UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DCC301– ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES– 2024

PROF. DR. HEBERT OLIVEIRA ROCHA

LEONARDO VINÍCIUS LIMA CASTRO

ÁLEFE ALVES DA COSTA

RELATÓRIO DO PROCESSOR DE RISC DE 8 BITS

BOA VISTA, RR

2025

LEONARDO VINÍCIUS LIMA CASTRO

ÁLEFE ALVES DA COSTA

RELATÓRIO DO PROCESSOR DE RISC DE 8 BITS

Trabalho da disciplina de Arquitetura e Organização de Computadores do ano de 2024.2 apresentado à Universidade Federal de Roraima do curso de Bacharelado em ciência da computação.

Docente: Prof. Dr. Hebert O. Rocha

BOA VISTA, RR

2025

Sumário

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

Este relatório técnico apresenta o projeto e a implementação de um processador RISC de 8 bits, desenvolvido utilizando a linguagem de descrição de hardware VHDL no software Quartus Prime 18.0. O processador projetado segue uma arquitetura semelhante ao MIPS e tem como objetivo a aplicação de conceitos fundamentais de arquiteturas computacionais, circuitos digitais e linguagens de descrição de hardware.

A construção do processador envolve a modelagem e implementação de seus componentes principais, incluindo a unidade de controle, o caminho de dados (datapath) e os barramentos de comunicação. Além disso, a descrição da estrutura das instruções suportadas é detalhada por classes e distribuições de bits, incluindo sua representação em linguagem assembly e binário.

O conjunto de instruções obrigatórias implementadas inclui operações essenciais, como load, store, soma, subtração, beq (branch if equal) e salto incondicional, garantindo funcionalidade mínima para a execução de programas simples. Para validar o correto funcionamento do processador, foram realizadas simulações detalhadas por meio de waveforms e testbenches, cobrindo cada instrução individualmente e um programa que integra todas as operações suportadas.

Este trabalho documenta todo o processo de desenvolvimento, desde a descrição dos componentes e suas conexões até a validação funcional do processador, contribuindo para o entendimento e a aplicação prática de arquiteturas RISC em projetos acadêmicos e profissionais.

2. ESPECIFICAÇÕES

Esta seção descreve as especificações do ambiente de desenvolvimento e o conjunto de instruções utilizadas no projeto.

2.1. AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento do projeto foi realizado utilizando o Quartus Prime 18.0 Lite Edition, um ambiente integrado para o desenvolvimento de circuitos digitais em FPGA.

O Quartus Prime Lite Edition é uma ferramenta da Intel FPGA, utilizada para design, síntese, simulação e programação de FPGAs. Neste projeto, foi empregado VHDL para a implementação de um processador RISC de 8 bits, permitindo a descrição e a simulação da arquitetura antes da gravação no FPGA. Além disso, o ambiente oferece recursos gráficos para facilitar a configuração e depuração do circuito.

2.2. CONJUTO DE INSTRUÇÕES

O processador RISC de 8 bits desenvolvido neste projeto possui um conjunto de instruções organizadas em três formatos principais: Tipo R, Tipo I e Tipo J. Cada instrução segue um padrão de codificação baseado em opcodes e operandos, distribuídos em um formato de 8 bits.

2.2.1. Instruções do tipo R

As instruções do tipo R são utilizadas para operações aritméticas e lógicas entre registradores. Elas seguem o seguinte formato:

Opcode	rs	rt
7 - 5	4 - 3	2 - 1
3 bits	2 bits	2 bits

2.2.2. Instruções do tipo I

As instruções do tipo I realizam operações que envolvem constantes imediatas ou acesso à memória. Elas são divididas em dois formatos:

Opcode	rs	Immediate
7 - 5	4 - 3	2 - 0
3 bits	2 bits	3 bits

2.2.3. Instruções do tipo J

As instruções do tipo J (Jump) são usadas para saltos incondicionais no programa, modificando diretamente o contador de programa (PC).

Opcode	Adress
7 - 5	4 - 0
3 bits	5 bits

Tabela 1 – Opcodes suportados pelo processador

Opcode	Operação	Formato	Significado
000	add	R	Soma
001	sub	R	Subtração
010	lw	I	Load Word
011	sw	I	Store Word
100	beq	J	Branch if equal
101	j	J	Jump Incondicional

3. CONSTRUÇÃO DO PROCESSADOR

Esta seção apresenta os componentes desenvolvidos para o processador, e suas especificações.

3.1. PC

O componente PC é responsável por armazenar e passar o endereço da próxima linha de código do programa que deve ser executada. Ele atua como um registrador paralelo que atualiza sua saída com base no sinal de clock e no sinal de reset.

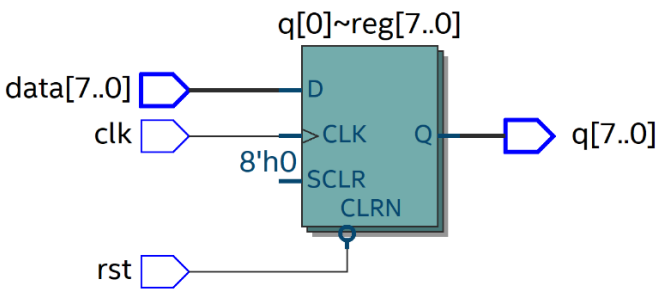
O componente PC recebe como entrada:

- CLOCK: Sinal de clock de 1 bit que sincroniza a atualização do endereço armazenado no PC.
- RESET: Sinal de reset de 1 bit que, quando ativo (nível baixo, '0'), zera o valor armazenado no PC.
- ADDRESS_IN: Dado de entrada de 8 bits (1 byte) que representa o novo endereço a ser armazenado no PC.

