PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR RISC 8 BITS

ALUNOS:

Leonam De Sousa Silva – 2022007093
Ryan Pimentel De Oliveira - 2022007020

Março de 2025
Boa Vista/Roraima

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORER

RELATÓRIO DO PROJETO: PROCESSADOR RISC 8 BITS

Março de 2025
Boa Vista/Roraima

Resumo

Este relatório documenta o projeto e a implementação do Processador 8 Bits RISC, desenvolvido em VHDL e sintetizado com Quartus Prime. O processador utiliza um conjunto básico de 8 instruções e possui uma arquitetura modular, onde cada componente – como o Program Counter, ROM, Decodificador, Unidade de Controle, Banco de Registradores, MuxULA, ULA e RAM – foi implementado em entidades separadas. Um programa alternativo de Fibonacci (simplificado) também foi incluído na ROM (como comentário) para demonstrar a capacidade do sistema de calcular uma sequência iterativa.

Conteúdo

Lista de Figuras

Lista de Tabelas

Relatório do Projeto – Processador 8 Bits RISC

1. Introdução

Flow Status	Successful - Mon Mar 03 15:41:43 2025
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	Processador8bits
Top-level Entity Name	CPU
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	368 / 56,480 (< 1 %)
Total registers	566
Total pins	65 / 268 (24 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Figura 1 - Especificações no Quartus

Este relatório documenta o projeto e a implementação do Processador 8 Bits RISC, desenvolvido em VHDL e sintetizado com Quartus Prime. O processador utiliza um conjunto básico de 8 instruções e possui uma arquitetura modular, onde cada componente – como o Program Counter, ROM, Decodificador, Unidade de Controle, Banco de Registradores, MuxULA, ULA e RAM – foi implementado em entidades separadas. Um programa alternativo de Fibonacci (simplificado) também foi incluído na ROM (como comentário) para demonstrar a capacidade do sistema de calcular uma sequência iterativa.

2. Conjunto de Instruções

Instrução	Tipo	Opcode	Significado	Exemplo e Interpretação
ADD	R	000	Soma os valores dos registradores rs e rd, armazenando o resultado em rd.	'00001110': ADD R1, R3 (R1 <- R1 + R3)
SUB	R	001	Subtrai o valor de rs de rd e armazena o resultado em rd.	'00110010': SUB R1, R2 (R1 <- R1 - R2)
LW	I	010	Carrega da memoria (endereco = imediato) para o registrador destino.	'01001010': LW R1, 010 (R1 <- Mem[2])
SW	I	011	Armazena o valor do registrador fonte na memória no endereço especificado pelo imediato.	'01110011': SW R2, 011 (Mem[3] <- R2)
ADDI	I	100	Soma o valor imediato ao conteúdo do registrador destino.	'10010001': ADDI R2, 001 (R2 <- R2 + 1)
LI	I	101	Carrega um valor imediato no registrador destino.	'10101101': LI R1, 101 (R1 <- 5)
BEQ	I	110	Se o valor do registrador for igual ao de R0, o PC recebe o imediato como novo endereço.	'11001010': BEQ R2, 010 (se R2 == R0, PC <- 2)
JUMP	J	111	Salta incondicionalmente para o endereço especificado.	'11100111': JUMP 00111 (PC <- 7)

Figura 2 - Conjunto de instruções

Cada instrução possui 8 bits e é classificada em três tipos: I-type (imediato), R-type (registro) e J-type (jump). A tabela a seguir resume cada instrução, sua formatação e significado.

Formato para escrita em código binário do tipo R:

Bits	Campo
7-5	opcode
4-3	rd
2-1	rs
0	Bit fixo em 0

Tabela 1 – Escrita em binário do tipo R

Formato para escrita em código binário do tipo J:

Bits	Campo
7-5	Opcode
4-0	endereço

Tabela 2 – Escrita em binário do tipo J.

Formato para escrita em código binário do tipo I:

Bits	Campo
7-5	opcode
4-3	Resgistrador
2-0	Imediato

Tabela 2 – Escrita em binário do tipo J.

