Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Engenharia de Computação

Relatório: Trabalho Prático 2

Registradores em VHDL

Professor(es): Antônio Hamilton Magalhães

Aluno(s): Bruno Luiz Dias Alves de Castro
Rafael Ramos de Andrade

Belo Horizonte
Campus Coração Eucarístico

20 de novembro de 2024

Conteúdo

1 Introdução **3**
- 1.1 Objetivos . . . 3
  - 1.1.1 Registradores . . . 3
  - 1.1.2 Pilhas . . . 3

2 w_reg **4**
- 2.1 Implementação . . . 4
- 2.2 Simulação . . . 5

3 fsr_reg **6**
- 3.1 Implementação . . . 6
- 3.2 Simulação . . . 7

4 status_reg **8**
- 4.1 Implementação . . . 8
- 4.2 Simulação . . . 9

5 stack **11**
- 5.1 Implementação . . . 11
- 5.2 Simulação . . . 12

6 pc_reg **14**
- 6.1 Implementação . . . 14
- 6.2 Simulação . . . 16

7 Conclusão **17**

1 Introdução

Durante as aulas da disciplina de Sistemas Reconfiguráveis, fomos introduzidos à linguagem VHDL. VHDL (**V**HSIC **H**ardware **D**escription **L**anguage) é uma linguagem de descrição de hardware. Com ela, podemos montar circuitos lógicos de maneira totalmente textual, o que garante à linguagem uma grande vantagem ante à soluções visuais.

1.1 Objetivos

O objetivo deste segundo trabalho prático é a implementação de estruturas diversas de registradores e pilhas. Essas estruturas são cruciais na construção de circuitos complexos como controladores e processadores. Uma breve descrição de cada um é apresentada à seguir:

1.1.1 Registradores

Registradores são estruturas capazes de armazenar valores binários. São extremamente úteis na construção de circuitos, e é o que permite à eles se "lembrar" de dados relevantes.

Possuem três operações básicas: escrita, leitura e *reset*. Com o auxílio de um sinal de *clock*, mantém suas operações sincronizadas.

Pode ser usada por um processador para guardar valores entre ciclos de *clock*, ou para obter a próxima instrucão à ser executada (PC). Este último é implementado no último circuito (pc_reg), juntamente com uma pilha.

1.1.2 Pilhas

Pilhas são conjuntos de registradores com duas operações básicas: *Push* (Empilhamento) e *pop* (Desempilhamento). Ao receber um sinal de *Push*, o dado na entrada da pilha e adicionado ao primeiro registrador da estrutura. O dado que estava neste registrador é então movido para o segundo, e assim por diante.

O contrário ocorre na operação de *pop*. O dado no último registrador é movido para o penúltimo, e assim por diante. O dado no primeiro registrador é então colocado para fora da pilha, para ser consumido.

Ambas as operações acontecem de maneira síncrona, com o auxílio de um sinal de *clock*. Um efeito interessante é que, como os dados são empilhados através de uma regra FILO (**F**irst **I**n **L**ast **O**ut), isso tem o efeito colateral de inverter a ordem dos dados que foram inseridos. Por exemplo, ao empilhar a sequência: "1", "2" e "3", ao desempilhar toda a pilha, observarem o sequência: "3", ""2" e "1" na saída.

2 w_reg

O registrador w_reg é um registrador simples. Possui um barramento de dados para escrita, habilitada por um sinal wr_in.

As entradas e saídas do circuito são descritas na tabela a baixo:

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
d_in	8 bits	*Input*	Entrada de dados para escrita.
wr_en	1 bit	*Input*	Entrada de habilitação de escrita.
w_out	8 bits	*Output*	Saída de dados.

