UNIVERSIDADE FEDERAL DE RORAIMA
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
CURSO BACHAREL EM CIÊNCIA DA COMPUTAÇÃO
ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

ANDERSSON SILVA PEREIRA
ANDREZA OLIVEIRA GONÇALVES
ARTHUR CORREIA DE OLIVEIRA RAMOS

PROJETO FINAL - IMPLEMENTAÇÃO DE UM PROCESSADOR RISC DE 8 BITS

BOA VISTA – RR
2025

ANDERSSON SILVA PEREIRA
ANDREZA OLIVEIRA GONÇALVES
ARTHUR CORREIA DE OLIVEIRA RAMOS

PROJETO FINAL - IMPLEMENTAÇÃO DE UM PROCESSADOR RISC DE 8 BITS

Relatório Científico apresentado ao Prof. Dr. Herbert Oliveira Rocha, com objetivo de obtenção de nota parcial para aprovação na disciplina DCC 301 - Arquitetura e Organização de Computadores, do Departamento de Ciência da Computação da Universidade Federal de Roraima.

BOA VISTA – RR
2025

1 INTRODUÇÃO

Este relatório documenta o desenvolvimento e a análise de um processador RISC de 8 bits. O objetivo principal é implementar e validar o processador, detalhando cada etapa de construção e os resultados dos testes realizados.

O trabalho usa a linguagem de programação VHDL, MIPS e técnica de *testbench*. Cada componente será descrito de forma detalhada, abrangendo:

1. **Descrição do Componente**:

Identificação da funcionalidade do componente, explicando:

- Operações realizadas internamente (como o funcionamento lógico ou sequencial).

2. **Imagem do Circuito**:

Capturas do circuito esquemático mostrando:

- Conexões por meio do RTL Viewer de cada componente.

3. **Testes Realizados**:

Detalhamento das estratégias de teste, incluindo:

- Configuração de simulações, explicando como os estímulos foram aplicados aos pinos de entrada e como as saídas foram observadas.

2 DESCRIÇÃO

2.1 Plataforma de desenvolvimento

Para implementação do processador foi utilizada a IDE *Quartus Prime.*

Flow Status	Successful - Thu Mar 13 08:38:24 2025
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	processador8bits
Top-level Entity Name	processador8bits
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A until Partition Merge
Total registers	N/A until Partition Merge
Total pins	N/A until Partition Merge
Total virtual pins	N/A until Partition Merge
Total block memory bits	N/A until Partition Merge
Total PLLs	N/A until Partition Merge
Total DLLs	N/A until Partition Merge

Figura 1 - Especificações do projeto na plataforma de desenvolvimento.

2.2 Conjunto de instruções

Nosso processador de 8 bits segue um formato de instrução padronizado, onde cada instrução é dividida em campos específicos para facilitar a execução. As instruções são classificadas em três principais tipos:

- **Tipo R**: Operam diretamente nos registradores.

Opcode (3 bits)	Registrador 1 (2 bits)	Registrador 2 (2 bits)	Funct (1 bit)
000	01	10	0

- **Tipo I**: Instruções que envolvem memória.

Opcode (3 bits)	Registrador 1 (2 bits)	Immediate (3 bits)
000	01	101

- **Tipo J**: Saltos incondicionais.

Opcode (3 bits)	Address (5 bits)
000	10101

As instruções foram implementadas:

- **Load (lw)**: Carrega um valor da memória para um registrador.
- **Store (sw)**: Armazena um valor de um registrador na memória.
- **Soma (add)**: Realiza a adição entre dois registradores.
- **Subtração (sub)**: Realiza a subtração entre dois registradores.
- **Beq (branch if equal)**: Salta para um endereço se dois registradores forem iguais.
- **Jump (j)**: Salto incondicional.

Cada instrução foi codificada em VHDL dentro da Unidade de Controle, com seus respectivos códigos Opcode, que é a parte da instrução que indica qual operação será executada pela Unidade de Controle, a saber:

000: Jump (j)
001: Beq (branch if equal)
010: Load word (lw)
011: Store word (sw)
100: Soma (add)
101: Subtração (sub)
110: Load imediato (li)
111: Soma imediata (addi)

Tipo de Instrução	**Bits 7-5 (3 bits)**	**Bits 4-3 (2 bits)**	**Bits 2-1 (2 bits)**	**Bit 0 (1 bit)**
R (Registrador)	Opcode	Registrador 1	Registrador 2	Função
I (Imediato)	Opcode	Registrador	Imediato (3 bits)	-
J (Jump)	Opcode	Endereço (5 bits)	-	-

2.3 Descrição da linguagem

- Nosso processador suporta um conjunto reduzido de instruções em Assembly, semelhante ao MIPS.
- A sintaxe do assembly é semelhante à do MIPS, com comandos como `add`, `sub`, `lw`, `sw`, `beq`, e `j`.
- O conjunto de instruções será traduzido para código binário compatível com o processador.

Instruções Suportadas

Instrução	Binário	Descrição
ADD Rd, Rs	`100 Rd Rs 0`	Soma Rd + Rs e armazena em Rd
SUB Rd, Rs	`101 Rd Rs 1`	Subtrai Rd - Rs e armazena em Rd
ADDI Rs, Imm	`111 Rs Imm`	Soma Rs + Imediato e armazena em Rs
LOAD Rd, Imm	`010 Rd Imm`	Carrega o valor da memória para Rd
LOADI Rd, Imm	`110 Rd Imm`	Carrega o valor da memória para Rd
STORE Rd, Imm	`011 Rd Imm`	Armazena Rd na memória
JUMP Addr	`000 Addr`	Salta para o endereço Addr
BEQ	`001 Addr`	

Exemplo de código em Assembly:

```
lw $t0, 0($s1)   # Carrega um valor da memória para o registrador $t0
add $t1, $t0, $s2 # Soma o valor de $t0 e $s2, armazenando em $t1
sw $t1, 4($s1)   # Armazena o resultado de volta na memória
```

