UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DCC301 – ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES – 2025

PROF. DR. HEBERT OLIVEIRA ROCHA

HELIAN VINCIUS FILINTO DA SILVA

SALVADOR DE JESUS MALAVÉ

LUCAS GUILHERME PEREIRA SANTIAGO

PROCESSADOR RISC DE 8 BITS

BOA VISTA

RR 2025

HELIAN VINCIUS FILINTO DA SILVA

SALVADOR DE JESUS MALAVÉ

LUCAS GUILHERME PEREIRA SANTIAGO

PROCESSADOR RISC DE 8 BITS

Trabalho da disciplina de Arquitetura e Organização de Computadores do ano de 2025 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 descreve o desenvolvimento de um processador RISC de 8 bits, projetado e implementado utilizando a linguagem de descrição de hardware VHDL no ambiente do software Intel Quartus Prime. A arquitetura adotada é baseada nos princípios do modelo MIPS e tem como finalidade consolidar, por meio de uma abordagem prática, os conceitos fundamentais de arquitetura e organização de computadores, circuitos digitais e projeto de hardware digital.

O conjunto de instruções implementadas contempla operações essenciais, como **add**, **sub**, **load**, **store**, **beq (branch if equal)** e **salto incondicional**, assegurando a funcionalidade necessária para a execução do programa no modelo básico de um processador RISC. O desenvolvimento do processador compreende a modelagem e a implementação de seus principais módulos, tais como a unidade de controle, o caminho de dados e os barramentos responsáveis pela comunicação entre os componentes internos.

Durante o desenvolvimento, foram consideradas restrições típicas de um processador de 8 bits, como o tamanho reduzido dos registradores, da memória e do conjunto de instruções, exigindo decisões de projeto voltadas à simplicidade e eficiência do hardware. Essas escolhas refletem princípios fundamentais da filosofia RISC, como a execução de instruções simples em poucos ciclos e a separação clara entre operações de acesso à memória e operações aritmético-lógicas.

2. CONJUNTO DE INSTRUÇÕES

O processador RISC de 8 bits implementa um conjunto de instruções reduzido, estruturado de acordo com os princípios da arquitetura RISC. As instruções são organizadas em três formatos principais: **Tipo R**, **Tipo I** e **Tipo J**, cada um definido conforme a natureza da operação realizada. Todas as instruções possuem comprimento fixo de 8 bits e seguem um padrão de codificação baseado em *opcode* e campos de operandos, garantindo simplicidade de decodificação e execução em hardware.

2.2. Instruções do tipo R

As instruções do Tipo R são destinadas à execução de operações aritméticas e lógicas entre registradores. Esse formato é utilizado quando todos os operandos envolvidos na operação estão armazenados internamente no banco de registradores. A estrutura da instrução é composta por um campo de *opcode* e dois campos que identificam os registradores da fonte.

Esse formato permite a realização de operações como soma e subtração diretamente entre registradores, reforçando o princípio RISC de operações simples e eficientes.

O formato das instruções do Tipo R é definido conforme a seguir:

- **Opcode**: bits 7 a 5 (3 bits)
- **rs**: bits 4 a 3 (2 bits)
- **rt**: bits 2 a 1 (2 bits)
- **bit 0**: não utilizado (reservado)

2.2. Instruções do tipo I

As instruções do Tipo I são utilizadas em operações que envolvem valores imediatos ou acesso à memória. Esse formato é empregado tanto para instruções de carga e armazenamento quanto para instruções que utilizam constantes embutidas na própria instrução. O campo immediate permite a representação de constantes ou endereços de memória de forma compacta, viabilizando instruções como load word e store word.

As instruções do Tipo I seguem o formato abaixo:

- **Opcode**: bits 7 a 5 (3 bits)
- **rs**: bits 4 a 3 (2 bits)
- **Immediate**: bits 2 a 0 (3 bits)

O campo *immediate* permite a representação de constantes ou endereços de memória de forma compacta, viabilizando instruções como *load word* e *store word*.

2.3. Instruções do tipo J

As instruções do Tipo J (*Jump*) são responsáveis pelo controle de fluxo do programa, realizando saltos incondicionais ou condicionais por meio da modificação direta do contador de programa (*Program Counter – PC*). Esse formato é utilizado quando o endereço de destino do salto está embutido na própria instrução.

O formato das instruções do Tipo J é definido da seguinte forma:

- **Opcode**: bits 7 a 5 (3 bits)
- **Address**: bits 4 a 0 (5 bits)

Esse formato possibilita a alteração do fluxo sequencial de execução, permitindo a implementação de estruturas de controle como desvios e laços.

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	jump	J	Jump Incondicional

O conjunto de instruções apresentado foi definido de modo a garantir a funcionalidade mínima necessária para a execução de programas simples, mantendo a coerência com a filosofia RISC e respeitando as limitações inerentes a um processador de 8 bits.

3. COMPONENTES DO PROCESSADOR

3.1. PC

O Contador de Programa (*Program Counter – PC*) é o componente responsável por armazenar e fornecer o endereço da próxima instrução do programa a ser executada. Seu funcionamento é baseado em um registrador paralelo síncrono, cuja atualização ocorre de acordo com o sinal de clock e o sinal de reset.

A cada ciclo de clock, o PC atualiza seu valor de saída com base no endereço de entrada fornecido, permitindo o avanço sequencial do fluxo de execução ou a realização de desvios e saltos, conforme definido pela lógica de controle do processador.

