC_Control_Jal : in std_logic_vector(1 downto 0);

        PC_Control_PCSRC : out std_logic_vector(1 downto 0)
        );
end entity;

architecture df of PC_Control is

begin

    process(PC_Control_Branch, PC_Control_Zero, PC_Control_Jal) begin

        if (PC_Control_Branch and (not PC_Control_Zero)) then -- Branch
            PC_Control_PCSRC <= "01";

        elsif (PC_Control_Jal = "01") then -- JAL
            PC_Control_PCSRC <= "01";

        elsif (PC_Control_Jal = "10") then -- JALR
            PC_Control_PCSRC <= "10";

        else -- PC + 4
            PC_Control_PCSRC <= "00";

        end if;

    end process;
end df;
```

Figure 21. Esquemático do Controle do PC

2.9. Registradores de Pipeline

Foram implementados 4 Registradores de Pipeline, os quais servem como intermediários e armazenam os valores necessários das Instruções anteriores para dar Segmento à execução da instrução no Pipeline. Esses Registradores são denominados como ID/Ex (entre os Estados de Fetch e Decode), EX/Mem (entre os Estados de Decode e Acesso à Memória) e Mem/WB (entre os Estados de Acesso à Memória e Write-Back). Entre os Sinais passados por esses Processadores, destacam-se os Sinais de Controle, os quais são gerados para cada instrução e passados de instrução para instrução e o Sinal de PC+4, o qual precisou ser passado para os próximos Estágios do Pipeline para garantir a Execução das Instruções LUI e AUIPC. A implementação desses Registradores pode ser vista em 22, 23, 24, 25, 26, 27 e 28.

```
entity IF_ID_PIPE is
        port (
                IF_ID_clk : in std_logic;

                IF_ID_PC_in : in std_logic_vector(31 downto 0);
        IF_ID_PC_out : out std_logic_vector(31 downto 0);

        IF_ID_Instr_in : in std_logic_vector(31 downto 0);
        IF_ID_Instr_out : out std_logic_vector(31 downto 0);

        IF_ID_rs1_out : out std_logic_vector(4 downto 0);
        IF_ID_rs2_out : out std_logic_vector(4 downto 0);
        IF_ID_rd_out : out std_logic_vector(4 downto 0);

        IF_ID_PC_PLUS_4_in : in std_logic_vector(31 downto 0);
        IF_ID_PC_PLUS_4_out : out std_logic_vector(31 downto 0)
        );
end entity;

architecture df of IF_ID_PIPE is

begin

    process(IF_ID_clk) begin
        if (rising_edge(IF_ID_clk)) then
            IF_ID_PC_out <= IF_ID_PC_in;

            IF_ID_Instr_out <= IF_ID_Instr_in;

            IF_ID_rs1_out <= IF_ID_Instr_in(19 downto 15);
            IF_ID_rs2_out <= IF_ID_Instr_in(24 downto 20);
            IF_ID_rd_out <= IF_ID_Instr_in(11 downto 7);

            IF_ID_PC_PLUS_4_out <= IF_ID_PC_PLUS_4_in;
        end if;
    end process;

end df;
```

Figure 22. Esquemático do Registrador IF/ID

```
entity ID_EX_PIPE is
        port (
                ID_EX_clk : in std_logic;

                ID_EX_PC_in : in std_logic_vector(31 downto 0);
        ID_EX_PC_out : out std_logic_vector(31 downto 0);

        ID_EX_ro1_in : in std_logic_vector(31 downto 0);
        ID_EX_ro1_out : out std_logic_vector(31 downto 0);

        ID_EX_ro2_in : in std_logic_vector(31 downto 0);
        ID_EX_ro2_out : out std_logic_vector(31 downto 0);

        ID_EX_rd_in : in std_logic_vector(4 downto 0);
        ID_EX_rd_out : out std_logic_vector(4 downto 0);