2.3.1 Datapath

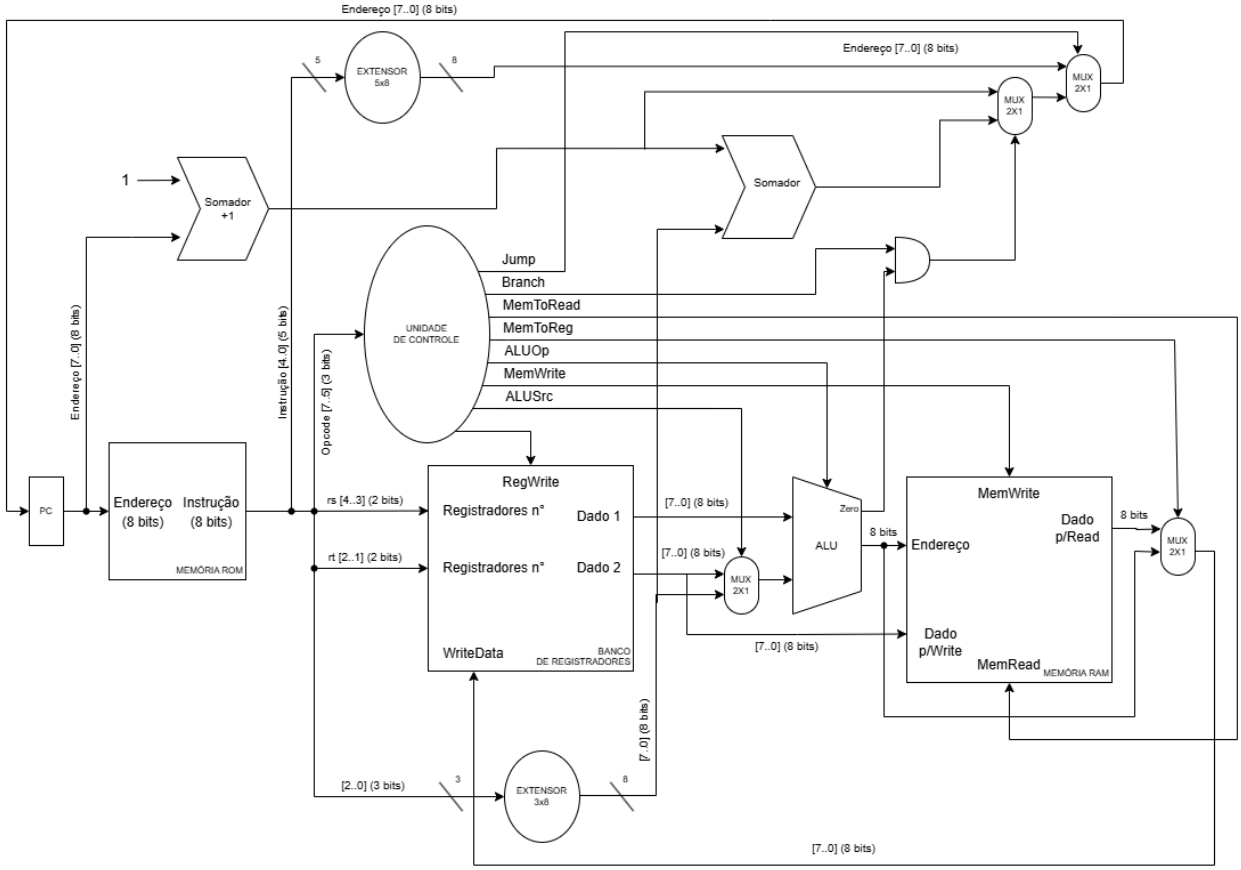

Figura 2 - Datapath do processador.

3 COMPONENTES

3.1 Banco de Registradores de 8 bits

O componente BancoRegistradores tem como principal objetivo escrever, ler e armazenar valores nos registradores.

O componente BancoRegistradores recebe como entrada quatro valores:

• Clock: dado de 1 bit;

• Data: dado de 1 byte a ser escrito em registrador.

• Endereco01 e Endereco02: dados de dois bits que indicam sobre quais registradores devem ser executados as operações;

• RegWrite: dado de 1 bit que indica se há escrita em registradores;

O componente BancoRegistradores tem duas saídas:

• Output01 e Output02: dados de 1 byte armazenados nos registradores.

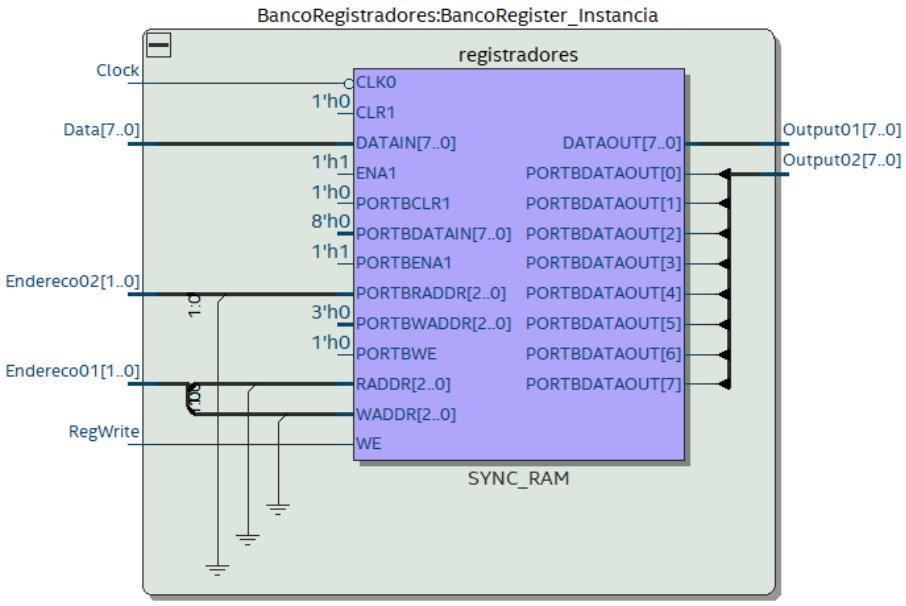

Figura 2 - RTL Viewer do banco de registradores.

3.2 Contador de programa

O componente PC é responsável por passar a linha de código do programa que deve ser executada. O componente PC recebe como entrada:

• Clock: dado de 1 bit;

• enderecoEntrada: dado de 1 byte com a linha atualizada.

O componente PC tem como saída:

• enderecoSaida: dado de 1 byte com a linha atual.

