
Universidade de Brasília

Departamento de Ciência da Computação


CIC0099–Organização e Arquitetura de Computadores - 2025.1

# Relatório Técnico
# Laboratório 2
# CPU RISC-V UNICICLO


**Grupo 6:**
Albert Teixeira Soares – 21/1035108
Rychard Ryan Alves de Moraes - 19/0095229
Fernando Nunes de Freitas – 22/2014661
Humberto Alves Mesquita – 21/1026566
Wesley Reno da Silva Lira – 17/ 0072291

Professora: Carla Maria Chagas e Cavalcante Koike
Turma 4



Brasília- DF, Junho de 2025

# Sumário

# 1 Resumo

Este relatório descreve a implementação de um processador RISC-V Uniciclo, realizada como parte do Laboratório 2 da disciplina CIC0099 - Organização e Arquitetura de Computadores - Unificado. Foi implementado um processador Uniciclo compatível com a ISA RV32I reduzida. As instruções suportadas incluem add, sub, and, or, slt, lw, sw, beq, jal, jalr, e addi. Para tal, foram desenvolvidos os seguintes módulos principais:

- Um Banco de Registradores com três portas de leitura simultâneas (rs1, rs2 e disp);
- Um Gerador de Imediatos;
- Uma Unidade Lógica Aritmética (ULA) que suporta as operações add, sub, and, or, slt, e zero;
- O Controlador da ULA e o Bloco Controlador geral do processador;
- O processador Uniciclo completo, integrando todos os módulos desenvolvidos.

O objetivo principal deste laboratório foi capacitar os alunos da disciplina com a Linguagem de Descrição de Hardware (HDL) Verilog e familiarizá-los com o software de síntese QUARTUS Prime. Além disso, buscou-se desenvolver a capacidade de análise e síntese de sistemas digitais utilizando uma HDL, culminando na implementação prática de uma CPU Uniciclo.

A implementação foi realizada utilizando a linguagem Verilog e o software QUARTUS Prime v24.1, com base no arcabouço fornecido no projeto TopDE.qar. Um programa de teste (de1.s) foi escrito para verificar a corretude de todas as instruções implementadas, sendo este testado primeiramente no simulador Rars. As memórias de instrução e dados foram geradas a partir dos arquivos .mif exportados do Rars (formato MIF 32). Após a implementação, foram realizados os seguintes passos para validação e análise:

- Visualização e registro do netlist RTL view dos módulos;
- Levantamento dos requisitos físicos e temporais do processador completo, verificando o cumprimento dos slacks;
- Simulações funcionais e temporais por forma de onda utilizando o programa de1.s para demonstrar o funcionamento correto da CPU;
- Determinação experimental da máxima frequência de clock utilizável da CPU, ajustando a frequência no arquivo .vwf e analisando a simulação temporal.

A implementação do processador Uniciclo foi bem-sucedida, conforme demonstrado pelas simulações funcionais e temporais com o programa de1.s, que confirmaram o comportamento correto das instruções implementadas. A análise dos requisitos físicos e temporais permitiu verificar o desempenho do processador, e a determinação da frequência máxima de clock forneceu uma métrica crucial para a avaliação da implementação. Apesar dos mais diversos problemas com as ferramentas e com o entendimento do comportamento dos dados para cada instrução, a equipe obteve sucesso na implementação e nos testes, o que portanto, valida os objetivos de aprendizado propostos pelo laboratório.

# 2 Introdução

**E**ste relatório tem como objetivo documentar o processo de implementação de um processador RISC simplificado, conforme proposto no projeto da disciplina de Organização e Arquitetura de Computadores. O desenvolvimento foi realizado em linguagem Verilog, utilizando como base o simulador Logisim Evolution para os testes e validações dos módulos digitais.

O foco do projeto é implementar uma arquitetura de processador que, embora simplificada, seja funcional e capaz de executar instruções básicas do conjunto de instruções RISC-V. Entre os principais módulos desenvolvidos estão: Unidade Lógica e Aritmética (ULA), Controlador da ULA, Bloco Controlador, Banco de Registradores, Memória de Dados, Unidade de Controle de Programa e o Datapath. Cada componente foi desenvolvido de forma modular, buscando respeitar os princípios de escalabilidade, legibilidade e reutilização.

A relevância deste projeto é multifacetada. No cenário tecnológico atual, a compreensão de como os processadores são projetados e implementados é fundamental para futuros engenheiros e cientistas da computação. A implementação de uma CPU Uniciclo com uma ISA RV32I reduzida oferece uma oportunidade prática de aplicar conceitos teóricos de arquitetura de computadores, desenvolver a capacidade de análise e síntese de sistemas digitais, e observar o impacto das decisões de design no desempenho e nos requisitos físicos do hardware.

Ao longo deste relatório, serão descritos os detalhes técnicos das implementações. O objetivo final é demonstrar o correto funcionamento do processador para um subconjunto representativo de instruções, validando sua lógica de controle, execução e memória.

# 3 Desenvolvimento

## 3.1 Implementação de Instruções e Teste no RARS (item 1.1)

**Enunciado:** Escreva um programa de1.s que teste a corretude da implementação de todas as 9 + 2 instruções e teste no Rars. Dica: Use o registrador t0 para visualizar resultados!

**Implementação:** De acordo com o enunciado, foi criado um programa simples com as instruções add, sub, and, or, slt, lw,sw, beq, jal, e ainda jalr e addi. Além disso, o registrador t0 foi usado como parâmetro de destino em todas as instruções com o objetivo de visualizar os resultados dos processamentos.

```
.word 1

.text
        li gp,0x10010000 # gp = 0x10010000
        addi t1, zero, 4 # t1 = 4
        lw t2, 0(gp) # t2 = 1
        addi t0, t2, 0  #t0 = 1
        add t0, t0, t1 # t0 = 5
        sub t0, t0, t1 # t0  = 1
        slt t0, t1, t0 # t0 = 0
        or t0, t0, t2 # t0 = 1
        and t0, t0, zero # t0 = 0
        beq t0, t1, exit # Não Pula só pra testar
        beq t0, zero, pula # pula para "pula"
exit:
        jalr zero, ra, 0 # pula para o endereÃ§o salvo em ra

pula:
        jal ra, exit # pula para o label exit e salva o endereco em ra
        addi t0,t1, 10 # t0 = 14
        sw t0, 4(gp) # armazena o t0(14) em gp+4
```