O componente PC tem como saída:

- ADDRESS_OUT: Dado de saída de 8 bits (1 byte) que representa o endereço atual armazenado no PC.

Figura 1 - PC viewer



3.2. MEMÓRIA DE INSTRUÇÕES

O componente memoria_instrucao é responsável por armazenar as instruções de um programa e fornecer a instrução correspondente ao endereço solicitado. Ele funciona como uma memória de leitura, onde o endereço de entrada é usado para buscar a instrução armazenada naquela posição.

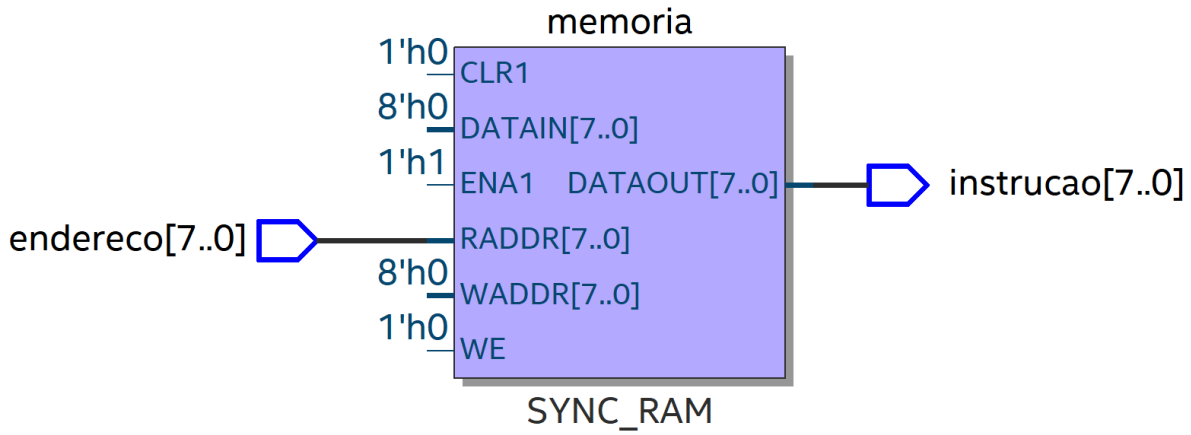
O componente memoria_instrucao recebe como entrada:

- ENDERECO: Dado de entrada de 8 bits (1 byte) que representa o endereço da instrução a ser buscada na memória.

O componente memoria_instrucao tem como saída:

- INSTRUCAO: Dado de saída de 8 bits (1 byte) que representa a instrução armazenada no endereço solicitado.

Figura 2 - Memória de instruções



3.3. SOMADOR

O componente somador é responsável por realizar a soma de dois valores de entrada e fornecer o resultado na saída.

O componente somador recebe como entrada:

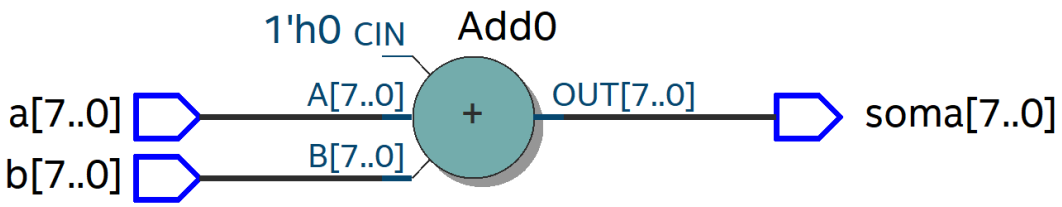
A: Dado de entrada de 8 bits (1 byte) que representa o primeiro operando da soma.

B: Dado de entrada de 8 bits (1 byte) que representa o segundo operando da soma.

O componente somador tem como saída:

SOMA: Dado de saída de 8 bits (1 byte) que representa o resultado da soma dos dois operandos de entrada.

Figura 3 - Somador de 8 bits



3.4. EXTENSOR DE BITS 3X8

O componente extensor_3x8 é responsável por estender um sinal de 3 bits para um sinal de 8 bits, preenchendo os bits mais significativos com zeros.

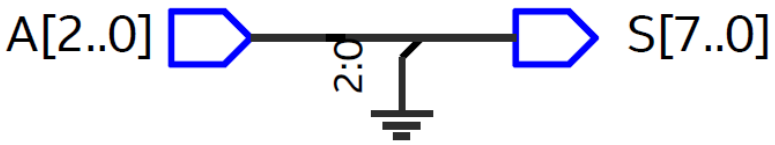
O componente extensor_3x8 recebe como entrada:

- A: Dado de entrada de 3 bits que representa o valor a ser estendido.

