Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari

# Sistemas Reconfiguráveis
# Trabalho 3 - Port_io e Ram_256x8

Brasil

30/05/2024

Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari

# Sistemas Reconfiguráveis
# Trabalho 3 - Port_io e Ram_256x8

Este documento é um relatório técnico para o curso de Sistemas Reconfiguráveis, apresentado conforme as normas ABNT e utilizando a plataforma LaTeX, para detalhar o projeto e implementação de um Port_io e Ram_256x8.

Pontifícia Universidade Católica de Minas Gerais

Faculdade de Engenharia de Computação

Brasil

30/05/2024

Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari
Sistemas Reconfiguráveis
Trabalho 3 - Port_io e Ram_256x8/ Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari. – Brasil, 30/05/2024-

Relatório Técnico – Pontifícia Universidade Católica de Minas Gerais
Faculdade de Engenharia de Computação , 30/05/2024.

CDU 02:141:005.7

# Resumo

Este documento apresenta a concepção, implementação e simulação de dois módulos fundamentais em VHDL: uma porta de entrada/saída paralela de 8 bits (*port_io*) e uma memória RAM de 256 bytes (*ram_256x8*). Cada módulo foi projetado para operar com sinais do tipo *std_logic* ou *std_logic_vector*, e desenvolvido utilizando a linguagem VHDL com a ferramenta Quartus. A metodologia adotada incluiu uma descrição detalhada da lógica de cada módulo, seguida por simulações para validar todas as funcionalidades propostas. Os resultados das simulações demonstraram que cada módulo atende às especificações técnicas requeridas, executando suas operações de maneira correta e eficiente. Este relatório segue as normas ABNT para documentação técnica e acadêmica, e inclui todos os arquivos de projeto e simulações realizadas.

**Palavras-chave**: VHDL, simulação, port_io, memória RAM, Quartus.

# Sumário

# Parte I

# Introdução

# 1 Contexto

Este relatório cobre a implementação e simulação de dois componentes fundamentais de sistemas digitais: uma porta de entrada/saída paralela de 8 bits (*port_io*) e uma memória RAM de 256 bytes (*ram_256x8*). Esses componentes são essenciais em várias aplicações de sistemas embarcados e microprocessadores, fornecendo interfaces de entrada/saída e armazenamento de dados de maneira eficiente e confiável. Através da utilização da linguagem VHDL, esses sistemas são descritos e simulados para validar sua funcionalidade e desempenho.

# 2 Objetivos do Trabalho

O objetivo deste projeto é projetar, implementar e simular dois módulos distintos usando VHDL. Cada módulo foi cuidadosamente projetado para operar com sinais do tipo *std_logic* ou *std_logic_vector*, adequando-se às especificações fornecidas para:

- Uma porta de entrada/saída paralela de 8 bits (*port_io*) que permite o interfaceamento do controlador com sensores e atuadores digitais.
- Uma memória RAM de 256 bytes (*ram_256x8*) que oferece capacidade de armazenamento volátil para dados temporários.

Este relatório detalhará a implementação desses sistemas em VHDL, incluindo simulações que demonstram suas operações e discutem seus possíveis usos em aplicações práticas.

# 3 Estrutura do Documento

O documento está estruturado da seguinte forma para facilitar a compreensão e a localização de informações específicas:

1. **Introdução:** Apresenta o contexto e os objetivos deste trabalho.
2. **Desenvolvimento e Resultados:** Descreve a lógica e o desenvolvimento de cada um dos módulos em VHDL e seus resultados.
3. **Implementação:** Detalha especificamente como cada módulo foi implementado, incluindo trechos de código e explicações.
4. **Conclusões:** Resume os resultados alcançados e sugere direções futuras para pesquisa e desenvolvimento.

# Parte II

# Desenvolvimento e Resultados

# 1 port_io

## 1.1 Descrição da Implementação

O módulo `port_io` foi desenvolvido para simular uma porta de entrada/saída paralela de 8 bits, que permite o interfaceamento do controlador com sensores e atuadores digitais. Cada pino pode ser individualmente programado para funcionar como entrada ou saída através do registrador `dir_reg`. Um valor '0' no bit correspondente do `dir_reg` configura o pino como entrada, enquanto um valor '1' o configura como saída. Os pinos configurados como saída recebem o valor escrito no registrador `port_reg`, enquanto o estado de todos os pinos pode ser lido.

