Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari

Sistemas reconfiguráveis
Trabalho 2 - Registrador, Contador, Pilha

Brasil

25/04/2024

Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari

Sistemas reconfiguráveis
Trabalho 2 - Registrador, Contador, Pilha

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 Registrador, Contador e uma Pilha.

Pontifícia Universidade Católica de Minas Gerais

Faculdade de Engenharia de Computação

Brasil
25/04/2024

Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari
Sistemas reconfiguráveis
Trabalho 2 - Registrador, Contador, Pilha/ Philipi Gariglio Carvalho Faustino Altoé
Izabela Galinari. – Brasil, 25/04/2024-

Relatório Técnico – Pontifícia Universidade Católica de Minas Gerais
Faculdade de Engenharia de Computação , 25/04/2024.

CDU 02:141:005.7

Resumo

Este documento apresenta a concepção, implementação e simulação de três módulos fundamentais em VHDL: um banco de registradores (*reg_bank*), uma pilha de dados (*stack*) e um contador programável (*Prog_cnt*). 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 na versão 9.1sp2. 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, reg_bank, registrador, stack, pilha, Prog_cnt, contador, Quartus.

Sumário

Parte I

Introdução

1 Contexto

Este relatório cobre a implementação e simulação de três componentes fundamentais de sistemas digitais: um banco de registradores (reg_bank), uma pilha (stack) e um contador programável (prog_cnt). Cada um desses componentes desempenha um papel crucial em várias aplicações de sistemas embarcados e microprocessadores, manipulando e armazenando dados de maneira eficiente e confiável. Através da utilização da linguagem VHDL, esses sistemas são descritos e simulados para validar sua funcionalidade e performance.

2 Objetivos do Trabalho

O objetivo deste projeto é projetar, implementar e simular três 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:

- Um banco de registradores (reg_bank) que gerencia múltiplas unidades de armazenamento.
- Uma pilha (stack) que opera com o princípio Last In, First Out (LIFO).
- Um contador programável (prog_cnt) que suporta operações de incremento e carga baseadas em sinais de controle.

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 Resultado:** Descreve a lógica e o desenvolvimento de cada um dos três 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 regBank

1.1 Descrição da Implementação

O módulo `reg_bank` foi desenvolvido para simular um banco de registradores que consiste em oito registradores de 8 bits. Cada registrador, de R0 até R7, é capaz de armazenar um byte de dados. O registrador R7 é utilizado para armazenar flags de status, especificamente as flags de carry (C), zero (Z) e overflow (V), que são essenciais para o controle de operações lógicas e aritméticas em processadores.

As operações principais do `reg_bank` incluem:

- **Escrita:** Realizada sincronicamente com o sinal de clock quando o sinal de habilitação `regn_wr_ena` está ativo. A seleção do registrador destino é feita através do sinal `regn_wr_sel`.
- **Leitura:** Executada de maneira assíncrona através das saídas `regn_do_a` e `regn_do_b`, que permitem a leitura simultânea de dois registradores distintos, selecionados pelos sinais `regn_rd_sel_a` e `regn_rd_sel_b`.

1.2 Resultados da Simulação

Durante a simulação, o módulo `reg_bank` demonstrou capacidade de armazenamento e recuperação de dados, conforme esperado. Os testes incluíram múltiplas escritas sequenciais nos registradores e subsequentes leituras, verificando a integridade e a precisão dos dados armazenados.

As flags de status, atualizadas durante operações específicas, refletiram mudanças precisas conforme manipulações dos dados, indicando a funcionalidade eficaz do sistema de flags embutido no registrador R7.

1.3 Discussão

O waveform da simulação claramente demonstra que o `reg_bank` mantém a consistência dos dados através de diversas operações de escrita e leitura, evidenciando a robustez do módulo. A habilidade de atualizar e consultar as flags de status de forma independente dos registradores de dados também foi validada, reforçando a utilidade do módulo em cenários de aplicação real, como parte de um processador mais complexo ou sistema embarcado.