        ID_EX_imm_in : in std_logic_vector(31 downto 0);
        ID_EX_imm_out : out std_logic_vector(31 downto 0);

        ID_EX_instr_in : in std_logic_vector(31 downto 0);
        ID_EX_instr_out : out std_logic_vector(31 downto 0);

        ID_EX_PC_PLUS_4_in : in std_logic_vector(31 downto 0);
        ID_EX_PC_PLUS_4_out : out std_logic_vector(31 downto 0);

        -- control
        ID_EX_CONTROL_ALUSrc_in : in std_logic;
        ID_EX_CONTROL_ALUOp_in : in std_logic_vector(1 downto 0);
        ID_EX_CONTROL_Branch_in : in std_logic;
        ID_EX_CONTROL_Jal_in : in std_logic_vector(1 downto 0);
        ID_EX_CONTROL_MemWrite_in : in std_logic;
        ID_EX_CONTROL_RegWrite_in : in std_logic;
        ID_EX_CONTROL_ResultSrc_in : in std_logic_vector(2 downto 0);

        ID_EX_CONTROL_ALUSrc_out : out std_logic;
        ID_EX_CONTROL_ALUOp_out : out std_logic_vector(1 downto 0);
        ID_EX_CONTROL_Branch_out : out std_logic;
        ID_EX_CONTROL_Jal_out : out std_logic_vector(1 downto 0);
        ID_EX_CONTROL_MemWrite_out : out std_logic;
        ID_EX_CONTROL_RegWrite_out : out std_logic;
        ID_EX_CONTROL_ResultSrc_out : out std_logic_vector(2 downto 0)
```

Figure 23. Esquemático do Registrador EX/MEM (Primeira Parte)

```
architecture df of ID_EX_PIPE is

begin

    process(ID_EX_clk) begin
        if (rising_edge(ID_EX_clk)) then
            ID_EX_PC_out <= ID_EX_PC_in;

            ID_EX_ro1_out <= ID_EX_ro1_in;

            ID_EX_ro2_out <= ID_EX_ro2_in;

            ID_EX_rd_out <= ID_EX_rd_in;

            ID_EX_imm_out <= ID_EX_imm_in;

            ID_EX_instr_out <= ID_EX_instr_in;

            ID_EX_PC_PLUS_4_out <= ID_EX_PC_PLUS_4_in;

            ID_EX_CONTROL_ALUSrc_out <= ID_EX_CONTROL_ALUSrc_in;
            ID_EX_CONTROL_ALUOp_out <= ID_EX_CONTROL_ALUOp_in;
            ID_EX_CONTROL_Branch_out <= ID_EX_CONTROL_Branch_in;
            ID_EX_CONTROL_Jal_out <= ID_EX_CONTROL_Jal_in;
            ID_EX_CONTROL_MemWrite_out <= ID_EX_CONTROL_MemWrite_in;
            ID_EX_CONTROL_RegWrite_out <= ID_EX_CONTROL_RegWrite_in;
            ID_EX_CONTROL_ResultSrc_out <= ID_EX_CONTROL_ResultSrc_in;
        end if;
    end process;

end df;
```

Figure 24. Esquemático do Registrador IF/EX (Segunda Parte)

```
entity EX_MEM_PIPE is
        port (
                EX_MEM_clk : in std_logic;

                EX_MEM_PC_plus_Imm_in : in std_logic_vector(31 downto 0);
        EX_MEM_Pc_plus_Imm_out : out std_logic_vector(31 downto 0);

        EX_MEM_zero_in : in std_logic;
        EX_MEM_zero_out : out std_logic;

        EX_MEM_ro2_in : in std_logic_vector(31 downto 0);
        EX_MEM_ro2_out : out std_logic_vector(31 downto 0);

        EX_MEM_rd_in : in std_logic_vector(4 downto 0);
        EX_MEM_rd_out : out std_logic_vector(4 downto 0);

