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

Lucas Gabriel Rocha Constancio

Yan dos Santos Teixeira

Relatorio trabalho final Processador RISC 8 bits

Boa Vista - RR
2024

Relatorio Trabalho Final AOC – [Task 01] Projetar e implementar um processador RISC de 8 bits (semelhante ao MIPS)]

Trabalho apresentado à disciplina de Arquitetura e Organização de Computadores, do Curso de Ciência da Computação, da UFRR (Universidade Federal de Roraima), como requisito parcial para obtenção de nota na disciplina.

Orientador(a): Prof. Dr. Hebert Oliveira Rocha

Boa Vista - RR
2024

RESUMO

O presente trabalho descreve o projeto e a implementação de um processador RISC de 8 bits, desenvolvido com base na arquitetura reduzida de instruções computacionais (RISC), utilizando a linguagem VHDL e a ferramenta Intel Quartus Prime Lite. Ao longo deste relatório, são apresentados em detalhes os componentes essenciais para o correto funcionamento do processador, bem como as estratégias de implementação adotadas e os resultados de testes realizados durante o desenvolvimento do projeto.

Palavras-chave: 8 bits; VHDL; Intel Quartus Prime Lite; Arquitetura e Organização de Computadores.

Sumario

1. Introdução e Especificação

Neste projeto, desenvolveu-se um **processador RISC de 8 bits**, descrito em **VHDL**. O trabalho foi dividido em diversos módulos (componentes), que juntos implementam a arquitetura básica de um processador, incluindo unidades de controle, aritmética, memória e gerenciamento de dados. Os principais arquivos VHDL analisados foram:

- **Divisor.vhd**
- **Extensor4x16.vhd**
- **multiplexador.vhd**
- **Multiplexador_2x1.vhd**
- **PC.vhd**
- **Processador.vhd** (arquivo *top-level* que integra todos os módulos)
- **RAM.vhd**
- **Registradores.vhd**
- **Rom.vhd**
- **somador.vhd**
- **Uc.vhd**
- **ULA.vhd**
- **ULA_tb.vhd**

2. Plataforma de Desenvolvimento

O desenvolvimento do processador ocorreu na IDE **Intel Quartus Prime Lite Edition**, com a **linguagem VHDL** para descrição de hardware e o **ModelSim** para verificação e teste dos módulos por meio de simulações. Essa plataforma permitiu explorar desde a criação de circuitos combinacionais e sequenciais até a integração completa dos componentes.

3. Conjunto de Instruções

O processador trabalha com instruções de **8 bits**, divididas em campos para **opcode** (4 bits) e **operando** (4 bits). Embora o tamanho dos dados seja reduzido, o conjunto de instruções contempla operações aritméticas, lógicas, de acesso à memória e saltos. A estrutura básica é:

- **Opcode (4 bits)**: Define a operação a ser executada.
- **Operando (4 bits)**: Representa o registrador ou valor imediato, dependendo do tipo de instrução.

Instruções Implementadas

- **ADD**: Soma o conteúdo de dois registradores.
- **ADDi**: Soma com valor imediato.
- **SUB/SUBi**: Subtração entre registradores ou com valor imediato.
- **LW / SW**: Acesso à memória para carregamento ou armazenamento de dados.
- **JUMP**: Salto incondicional para uma determinada posição de memória.
- **BeQ**: Salto condicional.
- **END Gate**: Verifica se todas as entradas são verdadeiras.
- **OR Gate**: Verifica se apenas uma entrada é verdadeira.

Campo	Tamanho (bits)	Descrição
Opcode	4	Código da operação
Operando	4	Registrador ou valor imediato

4. Descrição do Hardware

Nesta seção, descrevem-se cada um dos componentes (módulos em VHDL) que compõem o processador RISC de 8 bits.

4.1 Extensor de Sinal (Extensor_4x8.vhd)

O componente **Extensor_4x8** tem como principal objetivo efetuar a extensão de sinal de um valor imediato de 4 bits para um formato de 8 bits, garantindo que o valor estendido mantenha a mesma representação numérica (em complemento de dois) do valor original.

O elemento recebe como **entrada**:
• **in4** – Valor imediato de 4 bits.

E fornece como **saída**:
• **out8** – Valor estendido para 8 bits.

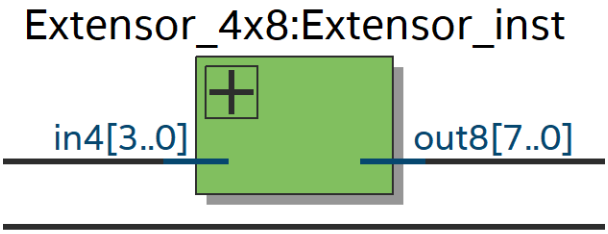
Funcionamento:

O módulo realiza a extensão com sinal replicando o bit mais significativo do valor imediato (in4(3)) nos 4 bits superiores da saída, seguido pela concatenação dos 4 bits originais. Dessa forma, se o valor imediato for negativo (bit de sinal igual a '1'), os 4 bits superiores em out8 também serão '1', preservando corretamente o sinal do número.

Exemplo de operação:

- Se **in4** = "1010", então **out8** será "11111010".
- Se **in4** = "0101", então **out8** será "00000101".

Figura 1



4.2 Multiplexadores (Multiplexador_2x1.vhd)

O componente **Multiplexador** tem como principal objetivo selecionar, entre duas entradas de 8 bits, qual delas será encaminhada para a saída, de acordo com o sinal de seleção.

O elemento recebe como **entradas**:
• **sel** – Sinal de seleção (1 bit), que determina qual entrada será escolhida.
• **d0** – Entrada 0 (8 bits).
• **d1** – Entrada 1 (8 bits).

E fornece como **saída**:
• **y** – Saída (8 bits), que recebe o valor de **d0** ou **d1** conforme o estado de **sel**.

Funcionamento:
O multiplexador 2x1 de 8 bits implementado utilizando uma estrutura processual:

- Quando **sel** é igual a '0', a saída **y** recebe o valor contido em **d0**.
- Caso contrário, se **sel** for '1', a saída **y** recebe o valor contido em **d1**.

Figura 2

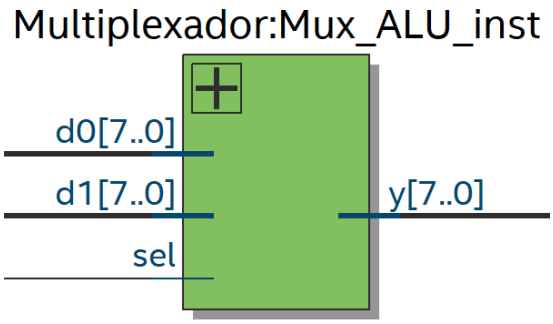
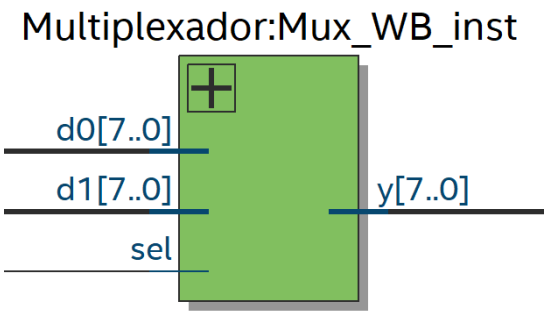


Figura 3



4.3 Contador de Programa – PC (PC.vhd)

O componente **PC** (Contador de Programa) tem como principal objetivo armazenar o endereço da instrução atual e fornecer o endereço para a próxima instrução, sincronizando essa operação com o sinal de clock.