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

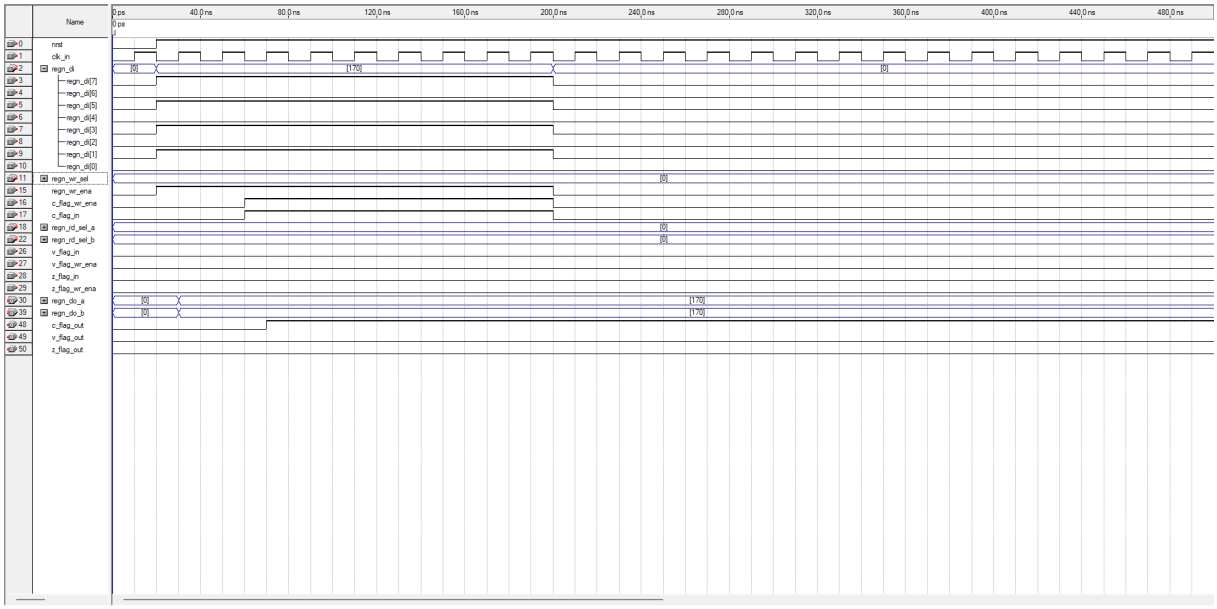

Figura 1 – Waveform ilustrando a simulação do reg_bank com múltiplas operações de escrita e leitura.

1.4 Conclusões

Os resultados obtidos confirmam o sucesso na implementação do `reg_bank`, 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 Stack

2.1 Descrição da Implementação

O módulo `Stack` foi desenvolvido para funcionar como uma pilha de dados com 8 registradores de 11 bits cada. Este tipo de estrutura de dados opera sob o princípio LIFO (Last In, First Out), o que é ideal para diversas aplicações de controle e temporização em sistemas embarcados e computacionais.

As principais funcionalidades implementadas no módulo `Stack` incluem:

- **Push:** A inserção de dados é realizada na borda de subida do clock quando o sinal `stack_push` está ativo. O dado inserido no topo da pilha é recebido pelo sinal `stack_in`, e todos os dados existentes são deslocados para baixo.
- **Pop:** A remoção de dados também ocorre na borda de subida do clock quando o sinal `stack_pop` está ativo. O dado do topo é removido e todos os dados abaixo são

Figura 2 – Waveform mostrando a operação da pilha com sequências de push e pop.

deslocados para cima, deixando o último registrador vazio.

2.2 Resultados da Simulação

A simulação do módulo `Stack` validou sua capacidade de armazenar e recuperar dados de forma correta e sequencial, conforme as operações de push e pop foram executadas.

Os testes mostram a funcionalidade da pilha ao demonstrar o armazenamento sequencial de dados e sua recuperação na ordem inversa, validando o princípio LIFO. Além disso, a manipulação de registros vazios e cheios foi gerenciada corretamente, prevenindo erros como underflow ou overflow não tratados.