        EX_MEM_Alu_in : in std_logic_vector(31 downto 0);
        EX_MEM_Alu_out : out std_logic_vector(31 downto 0);

        EX_MEM_imm_in : in std_logic_vector(31 downto 0);
        EX_MEM_imm_out : out std_logic_vector(31 downto 0);

        EX_MEM_PC_PLUS_4_in : in std_logic_vector(31 downto 0);
        EX_MEM_PC_PLUS_4_out : out std_logic_vector(31 downto 0);

        EX_MEM_address_out : out std_logic_vector(7 downto 0);

        -- control
        EX_MEM_CONTROL_Branch_in : in std_logic;
        EX_MEM_CONTROL_Jal_in : in std_logic_vector(1 downto 0);
        EX_MEM_CONTROL_MemWrite_in : in std_logic;
        EX_MEM_CONTROL_RegWrite_in : in std_logic;
        EX_MEM_CONTROL_ResultSrc_in : in std_logic_vector(2 downto 0);

        EX_MEM_CONTROL_Branch_out : out std_logic;
        EX_MEM_CONTROL_Jal_out : out std_logic_vector(1 downto 0);
        EX_MEM_CONTROL_MemWrite_out : out std_logic;
        EX_MEM_CONTROL_RegWrite_out : out std_logic;
        EX_MEM_CONTROL_ResultSrc_out : out std_logic_vector(2 downto 0)
        );
end entity;
```

Figure 25. Esquemático do Registrador EX/MEM (Primeira Parte)

```
architecture df of EX_MEM_PIPE is

begin

    process(EX_MEM_clk) begin
        if (rising_edge(EX_MEM_clk)) then
            EX_MEM_Pc_plus_Imm_out <= EX_MEM_PC_plus_Imm_in;

            EX_MEM_zero_out <= EX_MEM_zero_in;

            EX_MEM_ro2_out <= EX_MEM_ro2_in;

            EX_MEM_rd_out <= EX_MEM_rd_in;

            EX_MEM_Alu_out <= EX_MEM_Alu_in;

            EX_MEM_imm_out <= EX_MEM_imm_in;

            EX_MEM_PC_PLUS_4_out <= EX_MEM_PC_PLUS_4_in;

            EX_MEM_address_out <= EX_MEM_Alu_in(7 downto 0);

            EX_MEM_CONTROL_Branch_out <= EX_MEM_CONTROL_Branch_in;
            EX_MEM_CONTROL_Jal_out <= EX_MEM_CONTROL_Jal_in;
            EX_MEM_CONTROL_MemWrite_out <= EX_MEM_CONTROL_MemWrite_in;
            EX_MEM_CONTROL_RegWrite_out <= EX_MEM_CONTROL_RegWrite_in;
            EX_MEM_CONTROL_ResultSrc_out <= EX_MEM_CONTROL_ResultSrc_in;
        end if;
    end process;

end df;
```

Figure 26. Esquemático do Registrador EX/MEM (Segunda Parte)

```
entity MEM_WB_PIPE is
        port (
                MEM_WB_clk : in std_logic;

                MEM_WB_mem_data_in : in std_logic_vector(31 downto 0);
        MEM_WB_mem_data_out : out std_logic_vector(31 downto 0);

        MEM_WB_Alu_in : in std_logic_vector(31 downto 0);
        MEM_WB_Alu_out : out std_logic_vector(31 downto 0);

        MEM_WB_rd_in : in std_logic_vector(4 downto 0);
        MEM_WB_rd_out : out std_logic_vector(4 downto 0);

        MEM_WB_PC_plus_Imm_in : in std_logic_vector(31 downto 0);
        MEM_WB_Pc_plus_Imm_out : out std_logic_vector(31 downto 0);

        MEM_WB_imm_in : in std_logic_vector(31 downto 0);
        MEM_WB_imm_out : out std_logic_vector(31 downto 0);