Figura 1: Arquivo De1.s aberto no RARS

| Instruções | Saída esperada | Observações |
|---|---|---|
| `li gp, 0x10010000` | `gp = 0x10010000` | Carrega endereço base do segmento de dados |
| `addi t1, zero, 4` | `t1 = 4` | Inicializa t1 com o valor 4 |
| `lw t0, 0(gp)` | `t0 = 1` | Carrega palavra da memória no endereço apontado pelo gp`gp` |
| `add t0, t0, t1` | `t0 = 5` | Soma t0 (1) com t1 (4) |
| `sub t0, t0, t1` | `t0 = 1` | Subtrai t1 (4) de t0 (5) |
| `slt t0, t0, t1` | `t0 = 1` | 1 ¡ 4, então t0 = 1 |
| `or t0, t0, zero` | `t0 = 1` | OR com 0 mantém valor |
| `and t0, t0, zero` | `t0 = 0` | AND com 0 zera t0 |
| `beq t0, t1, exit` | Não salta | `t0 = 0`, `t1 = 4`, não são iguais |
| `beq t0, zero, pula` | Salta para `pula` | `t0 = 0`, condição satisfeita |
| `jal ra, exit` | Salta para `exit` | Salto incondicional, salva `PC+4` em `ra` |
| `addi t0, zero, 10` | Não executada | Ignorada por salto anterior |
| `sw t0, 4(gp)` | Não executada | Ignorada também |
| `exit: jalr ra` | Retorna | Retorna para o endereço salvo em `ra` |

Tabela 1: Execução de instruções no simulador RARS

## 3.2 Leituras Simultâneas no Banco de Registradores (item 1.2)

**Enunciado:** Implemente o Banco de Registradores com 3 leituras simultâneas: rs1, rs2 e disp. Stack Pointer (sp) inicial: 0x1001_03FC e Global Pointer (gp) inicial: 0x1001_0000;

**Implementação:** De posse do projeto TopDE.qar, foi possível obter a implementação em verilog do Banco de Registradores a ser usado no construção do Processador Uniciclo. O código começa declarando o módulo Registers que modela os 32 registradores do RISC-V, com capacidade de leitura, escrita, e visualização de um registrador específico.
No módulo é declarado as entradas: clock e reset, sinal de controle para escrita, os índices dos registradores a serem lidos (iReadRegister1 e iReadRegister2), o índice do registrador a ser escrito, valor a ser escrito(iWriteData), e o índice do registrador a ser exibido (iRegDispSelect). Também é declarado as saídas: os dados lidos dos registradores(oReadData1e oReadData2), e oRegDisp - conteúdo de um registrador específico (para debug ou visualização). Podemos notar também, pela sintaxe que nessa etapa também é definido a quantidade de bits usadas em cada fio de saída e entrada.
Logo após é declarado os 32 regitradores de 32 bits cada,na linha 18, e a nomeação dos registradores 2 e 3 como SP (stack pointer) e GP (global pointer), de acordo com o que o enunciado pede. O código define parameter SP = 5'd2 e GP = 5'd3, que são os registradores x2 e x3, exatamente conforme o padrão RISC-V. É feito também uma inicialização com zero de todos os registradores e a atribuição dos registradores SP e GP com STACK_ADRESS E DATA_ADRESSS.
Por fim, são implementadas 3 instruções que realizam leituras combinacionais simultâneas dos registradores indicados pelos sinais de entrada, enviando os dados lidos para as respec-

tivas saídas. Nas linhas 33 a 36, cada linha conecta uma saída a um valor lido diretamente de um registrador, sem depender do clock. Na linha 33 por exemplo, lê-se o valor do registrador cujo índice está em iReadRegister1 e envia esse valor para a saída oReadData1. O bloco a partir da linha 40 implementa a escrita síncrona no banco de registradores, que só acontece na borda de subida do clock (iCLK) e quando o sinal iRegWrite está ativado. Também garante a inicialização dos registradores durante o reset e protege o registrador x0 contra escrita indevida. Se o reset for ativado, todos os registradores são zerados e os registradores x2 (SP) e x3 (GP) recebem valores iniciais específicos. Caso contrário, se o sinal iRegWrite estiver ativo durante a borda de subida do clock, o dado presente em iWriteData é gravado no registrador indicado por iWriteRegister, desde que esse registrador não seja o x0. Esse comportamento, portanto, caracteriza uma escrita síncrona, controlada por clock, habilitação e endereço.

```
1   `ifndef PARAM
2      `include "Parametros.v"
3   `endif
4
5   module Registers (
6       input wire          iCLK, iRST, iRegWrite,
7       input wire  [4:0]   iReadRegister1, iReadRegister2, iWriteRegister,
8       input wire  [31:0]  iWriteData,
9       output wire [31:0]  oReadData1, oReadData2,
10
11      input wire  [4:0]   iRegDispSelect,
12      output reg  [31:0]  oRegDisp
13
14      );
15
16
17  /* Register file */
18  reg [31:0] registers[31:0];
19
20  parameter  SPR=5'd2, GPR=5'd3;                    // SP e GP
21
22  reg [5:0] i;
23
24  initial
25     begin
26        for (i = 0; i <= 31; i = i + 1'b1)
27           registers[i] = 32'd0;
28        registers[SPR] <= STACK_ADDRESS;
29        registers[GPR] <= DATA_ADDRESS;
30     end
31
32
33  assign oReadData1 =  registers[iReadRegister1];
34  assign oReadData2 =  registers[iReadRegister2];
35
36  assign oRegDisp   =  registers[iRegDispSelect];
37
38
39
40  always @(posedge iCLK or posedge iRST)
41  begin
42      if (iRST)
43      begin // reseta o banco de registradores e pilha
44          for (i = 0; i <= 31; i = i + 1'b1)
45              registers[i] = 32'b0;
46          registers[SPR] <= STACK_ADDRESS;
47          registers[GPR] <= DATA_ADDRESS;
48      end
49      else
50      begin
51        i<=6'b0; // para não dar warning
52        if(iRegWrite && (iWriteRegister != 5'b0))
53              registers[iWriteRegister] <= iWriteData;
54      end
55  end
56
57  endmodule
58
```

Figura 2: Arquivo .v do Gerador de Imediatos

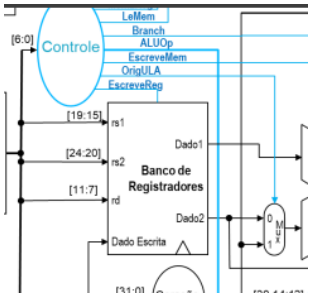



Figura 3: Entradas e Saídas do Banco de Registradores

## 3.3 Gerador de Imediatos (item 1.3)

**Enunciado:** Implemente o Gerador de Imediatos.

**Implementação:** Primeiro é feito um módulo ImmGen que interpreta o opcode da instrução e extrai corretamente o campo de imediato, com extensão de sinal, para ser usado por outras partes do processador como a ALU ou o PC (contador de programa). É definido uma entrada de 32 bits e uma saída que é o valor imediato gerado, já com extensão de sinal. Em always é usado uma lógica combinacional( ou seja, sem depender do clock) que faz a geração do valor imediato (oImm) a partir da instrução "iInstrucao", com base no seu tipo (determinado pelo opcode, que são os bits iInstrucao[6:0]).

```
1  `ifndef PARAM
2      `include "Parametros.v"
3  `endif
4
5  /* Unidade de geração do imediato */
6
7  module ImmGen (
8      input        [31:0] iInstrucao,
9      output logic [31:0] oImm
10 );
11
12
13 always @ (*)
14     case (iInstrucao[6:0])
15         OPC_LOAD,
16         OPC_OPIMM,
17         OPC_JALR:
18             oImm <= {{20{iInstrucao[31]}}, iInstrucao[31:20]};
19         OPC_STORE:
20             oImm <= {{20{iInstrucao[31]}}, iInstrucao[31:25], iInstrucao[11:7]};
21         OPC_BRANCH:
22             oImm <= {{20{iInstrucao[31]}}, iInstrucao[7], iInstrucao[30:25], iInstrucao[11:8], 1'b0};
23         OPC_JAL:
24             oImm <= {{12{iInstrucao[31]}}, iInstrucao[19:12], iInstrucao[20], iInstrucao[30:21], 1'b0};
25         default:
26             oImm <= ZERO;
27     endcase
28
29
30 endmodule
```