2.3 Discussão

Os resultados da simulação confirmam que o módulo `Stack` é robusto e confiável para uso em ambientes que requerem gerenciamento rápido e eficiente de dados temporários. A implementação testada demonstrou alto grau de confiabilidade nas operações de inserção e remoção, sendo uma ferramenta adequada para tarefas que dependem de reversibilidade de dados e histórico de operações.

O desempenho observado sugere que o módulo pode ser integrado em sistemas maiores como parte de um subsistema de gerenciamento de memória ou como um buffer de operações em processadores.

2.4 Conclusões

Este segmento do projeto mostrou não apenas a viabilidade técnica da implementação de uma pilha em VHDL mas também reforçou a importância de estruturas de dados

clássicas no design de hardware moderno. A pilha implementada demonstrou flexibilidade e robustez, indicando sua utilidade em uma variedade de aplicações práticas.

3 Prog_cnt

3.1 Descrição da Implementação

O módulo `Prog_cnt` é um contador programável projetado para contar até 2047 (módulo 2048) com funcionalidades adicionais de carregamento e incremento controlados por entrada. Este contador oferece flexibilidade para ser utilizado em uma variedade de aplicações, desde simples temporizações até controles complexos de processo.

Funcionalidades chave incluem:

- **Incremento:** O contador incrementa automaticamente em cada borda de subida do clock se o controle estiver configurado para esta operação (`pc_ctrl` igual a "11").
- **Carregamento Direto:** Permite o carregamento de um valor específico (`new_pc_in`) diretamente no contador se `pc_ctrl` estiver configurado para "01".
- **Carregamento da Pilha:** Carrega valores do topo de uma pilha (`from_stack`) diretamente no contador se `pc_ctrl` estiver configurado para "10".

3.2 Resultados da Simulação

O módulo `Prog_cnt` foi submetido a uma série de simulações para validar suas múltiplas funções. As simulações confirmaram que o contador responde corretamente aos comandos de controle, incrementa de maneira precisa e carrega valores apropriados conforme especificado.

O waveform mostra as operações de incremento, assim como os carregamentos diretos e do topo da pilha, demonstrando a capacidade do contador de se adaptar rapidamente aos comandos de entrada enquanto mantém a precisão em sua contagem.

3.3 Discussão

A capacidade do `Prog_cnt` de operar em modos de incremento e de carregamento oferece uma ferramenta versátil para desenvolvimento de sistemas embutidos e circuitos de controle. A precisão e a resposta rápida às mudanças de entrada são particularmente valiosas para aplicações que requerem manipulação precisa de temporização e eventos baseados em contagem.

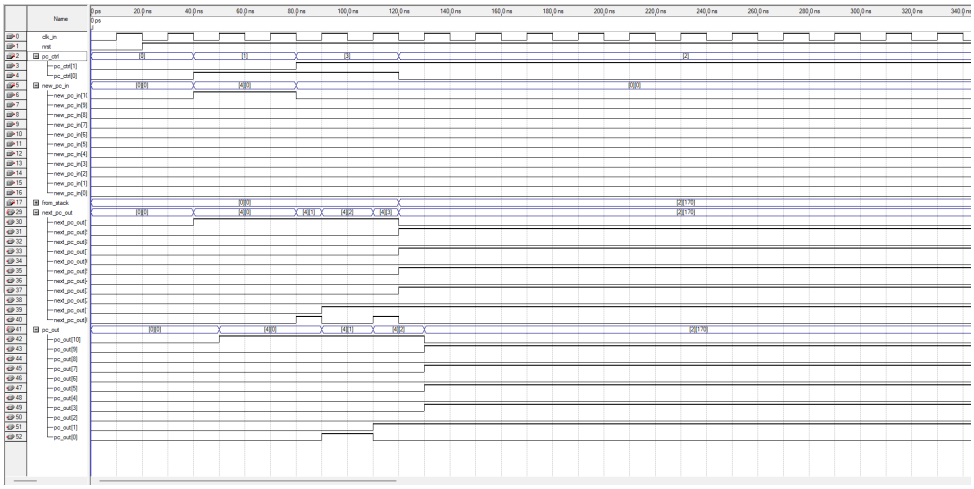

Figura 3 – Waveform demonstrando a funcionalidade do contador programável com várias operações de controle.