3. Datapath e Descrição dos Componentes

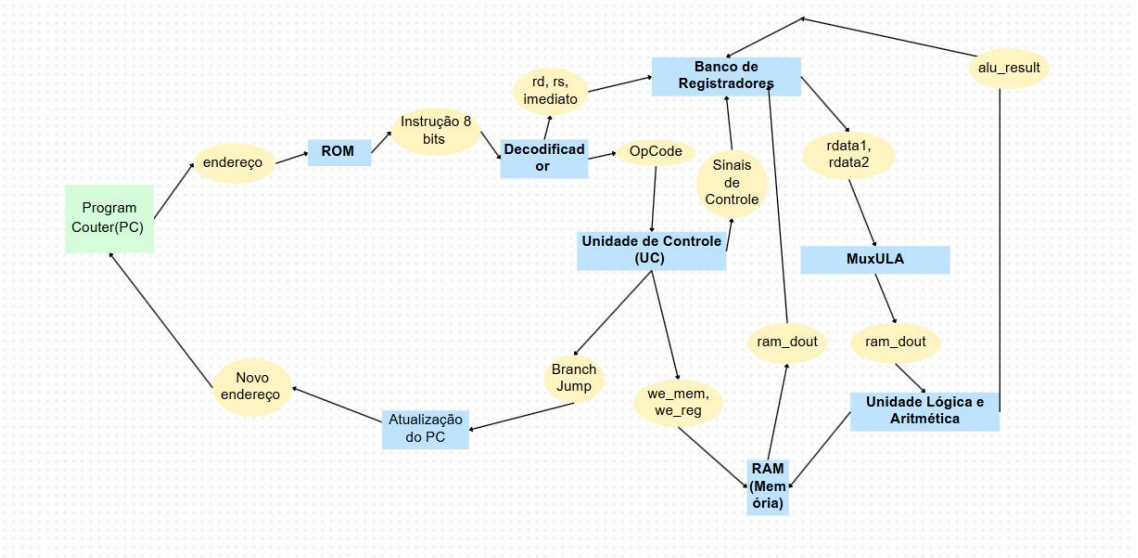

Figura 3 -Datapath e Descrição dos Componentes

O processador 8 Bits RISC é composto por diversos módulos, interconectados para formar o datapath completo. A seguir, descrevemos cada componente e sua função no fluxo de dados:

3.1 Program Counter

- **Descrição:**

O **Contador de Programa (PC)** é um registrador responsável por armazenar o endereço da **próxima instrução** a ser executada no processador. Ele é atualizado a cada ciclo de clock para garantir a execução sequencial do programa, podendo ser modificado por instruções de salto e desvios condicionais.

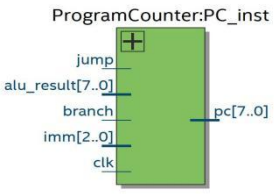

Figura 4 - Bloco simbólico do componente PC gerado pelo Quartus

• **Função:**

– Mantém o endereço da instrução atual.

– Incrementa o PC em 1 a cada ciclo de clock.

– Se o sinal “jump” estiver ativo, o PC é carregado com o valor de alu_result; se “branch” estiver ativo e a condição (comparação com R0) for satisfeita, o PC recebe o imediato estendido como novo endereço.

• **Instanciação:**

No módulo principal (CPU), o PC é instanciado como:

PC_inst: entity work.ProgramCounter

port map(clk => clk, branch => branch, jump => jump, alu_result => alu_result, imm => imm, pc => pc).

- **Execução Sequencial:** Se **jump = 0** e **branch = 0**, o PC simplesmente incrementa seu valor em **1**.
- **Salto Incondicional (Jump):** Se **jump = 1**, o PC assume o valor da saída da ULA (**alu_result**), permitindo mudanças diretas de fluxo no programa.
- **Desvio Condicional (Branch):** Se **branch = 1** e a condição for atendida (geralmente uma comparação envolvendo **R0**), o PC recebe um novo endereço baseado no valor imediato (**imm**).

3.2 ROM

- **Descrição:**

A **ROM (Read-Only Memory)** é um tipo de memória utilizada para armazenar instruções ou dados que não podem ser alterados durante a execução do programa. No contexto do processador, a ROM é responsável por armazenar o código do programa que será executado.

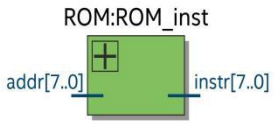

Figura 5 - Bloco simbólico do componente ROM gerado pelo Quartus

- **Função:**
  – Armazena as instruções do programa (até 256 linhas).
- **Instanciação:**
  O PC é usado para endereçar a ROM, que fornece a instrução atual:
  ROM_inst: entity work.ROM
  port map(addr => pc, instr => instr).