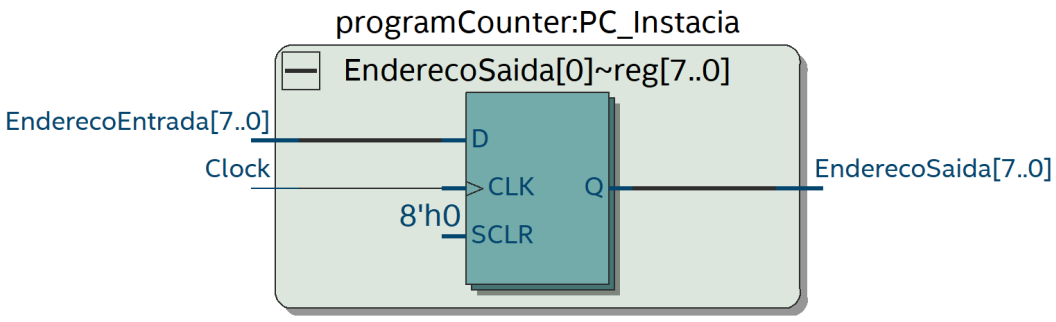

Figura 3 - RTL Viewer do PC.

3.3 Extensor de sinal 3x8

O componente Extensor3para8 estende um sinal de 3 bits para um de 1 byte. O componente Extensor3para8 recebe como entrada:

• Entrada: dado de 3 bits a ser extendido.

O componente Extensor3para8 tem como saída:

• Saida: dado de 1 byte extendido.

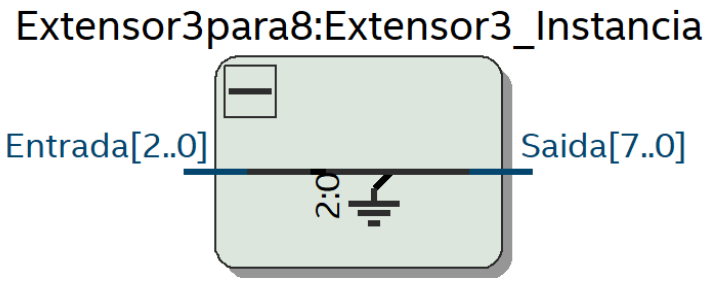

Figura 4 - RTL Viewer do extensor 3x8.

3.4 Extensor de sinal 5x8

O componente Extensor5para8 estende um sinal de 3 bits para um de 1 byte. O componente Extensor5para8 recebe como entrada:

• Entrada: dado de 5 bits a ser extendido.

O componente Extensor5para8 tem como saída:

• Saida: dado de 1 byte extendido.

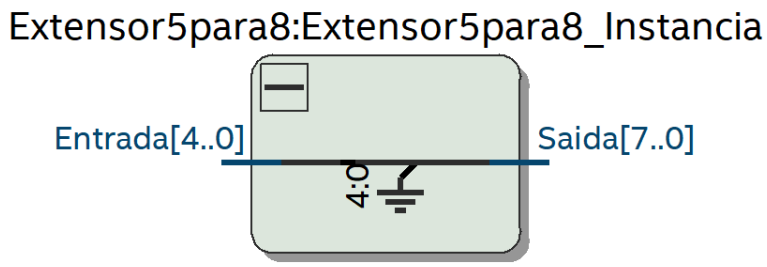

Figura 5 - RTL Viewer do extensor 5x8.

3.5 Memória ROM de 8 bits

O componente ROM é responsável por armazenar o programa, com até 256 linhas de código.

O componente recebe como entrada:

• Clock: dado de 1 bit;

• Address: dado de 1 byte com o endereço da linha a ser lida.

O componente ROM tem como saída:

• Data_out: dado de 1 byte armazenado naquela linha.

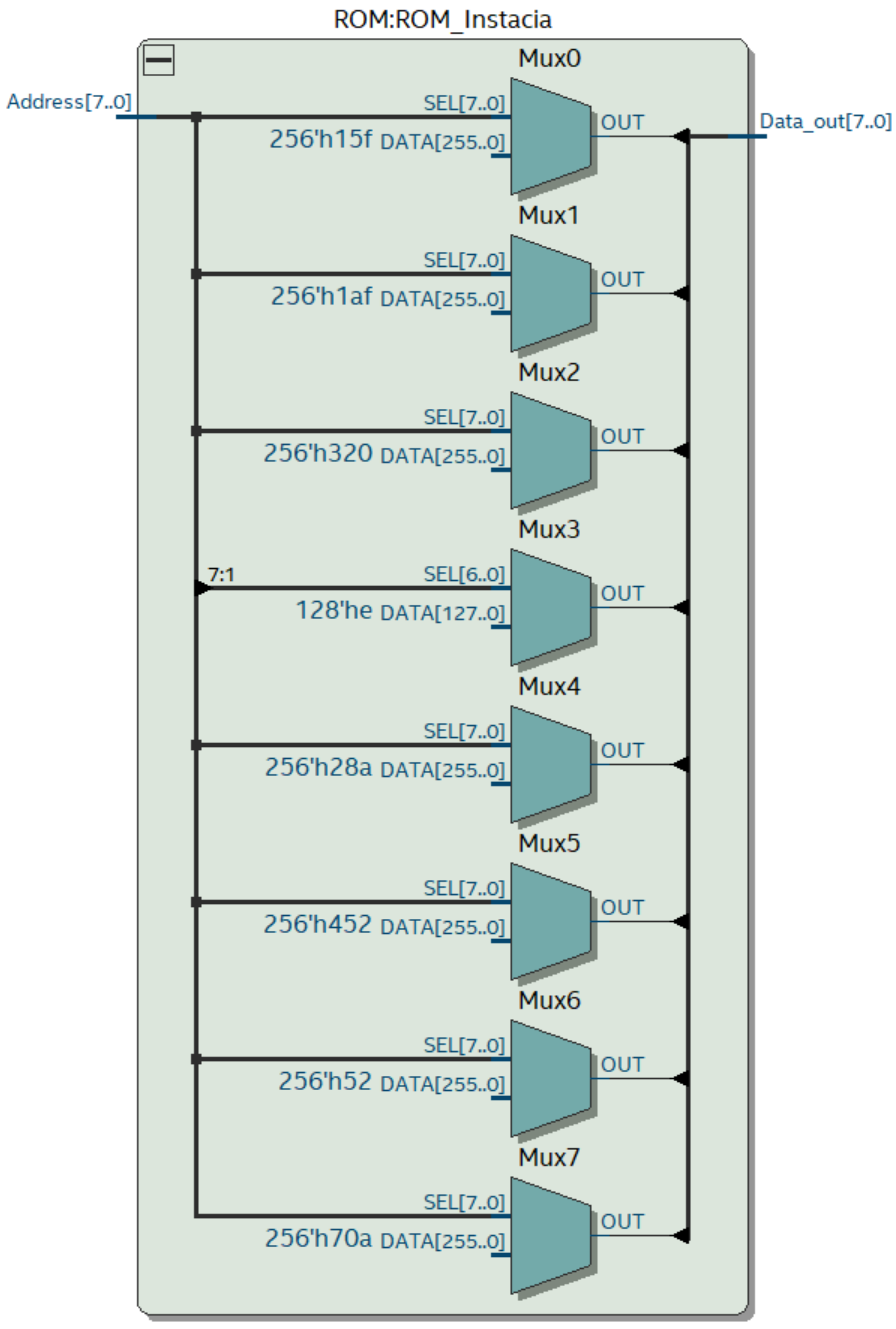

Figura 6 - RTL Viewer da memória ROM.