3.4 Conclusões

Os resultados das simulações apoiam a utilização do `Prog_cnt` em aplicações reais, onde a confiabilidade e a precisão são críticas. As funcionalidades implementadas permitem uma integração fácil em sistemas mais complexos, demonstrando o valor deste módulo como uma parte fundamental de sistemas digitais mais amplos.

Parte III

Implementação

1 Implementação do reg_bank

A implementação do `reg_bank` focou em fornecer um mecanismo eficiente e flexível para armazenamento e recuperação de dados. O design foi elaborado para permitir operações de leitura e escrita de forma sincronizada com o clock do sistema.

1.1 Código VHDL

O seguinte trecho de código VHDL ilustra a definição dos registradores e a lógica de controle para escrita e leitura:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RegBank is
    port (
        clk_in        : in  std_logic;
        nrst          : in  std_logic;
        regn_di       : in  std_logic_vector(7 downto 0);
        regn_wr_sel   : in  std_logic_vector(2 downto 0);
        regn_wr_ena   : in  std_logic;
        regn_rd_sel_a: in  std_logic_vector(2 downto 0);
        regn_rd_sel_b: in  std_logic_vector(2 downto 0);
        c_flag_in     : in  std_logic;
        z_flag_in     : in  std_logic;
        v_flag_in     : in  std_logic;
        c_flag_wr_ena: in  std_logic;
        z_flag_wr_ena: in  std_logic;
        v_flag_wr_ena: in  std_logic;
        regn_do_a     : out std_logic_vector(7 downto 0);
        regn_do_b     : out std_logic_vector(7 downto 0);
        c_flag_out    : out std_logic;
        z_flag_out    : out std_logic;
        v_flag_out    : out std_logic
    );
end RegBank;

architecture Behavioral of RegBank is
    type reg_array is array (0 to 7) of std_logic_vector(7 downto 0);
    signal registers : reg_array;
```

```
    signal c_flag, z_flag, v_flag : std_logic;
begin
    process (clk_in, nrst)
    begin
        if nrst = '0' then
            registers <= (others => (others => '0'));
            c_flag <= '0';
            z_flag <= '0';
            v_flag <= '0';
        elsif rising_edge(clk_in) then
            if regn_wr_ena = '1' then
                registers(to_integer(unsigned(regn_wr_sel))) <= regn_di;
            end if;
            if c_flag_wr_ena = '1' then
                c_flag <= c_flag_in;
            end if;
            if z_flag_wr_ena = '1' then
                z_flag <= z_flag_in;
            end if;
            if v_flag_wr_ena = '1' then
                v_flag <= v_flag_in;
            end if;
        end if;
    end process;

    regn_do_a <= registers(to_integer(unsigned(regn_rd_sel_a)));
    regn_do_b <= registers(to_integer(unsigned(regn_rd_sel_b)));

    c_flag_out <= c_flag;
    z_flag_out <= z_flag;
    v_flag_out <= v_flag;
end Behavioral;
```

1.2 Detalhes da Implementação

O módulo inclui sinais de controle que permitem a seleção específica do registrador para operações de leitura e escrita, garantindo assim a flexibilidade e a eficiência necessárias para aplicações que dependem de acesso rápido e confiável à memória de dados.

2 Implementação do Stack

O módulo `Stack` foi projetado para funcionar como uma pilha tradicional, com operações de push e pop.

2.1 Código VHDL

Aqui está um trecho que mostra a implementação do módulo `Stack`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Stack is
    port (
        clk_in      : in std_logic;
        nrst        : in std_logic;
        stack_in    : in std_logic_vector(10 downto 0);
        stack_push  : in std_logic;
        stack_pop   : in std_logic;
        stack_out   : out std_logic_vector(10 downto 0)
    );
end Stack;

architecture Behavioral of Stack is
    type stack_array is array (0 to 7) of std_logic_vector(10 downto 0);
    signal stack : stack_array := (others => (others => '0'));
begin
    process (clk_in, nrst)
    begin
        if nrst = '0' then
            stack <= (others => (others => '0'));
        elsif rising_edge(clk_in) then
            if stack_push = '1' and stack_pop = '0' then
                for i in 6 downto 0 loop
                    stack(i + 1) <= stack(i);
                end loop;
                stack(0) <= stack_in;
            elsif stack_pop = '1' and stack_push = '0' then

                for i in 0 to 6 loop
```

```
                stack(i) <= stack(i + 1);
            end loop;
            stack(7) <= (others => '0');
        end if;
    end if;
end process;

stack_out <= stack(0);
end Behavioral;
```