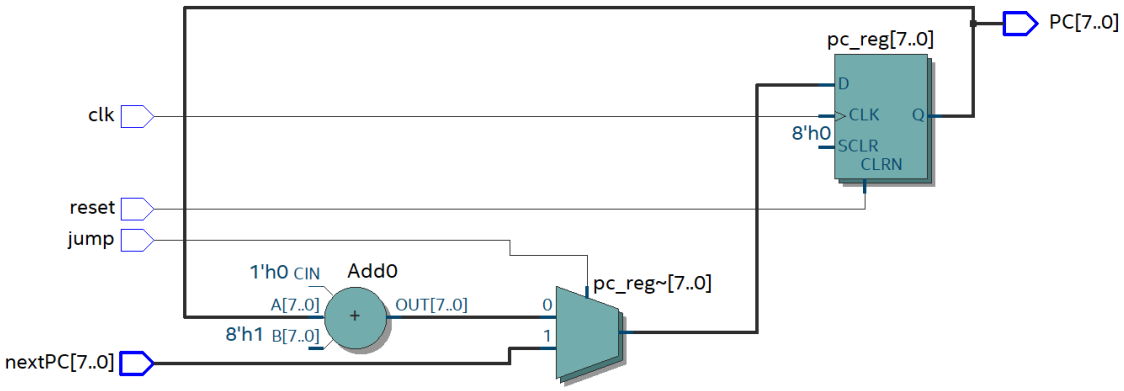
O elemento recebe como **entradas**:
• **clock** – Sinal de clock, que sincroniza a atualização do endereço;
• **enderecoDEentrada** – Vetor de 16 bits representando o endereço que deverá ser carregado.

E fornece como **saída**:
• **enderencoDEsaida** – Vetor de 16 bits que representa o endereço atual armazenado no PC.

Funcionamento:

O PC implementado conta com um processo sensível ao sinal de clock. Na borda de subida do clock (rising edge), o valor contido em **enderecoDEentrada** é transferido para **enderencoDEsaida**, atualizando assim o endereço do programa a ser executado.

Figura 4



4.4 Decodificador (Decodificador.vhd)

O componente Decodificador tem como principal objetivo interpretar a instrução de 8 bits e gerar os sinais de controle que determinam o comportamento dos demais módulos do processador.

Entrada:
• **instr** – Instrução de 8 bits.

E fornece como **saídas** os sinais de controle:
 RegWrite – Habilita escrita no registrador.
 ALUSrc – Seleciona a fonte da segunda entrada da ALU (registrador ou imediato).
 MemRead – Habilita leitura da memória (LW).
 MemWrite – Habilita escrita na memória (SW).
 Branch – Sinal de desvio condicional (BEQ).
 Jump – Sinal de salto incondicional (JUMP).
 ALUOp – Vetor de 2 bits para a operação na ALU (00 = ADD, 01 = SUB, 10 = AND, 11 = OR).

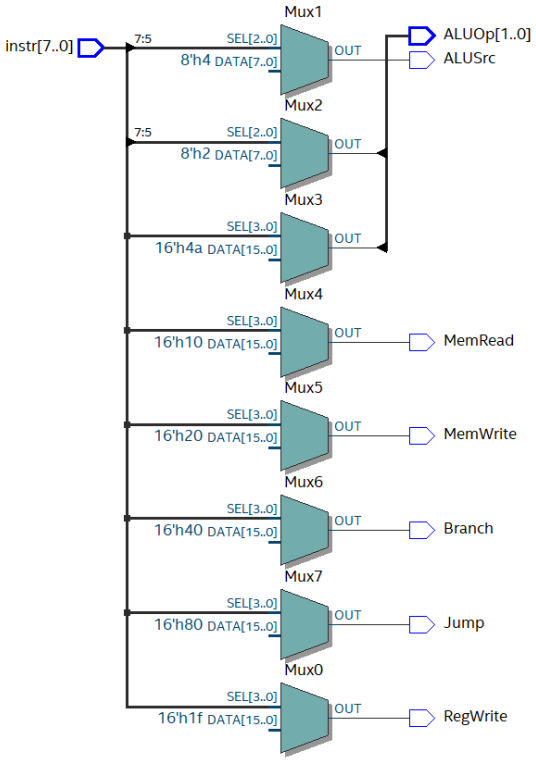
Funcionamento:

- O decodificador extrai o opcode dos 4 bits mais significativos da instrução (instr(7 downto 4)).
- Utilizando um processo sensível ao opcode, são atribuídos valores padrão para todos os sinais de controle, e em seguida, através de uma estrutura *case*, os sinais são configurados de acordo com o opcode:
    - Para a instrução **ADD** (opcode "0000"), ativa a escrita no registrador (**RegWrite** = '1'), utiliza o valor do registrador como segundo operando (**ALUSrc** = '0') e define a operação da ALU para adição (**ALUOp** = "00").
    - Para a instrução **SUB** (opcode "0001"), configura a operação da ALU para subtração (**ALUOp** = "01") e ativa a escrita no registrador.
    - Para as operações lógicas **AND** e **OR** (opcodes "0010" e "0011"), define os sinais de controle para execução da operação correspondente na ALU.
    - Para a instrução **LW** (opcode "0100"), ativa a leitura da memória (**MemRead** = '1') e configura a ALU para calcular o endereço (usando o imediato, com **ALUSrc** = '1' e **ALUOp** = "00").

- Para a instrução **SW** (opcode "0101"), ativa a escrita na memória (**MemWrite** = '1') e utiliza o imediato para o cálculo do endereço.
- Para a instrução **BEQ** (opcode "0110"), ativa o sinal de branch (**Branch** = '1') e configura a ALU para realizar uma subtração (comparação).
- Para a instrução **JUMP** (opcode "0111"), ativa o sinal de salto incondicional (**Jump** = '1').

- Caso o opcode não corresponda a nenhuma instrução conhecida, os sinais permanecem inativos, evitando comportamentos indesejados.

Figura 5



4.5 Memória RAM (RAM.vhd)

O componente **RAM** tem como principal objetivo armazenar dados temporariamente durante a execução do programa, possibilitando operações de leitura e escrita em um espaço de memória de 256 posições, onde cada posição armazena 8 bits.

O elemento recebe como **entradas**:
• **clk** – Sinal de clock, que sincroniza as operações de leitura e escrita;
• **we** – Sinal de habilitação de escrita (write enable). Quando '1', permite a escrita no endereço especificado;
• **addr** – Vetor de 8 bits que representa o endereço de acesso na memória;
• **din** – Vetor de 8 bits contendo o dado a ser escrito.