Tabela 1: Entradas e Saídas de fsr_reg

2.1 Implementação

O registrador w_reg foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  ENTITY w_reg IS
7      PORT (
8          -- Inputs
9          nrst : IN STD_LOGIC;                          -- Reset
10         clk_in: IN STD_LOGIC;                         -- Clock
11         d_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);        -- Dados
12         wr_en : IN STD_LOGIC;                         -- Enable
13
14         -- Outputs
15         w_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)      -- Dados
16     );
17 END ENTITY;
18
19 ARCHITECTURE w_reg OF w_reg IS
20     SIGNAL mem_reg: STD_LOGIC_VECTOR(7 DOWNTO 0);
21 BEGIN
22
23     PROCESS (nrst, clk_in)
24     BEGIN
25         IF nrst = '0' THEN                            -- reset
26             mem_reg <= "00000000";
27         ELSIF RISING_EDGE(clk_in) THEN
28             IF wr_en = '1' THEN                       -- write
29                 mem_reg <= d_in;
30             END IF;
31         END IF;
32     END PROCESS;
33
34     w_out <= mem_reg;                                 -- output
35
36 END w_reg;
```

Listing 1: Código VHDL w_reg

2.2 Simulação

Nesta imagem é realizado 3 testes para verificar a funcionalidade do registrador, nos primeiros 60ns é alterado os bits da entrada de dados (d_in) para nivel lógico alto, o bit de reset (nrst) que é ativo em baixa, é desativado, ou seja, nível lógico alto e o bit de ativação (wr_en) é colocoado em nível lógico alto após 10ns. Assim é possível verificar a mudança na saída (w_out) com um tempo de delay de 6ns. No segundo teste a partir de 60ns até 140ns é resetado os bits da memória do registrador colocando reset em nível lógico zero, o resultado é propagada para a saída após o tempo de delay de aproximadamente 6ns. No terceiro teste foi verificados se o bit de ativação de escrita está funcionando corretamente, portanto com o bit 6 da saída em nível lógico alto esse valor será escrito apenas no tempo 160ns quando é colocado a porta de ativação do registrador em nível lógico alto e o registrador é escrito.

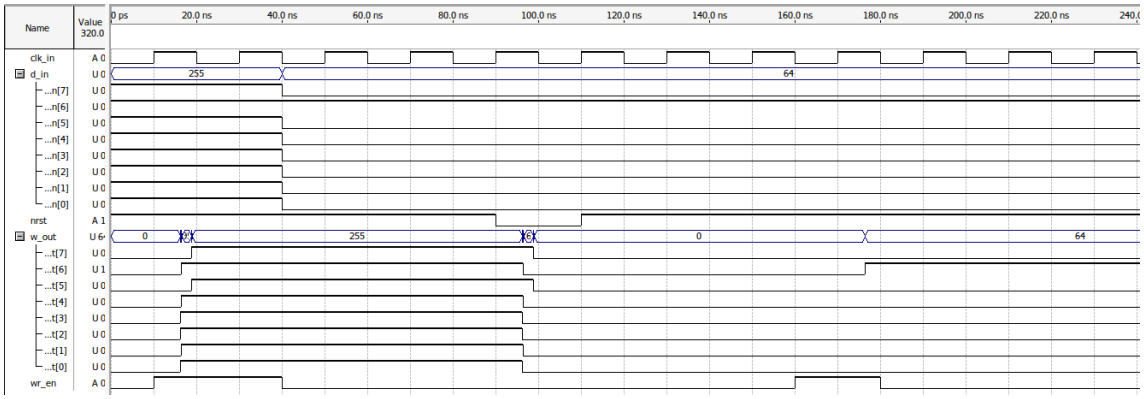

Figura 1: Simulação bloco w_reg

3 fsr_reg

O registrador FSR é um registrador semelhante ao implementado anteriormente. A principal diferença entre os dois está na presença de um sistema de endereçamento, e de duas entradas binárias independentes para habilitação da escrita e da leitura. Os requisitos são descritos na tabela abaixo.

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
abus_in	9 bit	*Input*	Entrada de enderençamento.
dbus_in	8 bits	*Input*	Entrada de dados para escrita.
wr_en	1 bit	*Input*	Entrada de habilitação de escrita.
rd_en	1 bit	*Input*	Entrada de habilitação de leitura.
dbus_out	8 bits	*Output*	Saída de dados hailitada por rd_en.

Tabela 2: Entradas e Saídas de fsr_reg

3.1 Implementação

O registrador fsr_reg foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  ENTITY fsr_reg IS
7      PORT (
8          -- Inputs
9          nrst : IN STD_LOGIC;                                -- Reset
10         clk_in: IN STD_LOGIC;                               -- Clock
11         abus_in: IN STD_LOGIC_VECTOR(8 DOWNTO 0);           -- Enderecamento
12         dbus_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);           -- Dados
13         wr_en : IN STD_LOGIC;                               -- Enable escrita
14         rd_en : IN STD_LOGIC;                               -- Enable leitura
15
16         -- Outputs
17         dbus_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);        -- Dados
18         fsr_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)          -- Registrador
19     );
20 END ENTITY;
21
22 ARCHITECTURE fsr_reg OF fsr_reg IS
23     SIGNAL mem_reg: STD_LOGIC_VECTOR(7 DOWNTO 0);
24 BEGIN
25     PROCESS (nrst, clk_in, mem_reg, abus_in, dbus_in)
26     BEGIN
27         IF nrst = '0' THEN                                  -- reset
28             mem_reg <= "00000000";
29         ELSIF abus_in(6 DOWNTO 0) = "0000100" THEN
30             IF RISING_EDGE(clk_in) THEN
31                 IF wr_en = '1' THEN                         -- write
32                     mem_reg <= dbus_in;
33                 END IF;
34             END IF;
35         END IF;
36     END PROCESS;
37
38     dbus_out <= mem_reg WHEN rd_en = '1' ELSE "ZZZZZZZZ"; -- read
39     fsr_out <= mem_reg;
40 END fsr_reg;
```

Listing 2: Código VHDL fsr_reg

3.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quatus II.