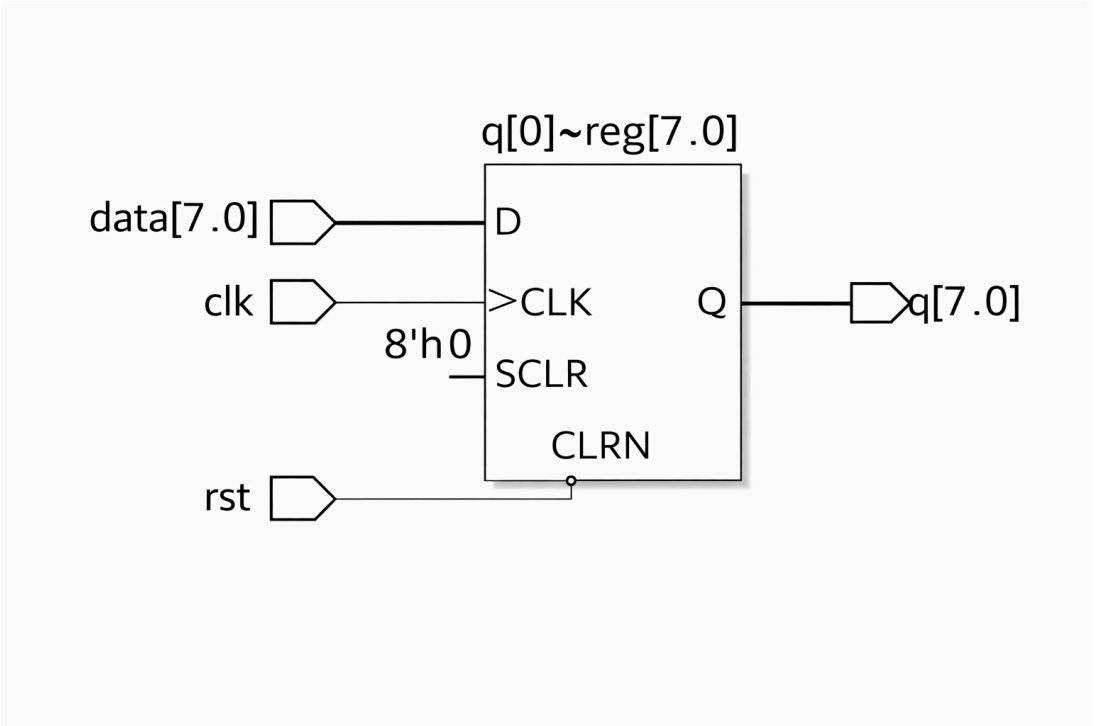
O componente PC apresenta as seguintes entradas:

- **CLOCK**: sinal de clock de 1 bit, responsável por sincronizar a atualização do valor armazenado no Contador de Programa.
- **RESET**: sinal de reset de 1 bit, ativo em nível lógico baixo (‘0’), que, quando acionado, força o PC a assumir o valor zero.
- **ADDRESS_IN**: sinal de dados de 8 bits (1 byte) que representa o novo endereço a ser carregado no Contador de Programa.

O componente PC disponibiliza a seguinte saída:

- **ADDRESS_OUT**: sinal de dados de 8 bits (1 byte) que corresponde ao endereço atualmente armazenado no Contador de Programa, utilizado para a busca da instrução corrente na memória de instruções.

Figura 1 - PC viewer



3.2. MEMÓRIA DE INSTRUÇÕES

O componente Memória de Instruções é responsável pelo armazenamento das instruções que compõem o programa a ser executado pelo processador, bem como pelo fornecimento da instrução correspondente ao endereço solicitado. Esse módulo opera como uma memória somente de leitura, na qual o endereço de entrada é utilizado para acessar a instrução previamente armazenada na posição indicada.

Durante o ciclo de busca de instruções, o endereço fornecido pelo Contador de Programa é aplicado à memória, que retorna a instrução associada àquele endereço, permitindo sua posterior decodificação e execução pelo processador.

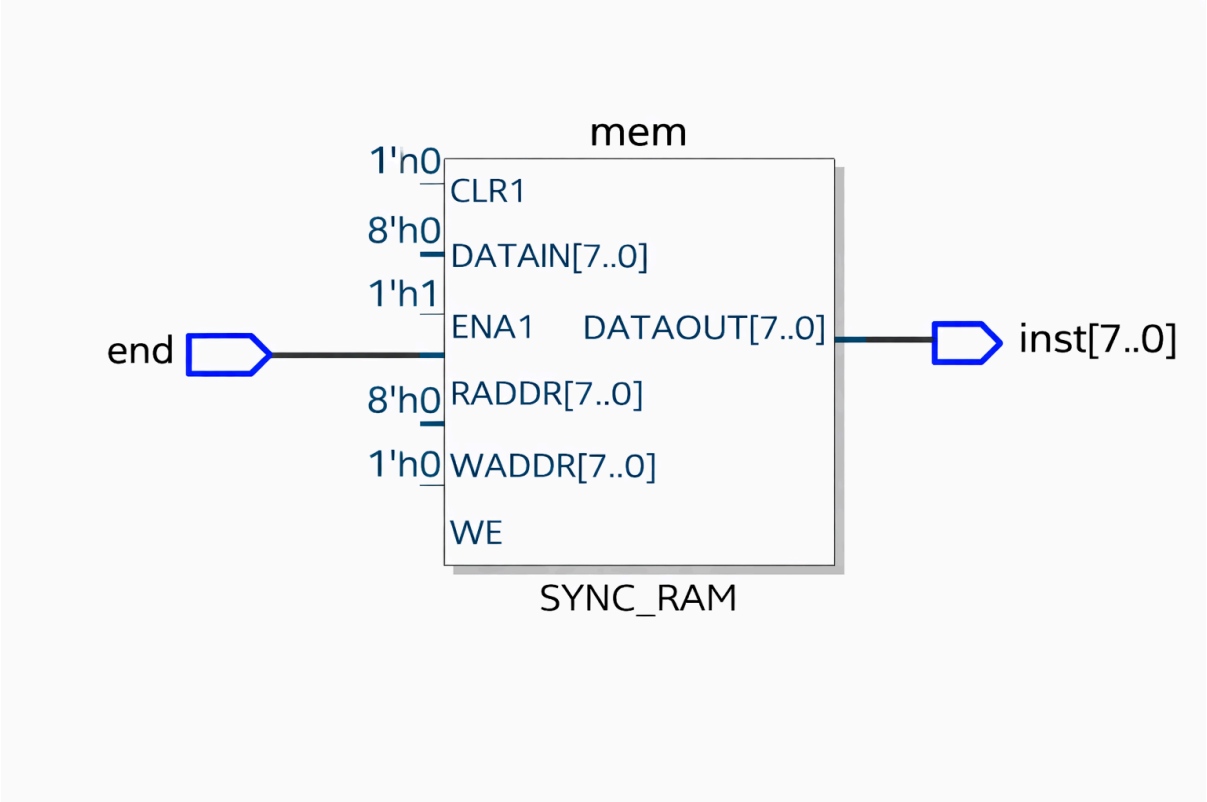
O componente **Memória de Instrução** possui a seguinte entrada:

- **Endereço**: sinal de dados de 8 bits (1 byte) que representa o endereço da instrução a ser acessada na memória.