        MEM_WB_PC_PLUS_4_in : in std_logic_vector(31 downto 0);
        MEM_WB_PC_PLUS_4_out : out std_logic_vector(31 downto 0);

        -- control
        MEM_WB_CONTROL_RegWrite_in : in std_logic;
        MEM_WB_CONTROL_ResultSrc_in : in std_logic_vector(2 downto 0);

        MEM_WB_CONTROL_RegWrite_out : out std_logic;
        MEM_WB_CONTROL_ResultSrc_out : out std_logic_vector(2 downto 0)
        );
end entity;
```

Figure 27. Esquemático do Registrador MEM/WB (Primeira Parte)

```
architecture df of MEM_WB_PIPE is

begin

    process(MEM_WB_clk) begin
        if (rising_edge(MEM_WB_clk)) then
            MEM_WB_mem_data_out <= MEM_WB_mem_data_in;

            MEM_WB_Alu_out <= MEM_WB_Alu_in;

            MEM_WB_rd_out <= MEM_WB_rd_in;

            MEM_WB_PC_plus_Imm_out <= MEM_WB_PC_plus_Imm_in;

            MEM_WB_imm_out <= MEM_WB_imm_in;

            MEM_WB_PC_PLUS_4_out <= MEM_WB_PC_PLUS_4_in;

            MEM_WB_CONTROL_RegWrite_out <= MEM_WB_CONTROL_RegWrite_in;
            MEM_WB_CONTROL_ResultSrc_out <= MEM_WB_CONTROL_ResultSrc_in;
        end if;
    end process;

end df;
```

Figure 28. Esquemático do Registrador MEM/WB (Segunda Parte)

2.10. Processador Principal

Para a Implementação do Processador, criaram-se todos os componentes supracitados no arquivo design.vhd e uniram-se seus módulos através de fios. O Processador possuía apenas dois Sinais de Entrada, reset e clock, e nenhum Sinal de Saída, tendo em vista que os valores desejados podem ser obtidos e testados através dos Sinais internos do Próprio Processador.

Para se averiguar o funcionamento do Processador, logo, monitoravam-se os Sinais internos do Processador e dos Componentes, em especial, os correspondentes à saída do Multiplexador presente no Estágio Write-Back, ao PC+4 e aos valores dos Registradores do Banco de Registradores (Xregs) por meio da Análise de forma de Onda correspondente.

2.11. Testbench

Para a realização do Testbench, foi necessária apenas a instanciação de um Componente Processador e a definição de um Clock e de um Reset.

Como não é possível assumir quais valores são correspondentes aos Sinais do Processador antes de seu início, escolheu-se iniciá-lo por meio de um Ciclo de Clock correspondente ao Reset de modo a tornar os Sinais Correspondentes à Instrução do PC, ao Banco de Registradores e a Memória de Instruções como zero. Para a realização do Clock, realizou-se um for de modo a se alternar entre o sinal ’1’ e ’0’ para o Clock até o fim da Execução do Programa. Sua implementação pode ser vista em 29.

```
begin

    dut: CPU port map (
        clock => clock_i,
        reset => reset_i
    );

    drive: process begin

        reset_i <= '0';
        clock_i <= '0';
        wait for 10 ns;

        reset_i <= '1';
        clock_i <= '1';
        wait for 10 ns;

        reset_i <= '0';
        clock_i <= '0';
        wait for 10 ns;

        for i in 0 to 30 loop

            clock_i <= '1';
            wait for 10 ns;

            clock_i <= '0';
            wait for 10 ns;
```

3. Testes

O Processador Implementado executou todas as instruções propostas, produzindo a saída esperada ao se analisar as formas de onda. Ao se testar Códigos Simples implementados no Software Rars, o Processador também apresentou desempenho Satisfatório, executando as Instruções conforme esperado. A seguir, são apresentados alguns exemplos de Resultados obtidos.

3.1. Teste do LUI