Figura 4: Decodificador de Imediatos

## 3.4 Geração de Arquivos a partir do RARS(item 1.4)

**Enunciado:** No Rars16_Custom2, vá em File/Dump Memory e exporte (MIF 32 Format) para o arquivo de1 (sem extensão). Os arquivos de1_text.mif e de1_data.mif serão gerados.

**Implementação:** Após implementar o De1.s e testar, os seguintes passos foram feitos:

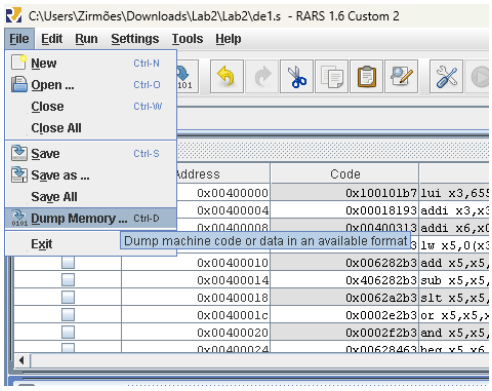



Figura 5: Passo 1- Dump Memory

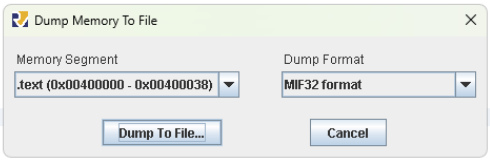



Figura 6: Passo 2- de1_text.mif

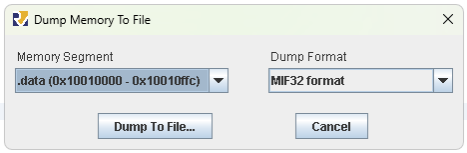



Figura 7: Passo 3- de1_data.mif

Após esses passos, os dois arquivos foram salvos dentro da pasta do projeto, para uso posterior.

## 3.5 Implementação da ULA Mínima(item 1.5)

**Enunciado:** Implemente a ULA mínima necessária (add, sub, and, or, slt, zero).

**Implementação:** Analisando os arquivos que foram disponibilizados para o desenvolvimento do processador, notou-se que na ULA há mais funções implementadas do que o que requerido no enunciado. Diante disso, a equipe optou por fazer simplicações nessas funções, transformando em comentário as linhas que não seriam usadas. Além disso, foi adicionado, nas linhas 13 e 82, códigos que permitirão o tratamento de procedimentos não permitidos pela ULA do processador desenvolvido aqui, como funções de multiplicação, divisão, erros entre outras. Trata-se da saída Zero, que geralmente serve para indicar se o resultado da operação foi zero, e tem usos importantes no controle do fluxo do programa, como por exemplo em instruções de desvio condicional. Sendo assim, a linha 82 atribui

à saída oZero um valor lógico, e a linha 13 declara o sinal oZero como saída do módulo ALU. De forma resumida, foi definida as entradas e saídas, e também as operações da ULA.

**As entradas da ULA:**
- iControl [4:0]: sinal que define a operação a ser executada;
- iA, iB [31:0]: operandos de 32 bits (com sinal).

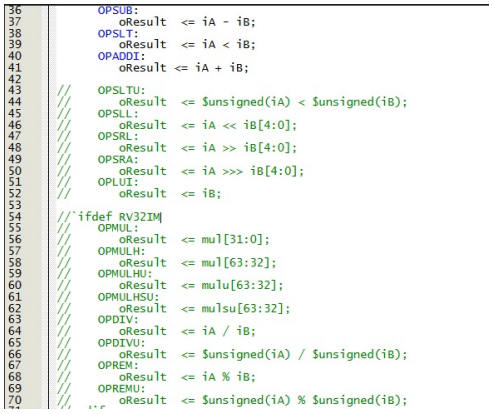
**As saídas da ULA:**
- oResult [31:0]: resultado da operação;
- oZero: saída adicional (linha adicionada, mas ainda não usada no código).

A partir da linha 25, está sendo implementado o bloco de seleção de operações da ULA (ALU), usando uma estrutura case, que escolhe qual operação executar com base no valor do sinal de controle iControl. Isso torna a ULA versátil e capaz de executar instruções diferentes, conforme requerido por programas em execução.
Os casos implementados são:

- OPAND: realiza iA & iB (AND bit a bit);
- OPOR: realiza iA — iB (OR bit a bit);
- OPADD: realiza iA + iB (adição);
- OPSUB: realiza iA - iB (subtração);
- OPSLT: realiza iA ¡ iB (set less than);
- OPADDI: realiza iA + iB (adição com imediato);
- OPNULL: realiza oResult ¡= ZERO (operação nula, saída zerada);
- default: realiza oResult ¡= ZERO (caso não identificado, saída zerada).

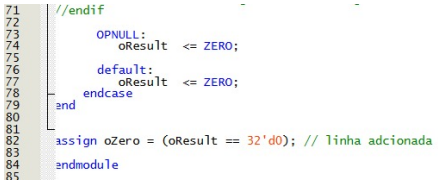
```
1   `ifndef PARAM
2       `include "Parametros.v"
3   `endif
4
5   //`define RV32I;
6   `define RV32IM;
7
8   module ALU (
9       input       [4:0]  iControl,
10      input signed [31:0] iA,
11      input signed [31:0] iB,
12      output logic [31:0] oResult,
13      output wire oZero // Linha adcionada
14      );
15
16  // wire [4:0] iControl=OPDIV;    // Usado para as analises
17
18  `ifdef RV32IM
19  wire [63:0] mul, mulu, mulsu;
20  assign mul  = $signed(iA) * $signed(iB);
21  assign mulu = $unsigned(iA) * $unsigned(iB);
22  assign mulsu= $signed(iA) * $unsigned(iB);
23  `endif
24
25  always @(*)
26  begin
27      case (iControl)
28          OPAND:
29              oResult  <= iA & iB;
30          OPOR:
31              oResult  <= iA | iB;
32  //      OPXOR:
33  //          oResult  <= iA ^ iB;
34          OPADD:
35              oResult  <= iA + iB;
```