Como saída, o componente disponibiliza:

- **Instrução**: sinal de dados de 8 bits (1 byte) correspondente à instrução armazenada no endereço solicitado.

Figura 2 - Memória de instruções



3.3. BANCO DE REGISTRADORES

O componente Banco de Registradores é responsável por armazenar e gerenciar um conjunto de registradores internos utilizados durante a execução dos programas. Esse módulo desempenha um papel fundamental na arquitetura do processador, permitindo o armazenamento temporário de dados e a transferência eficiente de valores entre as diversas unidades funcionais do sistema.

O componente **Banco de Registradores** possui as seguintes entradas:

- **CLOCK**: sinal de clock de 1 bit, responsável por sincronizar as operações de escrita no banco de registradores.

- **REG_WRITE**: sinal de controle de 1 bit que indica a habilitação da operação de escrita. Quando ativo em nível lógico alto (‘1’), o valor presente em **WRITE_DATA** é armazenado no registrador endereçado por REG1_IN.

- **REG1_IN**: sinal de entrada de 2 bits que especifica o endereço do registrador a ser acessado para leitura ou escrita.

- **REG2_IN**: sinal de entrada de 2 bits que especifica o endereço do segundo registrador a ser acessado para leitura.

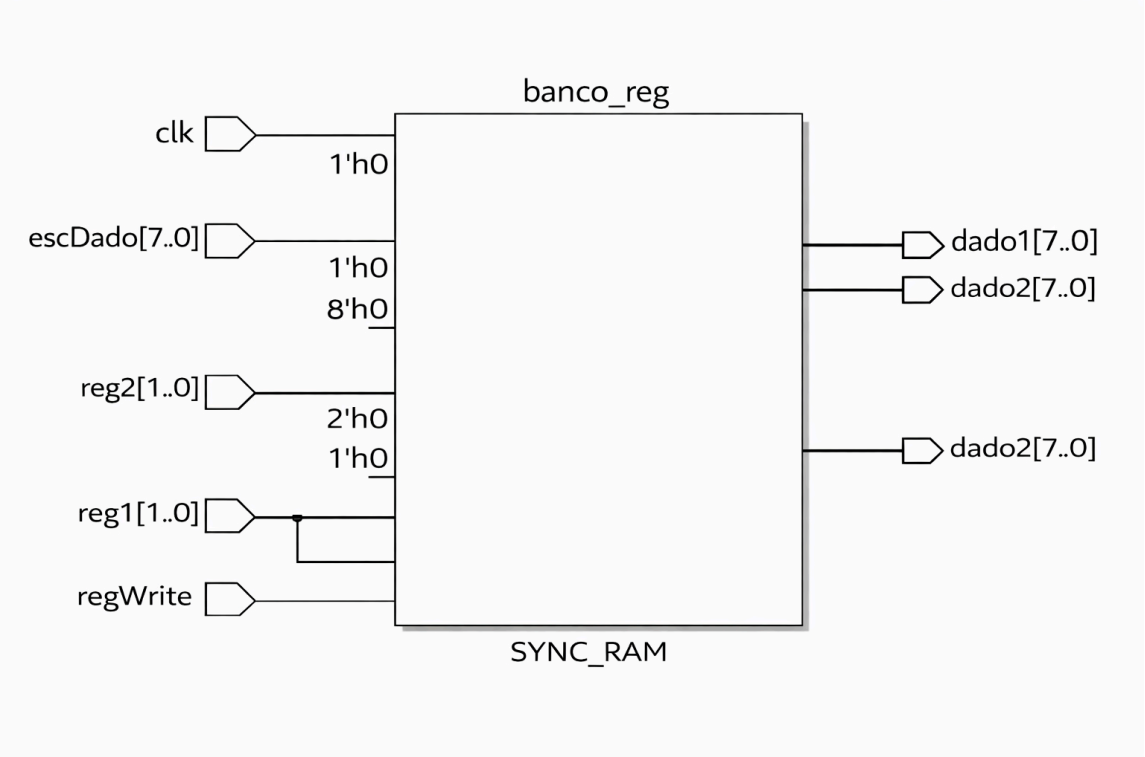
- **WRITE_DATA**: sinal de dados de 8 bits que contém o valor a ser escrito no registrador selecionado por REG1_IN, quando a escrita está habilitada.

Como saídas, o componente disponibiliza:

- **REG1_OUT**: sinal de dados de 8 bits correspondente ao valor armazenado no registrador endereçado por REG1_IN.

- **REG2_OUT**: sinal de dados de 8 bits correspondente ao valor armazenado no registrador endereçado por REG2_IN.

Figura 3 – Banco de Registradores



3.4. UNIDADE DE CONTROLE

O componente Unidade de Controle é responsável pela decodificação do *opcode* das instruções e pela geração dos sinais de controle necessários para sua correta execução no processador. Esse módulo exerce papel central na arquitetura, coordenando o funcionamento dos principais blocos do sistema, como a Unidade Lógica e Aritmética (ULA), o banco de registradores e os módulos de memória.