3.6 Memória RAM de 8 bits

O componente RAM é responsável por armazenar e ler até 8 dados de 1 byte por meio de instruções load e store.

O componente recebe como entrada:

• Clock: dado de 1 bit;
• InData: dado de 1 byte a ser escrito;
• Endereco: dado de 1 byte com endereço;

• MemWrite: dado de 1 bit que serve como flag;
• MemRead: dado de 1 bit que serve como flag.
O componente RAM tem como saída:
• S: dado de 1 byte com o resultado.

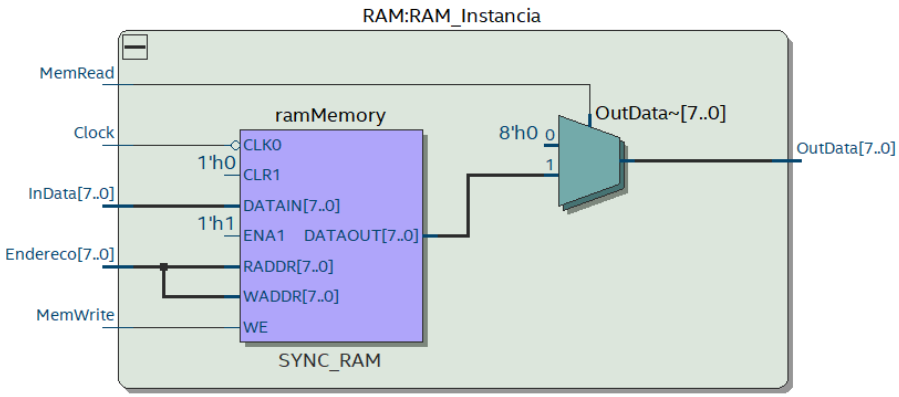

Figura 7 - RTL Viewer da memória RAM.

3.7 Multiplexador 2x1

O componente MUX seleciona um entre dois dados de 1 byte com base num seletor.
O componente recebe como entrada:
• InputA e InputB: dados de 1 byte;
• Chave: dado de 1 bit.
O componente tem como saída:
• Output: dado de 1 byte que foi selecionado.

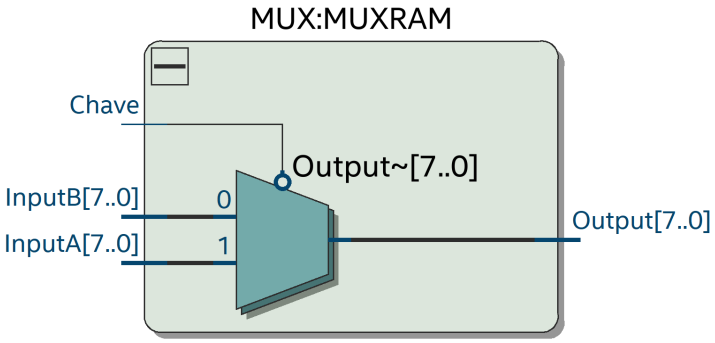

Figura 8 - RTL Viewer do multiplexador 2x1.

3.8 Somador mais 1

O componente SomadorMais1 tem como principal objetivo efetuar operações de soma entre dois dados de 1 byte.

O componente recebe como entrada:

• Entrada: dados de 1 byte com valores a serem somados A e B, onde o valor B é sempre 00000001 (1 em binário de 8 bits);

• CIN: dado de 1 byte com o carry in.

O componente SomadorMais1 tem como saída:

• OUT: dado de 1 byte com o resultado da soma de Entrada + 1.

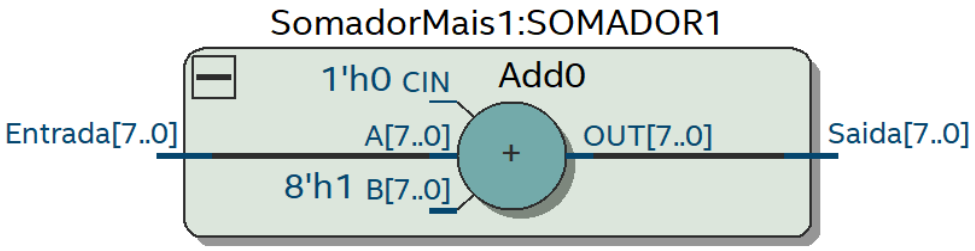

Figura 9 - RTL Viewer do somador mais 1.

3.9 Somador mais 2

O componente Somador2 tem como principal objetivo efetuar operações de soma entre dois dados de 1 byte.

O componente Somador2 recebe como entrada:

• Entrada1 e Entrada 2: dados de 1 byte com valores a serem somados;

• CIN: dado de 1 byte com o carry in.

O componente Somador2 tem como saída:

• Saida: dado de 1 byte com o resultado do somador, que será A + B.

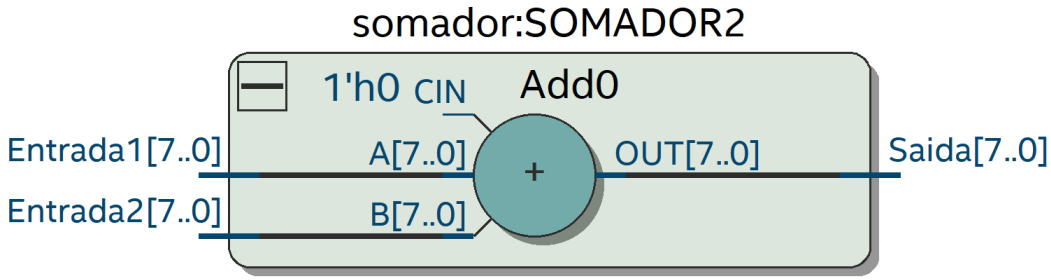

Figura 10 - RTL Viewer do somador mais 2.