Figura 8: Implementação da ULA - parte 1



```
36          OPSUB:
37              oResult  <= iA - iB;
38          OPSLT:
39              oResult  <= iA < iB;
40          OPADDI:
41              oResult <= iA + iB;
42
43      //      OPSLTU:
44      //          oResult  <= $unsigned(iA) < $unsigned(iB);
45      //      OPSLL:
46      //          oResult  <= iA << iB[4:0];
47      //      OPSRL:
48      //          oResult  <= iA >> iB[4:0];
49      //      OPSRA:
50      //          oResult  <= iA >>> iB[4:0];
51      //      OPLUI:
52      //          oResult  <= iB;
53
54      //`ifdef RV32IM
55      //      OPMUL:
56      //          oResult  <= mul[31:0];
57      //      OPMULH:
58      //          oResult  <= mul[63:32];
59      //      OPMULHU:
60      //          oResult  <= mulu[63:32];
61      //      OPMULHSU:
62      //          oResult  <= mulsu[63:32];
63      //      OPDIV:
64      //          oResult  <= iA / iB;
65      //      OPDIVU:
66      //          oResult  <= $unsigned(iA) / $unsigned(iB);
67      //      OPREM:
68      //          oResult  <= iA % iB;
69      //      OPREMU:
70      //          oResult  <= $unsigned(iA) % $unsigned(iB);
```

Figura 9: Implementação da ULA - parte 2



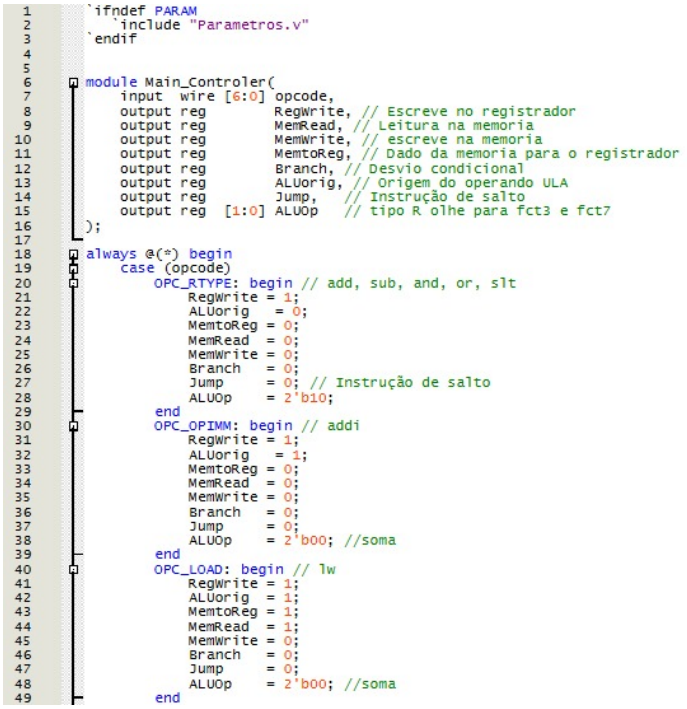
```
71      //endif
72
73            OPNULL:
74                oResult  <= ZERO;
75
76            default:
77                oResult  <= ZERO;
78        endcase
79    end
80
81
82    assign oZero = (oResult == 32'd0); // linha adcionada
83
84    endmodule
85
```

Figura 10: Implementação da ULA - parte 3

## 3.6 Controlador da ULA e o Bloco Controlador (item 1.6)

**Enunciado** : Implemente o Controlador da ULA e o Bloco Controlador.

**Implementação** : A implementação foi dividida em dois módulos principais: o Bloco Controlador (ou Main Controller) e o Controlador da ULA (ALUControl), ambos fundamentais para o correto funcionamento do processador.
O Bloco Controlador interpreta o campo opcode da instrução e gera os sinais de controle necessários para o fluxo de dados dentro do processador. Esses sinais incluem, por exemplo, RegWrite, MemRead, MemWrite, Branch, Jump, ALUOrig, MemtoReg, e ALUOp. Eles foram definidos com base na Tabela 2 do enunciado. Um caso default também foi incluído para lidar com instruções inválidas ou não implementadas, atribuindo a eles valores neutros (desativando os sinais de controle).
O Controlador da ULA recebe os sinais ALUOp, funct3 e funct7, utilizados nas instruções dos tipos R e I, para determinar a operação exata que a ULA deve executar. A partir desses sinais, ele gera a saída ALUControl, que especifica operações como add, sub, and, or e slt. Também foi prevista uma codificação de controle nulo (OPNULL) para instruções inválidas, como medida de segurança.
Essa separação de responsabilidades entre os dois módulos melhora a modularidade e facilita possíveis expansões do processador.

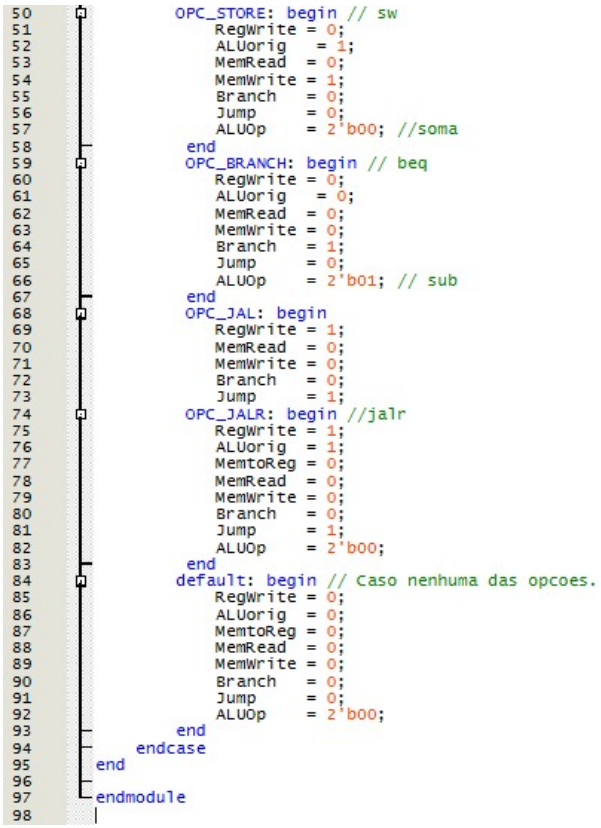


```
`ifndef PARAM
    `include "Parametros.v"
`endif


module Main_Controler(
    input  wire [6:0] opcode,
    output reg        RegWrite, // Escreve no registrador
    output reg        MemRead, // Leitura na memoria
    output reg        MemWrite, // escreve na memoria
    output reg        MemtoReg, // Dado da memoria para o registrador
    output reg        Branch, // Desvio condicional
    output reg        ALUorig, // Origem do operando ULA
    output reg        Jump,   // Instrução de salto
    output reg  [1:0] ALUOp   // tipo R olhe para fct3 e fct7
);

always @(*) begin
    case (opcode)
        OPC_RTYPE: begin // add, sub, and, or, slt
            RegWrite = 1;
            ALUorig  = 0;
            MemtoReg = 0;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 0; // Instrução de salto
            ALUOp    = 2'b10;
        end
        OPC_OPIMM: begin // addi
            RegWrite = 1;
            ALUorig  = 1;
            MemtoReg = 0;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 0;
            ALUOp    = 2'b00; //soma
        end
        OPC_LOAD: begin // lw
            RegWrite = 1;
            ALUorig  = 1;
            MemtoReg = 1;
            MemRead  = 1;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 0;
            ALUOp    = 2'b00; //soma
        end
```