A partir da instrução recebida, a Unidade de Controle determina quais operações devem ser realizadas em cada ciclo, habilitando ou desabilitando os componentes conforme o tipo de instrução em execução, garantindo a correta sincronização do fluxo de dados no *datapath*.

O componente **Unidade de Controle** possui a seguinte entrada:

- **opCode**: sinal de dados de 3 bits que representa o código da operação da instrução a ser executada.

Como saídas, a Unidade de Controle gera os seguintes sinais de controle:

- **Jump**: sinal de controle de 1 bit que indica a execução de uma instrução de salto incondicional (*jump*).

- **Branch**: sinal de controle de 1 bit que indica a execução de uma instrução de desvio condicional (*branch*).

- **MemRead**: sinal de controle de 1 bit que habilita a leitura da memória.

- **MemtoReg**: sinal de controle de 1 bit que define se o dado a ser escrito no banco de registradores deve ser proveniente da memória.

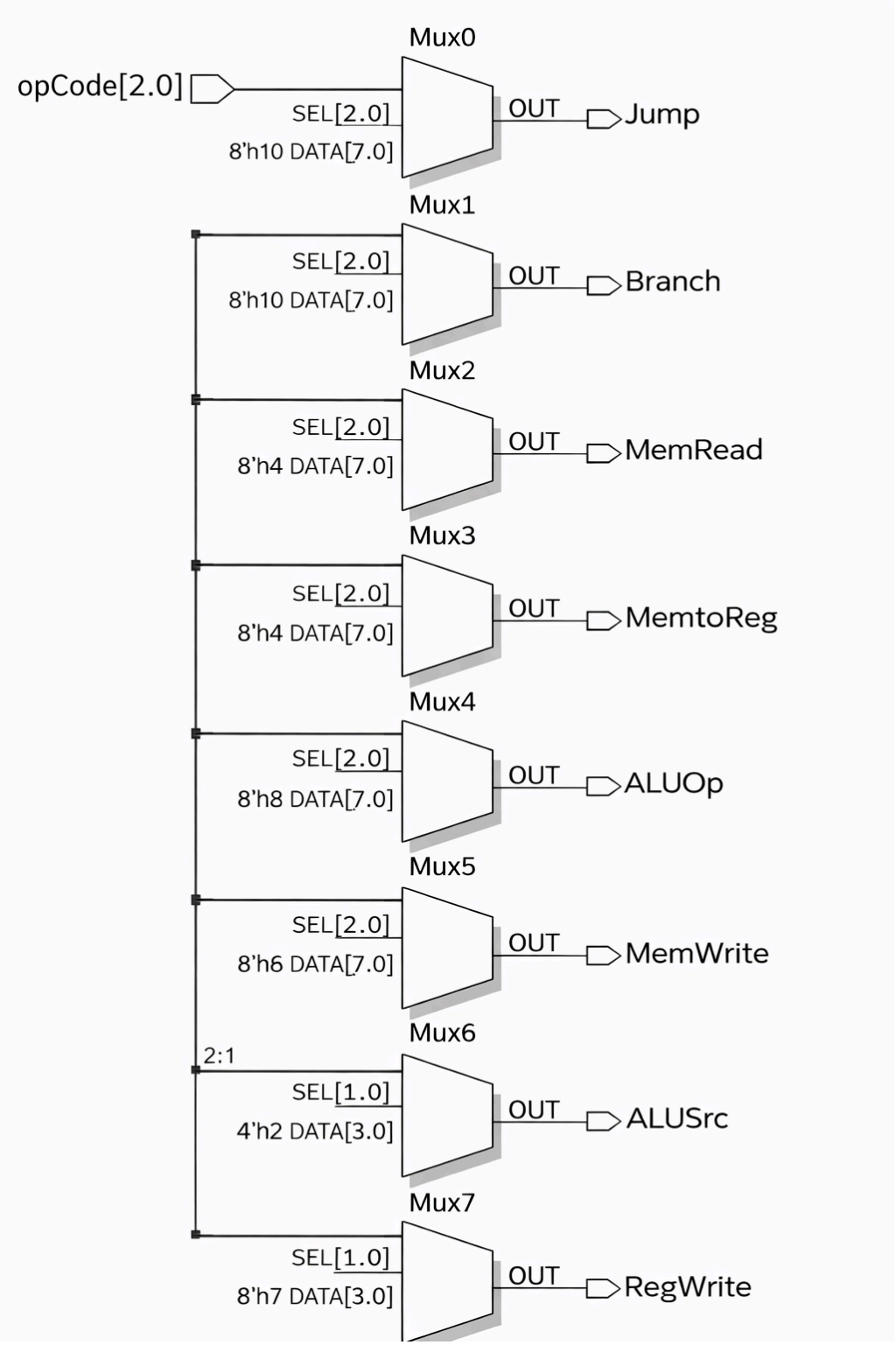
- **ALUOp**: sinal de controle de 1 bit que especifica a operação a ser realizada pela Unidade Lógica e Aritmética.

- **MemWrite**: sinal de controle de 1 bit que habilita a escrita na memória.

- **ALUSrc**: sinal de controle de 1 bit que seleciona a origem do segundo operando da ULA, podendo ser um valor proveniente de um registrador ou um valor imediato.

- **RegWrite**: sinal de controle de 1 bit que habilita a escrita no banco de registradores.

Figura 4 - Unidade de Controle



3.5. Memória de Dados

O componente Memória de Dados é responsável pelo armazenamento e gerenciamento das informações manipuladas durante a execução dos programas, sendo implementado como uma memória de acesso aleatório (RAM). Esse módulo possibilita operações de leitura e escrita em endereços específicos, conforme os sinais de controle gerados pela Unidade de Controle do processador.

As operações de escrita são sincronizadas pelo sinal de clock, enquanto as operações de leitura ocorrem de acordo com a habilitação do sinal correspondente, permitindo a correta interação entre a memória de dados e os demais componentes do *datapath*.

O componente **Memória de Dados** possui as seguintes entradas:

- **endereco**: sinal de dados de 8 bits que especifica o endereço da memória no qual será realizada a operação de leitura ou escrita.