As operações principais do `port_io` incluem:

- **Configuração da Direção:** Realizada através do registrador `dir_reg`, permitindo configurar cada pino individualmente como entrada ou saída.
- **Escrita:** Realizada no registrador `port_reg` para os pinos configurados como saída.
- **Leitura:** Pode ser feita tanto do estado atual dos pinos quanto do conteúdo dos registradores.

## 1.2 Resultados da Simulação

Durante a simulação, o módulo `port_io` demonstrou capacidade de configuração de direção e operações de leitura e escrita conforme esperado. Os testes incluíram múltiplas configurações de direção, escritas e leituras, verificando a integridade e a precisão das operações.

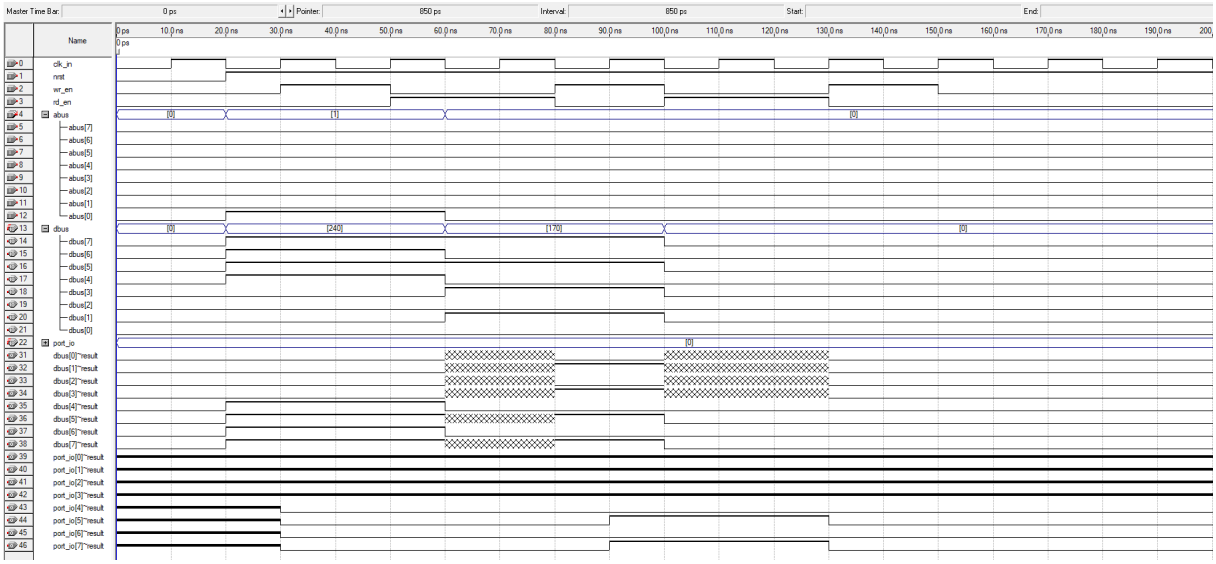


Figura 1 – Waveform ilustrando a simulação do `port_io` com múltiplas operações de configuração, escrita e leitura.

As operações de escrita e leitura demonstraram a funcionalidade do módulo, com os dados sendo corretamente armazenados e recuperados conforme as configurações de direção e os sinais de controle.

No waveform acima, podemos observar as seguintes operações ao longo do tempo:

- **0ps a 20ns**:
  - O sinal `nrst` está ativo baixo ('0'), mantendo todos os registradores zerados.
  - O `clk_in` inicia suas transições de clock.
  - Os sinais `wr_en` e `rd_en` estão desativados ('0').
- **20ns a 40ns**:
  - O sinal `nrst` é desativado ('1'), permitindo que o sistema funcione normalmente.
- **40ns a 60ns**:
  - O sinal `wr_en` é ativado ('1'), permitindo a escrita no registrador `port_reg`.
  - O endereço no barramento `abus` é `base_addr`, direcionando a escrita para o `port_reg`.
  - O barramento de dados `dbus` carrega o valor 240 (11110000 em binário), que será escrito no `port_reg`.
- **60ns a 80ns**:

  - O sinal `wr_en` é desativado ('0'), completando a operação de escrita.
  - O valor escrito (240) agora está armazenado no `port_reg`.