Os testes realizados foram os seguites:

1. Escrita com enderaçamento incorreto (diferente de XX0000100).
   - **Comportamento esperado:**
     - dbus_out em alta impedância;
     - fsr_out sem alteração;
2. Leitura habilitada e escrita desabilitada.
   - **Comportamento esperado:**
     - dbus_out = frs_out = último valor escrito;
3. Leitura desabilitada e escrita habilitada.
   - **Comportamento esperado:**
     - dbus_out em alta impendância;
     - frs_out = dbus_in;
4. *Reset* com leitura habilitada.
   - **Comportamento esperado:**
     - dbus_out = frs_out = “0b00000000”;

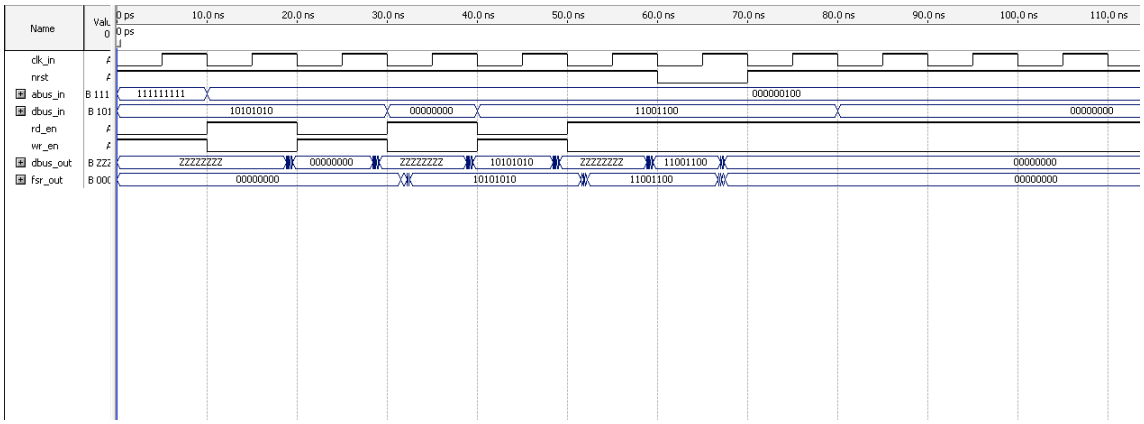

Figura 2: Simulação fsr_reg

4 status_reg

O status é um registrador semelhante ao implementado anteriormente. A diferença esta na presença de sinais de entrada e saída para controlar bits específicos. Assim como o anterior, existe um sistema de endereçamento que deve ser conferido para alterar o registrador.

As entradas e saídas do circuito são descritas na tabela a baixo:

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
abus_in	9 bit	*Input*	Entrada de enderençamento.
dbus_in	8 bits	*Input*	Entrada de dados para escrita.
wr_en	1 bit	*Input*	Entrada de habilitação de escrita.
rd_en	1 bit	*Input*	Entrada de habilitação de leitura.
z_in	1 bit	*Input*	Entrada de dado para escrita no bit 2 do registrador.
dc_in	1 bit	*Input*	Entrada de dado para escrita no bit 1 do registrador.
c_in	1 bit	*Input*	Entrada de dado para escrita no bit 0 do registrador.
z_wr_en	1 bit	*Input*	Entrada para habilitação da escrita no bit 2 do registrador.
dc_wr_en	1 bit	*Input*	Entrada para habilitação da escrita no bit 1 do registrador.
c_wr_en	1 bit	*Input*	Entrada para habilitação da escrita no bit 0 do registrador.
dbus_out	8 bits	*Output*	Saída de dados hailitada por rd_en.
irp_out	1 bit	*Output*	Saída correspondente ao bit 7 do registrador.
rp_out	2 bits	*Output*	Saída correspondente aos bits 6 e 5 do registrador.
z_out	1 bit	*Output*	Saída correspondente ao bit 2 do registrador.
dc_out	1 bit	*Output*	Saída correspondente ao bit 1 do registrador.
c_out	1 bit	*Output*	Saída correspondente ao bit 0 do registrador.