A ROM recebe um endereço **addr** e retorna a instrução **instr** armazenada nessa posição. Esse comportamento permite que a CPU busque e execute instruções do programa de forma sequencial ou controlada por desvios condicionais. Como a ROM é apenas leitura, seu conteúdo não pode ser modificado durante a execução.

3.3 Decodificador

- **Descrição** :

O **Decodificador de Instruções** é um módulo responsável por interpretar uma instrução de 8 bits e separá-la em seus respectivos campos. Esses campos são utilizados pelo processador para identificar a operação a ser executada e os operandos envolvidos.

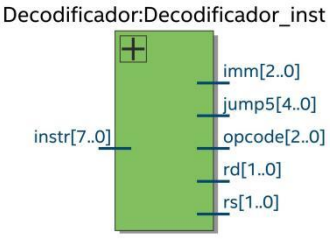

Figura 6 - Bloco simbólico do componente Decodificador gerado pelo Quartus

• **Função:**

– Separa a instrução de 8 bits em campos: opcode, rd, rs, imediato e campo de salto (jump5)

- **opcode**: Código da operação que define qual instrução será executada.
- **rd**: Registrador de destino, onde o resultado será armazenado.
- **rs**: Registrador de origem, que fornece um dos operandos.
- **imm**: Valor imediato, utilizado em operações que necessitam de um número constante.
- **jump5**: Campo de salto, usado para instruções que alteram o fluxo do programa.

• **Instanciação:**

```
Decodificador_inst: entity work.Decodificador
    port map( instr => instr, opcode => opcode, rd => rd, rs => rs, imm => imm, jump5 => jump5 ).
```

Quando uma nova instrução é recebida, o decodificador separa os bits e direciona cada parte para o componente correspondente. O **opcode** define a operação, enquanto os demais campos determinam os operandos e possíveis endereços de salto. Esse processo é essencial para garantir a correta execução das instruções pelo processador.

3.4 Unidade de Controle

- **Descrição:**

A **Unidade de Controle (UC)** é responsável por interpretar o código da operação (**opcode**) e gerar os sinais de controle necessários para a execução das instruções no processador. Esses sinais coordenam o funcionamento dos demais componentes, como registradores, memória e a ULA (Unidade Lógica e Aritmética).

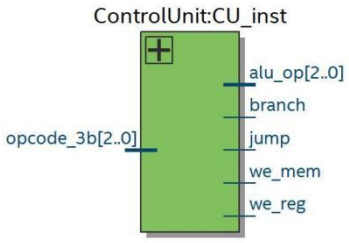

Figura 7 - Bloco simbólico do componente UC gerado pelo Quartus

• **Função:**
– Interpreta o opcode e gera os sinais de controle: we_reg, we_mem, branch, jump e alu_op.

- **we_reg** (Write Enable Register): Habilita a escrita em registradores.
- **we_mem** (Write Enable Memory): Controla a escrita na memória.
- **branch**: Indica se a instrução envolve um desvio condicional.
- **jump**: Define se a execução deve pular para um novo endereço.
- **alu_op**: Controla a operação que a ULA deve realizar.

• **Instanciação:**
CU_inst: entity work.ControlUnit
port map(opcode_3b => opcode, we_reg => we_reg, we_mem => we_mem, branch => branch, jump => jump, alu_op => alu_op).

Quando uma nova instrução é decodificada, a Unidade de Controle analisa o **opcode** e define quais sinais devem ser ativados. Dependendo da operação, a CU pode:

- Habilitar a escrita em registradores ou memória.
- Ativar sinais de controle de fluxo, como **branch** e **jump**.
- Definir a operação que a ULA deve executar.

3.5 Banco de Registradores

- **Descrição:**

O **Banco de Registradores** é um módulo responsável por armazenar e gerenciar **quatro registradores de 8 bits** (R0, R1, R2 e R3). Ele permite a leitura simultânea de dois registradores para fornecer operandos e possibilita a escrita em um registrador específico.

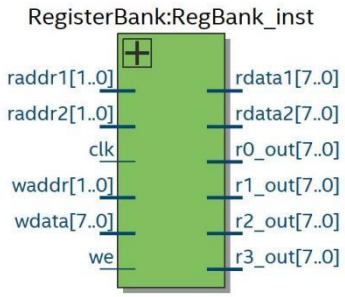

Figura 8 - Bloco simbólico do componente Registradores gerado pelo Quartus

- **Função:**
  – Armazena 4 registradores de 8 bits (R0 a R3).
  – Permite leitura dupla (para os operandos) e escrita em um registrador.
- **Detalhe:**
  – raddr1 é determinado pelo campo rd e, para instruções R-type, raddr2 é definido a partir do campo rs; para outras instruções, raddr2 é fixado em "00".