E fornece como **saída**:
• **dout** – Vetor de 8 bits com o dado lido da posição de memória especificada.

Funcionamento:

O módulo é implementado utilizando um processo sensível ao sinal de clock:

- Na borda de subida do clock (rising edge), se o sinal **we** estiver ativo ('1'), o valor contido em **din** é escrito na posição de memória correspondente ao endereço (convertido para inteiro).
- Ainda na borda de subida, o dado armazenado na posição indicada pelo endereço é atribuído à saída **dout**, garantindo que a leitura ocorra de forma síncrona.

Internamente, a RAM é modelada como um vetor de 256 posições (índices de 0 a 255), onde cada posição é um vetor de 8 bits. A utilização da conversão de tipos (de std_logic_vector para unsigned e depois para integer) permite o endereçamento correto durante as operações de leitura e escrita.

Figura 6

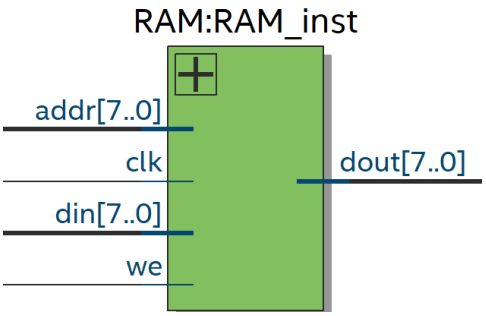
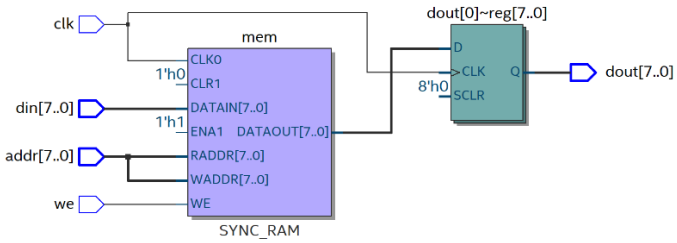


Figura 7



4.6 Banco de Registradores (Registradores.vhd)

O componente **Registradores** implementa um banco de 8 registradores, cada um com 8 bits, permitindo o armazenamento e a recuperação rápida dos dados utilizados durante a execução das operações do processador.

O elemento recebe como **entradas**:
• **clk** – Sinal de clock, que sincroniza as operações de escrita no banco de registradores;
• **regWrite** – Sinal que habilita a escrita, permitindo que os dados sejam gravados no registrador especificado;
• **readAddr1** – Endereço de leitura 1 (3 bits), que seleciona um dos 8 registradores para fornecer o dado;
• **readAddr2** – Endereço de leitura 2 (3 bits), que seleciona um segundo registrador para fornecer o dado;
• **writeAddr** – Endereço de escrita (3 bits), que determina em qual registrador os dados serão armazenados;
• **writeData** – Dados a serem escritos (8 bits).

E fornece como **saídas**:
• **readData1** – Dados lidos do registrador especificado por **readAddr1** (8 bits);
• **readData2** – Dados lidos do registrador especificado por **readAddr2** (8 bits).

Funcionamento:

- **Escrita:**
  Um processo sensível ao sinal de clock realiza a escrita no banco de registradores. Na borda de subida do clock, se o sinal **regWrite** estiver ativo ('1'), o valor presente em **writeData** é escrito no registrador identificado pelo endereço **writeAddr**. A conversão dos sinais de endereço de std_logic_vector para inteiro (por meio de unsigned e to_integer) permite a indexação correta do array de registradores.
- **Leitura:**
  As operações de leitura são implementadas de forma combinacional, onde os dados armazenados nos registradores apontados por **readAddr1** e **readAddr2** são encaminhados imediatamente para as saídas **readData1** e **readData2**, respectivamente, sem depender do clock.

Figura 8

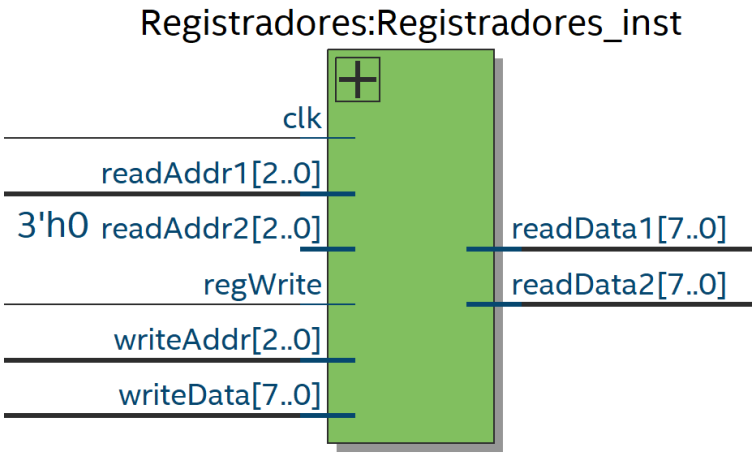
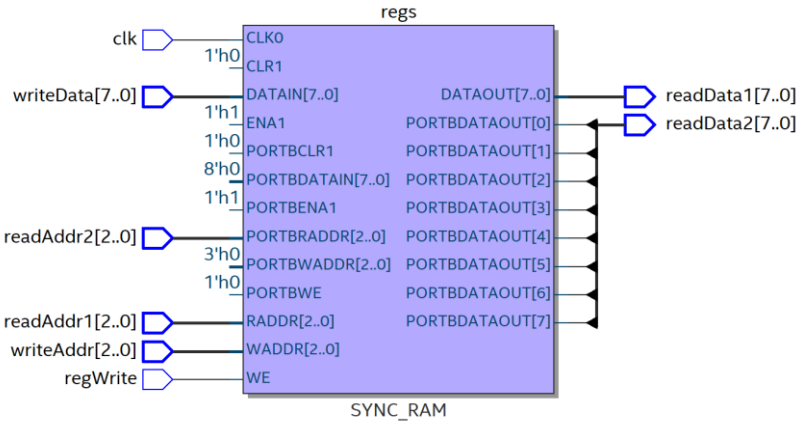


Figura 9



4.7 Memória ROM (ROM.vhd)

O componente **ROM** tem como principal objetivo armazenar permanentemente as instruções do programa, permitindo que o processador acesse e execute o código armazenado.

O elemento recebe como **entrada**:
• **addr** – Endereço de 8 bits (0 a 255) que indica a posição de memória a ser lida.

E fornece como **saída**:
• **instr** – Instrução de 8 bits correspondente ao endereço fornecido.

Funcionamento:

- **Armazenamento de Instruções:**
  A ROM é modelada como um array com 256 posições, onde cada posição armazena um vetor de 8 bits.
- **Inicialização:**
  O array constante ROM_Content é inicializado com os valores das instruções. No exemplo apresentado, as instruções correspondem a um teste do fatorial, com instruções de load imediato (LI), branch (BEQ), multiplicação (MULT), adição imediata (ADDI) e salto incondicional (JUMP).
- **Leitura Combinacional:**
  A instrução é fornecida de forma combinacional. A saída **instr** é atualizada indexando ROM_Content com o valor convertido do endereço (addr) para inteiro, utilizando as funções unsigned e to_integer.