O componente extensor_3x8 tem como saída:

- S: Dado de saída de 8 bits que representa o valor estendido, onde os 5 bits mais significativos são preenchidos com zeros e os 3 bits menos significativos correspondem ao valor de entrada A.

Figura 4 - Extensor de bits 3x8



3.5. EXTENSOR DE BITS 5X8

O componente extensor_5x8 é responsável por estender um sinal de 5 bits para um sinal de 8 bits, preenchendo os bits mais significativos com zeros.

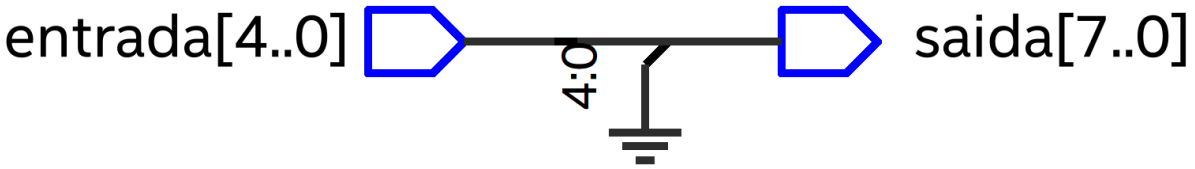
O componente extensor_5x8 recebe como entrada:

- entrada: Dado de entrada de 5 bits que representa o valor a ser estendido.

O componente extensor_5x8 tem como saída:

- saida: Dado de saída de 8 bits que representa o valor estendido, onde os 3 bits mais significativos são preenchidos com zeros e os 5 bits menos significativos correspondem ao valor de entrada, entrada.

Figura 5 - Extensor de bits 5x8



3.6. BANCO DE REGISTRADORES

O componente banco_de_registradores é responsável por armazenar e gerenciar um conjunto de registradores que podem ser lidos e escritos durante a execução de um programa. Ele é uma parte essencial de um processador, permitindo o armazenamento temporário de dados e a transferência de valores entre diferentes partes do sistema.

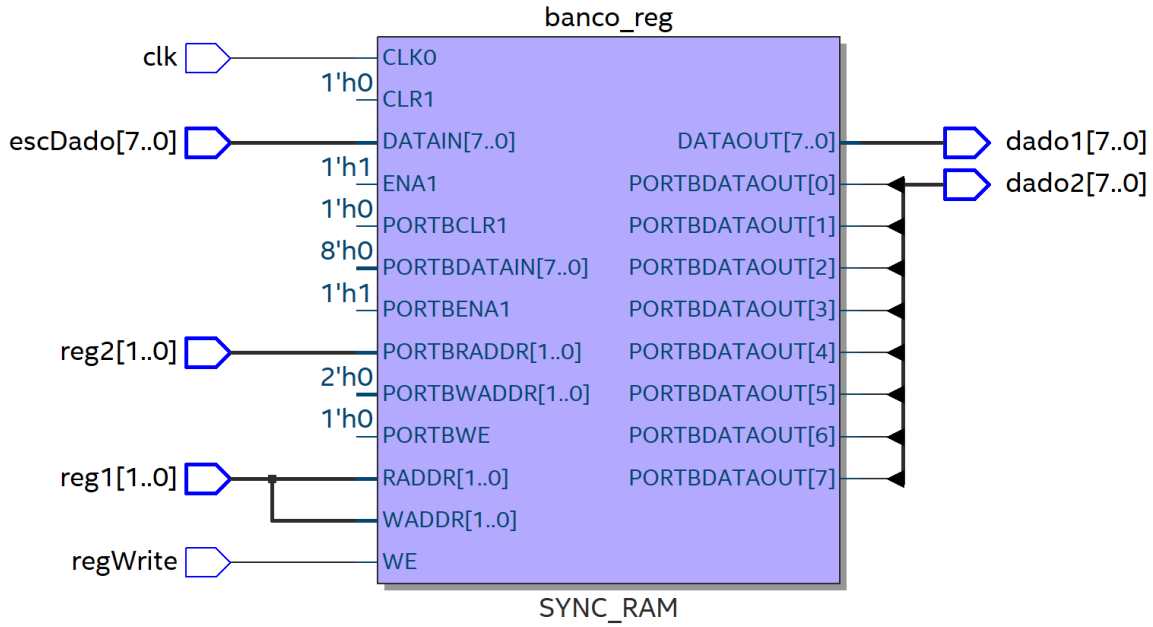
O componente banco_de_registradores recebe como entrada:

- CLOCK: Sinal de clock de 1 bit que sincroniza as operações de escrita no banco de registradores.
- REG_WRITE: Sinal de controle de 1 bit que indica se uma operação de escrita deve ser realizada. Quando ativo ('1'), o valor de WRITE_DATA é escrito no registrador especificado por REG1_IN.
- REG1_IN: Dado de entrada de 2 bits que especifica o endereço do registrador a ser lido ou escrito.
- REG2_IN: Dado de entrada de 2 bits que especifica o endereço do segundo registrador a ser lido.
- WRITE_DATA: Dado de entrada de 8 bits que contém o valor a ser escrito no registrador especificado por REG1_IN quando REG_WRITE está ativo.