Tabela 3: Entradas e Saídas de status_reg

4.1 Implementação

O status_reg foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.all;

ENTITY status_reg IS
    PORT (
        -- Inputs
        nrst: IN STD_LOGIC;                             -- Reset
        clk_in: IN STD_LOGIC;                           -- Clock
        abus_in: IN STD_LOGIC_VECTOR(8 DOWNTO 0);       -- Enderecamento
        dbus_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);       -- Dados
        wr_en: IN STD_LOGIC;                            -- Enable escrita
        rd_en: IN STD_LOGIC;                            -- Enable leitura
        z_in: IN STD_LOGIC;                             -- Dados bit 2
        dc_in: IN STD_LOGIC;                            -- Dados bit 1
        c_in: IN STD_LOGIC;                             -- Dados bit 0
        z_wr_en: IN STD_LOGIC;                          -- Enable escrita bit 2
        dc_wr_en: IN STD_LOGIC;                         -- Enable escrita bit 1
        c_wr_en: IN STD_LOGIC;                          -- Enable escrita bit 0

        -- Outputs
        dbus_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);     -- Dados
        irp_out: OUT STD_LOGIC;                         -- Dados bit 7
        rp_out: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);       -- Dados bit 6 e 5
```

```
        z_out: OUT STD_LOGIC;                           -- Dados bit 2
        dc_out: OUT STD_LOGIC;                          -- Dados bit 1
        c_out: OUT STD_LOGIC                            -- Dados bit 0
    );
END ENTITY;

ARCHITECTURE status_reg OF status_reg IS
    SIGNAL mem_reg: STD_LOGIC_VECTOR(7 downto 0);
BEGIN
    PROCESS(nrst, clk_in, mem_reg, wr_en, z_in, dc_in, c_in)
    BEGIN
        IF nrst = '0' THEN                              -- reset
            mem_reg <= "00000000";
        ELSIF RISING_EDGE(clk_in) THEN
            IF wr_en = '1' AND abus_in(6 DOWNTO 0) = "0000011" THEN
                mem_reg <= dbus_in;                     -- write
            END IF;
            IF z_wr_en = '1' THEN
                mem_reg(2) <= z_in;
            END IF;
            IF dc_wr_en = '1' THEN
                mem_reg(1) <= dc_in;
            END IF;
            IF c_wr_en = '1' THEN
                mem_reg(0) <= c_in;
            END IF;
        END IF;
    END PROCESS;