Exemplo de operação:

- Se **addr** for "00000000" (0), a saída **instr** será "00010011", correspondente à instrução *LI S0, 3*.
- Se **addr** for "00000001" (1), a saída **instr** será "00010101", correspondente à instrução *LI S1, 1*.

Figura 10

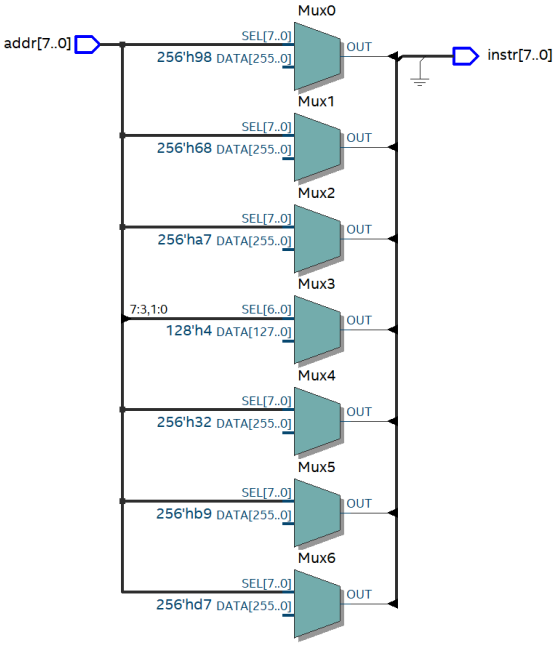
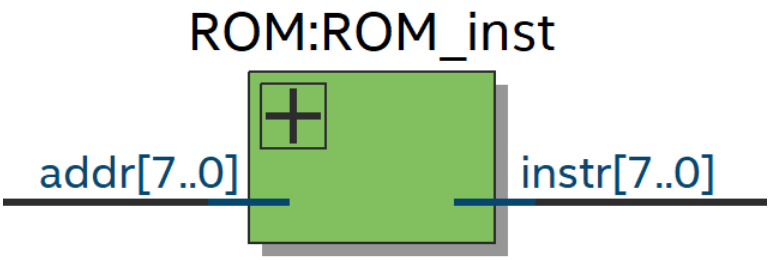


Figura 11



4.8 Somador (Somador.vhd)

O componente **Somador** tem como principal objetivo efetuar a soma aritmética de dois números de 8 bits, gerando uma saída de 8 bits para o resultado e um sinal de carry que indica overflow na operação.

O elemento recebe como entradas:
• **A** – Vetor de 8 bits representando o primeiro operando;
• **B** – Vetor de 8 bits representando o segundo operando.

E fornece como saídas:
• **Sum** – Vetor de 8 bits contendo o resultado da soma;
• **Carry** – Sinal de 1 bit que indica se ocorreu overflow (bit mais significativo do resultado de 9 bits).

Funcionamento:

- **Conversão para Unsigned:**
  As entradas **A** e **B** são convertidas para o tipo unsigned para permitir operações aritméticas, garantindo que a soma seja realizada corretamente em termos numéricos.
- **Realização da Soma:**
  Ao concatenar um bit '0' à esquerda de cada operando, forma-se um vetor de 9 bits, permitindo capturar o bit de overflow. A soma é então realizada, gerando um resultado de 9 bits.
- **Extração dos Resultados:**
  - A parte menos significativa (bits 7 downto 0) é atribuída a **Sum**.
  - O bit mais significativo (bit 8) é extraído e atribuído a **Carry**, indicando se houve overflow na soma.

Figura 12

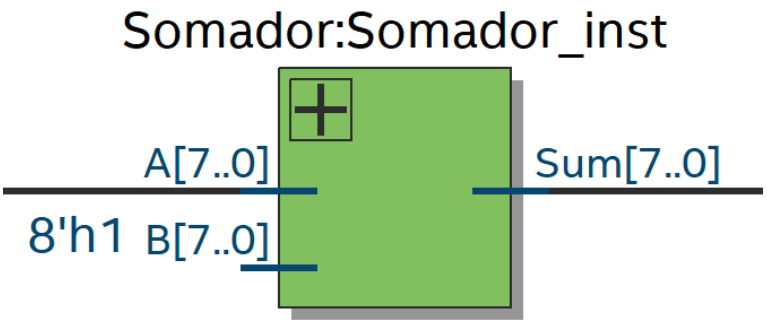
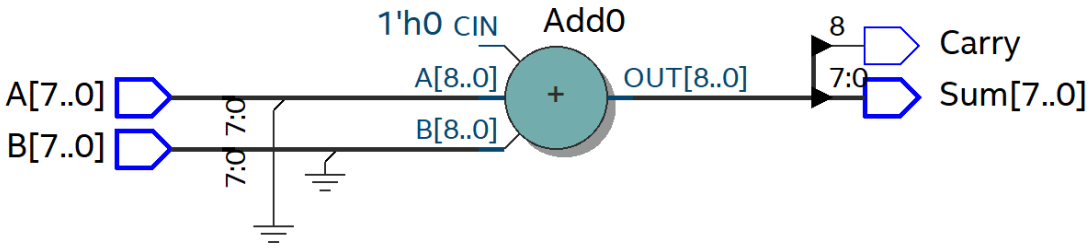


Figura 13



4.9 Unidade Lógica e Aritmética (ULA.vhd)

O componente ULA (Unidade Lógica e Aritmética) implementa as operações aritméticas e lógicas de um processador de 8 bits, possibilitando a realização de soma, subtração, AND e OR, bem como a sinalização de um resultado igual a zero.

O elemento recebe como entradas:

- **A** – Vetor de 8 bits representando o primeiro operando;
- **B** – Vetor de 8 bits representando o segundo operando;
- **AluOp** – Vetor de 2 bits, que determina qual operação será executada (00 = ADD, 01 = SUB, 10 = AND, 11 = OR).

E fornece como saídas:

- **result** – Vetor de 8 bits contendo o resultado da operação;
- **zero** – Sinal de 1 bit que indica se o resultado é igual a zero ('1' se for zero, caso contrário '0').

Funcionamento**:**

- **Conversão para unsigned:**
  As entradas A e B, do tipo std_logic_vector(7 downto 0), são convertidas para unsigned, permitindo a realização de operações aritméticas adequadas (soma e subtração).
- **Decodificação de AluOp:**
  O vetor de 2 bits AluOp define a operação a ser realizada na ULA:
  - "00": Soma (res <= unsigned(A) + unsigned(B))
  - "01": Subtração (res <= unsigned(A) - unsigned(B))
  - "10": AND bit a bit (res <= unsigned(A and B))
  - "11": OR bit a bit (res <= unsigned(A or B))
- **Formação do resultado:**
  O sinal interno res (tipo unsigned(7 downto 0)) é convertido de volta para std_logic_vector(7 downto 0), atribuindo esse valor à saída result.
- **Detecção de zero:**
  O sinal zero recebe '1' quando res = 0, indicando que o resultado final da operação é zero; caso contrário, recebe '0'.