3.10 Unidade de controle

A unidade de controle gera sinais para selecionar operações da ULA, ativar leitura/escrita na memória e determinar os caminhos do barramento. Para cada instrução, há um conjunto de sinais de controle que determina o comportamento do processador.

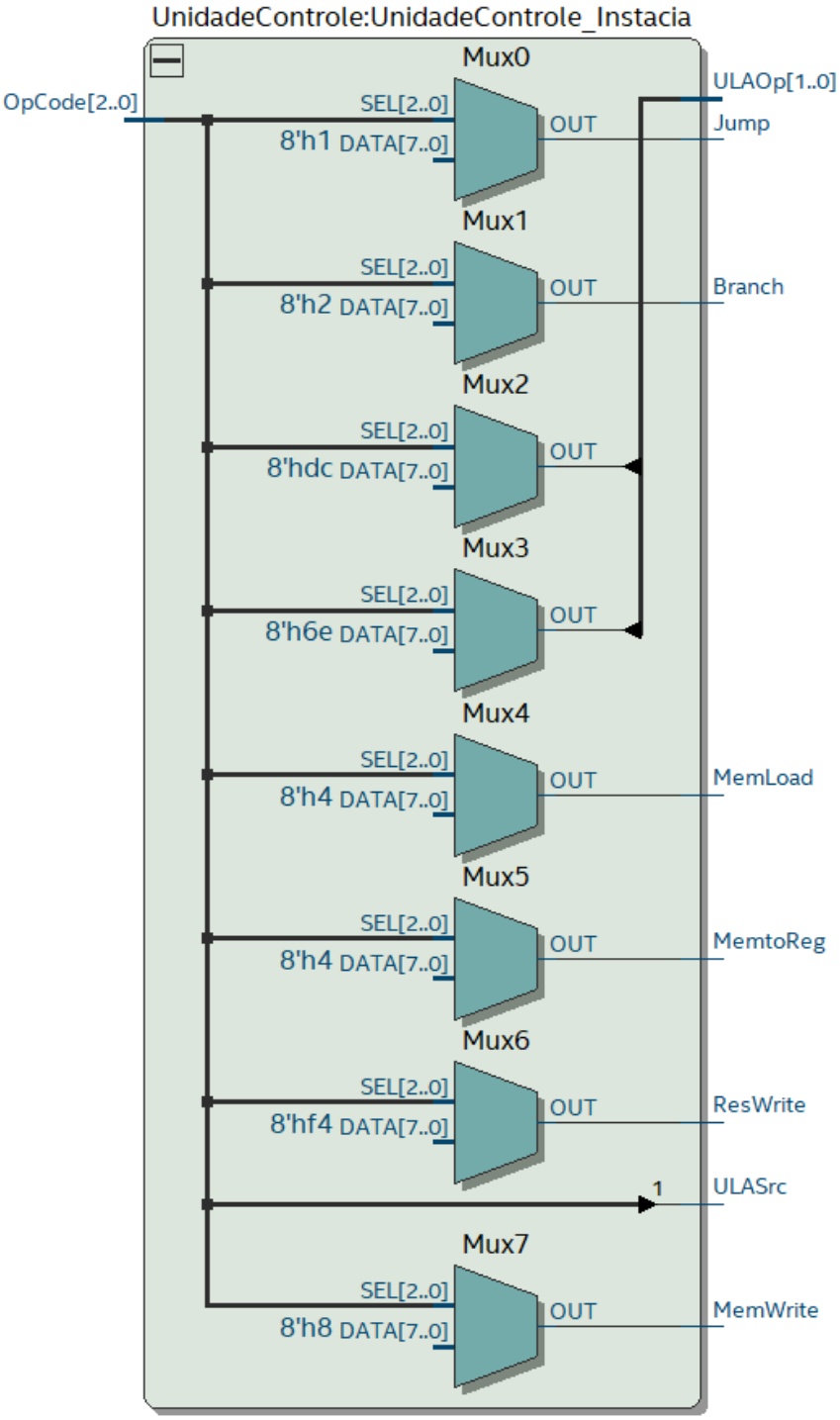

Figura 11 - RTL Viewer da unidade de controle.

O componente UnidadeControle recebe como entrada:

• Clock: dado de 1 bit;

• OpCode: dado de 3 bits, indicando a operação a ser realizada;

O componente UnidadeControle tem como saída:

• Jump: dado de 1 bit que serve de flag se há uma operação do tipo jump;

• Branch: dado de 1 bit que serve de flag se há uma operação característica de instrução tipo J;

• MemLoad: dado de 1 bit que server de flag se há leitura de memória;

• MemtoReg: dado de 1 bit que serve de flag se há escrita de um dado da memória num registrador;

• ULAOp: dado de 3 bits que indica a operação a ser realizada na ULA;

• MemWrite: dado de 1 bit que serve de flag se há escrita na memória;

• ULASrc: dado de 1 bit que serve de flag se há necessidade de executar operação com registrador ou imediato;

• ResWrite: dado de 1 bit que serve de flag se há escrita de dados no banco de registradores.

3.11 Unidade Lógica e Aritmética de 8 bits

O componente ALU (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas (considerando apenas resultados inteiros), dentre elas: soma e subtração. Adicionalmente, ela efetua operações de comparação de valor como BEQ.

O componente ALU recebe como entrada quatro valores:

• Clock: dado de 1 bit;

• Entrada1 e Entrada2: dados de 1 byte;

• Op: opcode de 4 bits.

A ALU possui duas saídas:

• Resultado: resultado de 1 byte;

• Zero: resultado de 1 bit para verificar se o valor retornado é zero;