Figura 11: Bloco Controlador (parte 1)



```
        OPC_STORE: begin // sw
            RegWrite = 0;
            ALUorig  = 1;
            MemRead  = 0;
            MemWrite = 1;
            Branch   = 0;
            Jump     = 0;
            ALUOp    = 2'b00; //soma
        end
        OPC_BRANCH: begin // beq
            RegWrite = 0;
            ALUorig  = 0;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 1;
            Jump     = 0;
            ALUOp    = 2'b01; // sub
        end
        OPC_JAL: begin
            RegWrite = 1;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 1;
        OPC_JALR: begin //jalr
            RegWrite = 1;
            ALUorig  = 1;
            MemtoReg = 0;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 1;
            ALUOp    = 2'b00;
        end
        default: begin // Caso nenhuma das opcoes.
            RegWrite = 0;
            ALUorig  = 0;
            MemtoReg = 0;
            MemRead  = 0;
            MemWrite = 0;
            Branch   = 0;
            Jump     = 0;
            ALUOp    = 2'b00;
        end
    endcase
end

endmodule
```

Figura 12: Bloco Controlador (parte 2)

```
`ifndef PARAM
    `include "Parametros.v"
`endif


module ALUControl (
    input  wire [1:0] ALUOp,
    input  wire [2:0] funct3,
    input  wire [6:0] funct7,
    output reg  [4:0] ALUControl
);

always @(*) begin
    case (ALUOp)
        2'b00: ALUControl = OPADD; // lw, sw
        2'b01: ALUControl = OPSUB; // beq
        2'b10: begin // R-type ou I-type
            case ({funct7, funct3})
                {7'b0000000, 3'b000}: ALUControl = OPADD; // add
                {7'b0100000, 3'b000}: ALUControl = OPSUB; // sub
                {7'b0000000, 3'b111}: ALUControl = OPAND; // and
                {7'b0000000, 3'b110}: ALUControl = OPOR;  // or
                {7'b0000000, 3'b010}: ALUControl = OPSLT; // slt
                default:              ALUControl = OPNULL;
            endcase
        end
        default: ALUControl = OPNULL;
    endcase
end

endmodule
```

Figura 13: Controlador da ULA

| Instrução | RegWrite | MemRead | MemWrite | MemtoReg | ALUorig | Branch | Jump | Jump2 | ALUop |
|---|---|---|---|---|---|---|---|---|---|
| R-type (add, sub, and, ...) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 |
| addi | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 00 |
| lw | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 00 |
| sw | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00 |
| beq | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 01 |
| jal | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 00 |
| jalr | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 00 |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 |

Tabela 2: Sinais de controle gerados pelo Main Controller com base no opcode

## 3.7 Processador Uniciclo Completo

**Enunciado:** Implemente o Processador Uniciclo completo

**Implementação:** A Figura 14 mostra o diagrama base do Processador Uniciclo - RiscV apresentado no PDF de questões do Laboratório 2. Porém, foi implementado alguns elementos além do que o arcabouço oferecia, com o objetivo de implementar corretamente o caminho de dados das intruções Jal e Jalr. Esses novos elementos foram desenhados de vermelho.

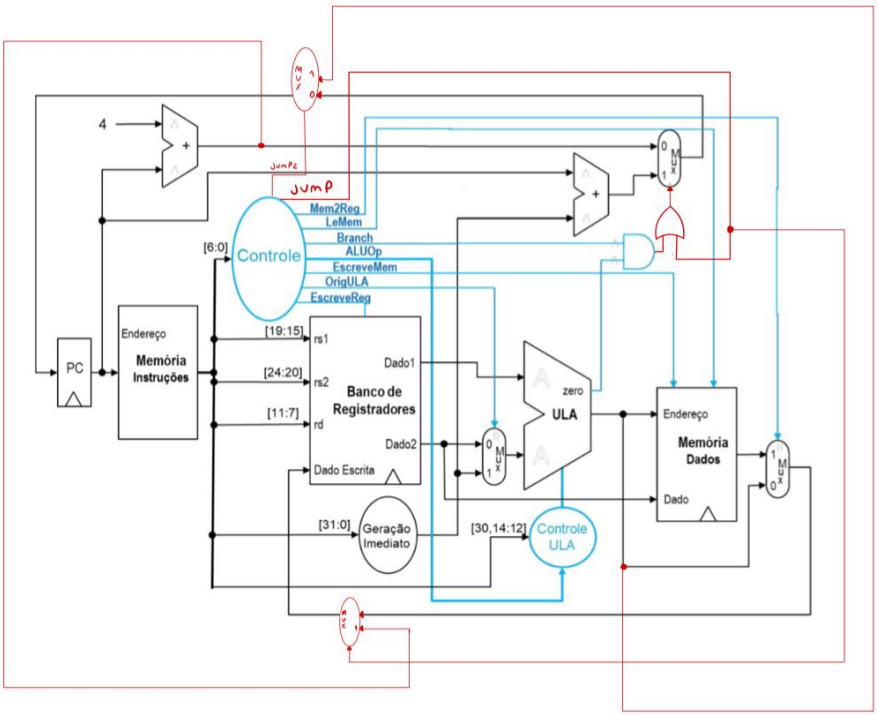



Figura 14: Diagrama do Uniciclo completo desenvolvido pela equipe

### 3.7.1 A) Visualização do Netlist RTL (item 1.7.1)

Após a implementação e integração de todos os componentes no módulo Uniciclo.v, o projeto foi sintetizado com a ferramenta Quartus Prime. A visualização RTL (Register Transfer Level) permite inspecionar a estrutura de hardware que o sintetizador interpretou a partir do código Verilog.

A figura 14 mostra todo o caminho de dados para implementação da logica estrutural dos caminhos para ser realizado o processamento de acordo com os dados que são colocados na entrada de acordo com as escolhas pre-definidas da controladora que faz o controle por meio de multiplexadores que foram criados e colocados na logica do arquivo uniciclo que junta todo o processador.

A Figura 15 mostra o diagrama de blocos de mais alto nível do projeto (TopDE), que encapsula o processador Uniciclo, exibindo suas principais portas de entrada e saída, como CLOCK, Reset e Regin. A Figura 16 apresenta a visualização RTL do interior do processador Uniciclo. Embora a disposição dos componentes seja diferente do diagrama conceitual da Figura 14, ela representa a mesma lógica de interconexão entre os módulos principais, mas de uma forma otimizada pela ferramenta de síntese. A visualização confirma que todos os componentes foram corretamente instanciados e conectados.