    -- output
    dbus_out <= mem_reg WHEN rd_en = '1' AND abus_in(6 DOWNTO 0) = "0000011" ELSE "ZZZZZZZZ";
    irp_out <= mem_reg(7);
    rp_out <= mem_reg(6 DOWNTO 5);
    z_out <= mem_reg(2);
    dc_out <= mem_reg(1);
    c_out <= mem_reg(0);
END status_reg;
```

Listing 3: Código VHDL status_reg

4.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quatus II.

Os testes realizados foram os seguites:

1. Escrita com enderaçamento incorreto (diferente de “0bXX0000011).
   - **Comportamento esperado:**
     - dbus_out em alta impedância;
     - rp_out = “0b00”
     - irp_out = z_out = dc_out = c_out = “0b0”
2. Leitura desabilitada e escrita habilitada;
   - dbus_in = “0b01011000”.
   - z_in = dc_in = c_in = 1;
   - z_wr_en = dc_wr_en = c_wr_en = 0;
   - **Comportamento esperado:**
     - dbus_out em alta impendância;
     - rp_out = “10b”
     - irp_out = z_out = dc_out = c_out = “0b0”
3. Leitura habilitada e escrita desabilitada.

- Valor salvo = “0b01011000”.
- **Comportamento esperado:**
  - dbus_out = “0b01011000”;
  - rp_out = “10b”
  - irp_out = z_out = dc_out = c_out = “0b0”

4. Leitura desabilitada e escrita habilitada;
   - dbus_in = “0b10100000”.
   - z_in = dc_in = c_in = 1;
   - z_wr_en = dc_wr_en = c_wr_en = 1;
   - **Comportamento esperado:**
     - dbus_out em alta impendância;
     - rp_out = “01b”
     - irp_out = z_out = dc_out = c_out = “0b1”
5. Leitura habilitada e escrita desabilitada.
   - Valor salvo = “0b10100111”.
   - **Comportamento esperado:**
     - dbus_out = “0b10100111”;
     - rp_out = “10b”
     - irp_out = z_out = dc_out = c_out = “0b1”
6. *Reset* com leitura habilitada.
   - **Comportamento esperado:**
     - dbus_out = “0b00000000”;
     - rp_out = “00b”
     - irp_out = z_out = dc_out = c_out = “0b0”

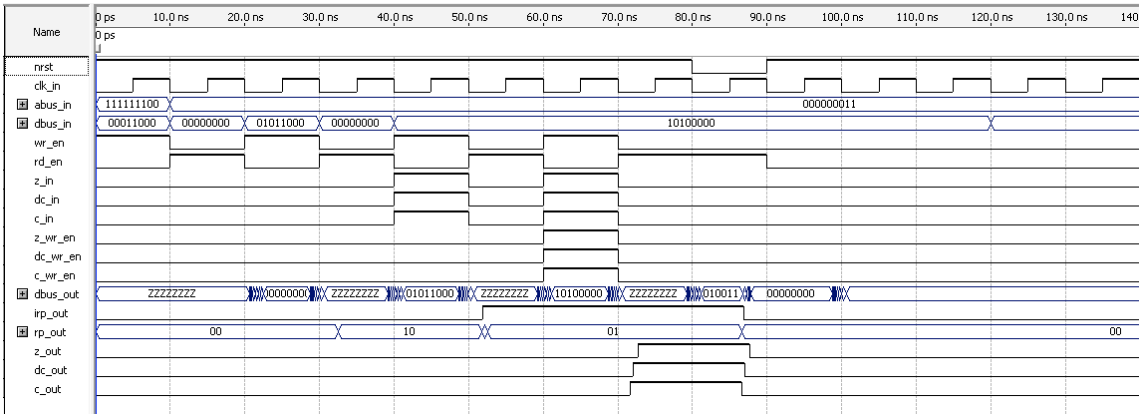

Figura 3: Simulação status_reg

5 stack

O bloco **stack** é um conjunto de 8 registradores de 13 bits. As operações de *push* e *pop* adicionam e removem dados da pilha, respectivamente.

As entradas e saídas deste bloco estão descritas na tabela abaixo:

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
stack_in	13 bit	*Input*	Entrada de dados para a pilha.
stack_push	1 bit	*Input*	Entrada de habilitação para colocar valores na pilha.
stack_pop	1 bit	*Input*	Entrada de habilitação para retirar valores da pilha.
stack_out	13 bits	*Output*	Saída correspondente à primeira posição da pilha.

Tabela 4: Entradas e Saídas do bloco stack

5.1 Implementação

O bloco stack foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  ENTITY stack IS
7      PORT (
8          -- Inputs
9          nrst: IN STD_LOGIC;                                  -- Reset
10         clk_in: IN STD_LOGIC;                                -- Clock
11         stack_in: IN STD_LOGIC_VECTOR(12 DOWNTO 0);          -- Dados
12         stack_push: IN STD_LOGIC;                            -- Enable push op
13         stack_pop: IN STD_LOGIC;                             -- Enable pop op
14
15         -- Outputs
16         stack_out: OUT STD_LOGIC_VECTOR(12 DOWNTO 0)         -- Stack output
17     );
18 END ENTITY;
19
20 ARCHITECTURE stack OF stack IS
21     SIGNAL mem_reg1, mem_reg2, mem_reg3, mem_reg4, mem_reg5, mem_reg6, mem_reg7, mem_reg8 :
       STD_LOGIC_VECTOR(12 DOWNTO 0);
22 BEGIN
23     PROCESS(nrst, clk_in, stack_push, stack_pop)
24     BEGIN
25         IF nrst = '0' THEN                                   -- registradores
26             mem_reg1 <= "0000000000000";
27             mem_reg2 <= "0000000000000";
28             mem_reg3 <= "0000000000000";
29             mem_reg4 <= "0000000000000";
30             mem_reg5 <= "0000000000000";
31             mem_reg6 <= "0000000000000";
32             mem_reg7 <= "0000000000000";
33             mem_reg8 <= "0000000000000";
34         ELSIF RISING_EDGE(clk_in) THEN
35       stack_out <= "0000000000000";
36             IF stack_pop = '1' THEN                          -- pop
37                 stack_out <= mem_reg1;                       -- output
38                 mem_reg1 <= mem_reg2;
39                 mem_reg2 <= mem_reg3;
40                 mem_reg3 <= mem_reg4;
41                 mem_reg4 <= mem_reg5;
```

```
42                  mem_reg5 <= mem_reg6;
43                  mem_reg6 <= mem_reg7;
44                  mem_reg7 <= mem_reg8;
45                  mem_reg8 <= "0000000000000";
46              ELSIF stack_push = '1' THEN                  -- push
47                  mem_reg8 <= mem_reg7;
48                  mem_reg7 <= mem_reg6;
49                  mem_reg6 <= mem_reg5;
50                  mem_reg5 <= mem_reg4;
51                  mem_reg4 <= mem_reg3;
52                  mem_reg3 <= mem_reg2;
53                  mem_reg2 <= mem_reg1;
54                  mem_reg1 <= stack_in;
55              END IF;
56          END IF;
57      END PROCESS;
58  END stack;
```

Listing 4: Código VHDL stack

5.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quatus II.