O componente banco_de_registradores tem como saída:

- REG1_OUT: Dado de saída de 8 bits que contém o valor armazenado no registrador especificado por REG1_IN.
- REG2_OUT: Dado de saída de 8 bits que contém o valor armazenado no registrador especificado por REG2_IN.

Figura 6 – Banco de Registadores



3.7. UNIDADE DE CONTROLE

O componente unidade_controle é responsável por decodificar o opCode de uma instrução e gerar os sinais de controle necessários para a execução dessa instrução em um processador. Ele atua como o cérebro do processador, coordenando as operações da Unidade Lógica e Aritmética (ULA), da memória, e dos registradores.

O componente unidade_controle recebe como entrada:

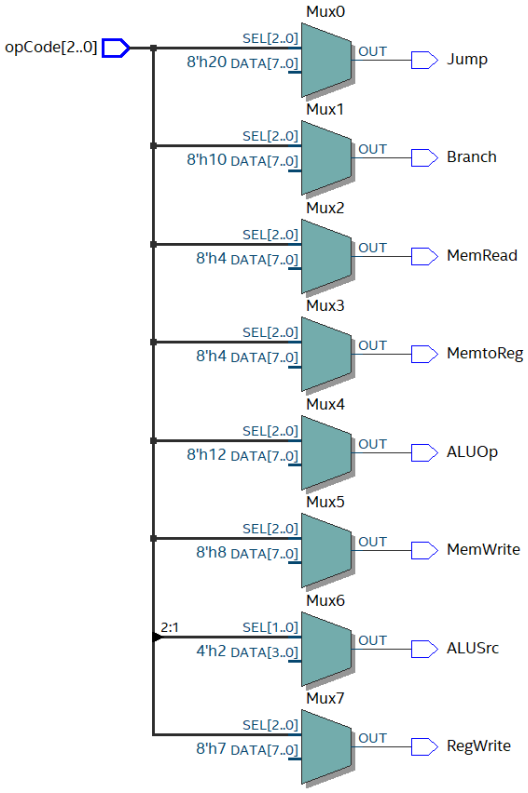
- opCode: Dado de entrada de 3 bits que representa o código da operação a ser executada.

O componente unidade_controle tem como saída:

- Jump: Sinal de 1 bit que indica se a instrução atual é um salto incondicional (jump).
- Branch: Sinal de 1 bit que indica se a instrução atual é um salto condicional (branch).
- MemRead: Sinal de 1 bit que indica se a memória deve ser lida.
- MemtoReg: Sinal de 1 bit que indica se o valor a ser escrito no registrador deve vir da memória.
- ALUOp: Sinal de 1 bit que controla a operação da ULA.
- MemWrite: Sinal de 1 bit que indica se a memória deve ser escrita.
- ALUSrc: Sinal de 1 bit que indica se o segundo operando da ULA deve vir de um registrador ou de um valor imediato.

- RegWrite: Sinal de 1 bit que indica se o banco de registradores deve ser escrito.

Figura 7 - Unidade de Controle



3.8. MULTIPLEXADOR 2X1

O componente mux_2x1 é um multiplexador de 2 para 1 que seleciona entre duas entradas de 8 bits com base em um sinal de seleção. Ele é utilizado para escolher entre dois sinais de entrada e direcionar um deles para a saída, dependendo do valor do sinal de controle.

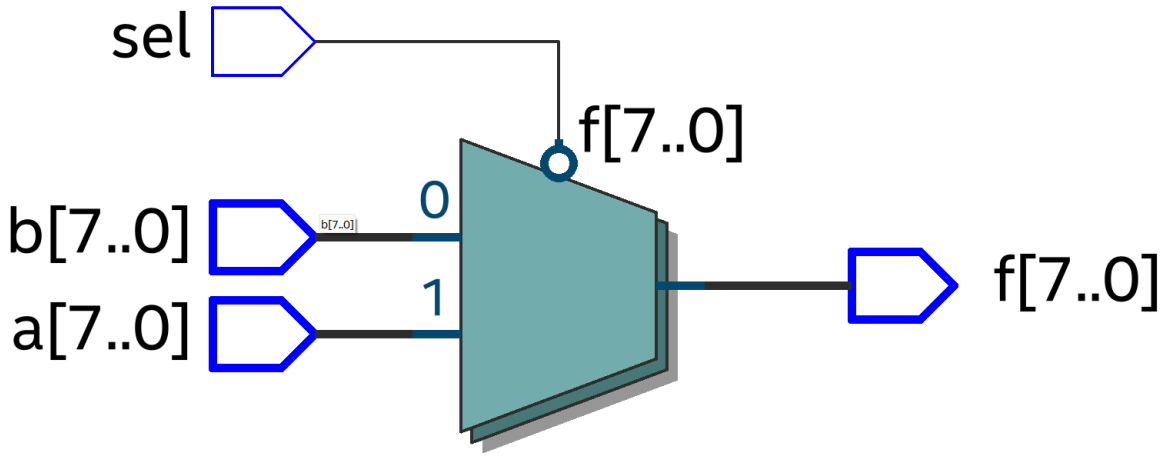
O componente mux_2x1 recebe como entrada:

- a: Dado de entrada de 8 bits que representa a primeira opção de entrada.
- b: Dado de entrada de 8 bits que representa a segunda opção de entrada.
- sel: Sinal de seleção de 1 bit que determina qual das duas entradas será direcionada para a saída.