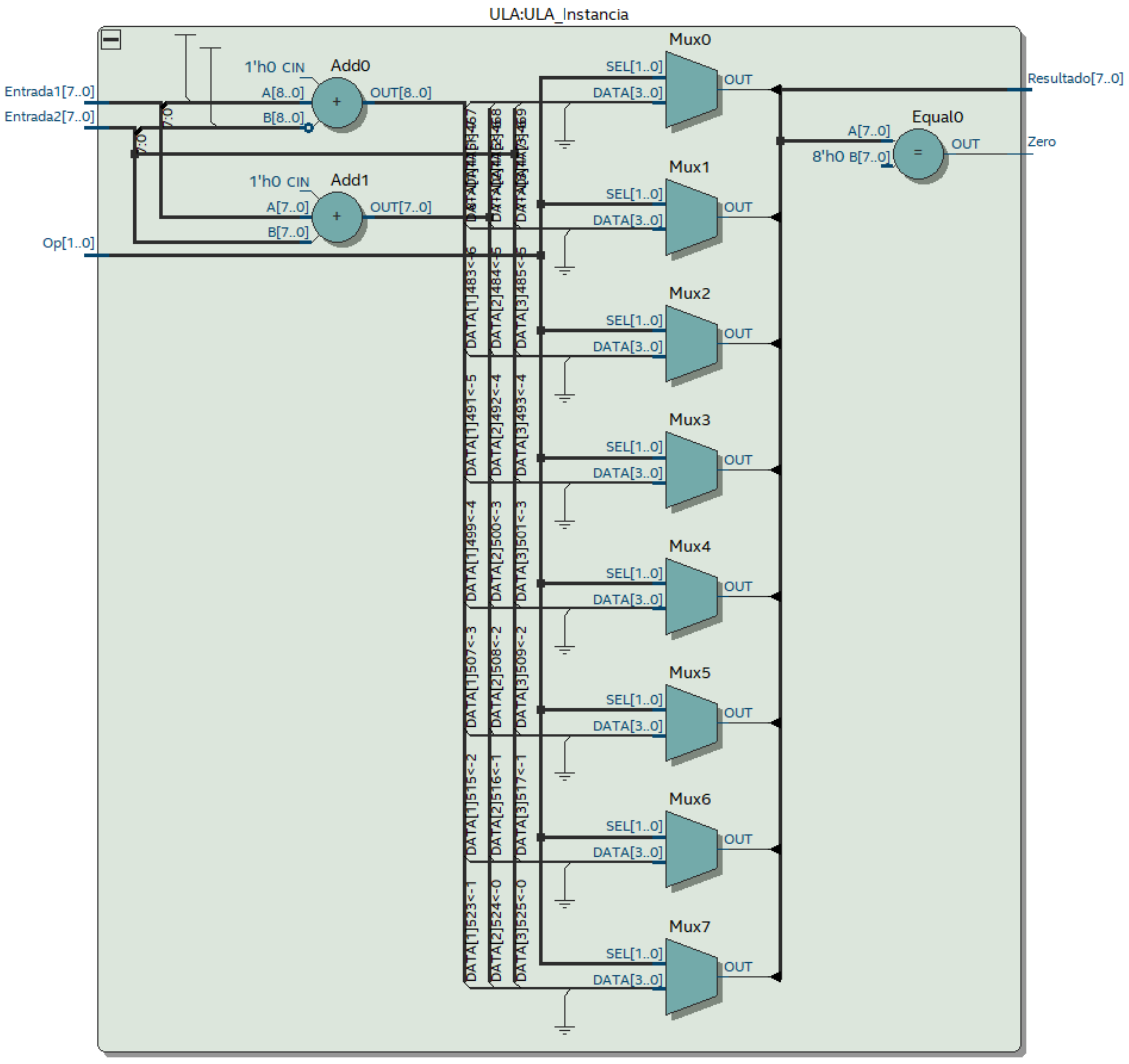

Figura 12 - RTL Viewer da ULA.

4 SIMULAÇÕES E TESTES

Usamos **waveforms** para visualizar a execução das operações e validar o fluxo de dados.

4.1 Teste ADD e ADDI

Esse teste verifica as operações de soma com registradores e soma imediata.

```
case address is
    -- Teste ADD e ADDI
    when "00000000" => data_out <= "11000010";  -- LI R1, 2
    when "00000001" => data_out <= "11001100";  -- LI R2, 4
    when "00000010" => data_out <= "10010110";  -- ADD R3, R4 (R3 = R1 + R2)
    when "00000011" => data_out <= "11110101";  -- ADDI R3, 5 (R3 = R3 + 5)

    when others => data_out <= "00000000";  -- Default (NOP)
```

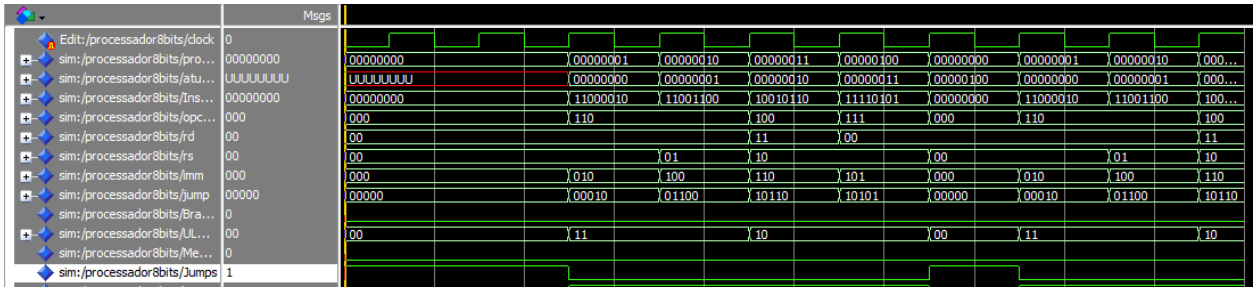

Figura 13 e 14 - Teste ADD e ADDI.

4.2 Teste SUB e SUBI

Aqui, validamos a subtração direta e com valores imediatos.

```
-- Teste SUB e SUBI
when "00000000" => data_out <= "11001111";  -- LI R1, 8
when "00000001" => data_out <= "11000101";  -- LI R2, 5
when "00000010" => data_out <= "10110110";  -- SUB R3, R4 (R3 = R1 - R2)
when "00000011" => data_out <= "00110001";  -- SUBI R3, 1 (R3 = R3 - 1)

when others => data_out <= "00000000";  -- Default (NOP)
```