Figura 14

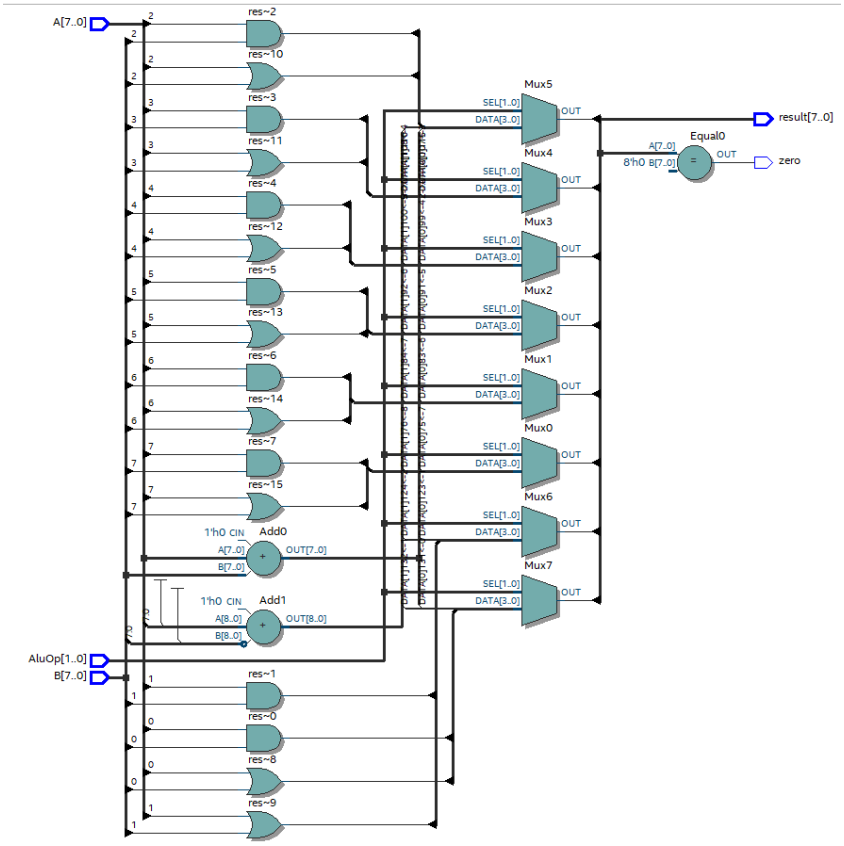
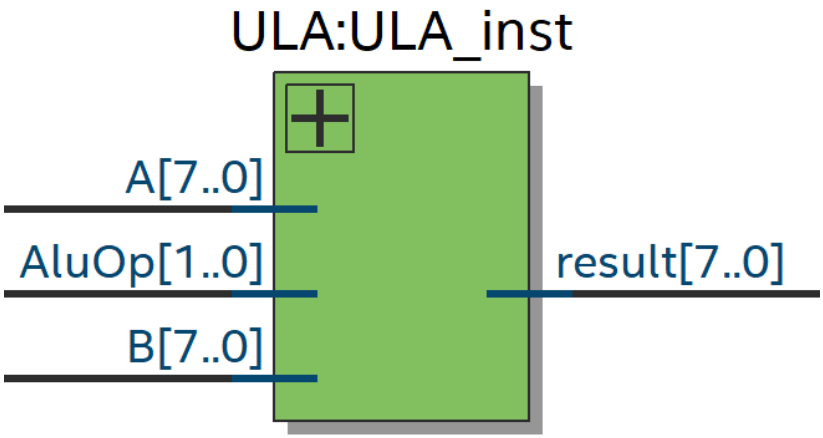


Figura 15



4.10 Unidade de Controle (Unidade_de_controle.vhd)

O componente Unidade de Controle tem como principal objetivo decodificar o campo de operação (opcode) e gerar os sinais de controle necessários para coordenar as demais unidades do processador. Assim, ele define como a ULA e outros blocos (memória, registradores, etc.) devem se comportar de acordo com cada instrução.

A unidade de controle recebe como **entradas:**

- **opcode** – Vetor de 4 bits que representa a instrução a ser executada (contendo os 4 bits mais significativos da palavra de instrução).

E fornece como **saídas:**

- **RegWrite** – Habilita ('1') ou não ('0') a escrita no Banco de Registradores.

- **ALUSrc** – Seleciona se a segunda entrada da ULA vem de um registrador ('0') ou de um imediato ('1').
- **MemRead** – Habilita ('1') a leitura da memória (usado em instruções como LW).
- **MemWrite** – Habilita ('1') a escrita na memória (usado em instruções como SW).
- **Branch** – Indica ('1') instrução de desvio condicional (BEQ).
- **Jump** – Indica ('1') instrução de desvio incondicional (JUMP).
- **MemToReg** – Seleciona se o dado que será escrito no registrador vem da ULA ('0') ou da memória ('1').
- **ALUOp** – Vetor de 2 bits que define a operação na ULA (00 = ADD, 01 = SUB, 10 = AND, 11 = OR).

Funcionamento:

- **Inicialização dos sinais**: Antes de analisar o opcode, todos os sinais são definidos com um valor padrão (normalmente desativados) e ALUOp é ajustado para "00" (ADD), caso a instrução não corresponda a nenhum caso específico.
- **Decodificação do opcode**:
    - **"0000" (ADD)**: Habilita escrita em registrador (RegWrite = '1'), seleciona operando do registrador (ALUSrc = '0'), envia o resultado da ULA diretamente para o registrador (MemToReg = '0') e define a ULA como soma (ALUOp = "00").
    - **"0001" (SUB)**: Semelhante ao ADD, mas muda a operação da ULA para subtração (ALUOp = "01").
    - **"0010" (AND)**: Habilita escrita no registrador e ajusta a ULA para operação AND (ALUOp = "10").
    - **"0011" (OR)**: Habilita escrita no registrador e ajusta a ULA para operação OR (ALUOp = "11").
    - **"0100" (LW - Load Word)**: Habilita escrita em registrador (RegWrite = '1'), ativa leitura de memória (MemRead = '1'), seleciona imediato para cálculo de endereço (ALUSrc = '1'), recebe dado da memória no registrador (MemToReg = '1') e utiliza soma na ULA para calcular o endereço (ALUOp = "00").
    - **"0101" (SW - Store Word)**: Desabilita escrita em registrador, ativa escrita na memória (MemWrite = '1'), usa imediato para endereço (ALUSrc = '1') e soma na ULA (ALUOp = "00").
    - **"0110" (BEQ - Branch if Equal)**: Desabilita escrita em registrador, marca instrução de desvio (Branch = '1') e define operação de subtração na ULA para comparar os valores (ALUOp = "01").
    - **"0111" (JUMP)**: Desabilita escrita em registrador e habilita salto incondicional (Jump = '1').