- **Armazena quatro registradores** (**R0 a R3**) de 8 bits cada.

- **Permite leitura dupla**: fornece dois operandos simultaneamente para operações da ULA.

- **Permite escrita** em um dos registradores, conforme determinado pelo campo de destino (**rd**).

- **Instanciação:**

```
RegBank_inst: entity work.RegisterBank
    port map( clk => clk, we => we_reg, raddr1 => rd, raddr2 => (rs when (opcode = "000" or opcode = "001") else "00"), waddr => rd, wdata => wdata, rdata1 => reg_data1, rdata2 => reg_data2, r0_out => r0_out, r1_out => r1_out, r2_out => r2_out, r3_out => r3_out );
```

O Banco de Registradores opera de forma síncrona com o **clock (clk)**, garantindo que os dados sejam armazenados e lidos corretamente a cada ciclo de processamento. A combinação do opcode e dos endereços de leitura determina quais registradores serão acessados em cada instrução.

3.6 MuxULA

- **Descrição :**

O módulo **MuxULA** tem a função de selecionar quais dados serão utilizados como entrada para a Unidade Lógica e Aritmética (ULA). Ele recebe diferentes sinais de entrada e, com base no valor do **opcode**, determina quais serão encaminhados para os sinais de saída **alu_a** e **alu_b.**

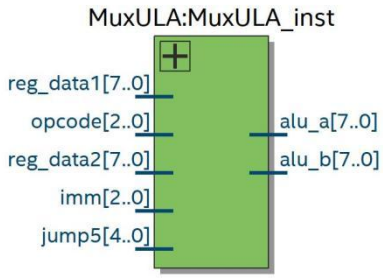

Figura 9 - Bloco simbólico do componente MUXULA gerado pelo Quartus

- **Função:**

– Seleciona os operandos para a ULA conforme o tipo de instrução.

– Para R-type, utiliza os valores lidos do Banco de Registradores; para I-type ou JUMP, converte o imediato ou campo de salto para 8 bits.

- **Instanciação:**

MuxULA_inst: entity work.MuxULA

port map(opcode => opcode, reg_data1 => reg_data1, reg_data2 => reg_data2, imm => imm, jump5 => jump5, alu_a => alu_a, alu_b => alu_b).

3.7 ULA (Unidade Lógica e Aritmética)

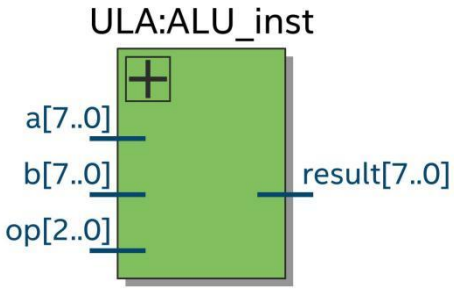

Figura 10 - Bloco simbólico do componente ULA gerado pelo Quartus

- **Função:**

– Executa as operações aritméticas e lógicas definidas pelo sinal alu_op, como soma (ADD), subtração (SUB), ADDI, LI, BEQ (comparação) e JUMP (passagem do endereço).

- **Instanciação:**

ALU_inst: entity work.ULA

port map(op => alu_op, a => alu_a, b => alu_b, result => alu_result).

3.8 RAM

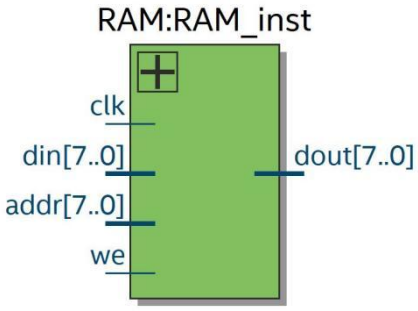

Figura 11 - Bloco simbólico do componente RAM gerado pelo Quartus

- **Função:**
  – Armazena e lê dados para as operações de load (LW) e store (SW).
- **Instanciação:**
  RAM_inst: entity work.RAM
  port map(clk => clk, addr => ram_addr, we => we_mem, din => reg_data1, dout => ram_dout).
  A saída ram_addr_out recebe o sinal ram_addr.