- **80ns a 100ns**:
  - O sinal `wr_en` é ativado novamente ('1'), mas desta vez o endereço no barramento `abus` é `base_addr + 1`, direcionando a escrita para o `dir_reg`.
  - O barramento de dados `dbus` carrega o valor 170 (10101010 em binário), que será escrito no `dir_reg`.

- **100ns a 120ns**:
  - O sinal `wr_en` é desativado ('0'), completando a operação de escrita.
  - O valor escrito (170) agora está armazenado no `dir_reg`.

- **120ns a 140ns**:
  - O sinal `rd_en` é ativado ('1'), iniciando uma operação de leitura.
  - O endereço no barramento `abus` é `base_addr`, direcionando a leitura para o latch, que contém os dados lidos do `port_io`.
  - O barramento de dados `dbus` reflete o valor dos pinos configurados como entrada no `port_io`.

- **140ns a 160ns**:
  - O sinal `rd_en` é desativado ('0'), completando a operação de leitura.
  - O barramento de dados `dbus` retorna à alta impedância.

- **160ns a 180ns**:
  - O sinal `rd_en` é ativado novamente ('1'), mas desta vez o endereço no barramento `abus` é `base_addr + 1`, direcionando a leitura para o `dir_reg`.
  - O barramento de dados `dbus` reflete o valor armazenado no `dir_reg` (170).

- **180ns a 200ns**:
  - O sinal `rd_en` é desativado ('0'), completando a operação de leitura.
  - O barramento de dados `dbus` retorna à alta impedância.

## 1.3 Discussão

O waveform da simulação claramente demonstra que o `port_io` mantém a consistência das operações através de diversas configurações de direção, escritas e leituras. A capacidade de configurar individualmente cada pino como entrada ou saída e realizar operações de leitura e escrita sem interferências mostra a robustez do módulo.

As operações de configuração da direção e de leitura/escrita foram validadas, reforçando a utilidade do módulo em cenários de aplicação real, como parte de um sistema embarcado ou controlador de processadores.

## 1.4 Conclusões

Os resultados obtidos confirmam o sucesso na implementação do `port_io`, com todas as funcionalidades operando conforme especificado. O módulo provou ser uma ferramenta eficaz para simulação e ensino de conceitos fundamentais de hardware, oferecendo uma base sólida para a expansão para sistemas mais complexos ou para uso em contextos educacionais.

# 2 ram_256x8

## 2.1 Descrição da Implementação

O módulo `ram_256x8` foi desenvolvido para simular uma memória RAM de 256 bytes. Esta memória é capaz de armazenar dados temporários de forma volátil, permitindo operações de leitura e escrita. A memória possui um barramento de dados bidirecional de 8 bits (`dio`), que comporta-se como saída durante as operações de leitura e como entrada durante as operações de escrita. As operações de escrita são sincronizadas com o sinal de clock, enquanto as operações de leitura são assíncronas.

As operações principais do `ram_256x8` incluem:

- **Escrita:** Realizada sincronicamente com o sinal de clock quando o sinal de habilitação `mem_wr_en` está ativo. O endereço da posição de memória é especificado pelo sinal `addr`, e os dados são fornecidos pelo barramento `dio`.
- **Leitura:** Executada de maneira assíncrona quando o sinal de habilitação `mem_rd_en` está ativo. O endereço da posição de memória é especificado pelo sinal `addr`, e os dados são lidos do barramento `dio`.

## 2.2 Resultados da Simulação

Durante a simulação, o módulo `ram_256x8` demonstrou capacidade de armazenamento e recuperação de dados conforme esperado. Os testes incluíram múltiplas escritas sequenciais em diferentes endereços de memória e subsequentes leituras para verificar a integridade e precisão dos dados armazenados.