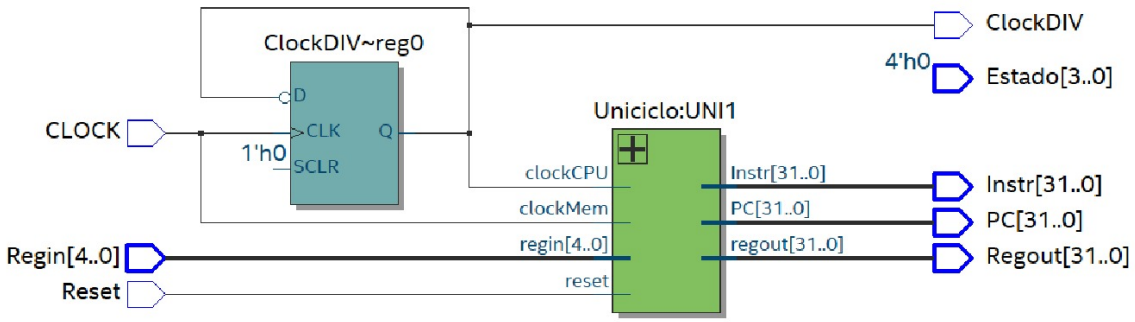



Figura 15: Módulo TopDE.

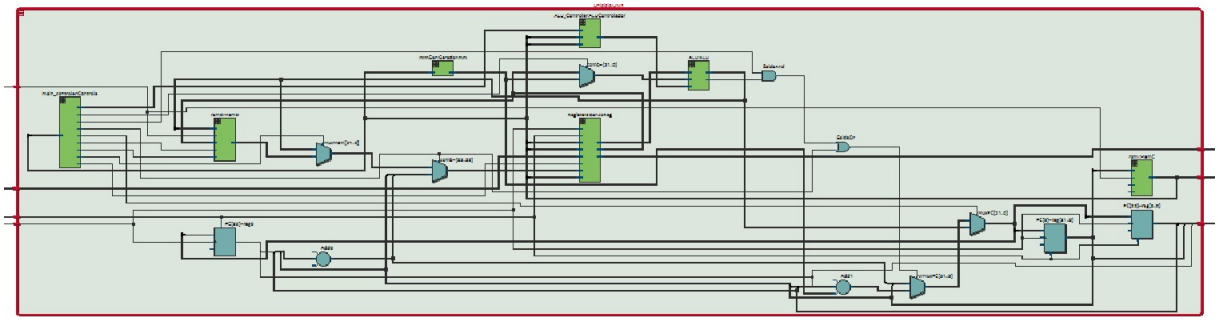


Figura 16: Módulo Uniciclo.

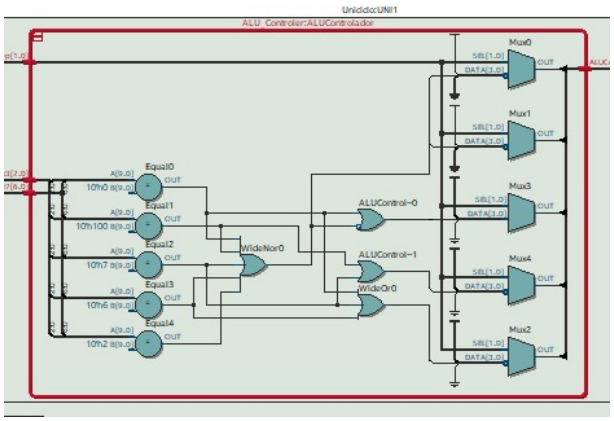


Figura 17: Módulo controlador da ULA.

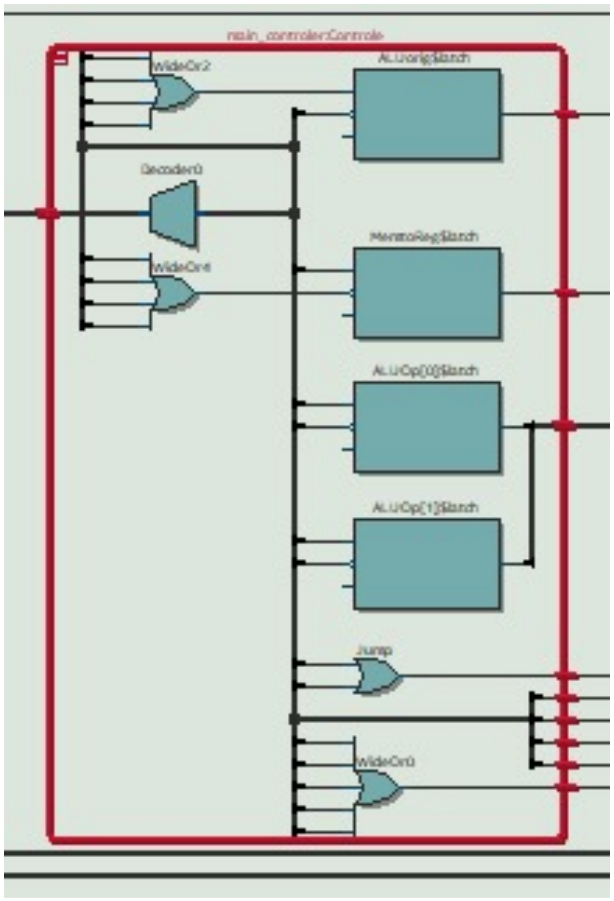



Figura 18: Módulo controlador principal.

### 3.7.2 B)

| | |
|---|---|
| Total logic elements | 3,166 / 114,480 ( 3 % ) |
| Total registers | 1269 |
| Total pins | 108 / 529 ( 20 % ) |
| Total virtual pins | 0 |
| Total memory bits | 65,536 / 3,981,312 ( 2 % ) |
| Embedded Multiplier 9-bit elements | 0 / 532 ( 0 % ) |
| Total PLLs | 0 / 4 ( 0 % ) |

Figura 19: Requisitos Funcionais do Processador Uniciclo Implementado

**-Fmax**= 82.4MHz e **Restricted Fmax** = 2.4MHz

| Tipo de Análise | Slack |
|---|---|
| Setup summary | 43.932 |
| Hold summary | 0.400 |
| Recovery Summary | 97.059 |
| Removal Summary | 1.002 |
| Minimum Pulse Width Summary | 49.577 |

Tabela 3: Requisitos Temporais do Processador Uniciclo Implementado

Nesta etapa, foram levantados os requisitos físicos e temporais do processador completo implementado no Quartus. As informações físicas foram obtidas a partir do relatório Flow Summary, incluindo uso de lógica, registradores, memória e pinos de I/O. Já os requisitos temporais foram extraídos do TimeQuest Timing Analyzer, utilizando o modelo Slow 1200mV 85°C, que fornece a análise dos slacks e da frequência máxima permitida. Com base nesses dados, foi possível verificar que todos os slacks estão sendo respeitados, garantindo que o processador opere de forma correta e estável dentro dos limites de temporização exigidos.

### 3.7.3 C)

Com base nas Figuras 20, 21 e 22, é possível verificar que a CPU implementada está funcionando corretamente tanto em simulação funcional quanto em simulação temporal com o programa de1.s.