Os testes realizados foram os seguites:

1. *Push* até pilha cheia.
   - stack_in: Sequência de “0” à “7”;
   - stack_push = “1”;
   - stack_pop = “0”;
   - **Comportamento esperado:**
     - stack_out = “0”;
2. *Pop* até pilha vazia.
   - stack_push = “0”;
   - stack_pop = “1”;
   - **Comportamento esperado:**
     - stack_out = Sequência de “7” à “0”;
3. *Push* até *Stack Overvlow*.
   - stack_in: Sequência de “0” à “9”;
   - stack_push = “1”;
   - stack_pop = “0”;
   - **Comportamento esperado:**
     - stack_out = “0”;
4. *Pop* até *Stack Underflow*.
   - stack: Sequência de “9” à “2”;
   - stack_push = “0”;
   - stack_pop = “1”;
   - **Comportamento esperado:**
     - stack_out = Sequência de “9” à “2”, depois, “0”;

5. *Push* e *Pop* simultâneo.

   - stack: “1”;
   - stack_push = “1”;
   - stack_pop = “1”;
   - **Comportamento esperado:**
     - Preferência do *pop*.
     - stack_out = “1”, depois, “0”;

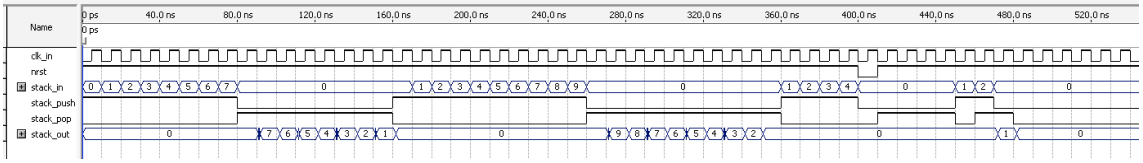

Figura 4: Simulação bloco stack

6 pc_reg

O bloco pc_reg controla o registrados pc de um processador. Possui operações para incremento de endereço, *load*, *reset* e escrita, bem como operações de *push* e *pop* adicionam e removem o valor corrente de pc à uma pilha pilha.

As entradas e saídas deste bloco estão descritas na tabela abaixo:

Nome	Tamanho	Tipo	Descrição
nrst	1 bit	*Input*	Entrada de *reset* assíncrono.
clk_in	1 bit	*Input*	Entrada de *clock*.
aadr_in	11 bits	*Input*	Entrada de dados para carga no registrador PC.
abus_in	9 bits	*Input*	Entrada de endereçamento para PCL e para o registrador PCLATH.
dbus_in	8 bits	*Input*	Entrada de dados para escrita em PCL e PCLATH.
inc_pc	1 bit	*Input*	Entrada de habilitação para incremento.
load_pc	1 bit	*Input*	Entrada de habilitação para carga.
wr_en	1 bit	*Input*	Entrada de habilitação para escrita nos registradores.
rd_en	1 bit	*Input*	Entrada de habilitação para leitura dos registradores.
stack_push	1 bit	*Input*	Entrada de habilitação para colocar valores na pilha.
stack_pop	1 bit	*Input*	Entrada de habilitação para retirar valores da pilha.
nextpc_out	13 bits	*Output*	Saída do valor a ser carregado no contador.
dbus_out	8 bits	*Output*	Saída de dados lidos com endereçamento por abus_in.

Tabela 5: Entradas e Saídas do bloco pc_reg

6.1 Implementação

O bloco pc_reg foi implementado utilizando a linguagem VHDL.