```
addi t1 x0 1
addi t2 x0 2
addi x0 x0 0 # nop
addi x0 x0 0 # nop
addi x0 x0 0 # nop
lui t0 7
```

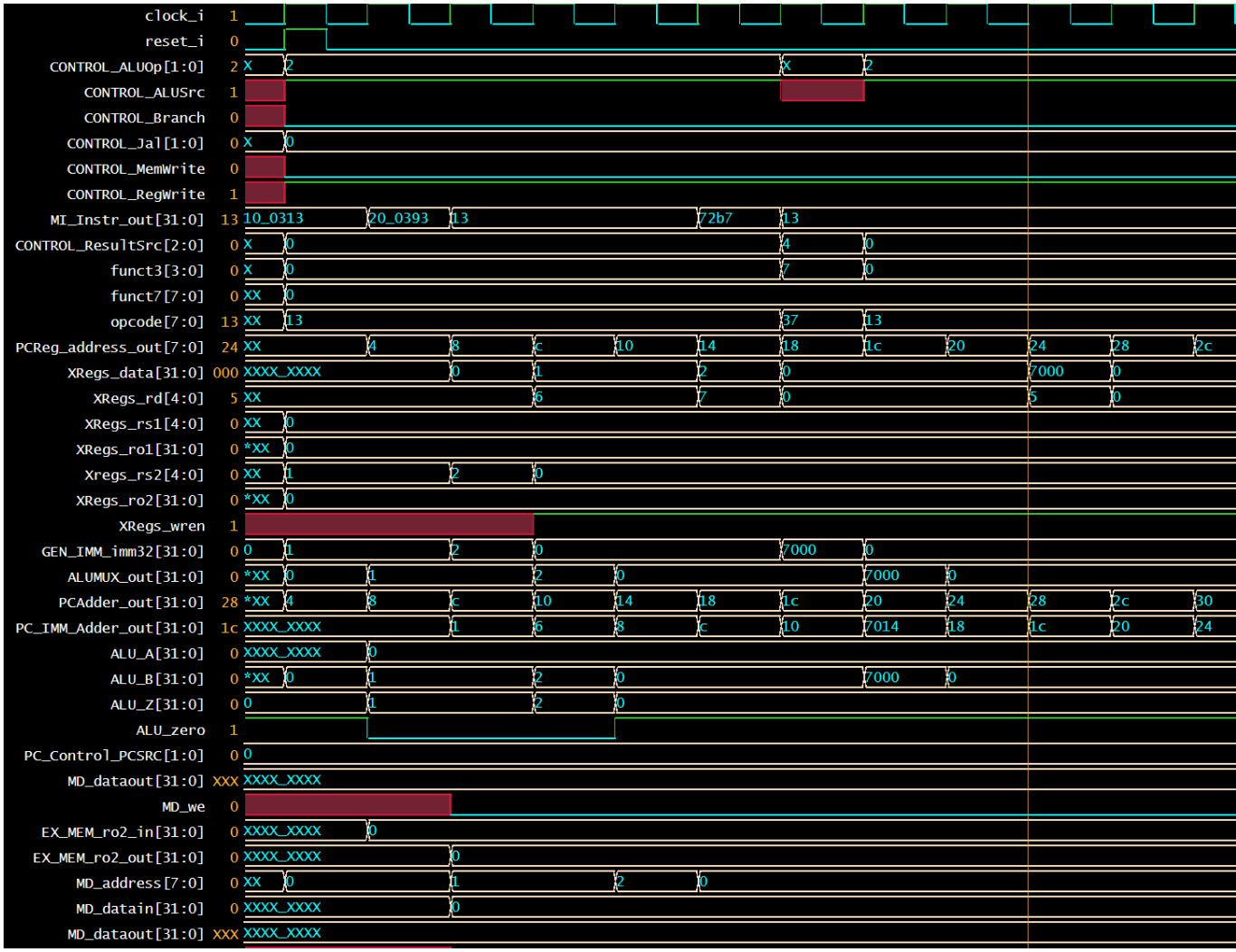

Figure 30. Diagrama de Sinais do exemplo de LUI

3.2. Teste do BEQ