- **escDado**: sinal de dados de 8 bits que contém o valor a ser escrito na memória quando a operação de escrita estiver habilitada.

- **memWrite**: sinal de controle de 1 bit que indica a habilitação da operação de escrita na memória.

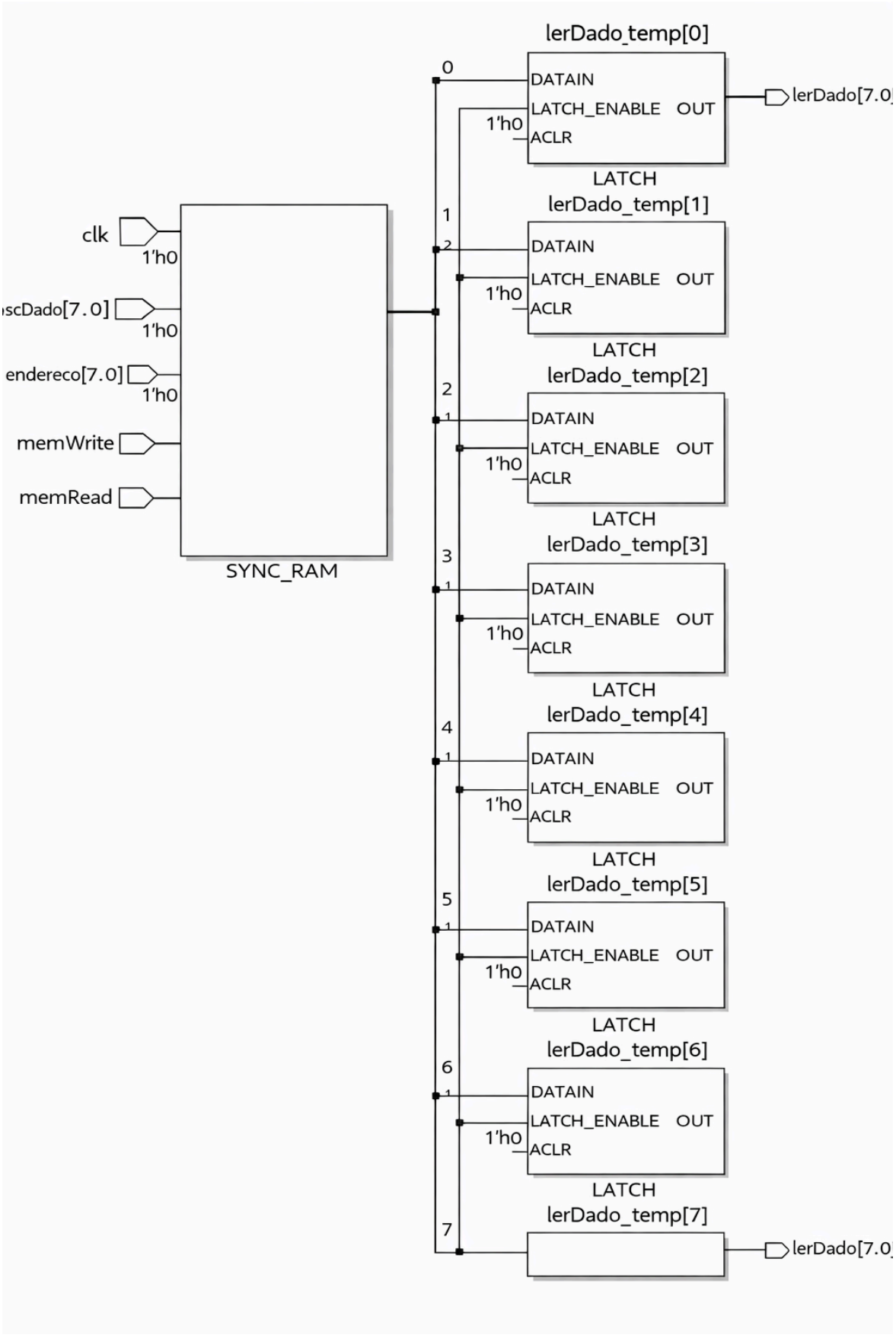
- **memRead**: sinal de controle de 1 bit que indica a habilitação da operação de leitura da memória.

- **clk**: sinal de clock de 1 bit, responsável por sincronizar as operações de escrita na memória de dados.

Como saída, o componente disponibiliza:

- **lerDado**: sinal de dados de 8 bits que corresponde ao valor lido da memória quando a operação de leitura estiver ativa.

Figura 5 – Memória de dados



3.6. ULA

O componente Unidade Lógica e Aritmética **(ULA)** é responsável pela execução de operações aritméticas e lógicas sobre dois operandos de 8 bits. No contexto deste projeto, a ULA implementa um conjunto reduzido de operações, contemplando a soma e a subtração, em conformidade com a filosofia RISC adotada. Adicionalmente, a ULA gera um sinal indicador de resultado zero, utilizado em instruções de desvio condicional.

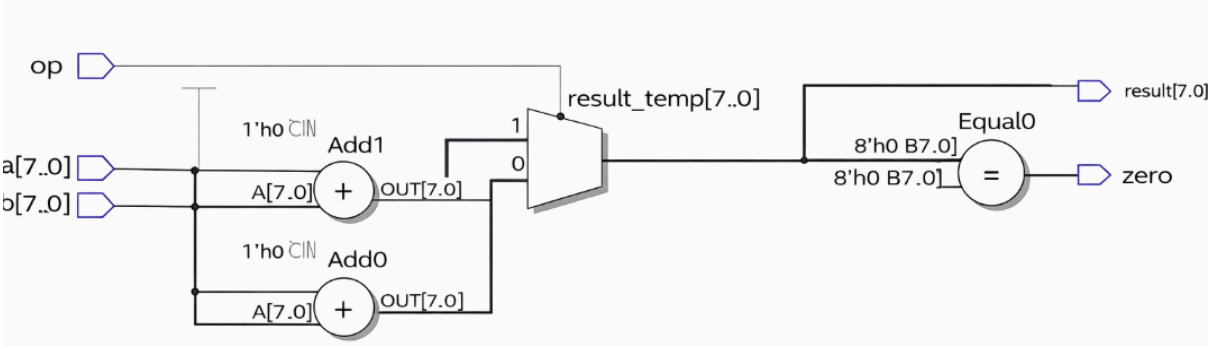
O componente **ula** possui as seguintes entradas:

- **a**: sinal de dados de 8 bits que representa o primeiro operando da operação aritmética ou lógica.
- **b**: sinal de dados de 8 bits que representa o segundo operando da operação aritmética ou lógica.
- **op**: sinal de controle de 1 bit que define a operação a ser realizada pela ULA, conforme descrito a seguir:
    - op = '0': realização da operação de soma;
    - op = '1': realização da operação de subtração.