O componente mux_2x1 tem como saída:

- f: Dado de saída de 8 bits que corresponde à entrada selecionada (a ou b).

Figura 8 – Multiplexador 2x1



3.9. MEMÓRIA DE DADOS

O componente memoria_dados é responsável por armazenar e gerenciar dados em uma memória de acesso aleatório (RAM). Ele permite a leitura e escrita de dados em endereços específicos da memória, com base em sinais de controle.

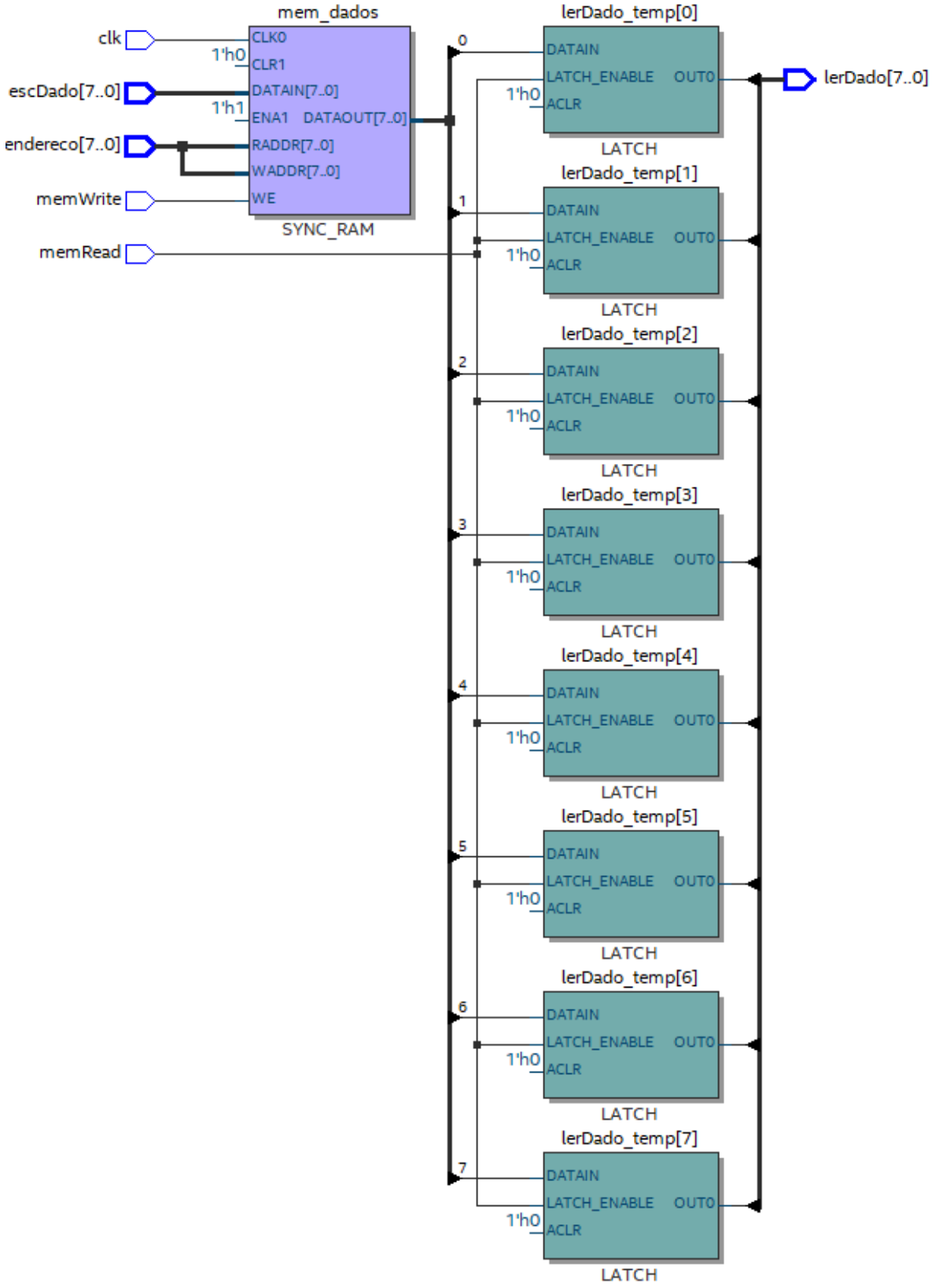
O componente memoria_dados recebe como entrada:

- endereco: Dado de entrada de 8 bits que especifica o endereço da memória onde a leitura ou escrita será realizada.
- escDado: Dado de entrada de 8 bits que contém o valor a ser escrito na memória quando a operação de escrita está ativa.
- memWrite: Sinal de controle de 1 bit que indica se uma operação de escrita deve ser realizada.
- memRead: Sinal de controle de 1 bit que indica se uma operação de leitura deve ser realizada.
- clk: Sinal de clock de 1 bit que sincroniza as operações de escrita na memória.

O componente memoria_dados tem como saída:

- lerDado: Dado de saída de 8 bits que contém o valor lido da memória quando a operação de leitura está ativa.