3.9 Lógica de Glue

- **Função:**
  – No módulo CPU, lógica combinacional determina:
    – wdata: para R-type, ADDI e LI, wdata = alu_result; para LW, wdata = ram_dout.
    – ram_addr: Para LW e SW, ram_addr = alu_result.
- **Implementação:**
  wdata <= alu_result when opcode = "000" or opcode = "001" or opcode = "100" or opcode = "101" else ram_dout when opcode = "010" else (others => '0');
  ram_addr <= alu_result when opcode = "010" or opcode = "011" else (others => '0');

4. Programa Fibonacci Simplificado e Testes

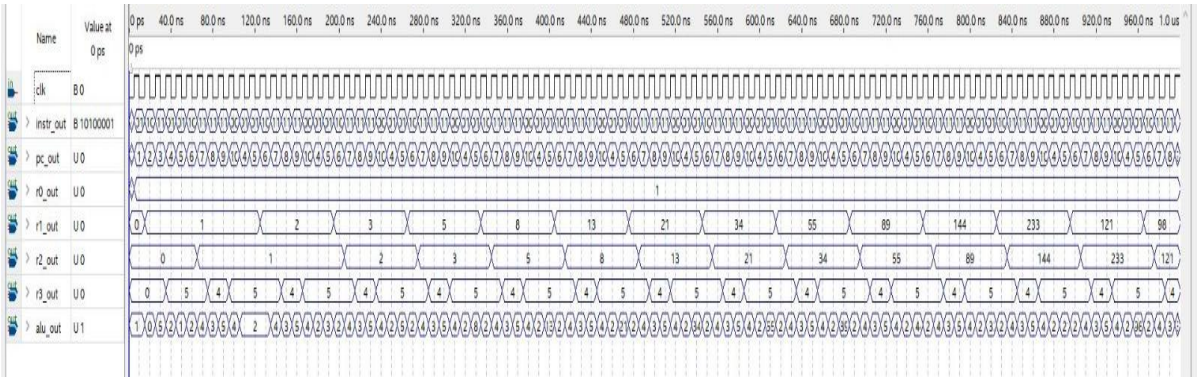

Figura 12 - Fibonacci gerado pelo Quartus

Na ROM, além do programa principal para testes gerais, há um exemplo de programa Fibonacci (em bloco de comentários) que implementa um loop simples: – Inicialmente, R1 é carregado com 1 (primeiro termo) e R2 com 0 (termo anterior).

– O valor de R1 é armazenado na memória e, em seguida, R1 é atualizado pela soma de R1 com R2.

– O antigo valor de R1 é carregado em R2 e o loop é reiniciado por meio de uma instrução JUMP.

O resultado da sequência Fibonacci (o termo corrente) é continuamente atualizado no registrador R1, e pode ser observado através da saída r1_out.