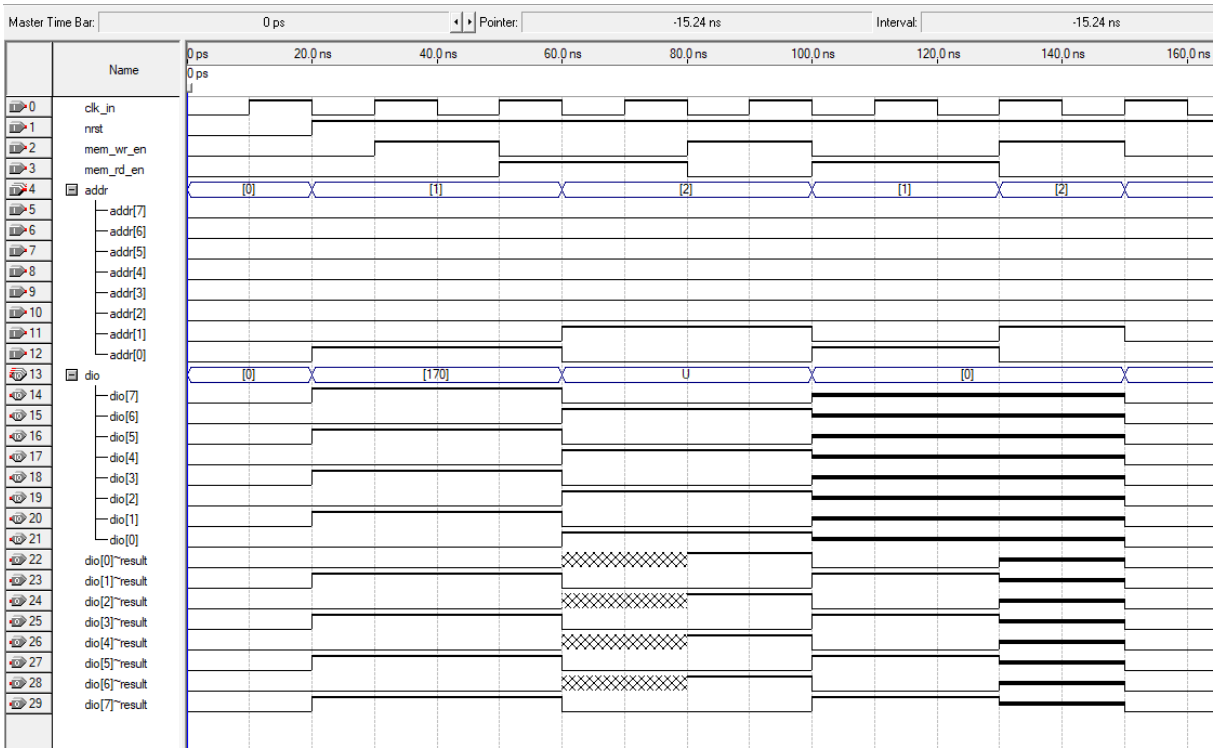


Figura 2 – Waveform ilustrando a simulação do `ram_256x8` com múltiplas operações de escrita e leitura.

As operações de escrita e leitura demonstraram a funcionalidade do módulo, com os dados sendo corretamente armazenados e recuperados conforme os sinais de controle.

No waveform acima, podemos observar as seguintes operações ao longo do tempo:

- **0ps a 20ns**:
  - O sinal `nrst` está ativo baixo ('0'), mantendo todas as posições da memória zeradas.
  - O `clk_in` inicia suas transições de clock.
  - Os sinais `mem_wr_en` e `mem_rd_en` estão desativados ('0').
  - O endereço no barramento `addr` é `00000000`.
  - O barramento de dados `dio` está em `00000000`.