Figura 9 – Memória de dados



3.10. ULA

O componente ula é responsável por realizar operações aritméticas e lógicas em dois operandos de 8 bits. Neste caso, a ULA suporta duas operações: soma e subtração. Além disso, ela gera um sinal de saída zero que indica se o resultado da operação é zero.

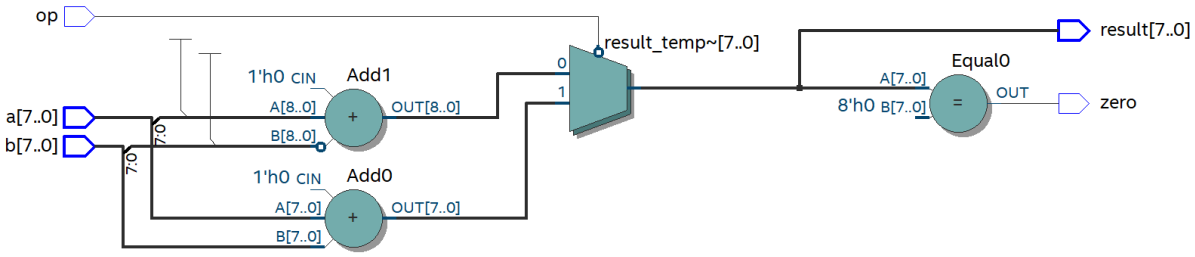
O componente ula recebe como entrada:

- a: Dado de entrada de 8 bits que representa o primeiro operando.
- b: Dado de entrada de 8 bits que representa o segundo operando.
- op: Sinal de controle de 1 bit que determina a operação a ser realizada:
    - op = '0': Realiza uma soma.
    - op = '1': Realiza uma subtração.

O componente ula tem como saída:

- result: Dado de saída de 8 bits que representa o resultado da operação realizada (soma ou subtração).
- zero: Sinal de saída de 1 bit que indica se o resultado da operação é zero:
    - zero = '1': O resultado é zero.
    - zero = '0': O resultado não é zero.