  Caso o opcode não corresponda a nenhum desses padrões, mantém os sinais em seus valores padrão, sem realizar nenhuma ação específica.

Dessa forma, a **Unidade de Controle** é responsável por interpretar o opcode e direcionar o fluxo de dados e as operações no processador, viabilizando a correta execução das instruções.

Figura 16

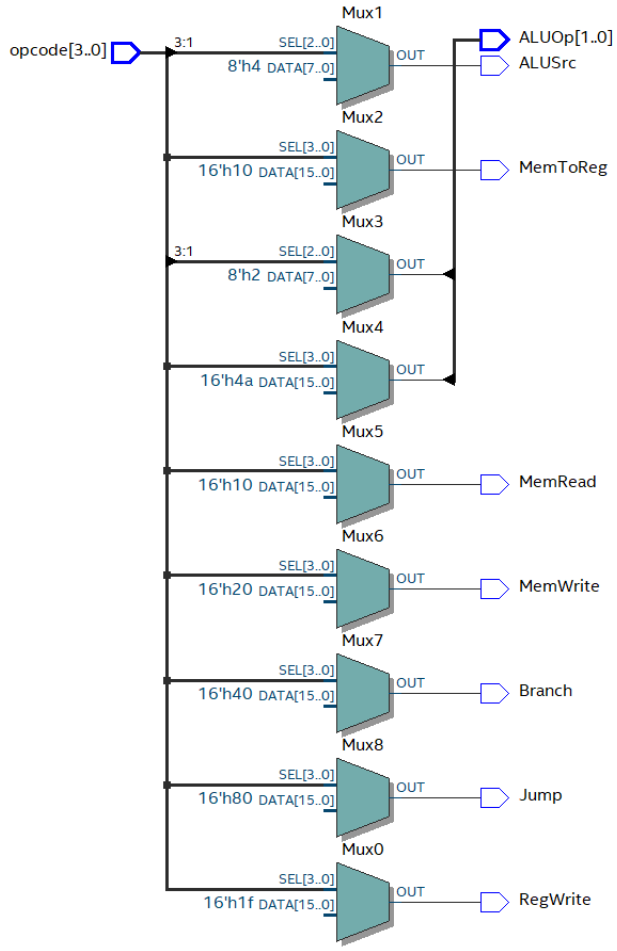
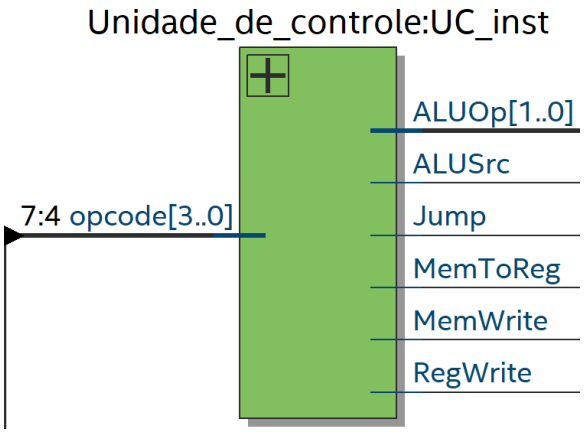


Figura 17



4.11 Processador (Processador.vhd)

O componente Processador tem como principal objetivo integrar todos os blocos de um sistema RISK de 8 bits, coordenando as operações de busca de instrução, decodificação, execução e acesso à memória para realizar cada instrução do conjunto definido.

O Processador recebe como **entradas:**

- **Clock** – Fornece o sinal de clock que sincroniza as operações de todos os componentes internos (contadores, registradores, memória, etc.).
- **Reset** – Sinal de reinicialização, utilizado para colocar o processador em um estado inicial (geralmente zerando o Contador de Programa, registradores, etc.).

Saídas: Esse processador simples fornece saídas especificas.

Funcionamento:

- **Contador de Programa (PC)**:
  É atualizado a cada ciclo de clock, podendo simplesmente incrementar em 1 (via Somador) ou receber um novo valor em casos de salto (Jump). O **ContadorPrograma** recebe Clock e Reset para, na borda ativa do clock, atualizar o PC com PC_next ou retornar ao estado inicial se Reset estiver ativo.
- **Busca da Instrução (ROM)**:
  O valor atual do PC endereça a memória de instruções (ROM). O conteúdo lido em Instr contém tanto o opcode (bits 7 a 4) quanto o imediato (bits 3 a 0).
- **Decodificação (Unidade de Controle)**:
  O opcode é encaminhado à **Unidade_de_controle**, que gera sinais como RegWrite, ALUSrc, MemRead, MemWrite, Branch, Jump, MemToReg e ALUOp, configurando como as outras unidades devem operar para cada instrução.
- **Banco de Registradores**:
  Utiliza RegWrite para decidir quando escrever dados em um registrador. Os endereços de leitura e escrita são derivados de campos na instrução (Instr). Os dados lidos são retornados em RegData1 e RegData2.
- **Extensão de Imediato**:
  O bloco **Extensor_4x8** converte o campo de 4 bits (Imm4) em 8 bits (Imm8), possibilitando instruções que usem constantes ou endereços mais extensos.
- **Seleção da Entrada B da ULA (MUX)**:
  O sinal ALUSrc determina se a segunda entrada da ULA (ALUB_in) será RegData2 (registrador) ou Imm8 (imediato).
- **ULA (Unidade Lógica e Aritmética)**:
  Recebe RegData1 e ALUB_in, bem como ALUOp para definir a operação (ADD, SUB, AND, OR). Entrega o resultado em ALUResult e gera um sinal Zero que indica se o resultado é nulo (utilizado, por exemplo, em instruções de desvio condicional).
- **Memória de Dados (RAM)**:
  Quando a instrução requer acesso à memória (LW ou SW), ALUResult é usado como endereço. O sinal MemWrite habilita a escrita (SW) e MemRead habilita a leitura (LW), cujo valor retornado fica em MemReadData.
- **Escrita de Volta ao Banco de Registradores**:
  O sinal MemToReg seleciona se o valor a ser escrito em um registrador é o ALUResult (para operações aritmético-lógicas) ou MemReadData (para instruções de leitura de memória). Esse valor é multiplexado em WriteData.

- **Atualização do PC**:
  Um **Somador** gera PC_next ao adicionar 1 a PC. Se houver salto (Jump), o **ContadorPrograma** encarrega-se de atualizar o PC para o endereço desejado. Caso contrário, o PC apenas incrementa sequencialmente para a próxima instrução.

Assim, o **Processador** integra todos esses módulos, realizando o fluxo básico de instrução: **buscar, decodificar, executar e acessar memória**, coordenando internamente por meio dos sinais de controle gerados e das conexões entre cada componente.