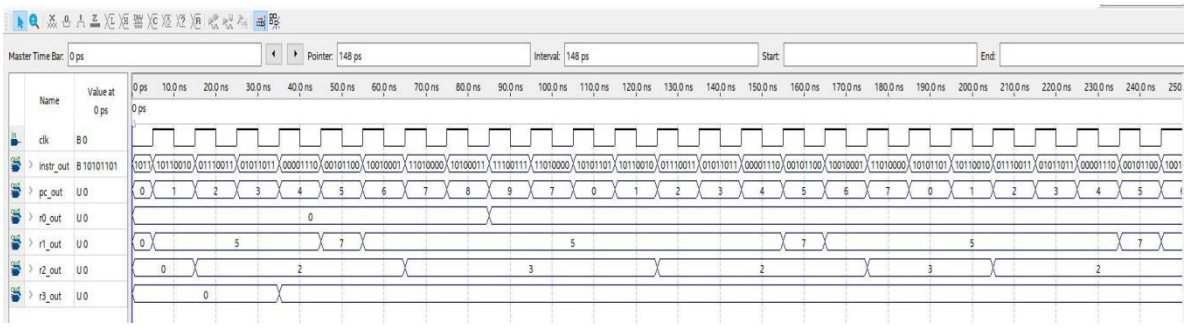

Figura 13 - Teste Programa principal

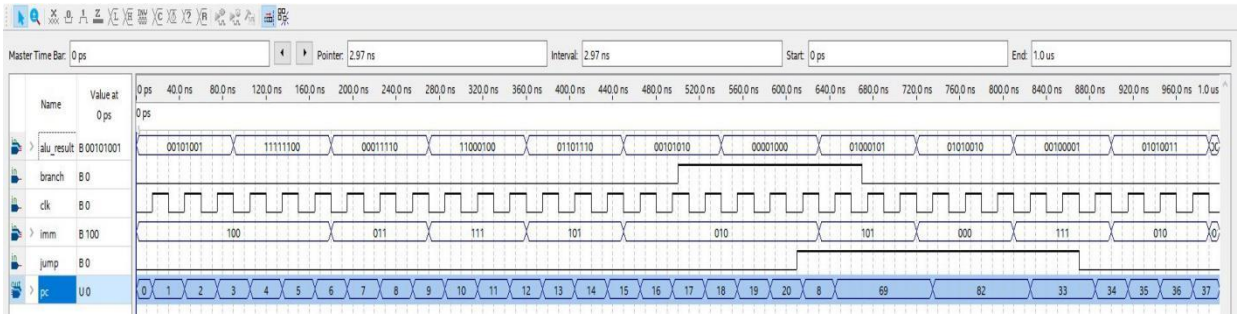

Figura 14 - Teste Program counter

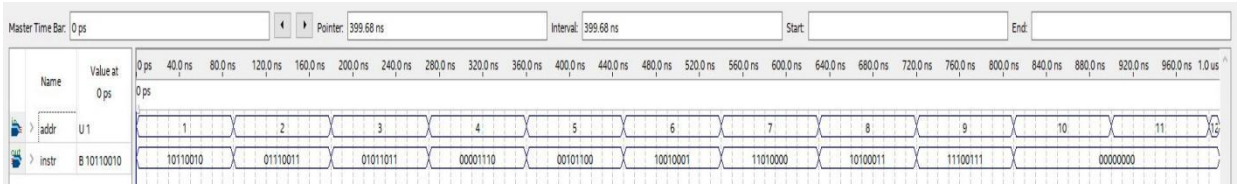

Figura 15 - Teste ROM

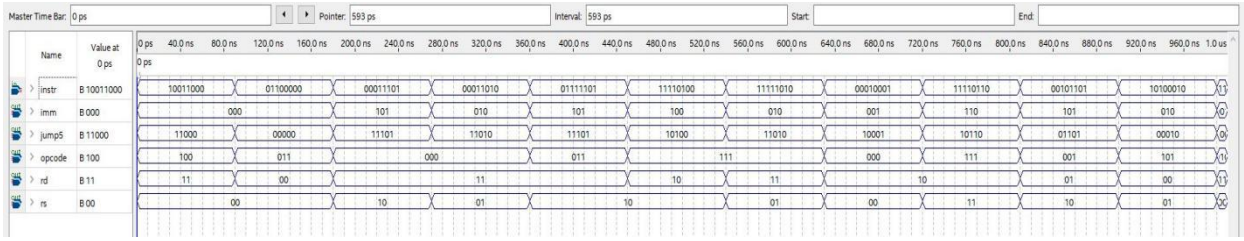

Figura 16 - Teste Decodificador

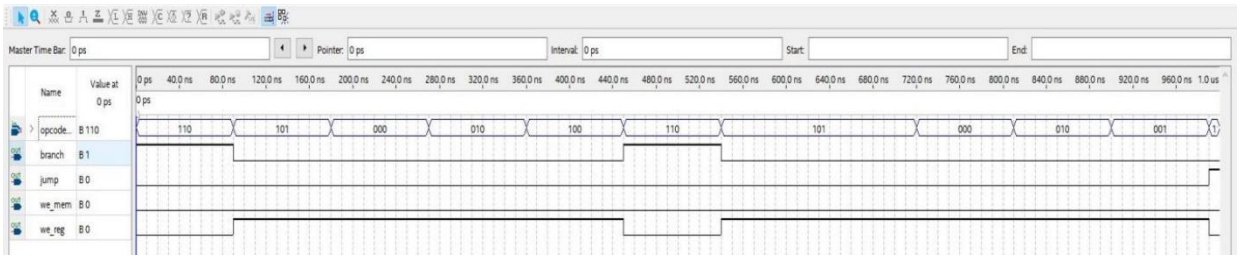

Figura 17 - Teste Unidade de Controle

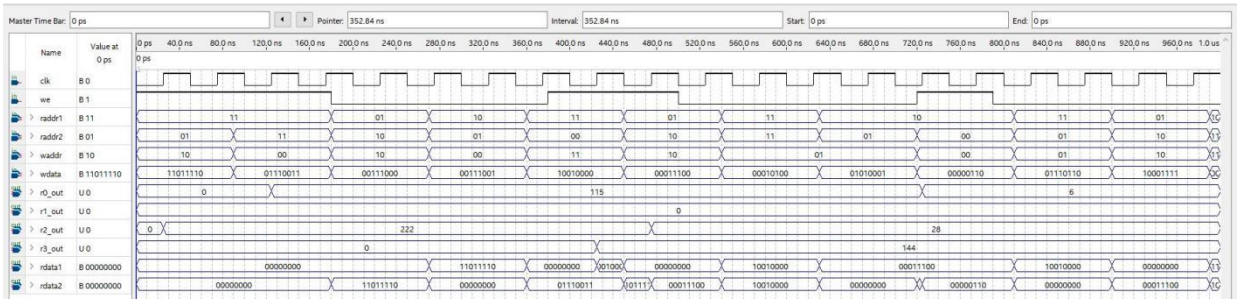

Figura 18 - Teste Banco de Registradores

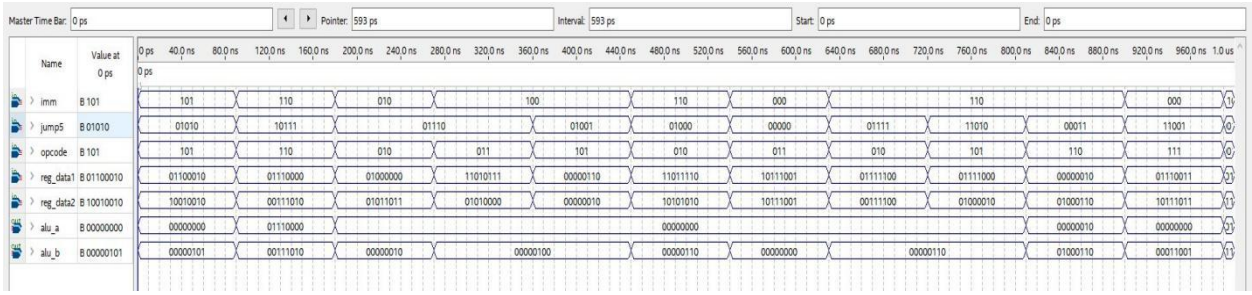

Figura 19 - Teste MuxULA

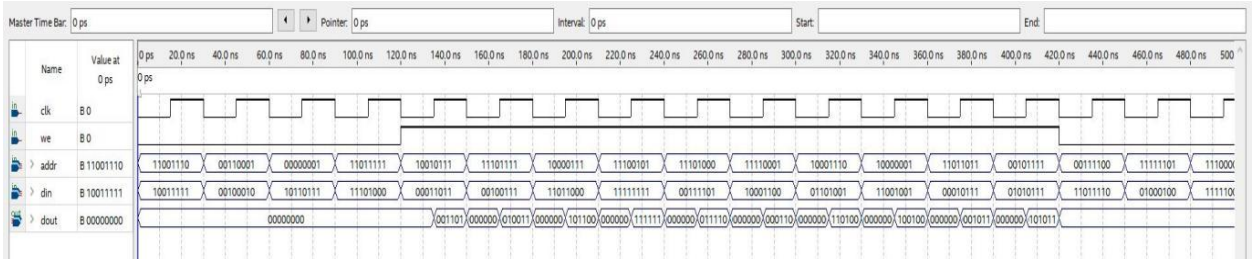

Figura 20 - Teste RAM

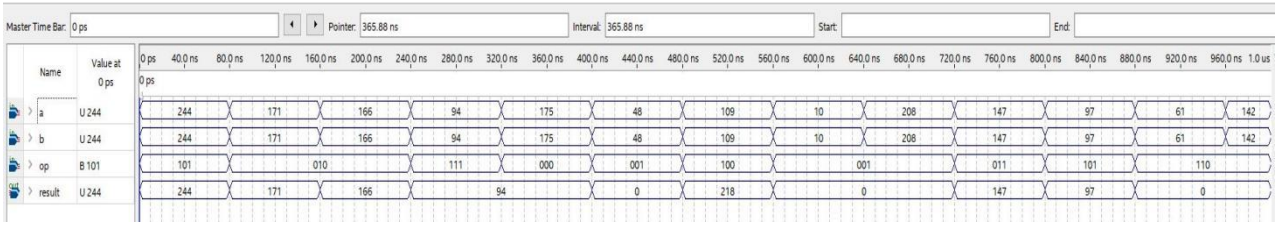

Figura 21 - Teste ULA

5. Datapath Geral

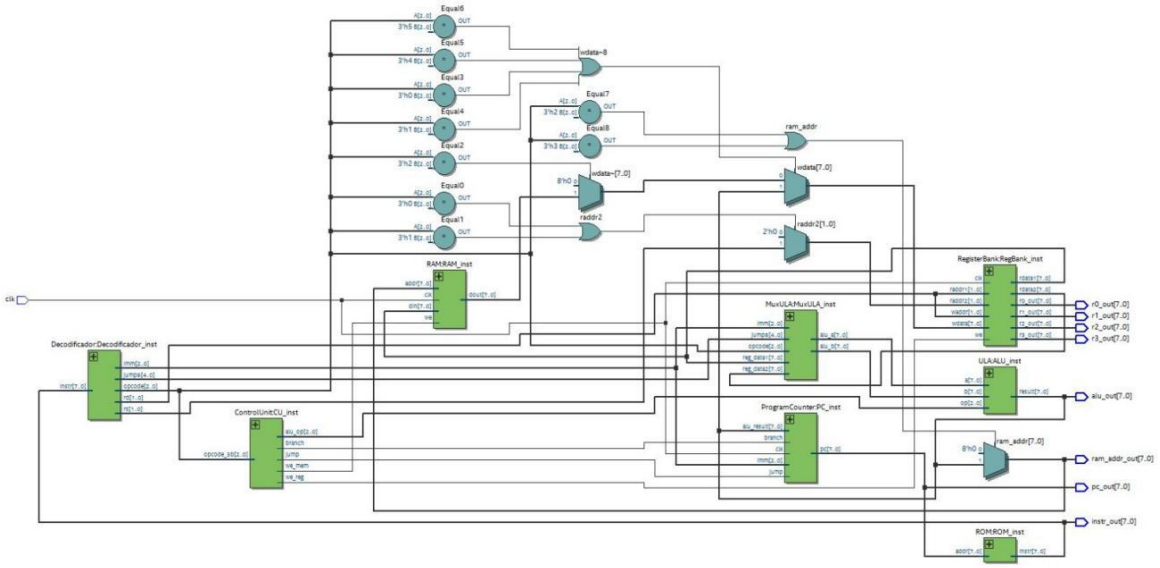

Figura 22 - Datapath Geral gerado pelo Quartus

O fluxo de dados no Processador 8 Bits RISC ocorre da seguinte forma:

a) O PC (Program Counter) determina o endereço da instrução na ROM.
b) A ROM fornece a instrução que é enviada ao Decodificador e à Unidade de Controle.