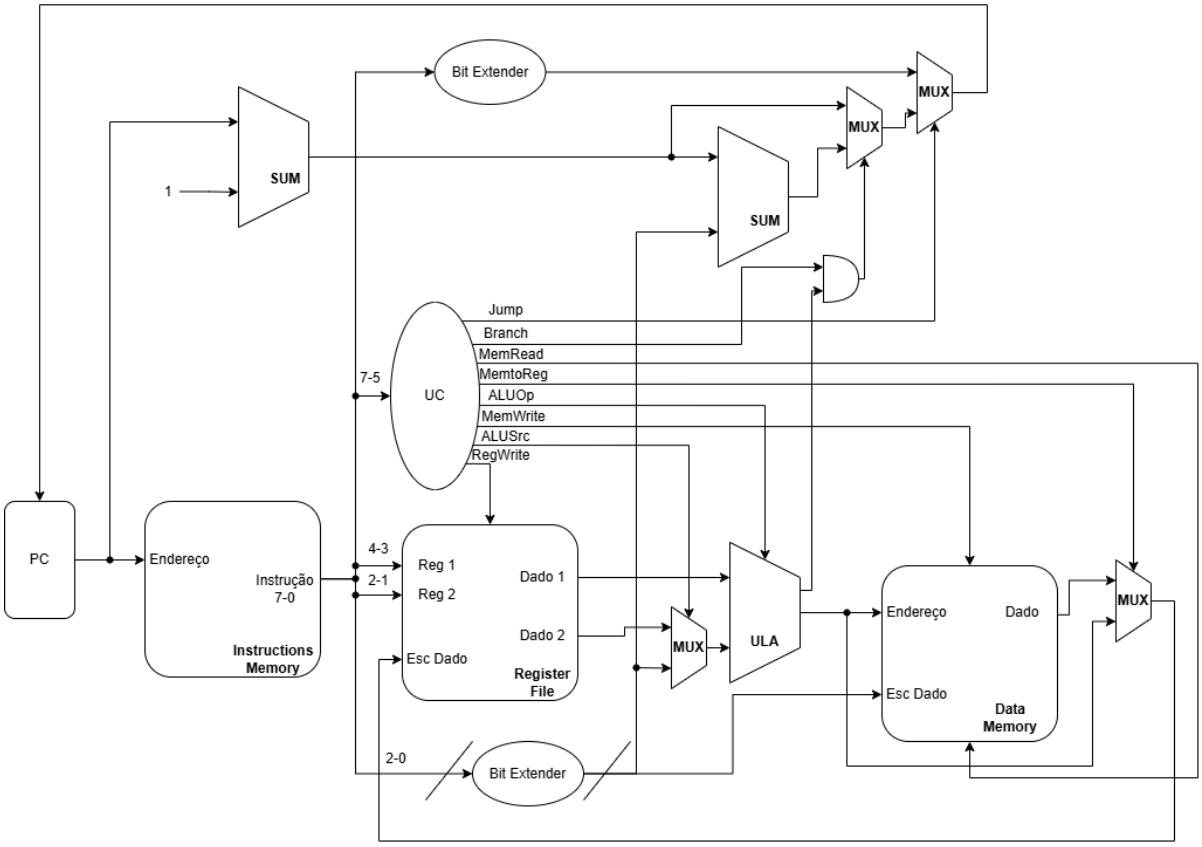
Figura 10 – ULA



3.11. DATAPATH

O datapath é a estrutura que permite a execução de instruções em um processador. Ele inclui componentes como o Bit Extender, que ajusta o tamanho dos dados, e a ALU, que realiza operações matemáticas. Multiplexadores (MUX) direcionam o fluxo de dados, enquanto sinais de controle, como Jump e Branch, gerenciam desvios no programa. Operações de memória são controladas por MemRead e MemWrite, e o resultado é armazenado em registradores com RegWrite. Juntos, esses elementos garantem a execução eficiente das instruções.

Figura 11 - Datapath



4. SIMULAÇÕES

4.1. TESTE ADD E SUB

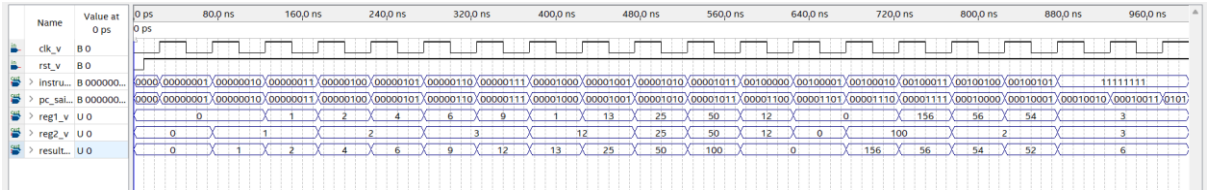
Figura 12 – Instruções do teste ADD e SUB

```
signal memoria: instrucoes := (  "00000000",--Instrução 1:  add $s0, $s0, $s0;
                                 "00000001",--Instrução 2:  add $s0, $s0, $s0;
                                 "00000010",--Instrução 3:  add $s0, $s0, $s1;
                                 "00000011",--Instrução 4:  add $s0, $s0, $s1;
                                 "00000100",--Instrução 5:  add $s0, $s0, $s2;
                                 "00000101",--Instrução 6:  add $s0, $s0, $s2;
                                 "00000110",--Instrução 7:  add $s0, $s0, $s3;
                                 "00000111",--Instrução 8:  add $s0, $s0, $s3;
                                 "00001000",--Instrução 9:  add $s1, $s1, $s0;
                                 "00001001",--Instrução 10: add $s1, $s1, $s0;
                                 "00001010",--Instrução 11: add $s1, $s1, $s1;
                                 "00001011",--Instrução 12: add $s1, $s1, $s1;

                                 "00100000",--Instrução 13: sub $s0, $s0, $s0;
                                 "00100001",--Instrução 14: sub $s0, $s0, $s0;
                                 "00100010",--Instrução 15: sub $s0, $s0, $s1;
                                 "00100011",--Instrução 16: sub $s0, $s0, $s1;
                                 "00100100",--Instrução 17: sub $s0, $s0, $s2;
                                 "00100101",--Instrução 18: sub $s0, $s0, $s2;

                                 others => (others => '1' )--Instruções Inválidas
```

Figura 13 – Waveform das instruções do teste ADD e SUB



4.2. TESTE LW E SW

Figura 14 – Instruções do teste LW, SW

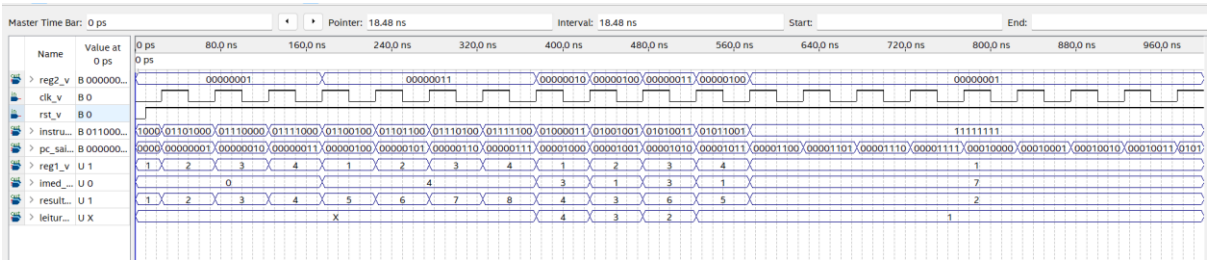
```
:= (  "01100000",--Instrução 1:  sw $s0, 0($s0);<=Salva 1 na posição 1 da memoria
      "01101000",--Instrução 2:  sw $s1, 0($s1);<=Salva 2 na posição 2 da memoria
      "01110000",--Instrução 3:  sw $s2, 0($s2);<=Salva 3 na posição 3 da memoria
      "01111000",--Instrução 4:  sw $s3, 0($s3);<=Salva 4 na posição 4 da memoria

      "01100100",--Instrução 5:  sw $s0, 0($s0);<=Salva 1 na posição 5 da memoria
      "01101100",--Instrução 6:  sw $s1, 0($s1);<=Salva 2 na posição 6 da memoria
      "01110100",--Instrução 7:  sw $s2, 0($s2);<=Salva 3 na posição 7 da memoria
      "01111100",--Instrução 8:  sw $s3, 0($s3);<=Salva 4 na posição 8 da memoria

      "01000011",--Instrução 9:  lw $s0, 4($s0);<=Carrega da posição 4 da memoria o valor 4 para o reg 0
      "01001001",--Instrução 10: lw $s1, 1($s1);<=Carrega da posição 3 da memoria o valor 3 para o reg 1
      "01010011",--Instrução 11: lw $s2, 3($s2);<=Carrega da posição 6 da memoria o valor 2 para o reg 2
      "01011001",--Instrução 12: lw $s3, 2($s3);<=Carrega da posição 5 da memoria o valor 1 para o reg 3

      others => (others => '1' )--Instruções Inválidas
```