Figura 18

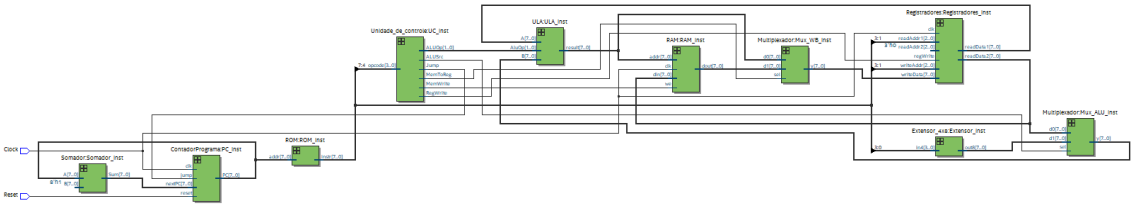


Figura 19

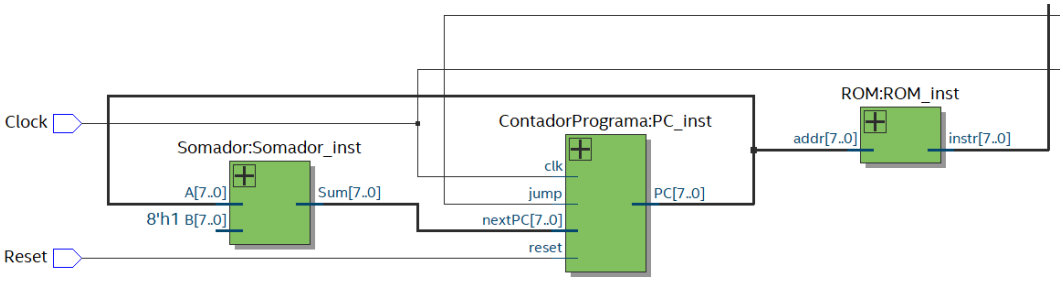


Figura 20

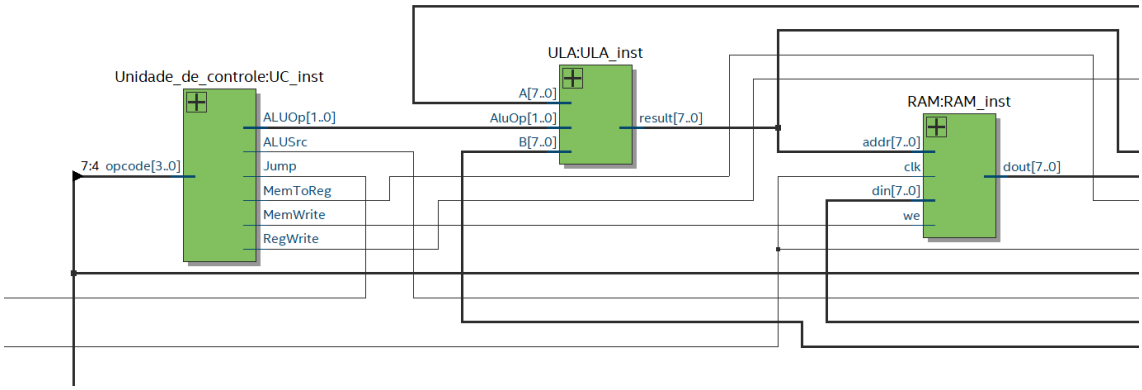
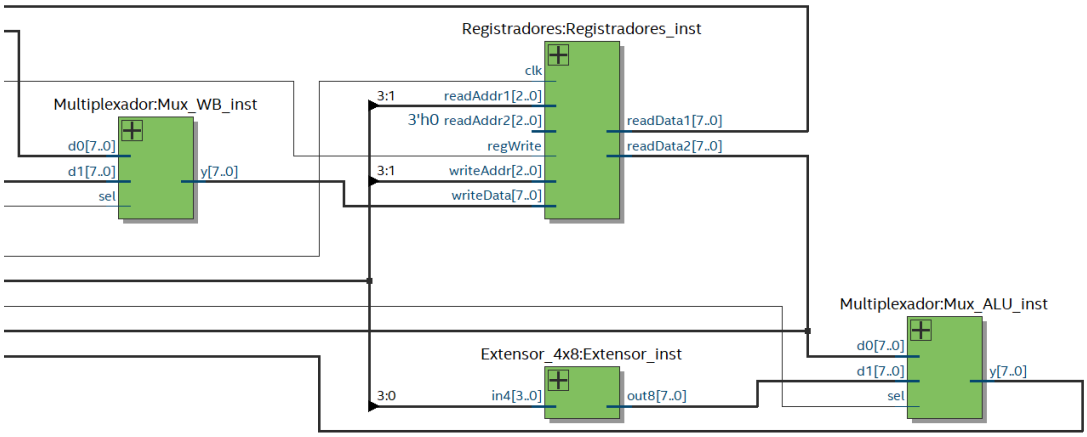


Figura 21



5. Simulações e Testes

Realizaram-se testes unitários para cada módulo e simulações integradas, a fim de verificar comunicação e fluxo de dados entre componentes. Utilizando o ModelSim, observou-se:

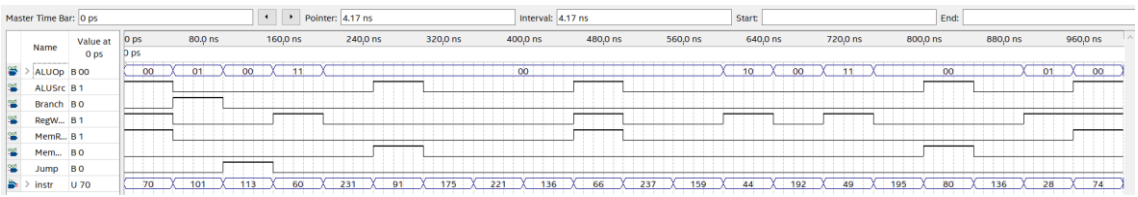
- **Decodificação** correta das instruções (Decodificador/Unidade de Controle).
- **Encaminhamento** apropriado dos sinais de controle.
- **Execução** das operações aritméticas/lógicas pela ULA e retorno do resultado.
- **Atualização do PC** e uso das memórias (RAM e ROM).
- **Gerenciamento** de dados pelo banco de registradores.

Exemplo de Sequência de Teste

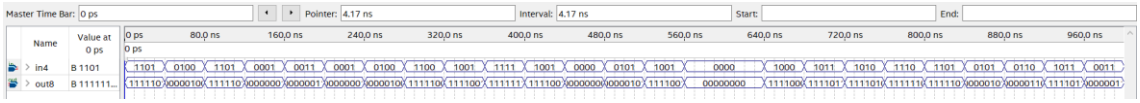
Endereço	Instrução em Linguagem	Código Binário	Observação
0	ADDi R0, 0	0001 0000	Inicializa R0 com 0
1	ADDi R1, 1	0001 0001	Inicializa R1 com 1
2	ADD R0, R0, R1	0000 0001	Soma: R0 = R0 + R1
3	SW R0, [endereço]	0101 0000	Armazena o resultado na RAM
4	J 0	0110 0000	Salto incondicional para 0

A simulação (waveform) mostrou o funcionamento sequencial dessas instruções, confirmando a **sintaxe** e **semântica** esperadas.