c) O Decodificador divide a instrução em campos (opcode, rd, rs, imediato, campo de salto).
d) A Unidade de Controle gera sinais de controle que direcionam o funcionamento do Banco de Registradores, da ULA e da RAM.
e) O Banco de Registradores fornece os operandos ao MuxULA, que seleciona os dados corretos para a ULA.
f) A ULA executa a operação (soma, subtração, etc.) e gera um resultado.
g) A lógica de glue direciona o resultado da ULA para atualizar os registradores (wdata) ou para formar o endereço de acesso à RAM (ram_addr).
h) Finalmente, o PC é atualizado com base no valor de alu_result, jump e branch, determinando a próxima instrução.

6. Conclusão

O Processador 8 Bits RISC demonstra uma arquitetura modular simples, porém completa, que integra a busca e execução de instruções com um conjunto de 8 instruções fundamentais. Cada componente – do Program Counter à RAM – foi implementado de forma independente e interconectado por uma lógica de glue, permitindo a execução de programas tanto para testes gerais quanto para aplicações específicas, como o cálculo da sequência

Fibonacci. No programa Fibonacci, o registrador R1 é continuamente atualizado com os novos termos, servindo como saída para verificação do algoritmo.

Imagens ilustrativas do projeto (exibidas no RTL Viewer) podem ser referenciadas como Figura 1, Figura 2, Figura 3 , etc., correspondendo aos componentes individuais (ALU, Banco de Registradores, PC, etc.) e ao datapath completo.

Este relatório fornece uma visão detalhada do funcionamento e da estrutura do Processador 8

Bits RISC, servindo como base para futuras análises, simulações e aprimoramentos do design.

7. Referências

PATTERSON, D.; HENESSY, J. L. Organização e projeto de computadores: a interface hardware/software. 3ª Edição. São Paulo: Elsevier, 2005.

Victor Hugo - Overview. Disponível em: https://github.com/VictorH456