Como saídas, o componente disponibiliza:

- **result**: sinal de dados de 8 bits que corresponde ao resultado da operação realizada pela ULA.
- **zero**: sinal de saída de 1 bit que indica se o resultado da operação é igual a zero, sendo ativado em nível lógico alto (‘1’) quando o resultado é zero, e em nível lógico baixo (‘0’) caso contrário.

Figura 6 – ULA



3.7. EXTENSOR DE BITS 3X8

O componente Extensor de Bits 3×8 é responsável por realizar a extensão de um sinal de 3 bits para um sinal de 8 bits, por meio do preenchimento dos bits mais significativos com zeros. Esse processo, conhecido como *zero extension*, é utilizado para adequar valores imediatos ao tamanho padrão de dados do processador.

O componente **extensor 3x8** possui a seguinte entrada:

- **A**: sinal de dados de 3 bits que representa o valor a ser estendido.

Como saída, o componente disponibiliza:

- **S**: sinal de dados de 8 bits que corresponde ao valor estendido, no qual os cinco bits mais significativos são preenchidos com zeros, enquanto os três bits menos significativos mantêm o valor original do sinal de entrada A.

3.8. EXTENSOR DE BITS 5X8

O componente Extensor de Bits 5×8 tem a função de ampliar um sinal de 5 bits para um sinal de 8 bits, realizando o preenchimento dos bits mais significativos com zeros. Esse tipo de extensão, denominada *zero extension*, é empregado para adequar valores de menor largura ao padrão de dados utilizado internamente pelo processador.

O componente **extensor 5x8** possui a seguinte entrada:

- **entrada**: sinal de dados de 5 bits que representa o valor a ser estendido.

Como saída, o componente disponibiliza:

- **saída**: sinal de dados de 8 bits correspondente ao valor estendido, no qual os três bits mais significativos são preenchidos com zeros, enquanto os cinco bits menos significativos mantêm o valor original do sinal de entrada.

3.9 Multiplexador 2×1

O componente Multiplexador 2×1 é responsável por selecionar um entre dois sinais de entrada de 8 bits, direcionando o valor escolhido para a saída de acordo com o sinal de controle. Esse tipo de componente é amplamente utilizado no *datapath* do processador para controlar o fluxo de dados entre os diferentes módulos, permitindo a escolha dinâmica das fontes de dados durante a execução das instruções.

O componente **mux 2x1** possui as seguintes entradas:

- **a**: sinal de dados de 8 bits que representa a primeira opção de entrada.
- **b**: sinal de dados de 8 bits que representa a segunda opção de entrada.
- **sel**: sinal de seleção de 1 bit que define qual das entradas será encaminhada para a saída.

Como saída, o componente disponibiliza:

- **f**: sinal de dados de 8 bits correspondente à entrada selecionada, sendo a quando sel = '0' e b quando sel = '1'.

3.10. Somador

O componente Somador é responsável pela realização da operação de adição entre dois operandos de 8 bits, fornecendo o resultado correspondente em sua saída. Esse módulo é utilizado no *datapath* do processador para operações que exigem o incremento ou a combinação de valores, como o avanço do Contador de Programa ou cálculos intermediários.

O componente **somador** possui as seguintes entradas:

- **A**: sinal de dados de 8 bits (1 byte) que representa o primeiro operando da operação de soma.
- **B**: sinal de dados de 8 bits (1 byte) que representa o segundo operando da operação de soma.

Como saída, o componente disponibiliza:

- **SOMA**: sinal de dados de 8 bits (1 byte) que corresponde ao resultado da soma dos dois operandos de entrada.