```
testeBEQ:
addi  t1  x0  2
addi  t2  t2  2
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
beq  t1  t2  testeBEQ
```

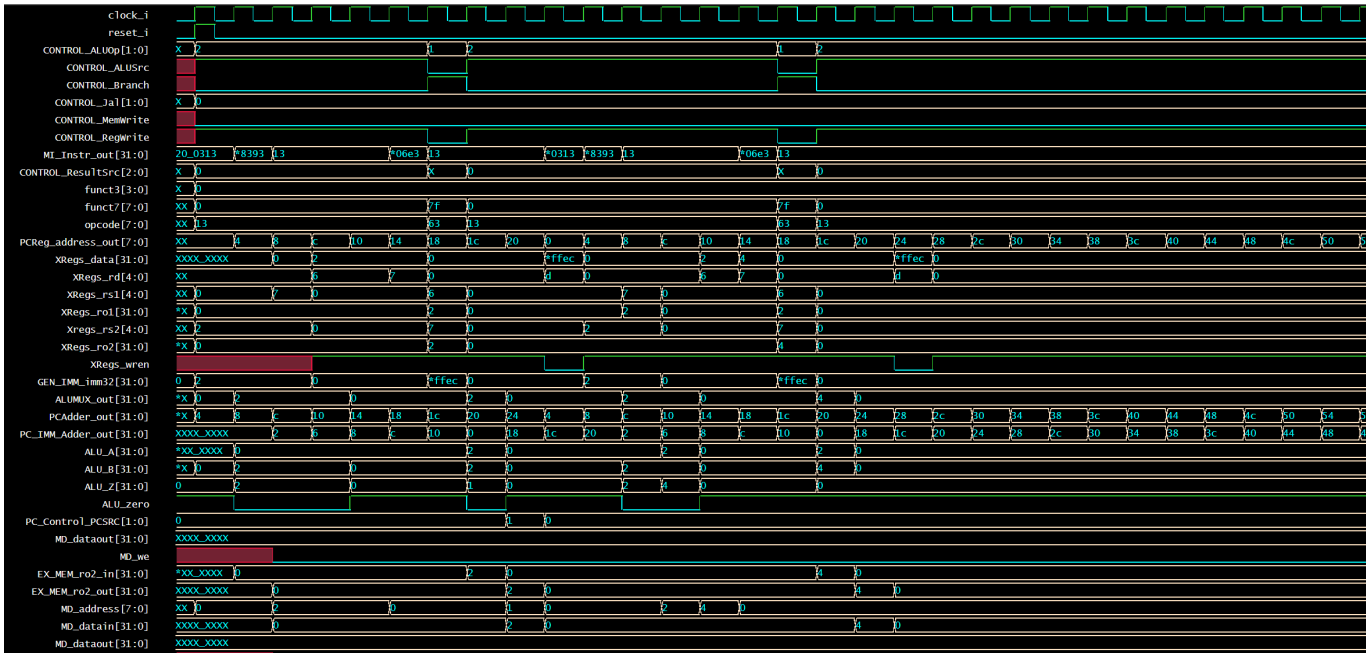

Figure 31. Diagrama de Sinais do exemplo de BEQ

3.3. Teste do SW e LW