Figura 15 – Waveform das instruções do teste LW e SW



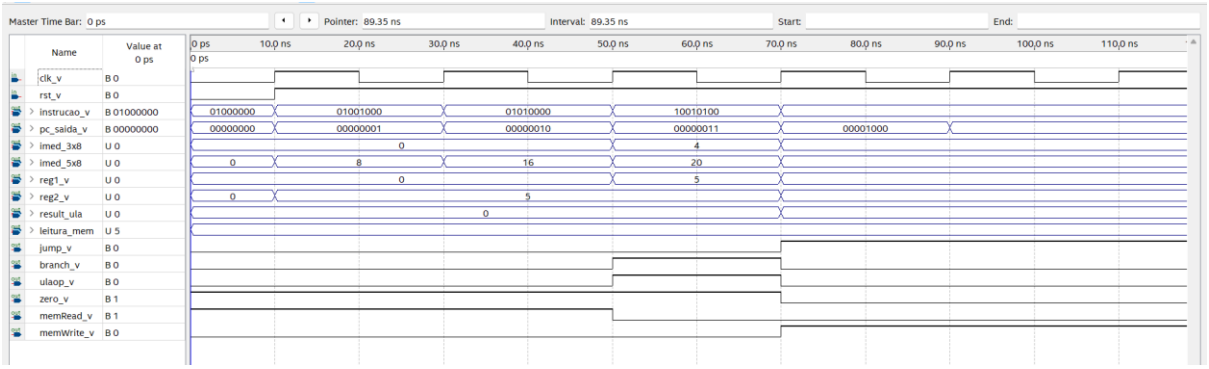
4.3. TESTE BEQ E JUMP

Figura 16 – Instruções do teste BEQ e JUMP

```
(   "01000000",--Instrução 1:  lw $s0, 0($s0);<=Carrega o valor da posição 0 da memoria para o reg 0
    "01001000",--Instrução 2:  lw $s1, 0($s1);<=Carrega o valor da posição 1 da memoria para o reg 1
    "01010000",--Instrução 3:  lw $s2, 0($s2);<=Carrega o valor da posição 3 da memoria para o reg 2
    "10010100",--Instrução 4:  beq $s3, inicio;<=Compara reg3 ao reg0 e pula para o fim do programa
    "00000010",--Instrução 5:  add $s0, $s0, $s1;
    "10101000",--Instrução 6:  jump 1000;<= Pula pra 8 instrução;
    "00110010",--Instrução 7:  sub $s2, $s2, $s1;<=Nunca executada.

    others => (others => '1' )--Instruções Inválidas
```

Figura 17 – Waveform das instruções do teste BEQ e JUMP



5. CONCLUSÃO

Este projeto, desenvolvido para a matéria de Arquitetura e Organização de Computadores (AOC), foi uma experiência muito importante para o nosso aprendizado. O objetivo era criar um processador RISC de 8 bits, e, ao longo do processo, enfrentamos alguns desafios, mas também tivemos muitas conquistas.

Uma das maiores dificuldades foi aprender a usar a linguagem VHDL, que é bem diferente das linguagens de programação que já conhecíamos. No começo, foi complicado entender como descrever o hardware de forma clara e funcional, mas, com prática e dedicação, conseguimos superar essa barreira. Outro desafio foi integrar todos os componentes do processador, como a unidade de controle, a ULA e os registradores. Foi preciso bastante atenção para garantir que tudo funcionasse corretamente junto.

Apesar das dificuldades, o projeto foi muito interessante porque nos ajudou a fixar melhor o conteúdo visto em sala de aula. Conseguimos ver na prática como os conceitos de arquitetura RISC, pipeline e funcionamento de um processador são aplicados. Além disso, as dúvidas que surgiram durante o desenvolvimento foram resolvidas em grupo, o que mostrou como a colaboração é essencial para o sucesso de um projeto.

No final, conseguimos concluir o processador e entender cada parte do seu funcionamento. Esse projeto não só reforçou nosso conhecimento em AOC, mas também nos preparou para enfrentar desafios maiores no futuro. Foi uma experiência que valeu muito a pena.

6. REFERÊNCIAS

STALLINGS, William; BOSNIC, Ivan; VIEIRA, Daniel. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Prentice Hall, 2006.

PATTERSON, David A. **Organização e projeto de computadores**. 3. ed. Rio de Janeiro: Elsevier, 2005.