- **20ns a 40ns**:
  - O sinal `nrst` é desativado ('1'), permitindo que o sistema funcione normalmente.
  - O endereço no barramento `addr` é alterado para `00000001`.
  - O barramento de dados `dio` é alterado para `10101010`.
- **40ns a 60ns**:
  - O sinal `mem_wr_en` é ativado ('1'), permitindo a escrita na posição de memória `00000001`.
  - O valor `10101010` é escrito na posição `00000001` da memória.
- **60ns a 80ns**:
  - O sinal `mem_wr_en` é desativado ('0'), completando a operação de escrita.
  - O endereço no barramento `addr` é alterado para `00000010`.
  - O barramento de dados `dio` é alterado para `01010101`.
- **80ns a 100ns**:
  - O sinal `mem_wr_en` é ativado novamente ('1'), permitindo a escrita na posição de memória `00000010`.
  - O valor `01010101` é escrito na posição `00000010` da memória.
- **100ns a 120ns**:
  - O sinal `mem_wr_en` é desativado ('0'), completando a operação de escrita.
  - O sinal `mem_rd_en` é ativado ('1'), permitindo a leitura da posição de memória `00000001`.
  - O valor `10101010` é lido da posição `00000001` e colocado no barramento `dio`.
- **120ns a 140ns**:
  - O sinal `mem_rd_en` é desativado ('0'), completando a operação de leitura.
  - O endereço no barramento `addr` é alterado para `00000010`.
- **140ns a 160ns**:
  - O sinal `mem_rd_en` é ativado novamente ('1'), permitindo a leitura da posição de memória `00000010`.
  - O valor `01010101` é lido da posição `00000010` e colocado no barramento `dio`.

## 2.3 Discussão

O waveform da simulação demonstra claramente que o `ram_256x8` mantém a consistência dos dados através de diversas operações de escrita e leitura, evidenciando a robustez do módulo. A capacidade de realizar operações de leitura e escrita de forma independente e precisa é essencial para a utilidade do módulo em aplicações reais, como parte de sistemas embarcados que exigem armazenamento temporário de dados.

As operações de escrita e leitura, sendo capazes de ocorrer sem interferências e de maneira controlada, garantem a confiabilidade necessária para aplicações críticas, onde a precisão e a estabilidade dos dados são vitais.

## 2.4 Conclusões

Os resultados obtidos confirmam o sucesso na implementação do `ram_256x8`, com todas as funcionalidades operando conforme especificado. O módulo provou ser uma ferramenta eficaz para simulação e ensino de conceitos fundamentais de hardware, oferecendo uma base sólida para a expansão para sistemas mais complexos ou para uso em contextos educacionais.

# Parte III

# Implementação

# 1 Implementação do port_io

A implementação do `port_io` focou em fornecer um mecanismo eficiente e flexível para o interfaceamento do controlador com sensores e atuadores digitais, permitindo operações de entrada e saída de dados com controle individual de direção para cada pino.

## 1.1 Código VHDL

O seguinte trecho de código VHDL ilustra a definição dos registradores de direção e porta, assim como a lógica de controle para as operações de leitura e escrita:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Defini  o da entidade port_io
entity port_io is
    generic (
        base_addr : std_logic_vector(7 downto 0) := "00000000"  -- Ender
    );
    port (
        nrst    : in  std_logic;  -- Entrada de reset ass ncrono
        clk_in  : in  std_logic;  -- Entrada de clock do sistema
        abus    : in  std_logic_vector(7 downto 0);  -- Barramento de end
        dbus    : inout std_logic_vector(7 downto 0);  -- Barramento bidir
        wr_en   : in  std_logic;  -- Habilita  o de escrita
        rd_en   : in  std_logic;  -- Habilita  o de leitura
        port_io: inout std_logic_vector(7 downto 0)  -- Porta bidirecion
    );
end port_io;

-- Arquitetura comportamental da entidade port_io
architecture Behavioral of port_io is
    -- Sinais internos para os registradores
    signal dir_reg : std_logic_vector(7 downto 0) := (others => '0');
-- Registrador de dire  o
    signal port_reg: std_logic_vector(7 downto 0) := (others => '0');
-- Registrador de porta
    signal latch    : std_logic_vector(7 downto 0);  -- Latch para leitura
    signal dbus_internal: std_logic_vector(7 downto 0);  -- Sinal interno
```

```
begin
    -- Processo para tratar o reset ass ncrono e as escritas s ncronas
    process(nrst, clk_in)
    begin
        if nrst = '0' then
            -- Reset ass ncrono: zera os registradores
            dir_reg <= (others => '0');
            port_reg <= (others => '0');
        elsif rising_edge(clk_in) then
            if wr_en = '1' then
                -- Escreve no registrador de porta ou de dire   o depend
                if abus = base_addr then
                    port_reg <= dbus;
                elsif abus = base_addr + 1 then
                    dir_reg <= dbus;
                end if;
            end if;
        end if;
    end process;