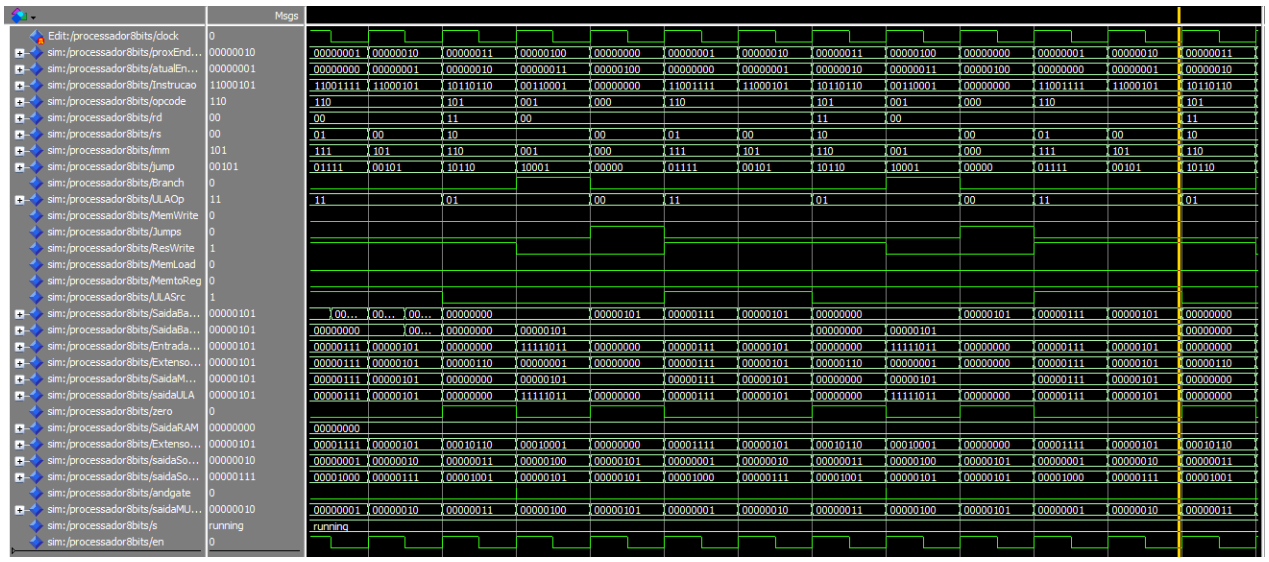

Figura 15 e 16 - Teste SUB e SUBI.

4.3 Teste BEQ e J

Esse teste valida desvios condicionais e incondicionais.

```
-- Teste BEQ e JUMP
when "00000000" => data_out <= "11001110";  -- LI R1, 7
when "00000001" => data_out <= "11001110";  -- LI R2, 7
when "00000010" => data_out <= "00101011";  -- BEQ R1, R2 (Se R1 == R2, pula para endereço 11)
when "00000011" => data_out <= "11000001";  -- LI R1, 1 (Se não pulou, carrega 1 em R1)
when "00000100" => data_out <= "00001000";  -- JUMP 8 (Pula para endereço 8)
when "00000101" => data_out <= "11000011";  -- LI R1, 3 (Nunca será executado se o BEQ funcionou)

when others => data_out <= "00000000";  -- Default (NOP)
```

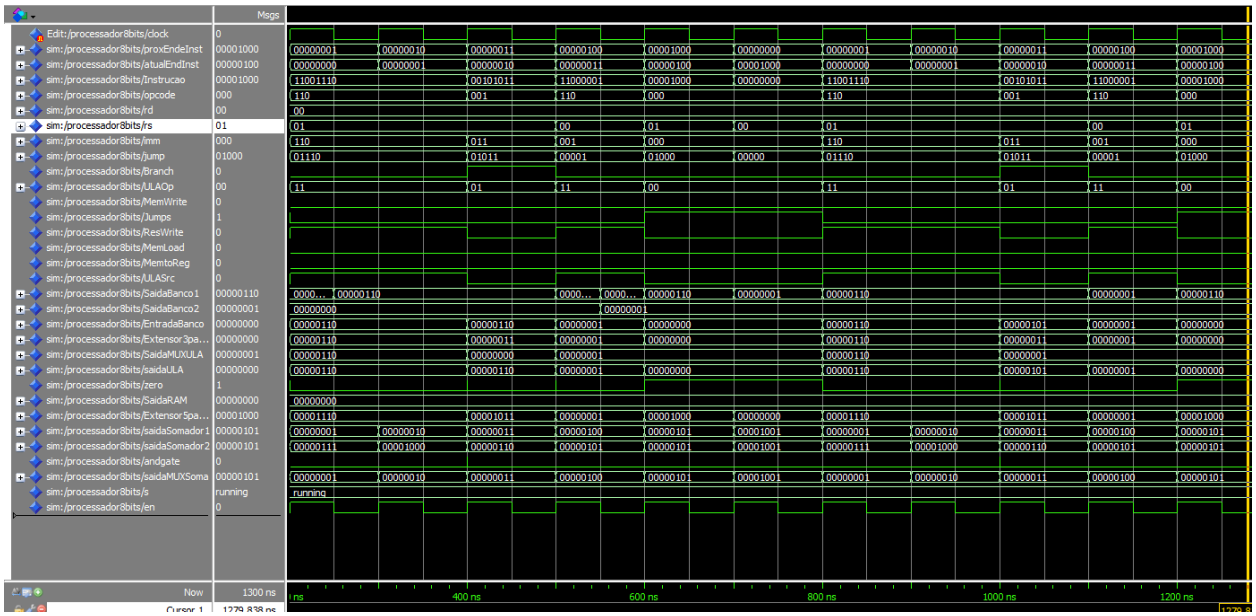

Figura 17 e 18 - Teste BEQ e JUMP.

4.4 Teste LW, SW e LI

Teste carregamento e armazenamento através do load word, store word e load imediato.

```
-- Teste LW, SW e LI
when "00000000" => data_out <= "11000010";   -- LI R1, 2
when "00000001" => data_out <= "01010000";   -- SW R1, 000 (Salva R1 na memória)
when "00000010" => data_out <= "01001100";   -- LW R3, 000 (Carrega R3 da memória)
when "00000011" => data_out <= "11000101";   -- LI R2, 5
when "00000100" => data_out <= "10010110";   -- ADD R3, R4 (R3 = R1 + R2)

when others => data_out <= "00000000";   -- Default (NOP)
```