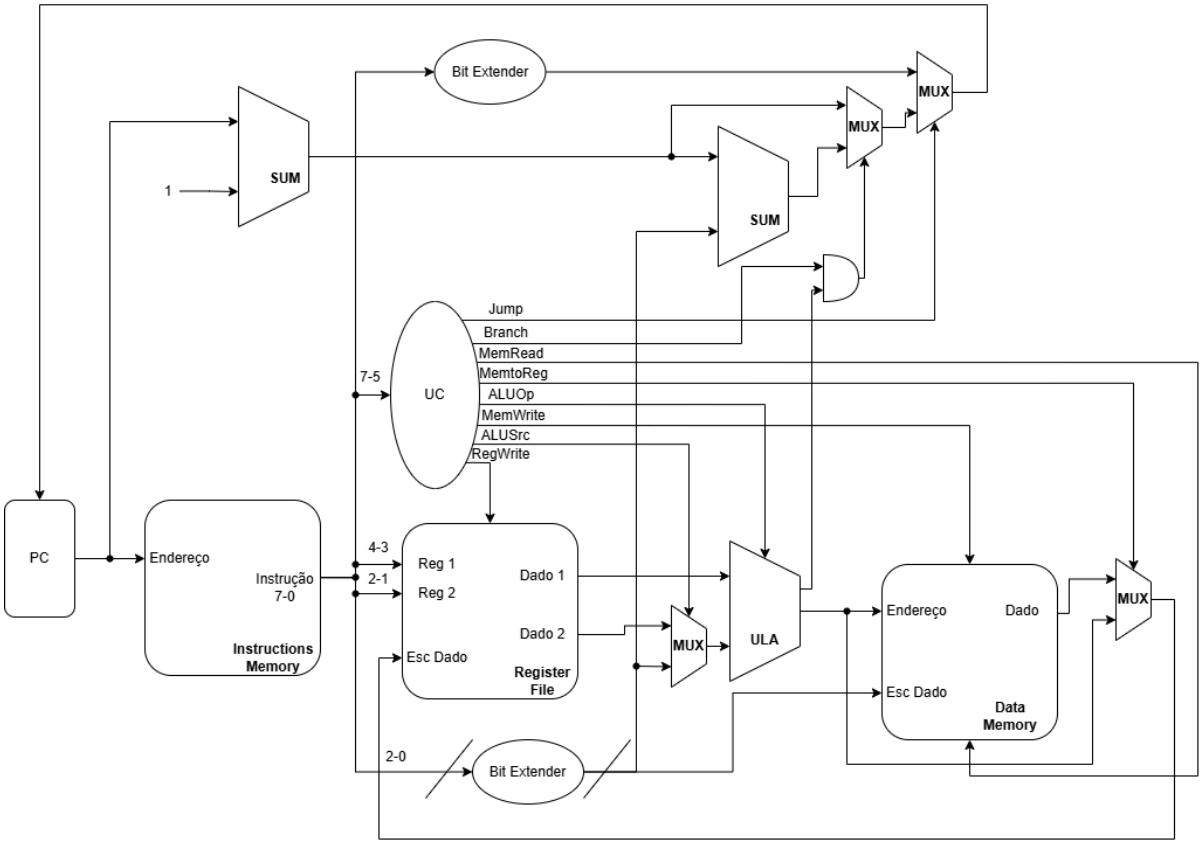
3.11. DATAPATH

O *datapath* corresponde à estrutura responsável pela execução das instruções no processador, sendo composto por um conjunto de componentes interconectados que realizam o fluxo e o processamento dos dados. Esse caminho de dados integra os principais módulos do sistema, permitindo que as operações definidas pelas instruções sejam executadas de forma coordenada e controlada.

Entre os componentes que compõem o *datapath* destacam-se o Bit Extender, responsável pela adequação do tamanho dos dados imediatos, e a Unidade Lógica e Aritmética (ULA), que realiza as operações aritméticas e lógicas. O direcionamento dos dados entre os diferentes blocos é realizado por meio de multiplexadores (MUX), os quais selecionam as fontes de dados apropriadas conforme os sinais de controle gerados pela Unidade de Controle.

O controle do fluxo de execução do programa é realizado por meio dos sinais Jump e Branch, que possibilitam a alteração do Contador de Programa em instruções de desvio. As operações de acesso à memória são gerenciadas pelos sinais MemRead e MemWrite, enquanto o armazenamento dos resultados no banco de registradores é habilitado pelo sinal RegWrite.

Figura 7 - Datapath



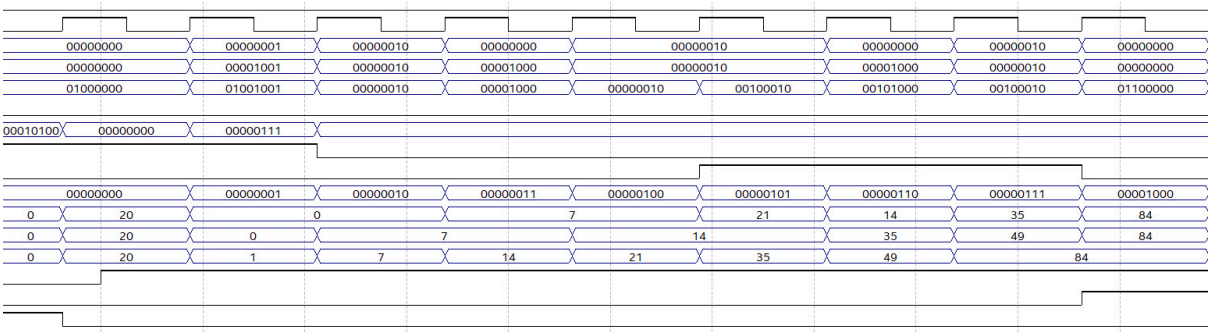
4. SIMULAÇÕES

4.1. TESTE ADD / SUB

Figura 8 – Instruções do teste ADD e SUB

```
signal memoria : instrucoes := (
    -- INÍCIO DO PROGRAMA DE TESTE (3 ADD / 3 SUB)
    0 => "01000000", -- LW $0, 0($0): Carrega 20 no Reg 0
    1 => "01001001", -- LW $1, 1($1): Carrega 7 no Reg 1

    -- PROCESSOS DE SOMA (ADD)
    2 => "00000010", -- ADD $0, $0, $1: Reg 0 = 20 + 7 = 27
    3 => "00001000", -- ADD $1, $1, $0: Reg 1 = 7 + 27 = 34
    4 => "00000010", -- ADD $0, $0, $1: Reg 0 = 27 + 34 = 61

    -- PROCESSOS DE SUBTRAÇÃO (SUB)
    5 => "00100010", -- SUB $0, $0, $1: Reg 0 = 61 - 34 = 27
    6 => "00101000", -- SUB $1, $1, $0: Reg 1 = 34 - 27 = 7
    7 => "00100010", -- SUB $0, $0, $1: Reg 0 = 27 - 7 = 20

    8 => "01100000", -- JUMP p/ Endereço 0: Reinicia o ciclo
    others => (others => '1')
);
```