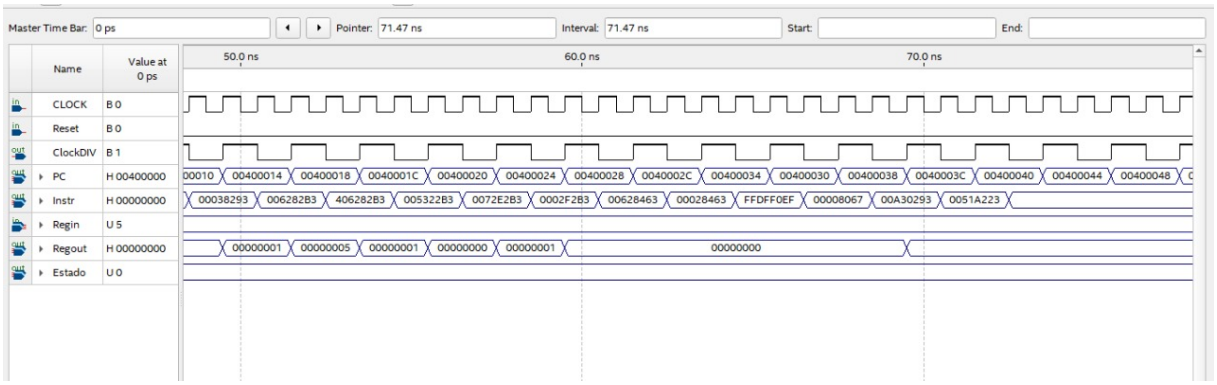



Figura 20: Simulação Funcional com Clock a 1nns

Na Figura 20, observa-se que:

O PC (Program Counter) está sendo incrementado corretamente a cada ciclo de clock, iniciando em 0x00400000 e progredindo em PC + 4.

A instrução (Instr) é atualizada conforme o valor do PC, refletindo corretamente as instruções do programa.

Os sinais Regin, Regout e Estado acompanham essa evolução sem apresentar valores inválidos, isso comprova o correto funcionamento lógico do processador, com resposta combinacional e sequencial operando de forma estável.

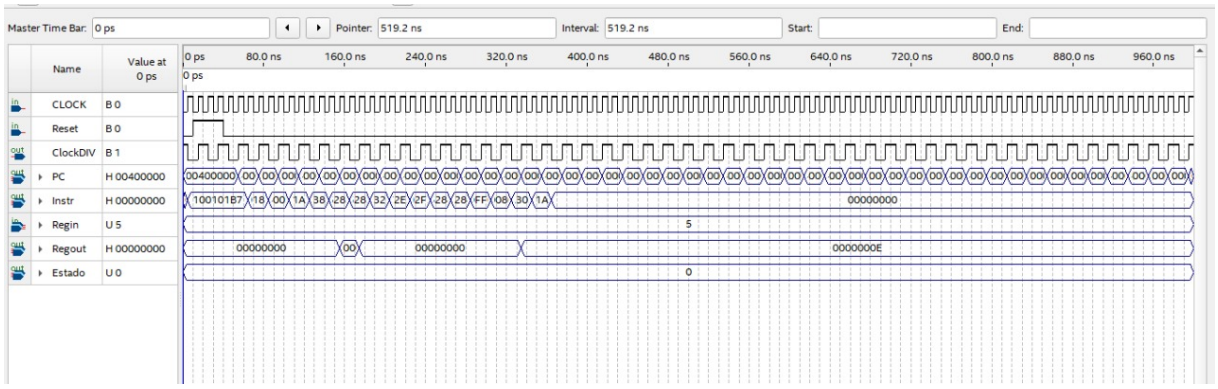



Figura 21: Simulação Funcional com Clock a 10 nns

A Figura 21 demostra o funcionamento que permanece correto mesmo com um clock mais lento, os sinais principais (PC, Instr, Regout) seguem variando conforme esperado.

O campo Instr mostra instruções decodificadas do programa sendo lidas corretamente da memória, isso demonstra que a CPU continua operando corretamente em outra escala temporal, evidenciando robustez do projeto mesmo com variações no período de clock.

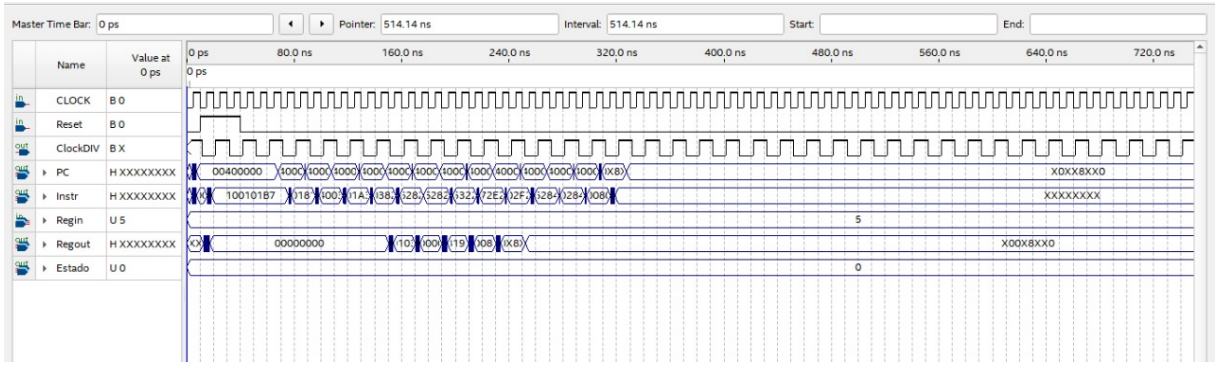



Figura 22: Simulação Temporal com Clock a 10 nns

Na figura 22 observamos uma simulação do tipo temporal:

Ainda que apareçam valores transitórios, o comportamento final estabiliza e confirma os mesmos padrões vistos na simulação funcional.

As instruções e os dados lidos/escritos nos registradores seguem o comportamento esperado, apenas com os atrasos naturais causados pela simulação mais detalhada, mesmo com atrasos internos simulados, a CPU executa o programa sem travamentos ou erros lógicos aparentes.

### 3.7.4 D)

A frequência máxima de operação (Fmax) determina a velocidade máxima com que o processador pode operar de forma confiável, conforme o relatório do Timing Analyzer do Quartus, a frequência máxima de operação para este design de processador uniciclo é Fmax = 82.4MHz.

**Explicação:** Este valor é limitado pelo "caminho crítico"do circuito, o caminho de sinal mais longo e, portanto, mais demorado. Em um processador uniciclo, este caminho é quase sempre o da instrução de load (lw), que precisa passar por todas as cinco etapas (busca da instrução, leitura dos registradores, cálculo na ULA, leitura da memória de dados e escrita no registrador) dentro de um único ciclo de clock. A Fmax de 82.4MHz indica que o período mínimo de clock que o sistema suporta é de aproximadamente 12.1ns (1 / 82.4MHz).