```
addi  t1  x0  16
addi  t2  x0  2
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
sw  t2 ,  0(t1)
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
addi  x0  x0  0  # nop
lw  t0 ,  0(t1)
```

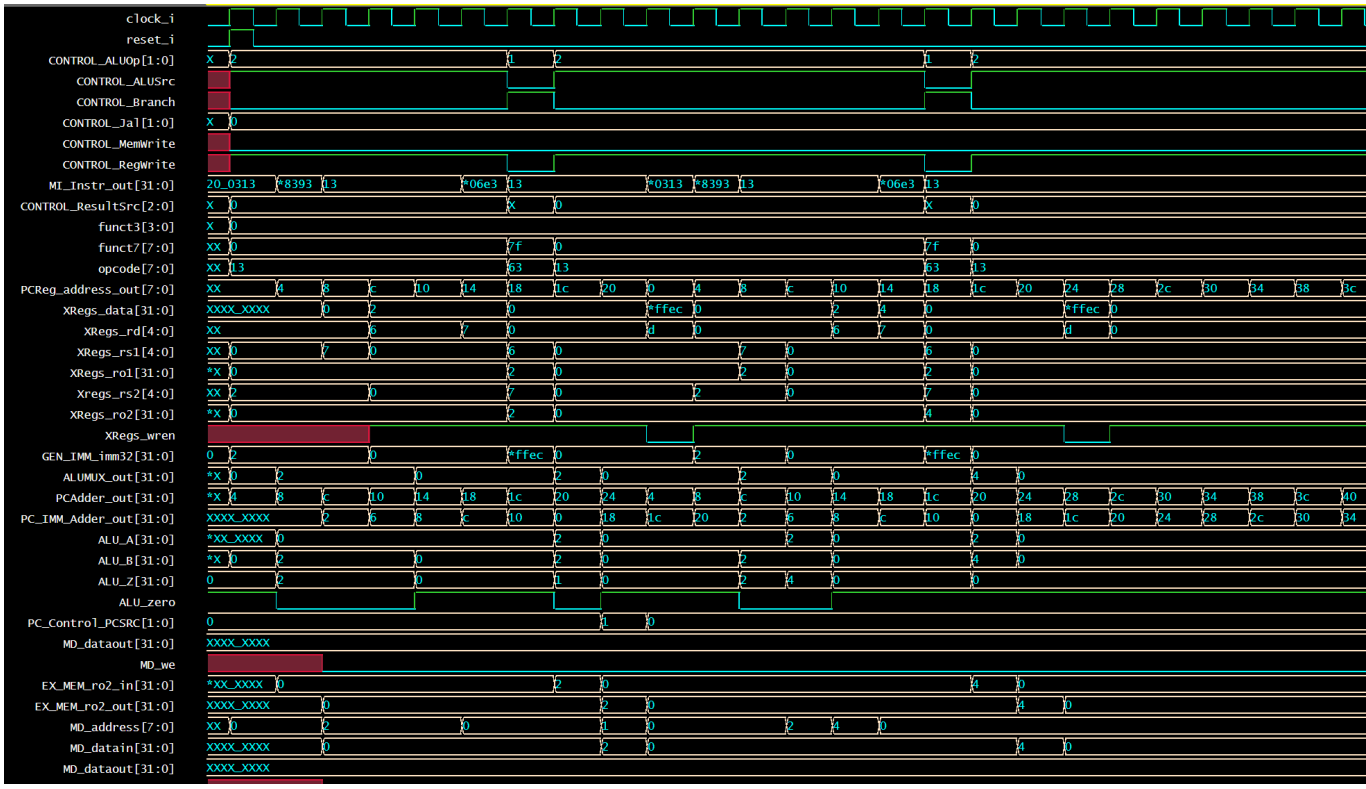

Figure 32. Diagrama de Sinais do exemplo de LW e SW

4. Conclusão

O Processador Risc-V Pipeline básico foi implementado de modo satisfatório, obtendo-se Resultados esperados para as instruções implementadas e Códigos Simples produzidos no RARS. A implementação necessitou algumas adaptações em relação à Arquitetura inicialmente proposta, de modo a fornecer suporte a algumas instruções e a simplificar o funcionamento do Processador.

References

[1] Ricardo Pezzuol Jacobi. *RISC-V Pipeline: Unidade Operativa e Controle*. https://aprender3.unb.br/pluginfile.php/2353912/mod$_resource/content/1/14_Pipeline_RISCV.pdf$. Jan. 2023.