Figura 9 – Waveform das instruções do teste ADD e SUB

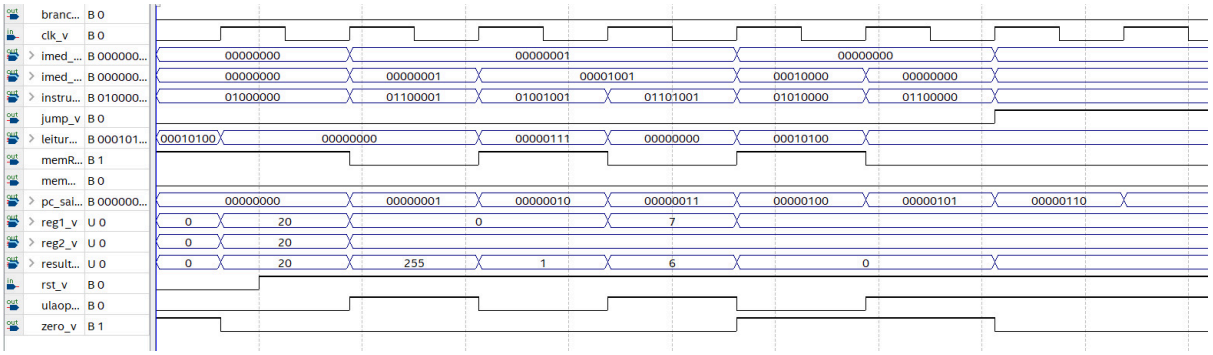


4.2. TESTE LW / SW

Figura 10 – Instruções do teste LW, SW

```
    -- Programa de Teste: LW e SW
signal memoria : instrucoes := (
    0 => "01000000", -- LW $0, 0($0): Reg 0 = RAM[0] (Carrega o 20)
    1 => "01100001", -- SW $0, 1($0): RAM[20 + 1] = 20 (Salva o 20 na posição 21)
    2 => "01001001", -- LW $1, 1($1): Reg 1 = RAM[1] (Carrega o 7)
    3 => "01101001", -- SW $1, 1($1): RAM[7 + 1] = 7  (Salva o 7 na posição 8)
    4 => "01010000", -- LW $2, 0($2): Reg 2 = RAM[0] (Lê o 20 da RAM novamente)
    5 => "01100000", -- JUMP p/ Endereço 0 (Loop)
    others => (others => '1')
);
```

Figura 11 – Waveform das instruções do teste LW e SW

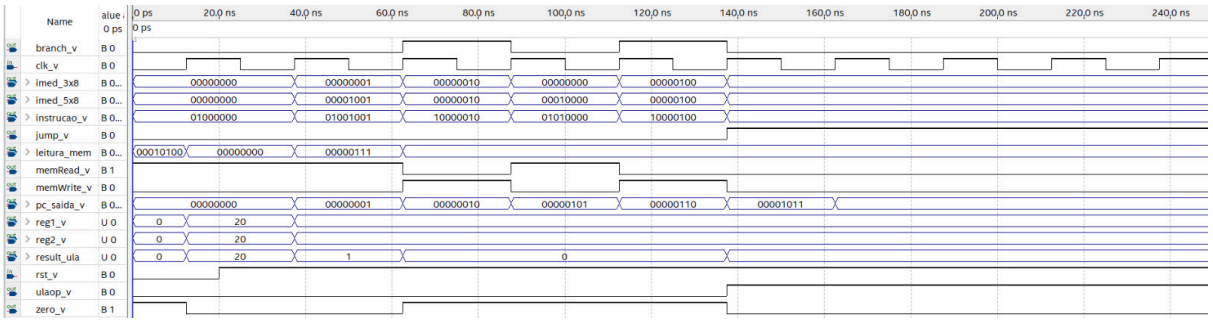


4.3. TESTE BEQ / JUMP

Figura 12 – Instruções do teste BEQ e JUMP

```
signal memoria : instrucoes := (
        0 => "01000000", -- LW $0, 0($0): Reg 0 = 20
        1 => "01001001", -- LW $1, 1($1): Reg 1 = 7

        -- TESTE BEQ (FALSO): 20 != 7, não deve pular.
        2 => "10000010", -- BEQ $0, $1, salto+2: Se iguais, pula para o endereço 5
        3 => "00000001", -- ADD $0, $0, $1: Executa isso porque o BEQ acima foi falso

        -- TESTE JUMP: Salto incondicional
        4 => "01100111", -- JUMP para Endereço 7: Pula direto para o final

        -- TESTE BEQ (VERDADEIRO):
        5 => "01010000", -- LW $2, 0($2): Reg 2 = 20 (Agora Reg 0 e Reg 2 são iguais)
        6 => "10000100", -- BEQ $0, $2, salto+2: Como 20 == 20, pula para o endereço 9

        7 => "00100001", -- SUB $0, $0, $1: Só chega aqui pelo JUMP do endereço 4
        8 => "01100000", -- JUMP para o Início (Loop)

        9 => "00001000", -- ADD $1, $1, $0: Alvo do segundo BEQ
        others => (others => '1')
    );
```

Figura 13 – Waveform das instruções do teste BEQ e JUMP



5. CONCLUSÃO

A realização deste projeto permitiu o desenvolvimento e a implementação de um processador RISC de 8 bits, consolidando na prática os conceitos abordados na disciplina de Arquitetura e Organização de Computadores (AOC). A construção do processador, desde a definição do conjunto de instruções até a integração dos módulos em VHDL, proporcionou uma compreensão mais aprofundada do funcionamento interno de arquiteturas de computadores.

Os resultados obtidos demonstram que o processador atende aos requisitos propostos, sendo capaz de executar corretamente o conjunto mínimo de instruções definido. As simulações realizadas confirmaram o funcionamento adequado dos principais componentes, como o *datapath*, a Unidade de Controle, a ULA e as memórias, validando as decisões de projeto adotadas.

O processador desenvolvido serve não apenas como uma ferramenta de validação prática dos conceitos teóricos estudados, mas também como uma base didática para o entendimento do fluxo de execução de instruções, do funcionamento da unidade de controle e da interação entre os diversos blocos que compõem uma arquitetura RISC. O estudo realizado contribui para a formação técnica na área de arquitetura de computadores e sistemas digitais, aproximando teoria e prática por meio de um projeto funcional e documentado.

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.