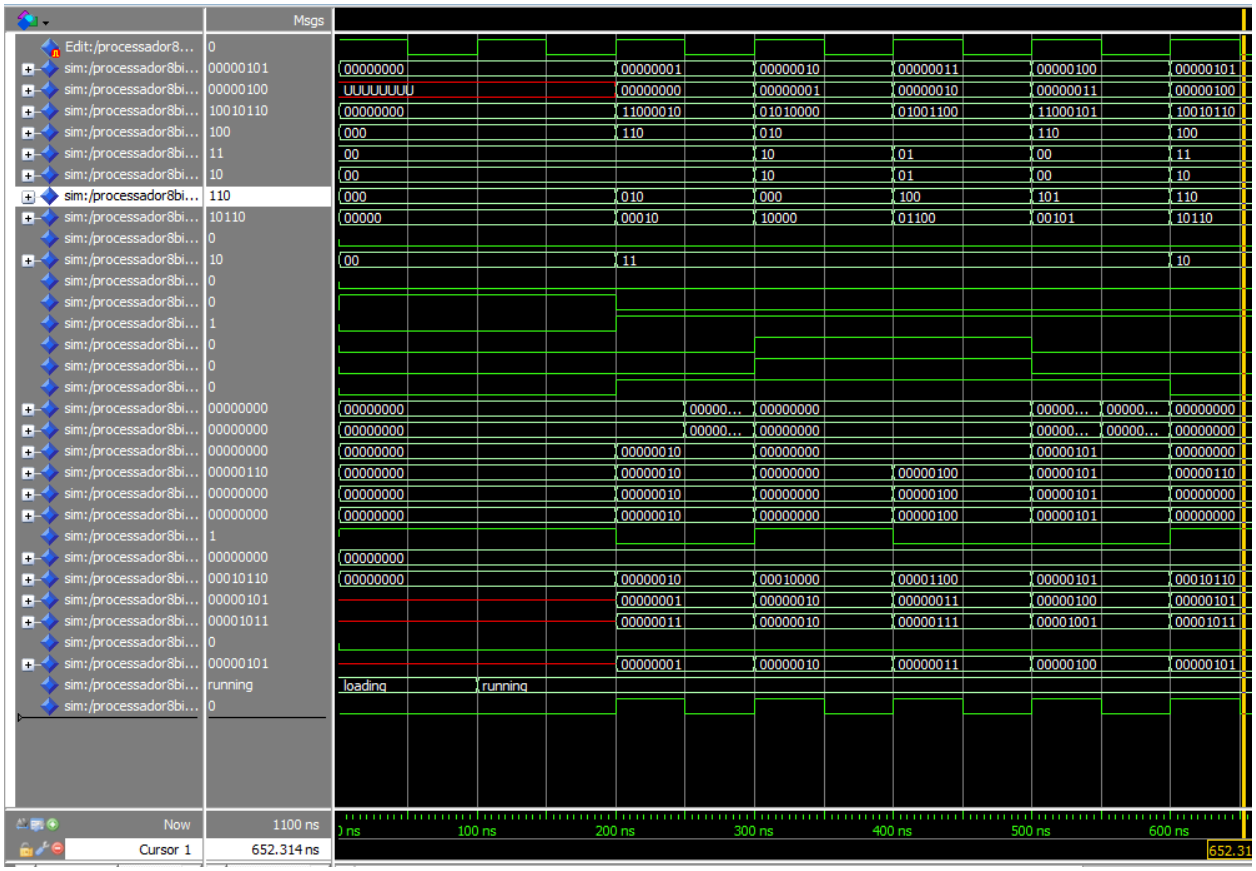

Figura 19 e 20 - Teste LW, SW e LI.

4.5 Teste FIBONACCI

Escrevemos um programa do algoritmo de Fibonacci que utiliza todas as instruções para garantir que o processador funcione corretamente, como temos resultado apresentado abaixo

```
case address is
    when "00000000" => data_out <= "11000000";   -- LI R1, 0
    when "00000001" => data_out <= "11001111";   -- LI R2, 8
    when "00000010" => data_out <= "11010000";   -- LI R3, 0
    when "00000011" => data_out <= "11011001";   -- LI R4, 1
    when "00000100" => data_out <= "10010110";   -- ADD R3, R4 (Fibonacci inicial)

    -- LOOP
    when "00000101" => data_out <= "01110000";   -- SW R3, 000 (Salva Fibonacci)
    when "00000110" => data_out <= "10010110";   -- ADD R3, R4 (Próximo Fibonacci)
    when "00000111" => data_out <= "01011000";   -- LW R4, 000 (Carrega R4 da memória)
    when "00001000" => data_out <= "11100001";   -- ADDI R1, 1 (contador++)
    when "00001001" => data_out <= "00101001";   -- BEQ R2, FIM (Se contador == 8, fim)
    when "00001010" => data_out <= "00000101";   -- JUMP LOOP (Volta para loop)

    -- FIM
    when "00001011" => data_out <= "00000000";   -- JUMP FIM (Loop infinito)

    when others => data_out <= "00000000";  -- Default (NOP)
end case;
```

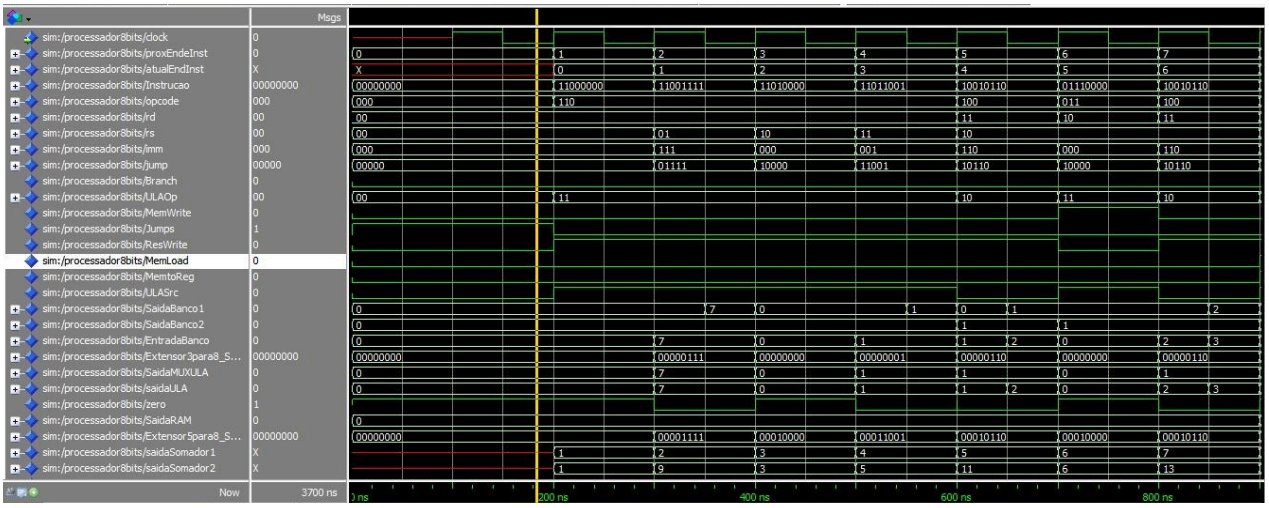

Figura 21 e 22 - Teste de Fibonacci.

5 REPOSITÓRIO

https://github.com/ArthurRamos26/AOC_Andersson_SilvaAndreza_GoncalvesArthur_Ramos_UFRR_2024