O código na íntegra está abaixo:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4  USE ieee.numeric_std.all;
5
6  ENTITY pc_reg IS
7      PORT (
8          -- Inputs
9          nrst: IN STD_LOGIC;                             -- Reset
10         clk_in: IN STD_LOGIC;                           -- Clock
11         addr_in: IN STD_LOGIC_VECTOR(10 DOWNTO 0);      -- Dados
12         abus_in: IN STD_LOGIC_VECTOR(8 DOWNTO 0);       -- Enderecamento PCL e PCLATH
13         dbus_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);       -- Dados PCL e PCLATH
14         inc_pc: IN STD_LOGIC;                           -- Enable incremento.
15         load_pc: IN STD_LOGIC;                          -- Enable carga.
16         wr_en: IN STD_LOGIC;                            -- Enable escrita.
17         rd_en: IN STD_LOGIC;                            -- Enable leitura.
18         stack_push: IN STD_LOGIC;                       -- Enable push op
19         stack_pop: IN STD_LOGIC;                        -- Enable pop op
20
21         -- Outputs
22         nextpc_out: OUT STD_LOGIC_VECTOR(12 DOWNTO 0);  -- Contador
23         dbus_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)      -- Dados
24     );
25 END ENTITY;
26
27 ARCHITECTURE pc_reg OF pc_reg IS
28     SIGNAL stack_reg1, stack_reg2, stack_reg3, stack_reg4, stack_reg5, stack_reg6, stack_reg7,
        stack_reg8 : STD_LOGIC_VECTOR(12 DOWNTO 0);
29     SIGNAL stack_popped: STD_LOGIC_VECTOR(12 DOWNTO 0);
30     SIGNAL pc: STD_LOGIC_VECTOR(12 DOWNTO 0);
```

```
    SIGNAL lath_pc: STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL nextpc: STD_LOGIC_VECTOR(12 DOWNTO 0);
BEGIN

    -- Stack
    PROCESS(nrst, clk_in, stack_push, stack_pop, stack_popped)
    BEGIN
        IF nrst = '0' THEN
            stack_reg1 <= "0000000000000";
            stack_reg2 <= "0000000000000";
            stack_reg3 <= "0000000000000";
            stack_reg4 <= "0000000000000";
            stack_reg5 <= "0000000000000";
            stack_reg6 <= "0000000000000";
            stack_reg7 <= "0000000000000";
            stack_reg8 <= "0000000000000";
        ELSIF RISING_EDGE(clk_in) THEN
            IF stack_pop = '1' THEN
                stack_popped <= stack_reg1;
                stack_reg1 <= stack_reg2;
                stack_reg2 <= stack_reg3;
                stack_reg3 <= stack_reg4;
                stack_reg4 <= stack_reg5;
                stack_reg5 <= stack_reg6;
                stack_reg6 <= stack_reg7;
                stack_reg7 <= stack_reg8;
                stack_reg8 <= "0000000000000";
            ELSIF stack_push = '1' THEN
                stack_reg8 <= stack_reg7;
                stack_reg7 <= stack_reg6;
                stack_reg6 <= stack_reg5;
                stack_reg5 <= stack_reg4;
                stack_reg4 <= stack_reg3;
                stack_reg3 <= stack_reg2;
                stack_reg2 <= stack_reg1;
                stack_reg1 <= pc;
            END IF;
        END IF;
    END PROCESS;

    -- logica combinacional para nextpc
    PROCESS(stack_pop, inc_pc, load_pc, wr_en, abus_in, pc, addr_in, lath_pc, stack_popped,
    dbus_in)
    BEGIN
        IF stack_pop = '1' THEN
            nextpc <= stack_popped;
        ELSIF inc_pc = '1' THEN
            nextpc <= pc + 1;
        ELSIF load_pc = '1' THEN
            nextpc(10 DOWNTO 0) <= addr_in;
            nextpc(12 DOWNTO 11) <= lath_pc(4 DOWNTO 3);
        ELSIF wr_en = '1' AND abus_in(6 DOWNTO 0) = "0000010" THEN
            nextpc <= lath_pc(4 DOWNTO 0) & dbus_in;
        ELSE
            nextpc <= pc;
        END IF;

        nextpc_out <= nextpc;
    END PROCESS;

    -- logica sequencial para PC_reg
    PROCESS(clk_in, nrst, pc, nextpc)
    BEGIN
        IF RISING_EDGE(clk_in) THEN
            pc <= nextpc;
        END IF;

        IF nrst = '0' THEN
            pc <= "0000000000000";
        END IF;
    END PROCESS;

    -- logica sequencial para PCLATH
    PROCESS(clk_in, nrst, wr_en, abus_in)
    BEGIN
        IF RISING_EDGE(clk_in) THEN
```

```
            IF wr_en = '1' AND abus_in(6 DOWNTO 0) = "0001010" THEN
                lath_pc <= dbus_in;
            END IF;
        END IF;

        IF nrst = '0' THEN
            lath_pc <= "00000000";
        END IF;
    END PROCESS;