    -- Processo para tratar a dire   o e os dados da porta
    process(dir_reg, port_reg)
    begin
        for i in 0 to 7 loop
            if dir_reg(i) = '0' then
                port_io(i) <= 'Z';  -- Alta imped ncia quando configurad
            else
                port_io(i) <= port_reg(i);  -- Valor de sa da quando co
            end if;
        end loop;
    end process;

    -- Processo para travar as entradas de port_io quando configuradas co
    process(port_io, dir_reg)
    begin
        for i in 0 to 7 loop
            if dir_reg(i) = '0' then
                latch(i) <= port_io(i);  -- L   o valor dos pinos de entr
            else
```

```
                latch(i) <= '0';
            end if;
        end loop;
    end process;

    -- Processo para tratar as opera  es de leitura
    process(rd_en, abus)
    begin
        if rd_en = '1' then
            -- Leitura dos registradores ou latch dependendo do endere o
            if abus = base_addr then
                dbus_internal <= latch;
            elsif abus = base_addr + 1 then
                dbus_internal <= dir_reg;
            else
                dbus_internal <= (others => 'Z');
            end if;
        else
            dbus_internal <= (others => 'Z');
        end if;
    end process;

    -- Buffer tri-state para o barramento de dados
    dbus <= dbus_internal when rd_en = '1' else (others => 'Z');

end Behavioral;
```

## 1.2 Detalhes da Implementação

O módulo `port_io` é composto por dois registradores principais: `dir_reg`, que define a direção dos pinos (entrada ou saída), e `port_reg`, que armazena os valores a serem escritos nos pinos configurados como saída. Além disso, um latch (`latch`) é utilizado para capturar os valores das entradas.

A configuração dos pinos é controlada pelo registrador `dir_reg`, onde cada bit indica se o respectivo pino é uma entrada ('0') ou uma saída ('1'). Após o reset, todos os pinos são configurados como entradas.

As operações de escrita são sincronizadas com o clock do sistema (`clk_in`), e ocorrem quando o sinal de habilitação de escrita (`wr_en`) está ativo. Dependendo do endereço fornecido no barramento de endereços (`abus`), o valor do barramento de dados

(dbus) é escrito no registrador de porta (port_reg) ou no registrador de direção (dir_reg).

As operações de leitura são controladas pelo sinal de habilitação de leitura (rd_en). Quando ativo, o conteúdo do registrador de direção (dir_reg) ou o latch (latch) é colocado no barramento de dados (dbus), dependendo do endereço fornecido.

A arquitetura foi projetada para garantir que as operações de leitura e escrita sejam realizadas de forma eficiente e sem interferências, permitindo um interfaceamento robusto e confiável com diversos dispositivos digitais. A utilização de buffers tri-state no barramento de dados (dbus) assegura que ele fique em alta impedância quando não está sendo utilizado, evitando conflitos de sinal.

# Parte IV

# Implementação

# 1 Implementação do ram_256x8

A implementação do `ram_256x8` focou em fornecer um mecanismo eficiente para armazenamento de dados volátil, permitindo operações de leitura e escrita de maneira síncrona e assíncrona, respectivamente.

## 1.1 Código VHDL

O seguinte trecho de código VHDL ilustra a definição da memória, assim como a lógica de controle para as operações de leitura e escrita:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL; -- Use numeric_std para to_integer e unsigned

-- Defini  o da entidade ram_256x8
entity ram_256x8 is
    port (
        nrst       : in  std_logic; -- Entrada de reset ass ncrono
        clk_in     : in  std_logic; -- Entrada de clock do sistema
        addr       : in  std_logic_vector(7 downto 0); -- Barramento de
        dio        : inout std_logic_vector(7 downto 0); -- Barramento
        mem_wr_en  : in  std_logic; -- Habilita  o de escrita na mem
        mem_rd_en  : in  std_logic  -- Habilita  o de leitura da mem
    );
end ram_256x8;

-- Arquitetura comportamental da entidade ram_256x8
architecture Behavioral of ram_256x8 is
    -- Defini  o do tipo de dado para a mem ria
    type memory_type is array (255 downto 0) of std_logic_vector(7 downto
    signal memory : memory_type := (others => (others => '0'));
-- Sinal que representa a mem ria
    signal data_out : std_logic_vector(7 downto 0); -- Sinal para os da
begin
    -- Processo para tratar o reset ass ncrono e a escrita s ncrona
    process(nrst, clk_in)
    begin
        if nrst = '0' then
            -- Reset ass ncrono: zera todas as posi  es da mem ria
            memory <= (others => (others => '0'));
```

```
        elsif rising_edge(clk_in) then
            if mem_wr_en = '1' then
                -- Escreve na mem ria na posi   o especificada por addr
                memory(to_integer(unsigned(addr))) <= dio;
            end if;
        end if;
    end process;

    -- Processo para tratar a leitura ass ncrona
    process(addr, mem_rd_en)
    begin
        if mem_rd_en = '1' then
            -- L   o valor da mem ria na posi   o especificada por addr
            data_out <= memory(to_integer(unsigned(addr)));
        else
            data_out <= (others => 'Z');  -- Alta imped ncia quando n o
        end if;
    end process;

    -- Buffer tri-state para o barramento de dados
    dio <= data_out when mem_rd_en = '1' else (others => 'Z');

end Behavioral;
```