Figura 23: Simulação Temporal com Clock a 10nns

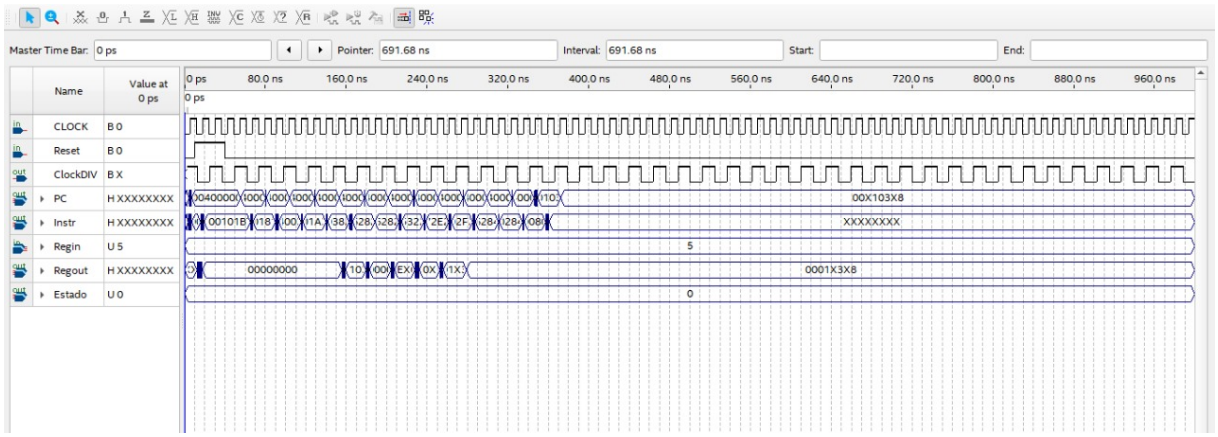



Figura 24: Simulação Temporal com Clock a 80Mghz

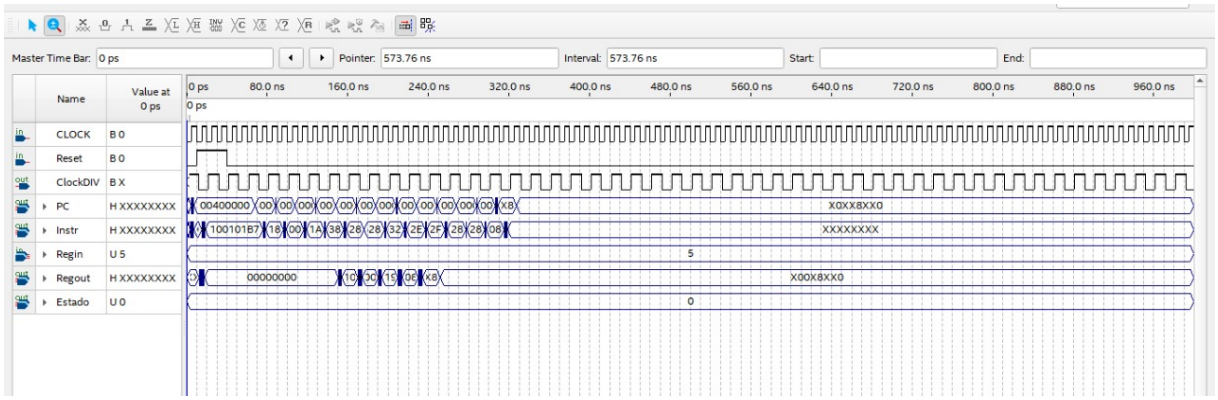



Figura 25: Simulação Temporal com Clock a 100 Mghz

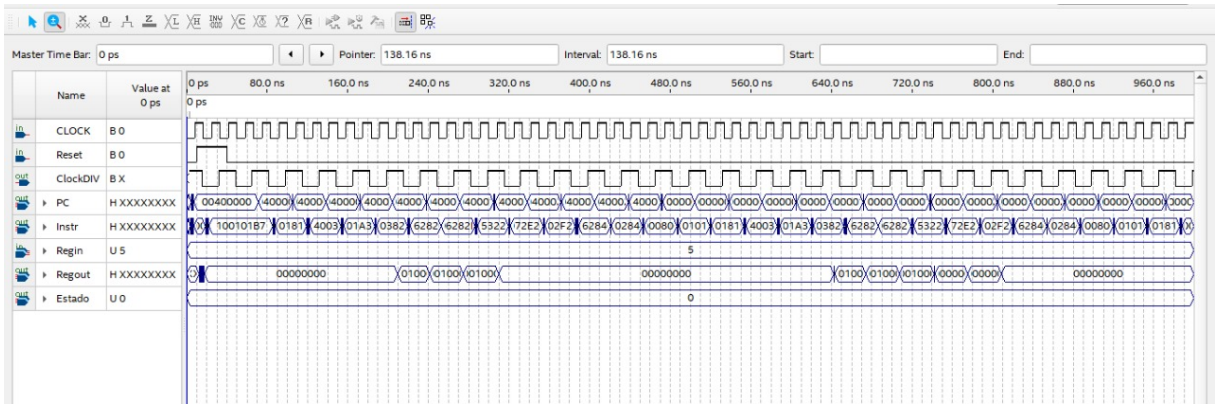



Figura 26: Simulação Temporal com Clock a 60 Mghz

# 4 Conclusão

O projeto e a implementação do processador RISC-V com ISA reduzida em Verilog foram concluídos com sucesso, cumprindo todos os objetivos propostos. A integração de todos os módulos (ULA, Banco de Registradores, Controles, etc.) foi validada através da visualização RTL e da compilação bem-sucedida no Quartus Prime.

A análise de recursos demonstrou uma implementação eficiente, utilizando uma pequena porcentagem da capacidade do FPGA. Mais importante, a análise temporal confirmou que o design é temporalmente válido, sem violações de slack, e capaz de operar a uma frequência máxima de 82.4MHz, porém foram realizadas simulações que mostraram que em algumas circunstancias a saída estava se comportando de forma ineseperada e foi visualizado um comportamento adequado com um clock de 60MHz. A corretude funcional foi comprovada por meio de simulações funcional e temporal, que mostraram o processador executando corretamente um programa de teste denominado como de1 que abrange todas as funcoes implementadas nele mostrando o comportamento adequado na saída e também a leitura correta de cada registrador.

Foi visto que os arquivos criados de controle fazem todo funcionamento entre todos os modulos fazendo com que eles se relacionem para que possa ter um caminho de dados eficiente com uma entrada de 32 bits e ele possa conseguir identificar todos os tipos de instrução e quais tipos de operações devem ser realizada de acordo com o comportamento esperado.

Este trabalho permitiu solidificar os conceitos de arquitetura de computadores e o fluxo de projeto de hardware.

# 5 Bibliografia

Patterson, D.A., Hennessy, J.L., Computer Organization and Design – The Hardware/Software Interface, RISC-V 2º edition, Morgan Kaufmann, 2021;