2.2 Detalhes da Implementação

Este módulo implementa um sistema de pilha clássico onde o último dado inserido é o primeiro a ser removido (LIFO). As operações de push e pop são controladas pelos sinais `stack_push` e `stack_pop`, respeitando as regras de sincronização com o sinal de clock do sistema.

3 Implementação do Prog_cnt

O módulo `Prog_cnt` é um contador avançado com capacidades de carregamento e incremento baseadas em sinais de controle.

3.1 Código VHDL

O código VHDL para `Prog_cnt` é mostrado a seguir:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Prog_cnt is
    port (
        clk_in      : in  std_logic;
        nrst        : in  std_logic;
        pc_ctrl     : in  std_logic_vector(1 downto 0);
        new_pc_in   : in  std_logic_vector(10 downto 0);
        from_stack  : in  std_logic_vector(10 downto 0);
        next_pc_out: buffer std_logic_vector(10 downto 0);
        pc_out      : out std_logic_vector(10 downto 0)
    );
```

```
end Prog_cnt;

architecture Behavioral of Prog_cnt is
    signal pc       : std_logic_vector(10 downto 0) := (others => '0');
begin

    with pc_ctrl select
        next_pc_out <= pc when "00",
                       new_pc_in when "01",
                       from_stack when "10",
                       std_logic_vector(unsigned(pc) + 1) when "11",
                       pc when others;

    process(clk_in, nrst)
    begin
        if nrst = '0' then
            pc <= (others => '0');
        elsif rising_edge(clk_in) then
            pc <= next_pc_out;
        end if;
    end process;

    pc_out <= pc;
end Behavioral;
```

3.2 Detalhes da Implementação

O contador programável `Prog_cnt` é capaz de operar em vários modos baseados no sinal de controle `pc_ctrl`, permitindo operações de incremento automático, carregamento direto de um valor externo, ou carregamento de um valor proveniente de uma pilha. Cada operação é sincronizada com o sinal de clock, garantindo a integridade dos dados e a precisão do contador.

Parte IV

Conclusões

1 Reflexões Gerais

Ao longo deste projeto, foram desenvolvidos e simulados três módulos distintos em VHDL: o `reg_bank`, o `Stack`, e o `Prog_cnt`. 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 reg_bank

O `reg_bank` provou ser extremamente confiável e eficiente, demonstrando a capacidade de gerenciar múltiplas operações de leitura e escrita de forma simultânea sem perder a integridade dos dados. Este módulo serve como um componente fundamental em sistemas que requerem acesso rápido e confiável a múltiplos registradores.

1.2 Conclusões sobre o Stack

O módulo `Stack` demonstrou ser robusto e versátil, facilitando operações de push e pop conforme o esperado em um ambiente LIFO. A pilha implementada é ideal para aplicações que necessitam de uma gestão temporária de dados, como em chamadas de função e controle de processos.

1.3 Conclusões sobre o Prog_cnt

O contador programável, `Prog_cnt`, destacou-se pela sua flexibilidade em atender a diferentes necessidades de contagem e carga através de sinais de controle externos. Esta capacidade permite a sua aplicação em uma ampla gama de cenários de uso, desde temporizadores simples até sistemas de controle complexos.

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.