## 1.2 Detalhes da Implementação

O módulo `ram_256x8` é composto por uma memória de 256 posições, onde cada posição armazena um byte de dados. A memória é implementada como um array de vetores de bits (`std_logic_vector`). O módulo permite operações de escrita e leitura nos endereços especificados pelo barramento de endereços (`addr`).

As operações de escrita são sincronizadas com o clock do sistema (`clk_in`) e são habilitadas pelo sinal `mem_wr_en`. Quando `mem_wr_en` está ativo, os dados presentes no barramento de dados (`dio`) são armazenados na posição da memória especificada por `addr`.

As operações de leitura são assíncronas e são habilitadas pelo sinal `mem_rd_en`. Quando `mem_rd_en` está ativo, os dados da posição de memória especificada por `addr` são colocados no barramento de dados (`dio`). Quando `mem_rd_en` não está ativo, o barramento de dados é colocado em alta impedância para evitar conflitos de sinal.

A arquitetura foi projetada para garantir que as operações de leitura e escrita sejam

realizadas de forma eficiente e sem interferências, proporcionando um armazenamento volátil robusto e confiável para dados temporários. A utilização de buffers tri-state no barramento de dados (`dio`) assegura que ele fique em alta impedância quando não está sendo utilizado, evitando conflitos de sinal e garantindo a integridade dos dados.

# Parte V

# Conclusões

# 1 Reflexões Gerais

Ao longo deste projeto, foram desenvolvidos e simulados dois módulos distintos em VHDL: o `port_io` e o `ram_256x8`. Cada um desses módulos desempenha funções essenciais em sistemas computacionais e de controle, e sua implementação correta é crucial para a integridade e eficiência desses sistemas.

## 1.1 Conclusões sobre o port_io

O `port_io` demonstrou a capacidade de gerenciar múltiplas operações de leitura e escrita, bem como a configuração individual de direção para cada pino. Este módulo serve como um componente fundamental em sistemas que requerem interfaceamento versátil e preciso com dispositivos de entrada e saída digitais.

## 1.2 Conclusões sobre o ram_256x8

O módulo `ram_256x8` facilitou operações de escrita síncronas e leituras assíncronas conforme o esperado. A memória implementada é ideal para aplicações que necessitam de armazenamento temporário de dados, oferecendo uma gestão confiável e rápida do acesso aos dados.

# 2 Considerações Finais

Durante a implementação dos módulos em VHDL, técnicas específicas como a função `rising_edge()` e conversões de tipo com `to_integer()` desempenharam papéis cruciais para garantir a correta sincronização e manipulação de dados. A função `rising_edge()` é particularmente valiosa para detectar a transição de subida do sinal de clock, garantindo que as operações lógicas e aritméticas sejam executadas no momento preciso, o que é essencial para o design de circuitos síncronos confiáveis.

As conversões de tipo, como `to_integer()` e `unsigned()`, facilitam a manipulação de vetores de sinais (`std_logic_vector`) para operações aritméticas e lógicas, permitindo uma interface mais flexível e poderosa entre diferentes domínios de dados no VHDL. Essas funções ajudam os desenvolvedores a escrever códigos que são não apenas sintaticamente corretos, mas também semanticamente robustos e otimizados para desempenho.