    -- logica combinacional para dbus_out
    PROCESS(clk_in, rd_en, abus_in, lath_pc, pc)
    BEGIN
        IF abus_in(6 DOWNTO 0) = "0001010" AND rd_en = '1' THEN
            dbus_out <= lath_pc(7 DOWNTO 0);
        ELSIF abus_in(6 DOWNTO 0) = "0000010" AND rd_en = '1' THEN
            dbus_out <= pc(7 DOWNTO 0);
        ELSE
            dbus_out <= "ZZZZZZZZ";
        END IF;
    END PROCESS;

END pc_reg;
```

Listing 5: Código VHDL pc_reg

6.2 Simulação

Para testar nosso código VHDL e certificar-nos de que nosso circuito funciona de maneira esperada, simulamos alguns casos de testes utilizando o software Quatus II.

Os testes realizados foram os seguites:

1. Incremento de PC (4x).
   - inc_pc: *High* (4x);
   - **Comportamento esperado:**
     - nextpc_out: Sequência de "0x1" à "0x4";
     - dbus_out: Alta impedância;
2. *Reset* do PC.
   - nrst: *High*;
   - **Comportamento esperado:**
     - nextpc_out: "0x00";
     - dbus_out: Alta impedância;
3. Empilhamento e Desempilhamento de PC ("0x0" e "0x2").
   - stack_push: *High* (2x) e *low* (2x);
   - stack_pop: *Low* (2x) e *high* (2x);
   - **Comportamento esperado:**
     - nextpc_out: "0x02" e "0x00", nesta ordem;
     - dbus_out: Alta impedância;
4. Escrita e leitura de PCL ("0x0" e "0x2").
   - abus_in: "0x002";
   - dbus_in: "0x0A"
   - wr_en: *High* na escrita. *low* na leitura;

- rd_en: *Low* na escrita. *high* na leitura;
- **Comportamento esperado:**
  - nextpc_out: “0x?0A”;
  - dbus_out: “0x0A”;

5. Carregamento de endereço.
   - addr_in: “0x3AB”;
   - load_pc: *High*;
   - **Comportamento esperado:**
     - nextpc_out: “0x3AB”;
     - dbus_out: Alta impedância;
6. Escrita e leitura de PCLATH (“0x0” e “0x2”).
   - abus_in: “0x00A”;
   - dbus_in: “0x18”;
   - wr_en: *High* na escrita. *low* na leitura;
   - rd_en: *Low* na escrita. *high* na leitura;
   - **Comportamento esperado:**
     - nextpc_out: “0x?18”;
     - dbus_out: “0x0A”;

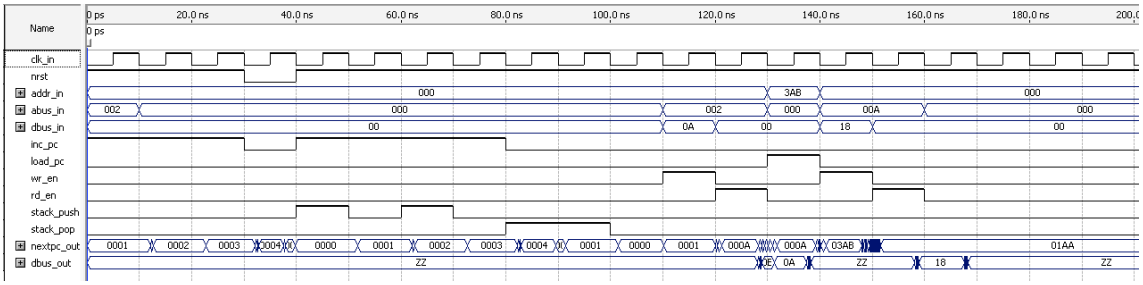

Figura 5: Simulação bloco pc_reg

7 Conclusão

A implementação de registradores através de liguagens de descrição de hardware, como a VHDL, são maneiras poderosas de construir circuitos computacionais de maneira menos complexa e mais rápida.

Apesar de parecerem “simples” à primeira vista, sua implementação possuem nuancias que exigem a atenção do engenheiro, e testes exaustivos para certificar que funcionam como deveriam.

Neste trabalho prático, aprendemos como construir registradores e pilhas para auxiliar a contrução de nossos circuitos computacionais, utilizando conhecimentos da prática anterior e novas técnicas, como a utilização do “PROCESS”, que permite a execução de trechos sequenciais no nosso circuito, e a tuilização do *clock*, que permite a sincronização de rotinas no nosso cicuito.

Com os circuitos implementados, estamos mais confiantes nas nossas capacidades, e estamos um passo mais perto de implementar circuitos mais complexos como controladores e processadores.