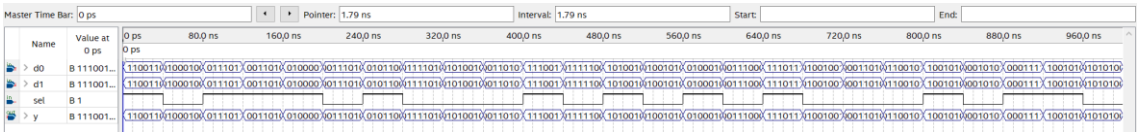
DECODIFICADOR



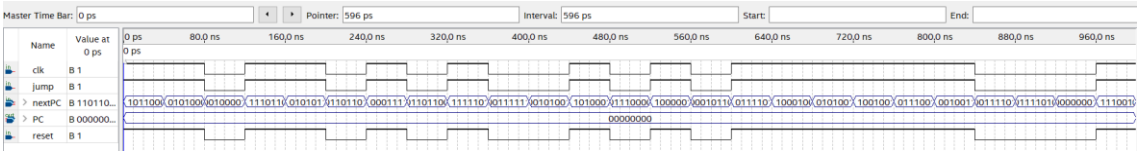
EXTENSOR_4x8



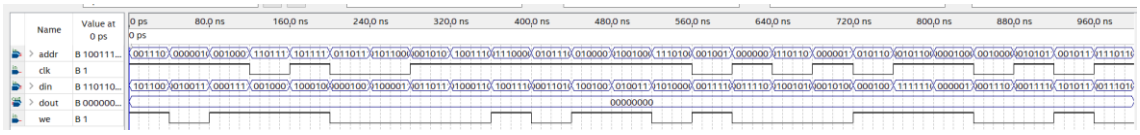
MULTIPLEXADOR



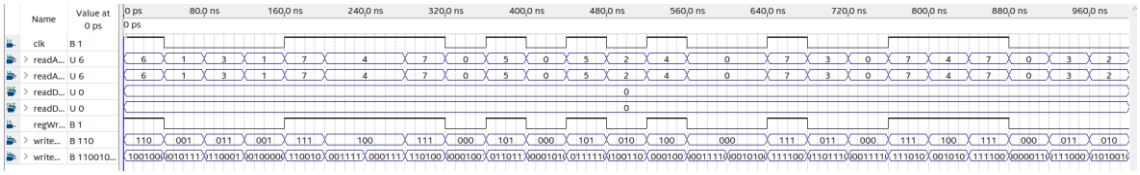
PC



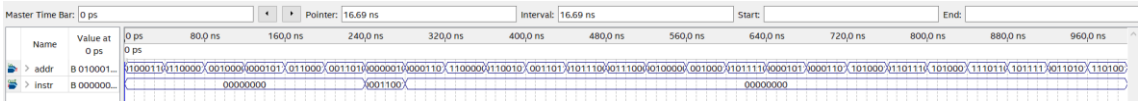
RAM



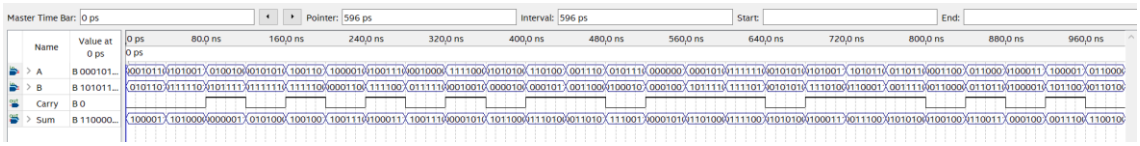
REGISTRADOR



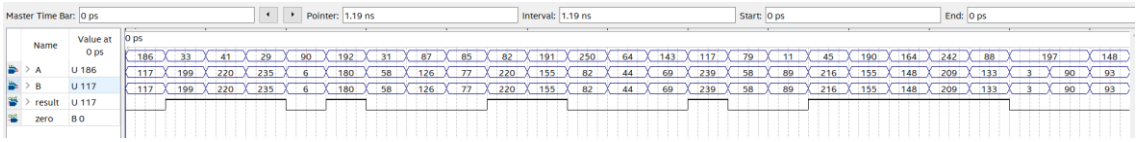
ROM



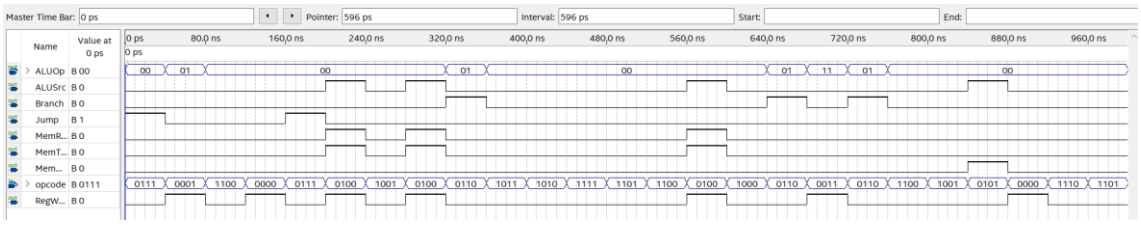
SOMADOR



ULA



UNIDADE DE CONTROLE



6. Considerações Finais

Este trabalho apresentou o **Processador RISK8**, um processador RISC de 8 bits desenvolvido em VHDL. Descreveram-se os componentes fundamentais (extensão de sinal, gerenciamento de fluxo de dados, execução aritmética, etc.) e a integração de todos os módulos por meio do arquivo *top-level* (**Processador.vhd**).

Desafios e Adaptações:

- **Compactar** a arquitetura para 8 bits, o que impõe restrições na manipulação de dados e sinalização.
- Implementar uma **Unidade de Controle** simples, mas funcional, com recursos limitados.
- Garantir **compatibilidade** de sinais entre módulos, considerando diferentes larguras e conversões de tipo (unsigned, std_logic_vector).

Apesar dessas limitações, o **RISK8** demonstrou-se operacional e eficiente para o conjunto reduzido de instruções proposto. A experiência possibilitou uma compreensão aprofundada de **design digital**, uso de **VHDL** e **integração de circuitos** no Quartus Prime Lite.

Referências

https://github.com/VictorH456/AOC_3VictorC-RyanKEGiovana_UFRR_2023/tree/main

Patterson, David Organização e projeto de computadores / David Patterson, John L. Hennessy. - 5. ed. - Rio de Janeiro : Elsevier, 2017.

https://youtube.com/playlist?list=PLYE3wKnWQbHDdnb3FsDkNx2tj8xoQAPtN&si=i-XlVHxUsfUtZrO5

Widmer. Neal S Sistemas digitas : princípios e aplicações I Neal S. Widmer, Gregory I. Moss. Rcrald J. Toco; (tradução Sérgio NasomentoJ - São Paulo: Pearson Education do Bras I. 2018. Titulo original: Digital systems: principles and applications 12. ed. americana. IS8N 978-85-430-2501 -3 1. Circuitos lógicos 2. Computadores 3. Eletrônica digital I. Moss. Gregory I. II. Tocci. Ronald J. II. Titulo.