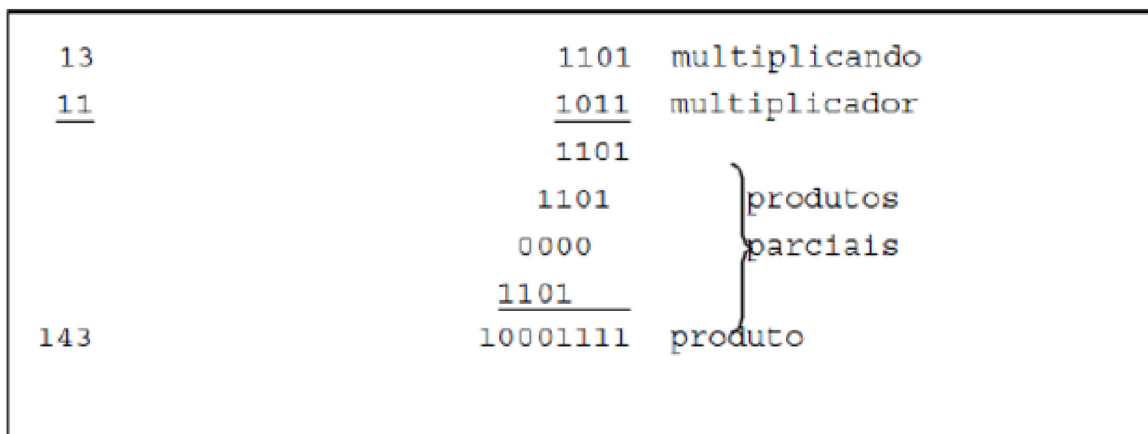
Prática V

Introdução

Nesta aula iremos programar o código de um multiplicador de 4 bits contido no livro de Ordonez et al. (2003), encontrar o erro do código de multiplicação de 4 bits que consta no livro de Ordonez et al.(2003), e por fim, escrevermos nosso próprio código de multiplicação. Para assim, compará-los e identificar o melhor desempenho entre eles. Antes de iniciarmos, descreve-se abaixo uma pequena definição do Multiplicador Binário segundo Midorikawa (2001), que diz:³O Multiplicador Binário é responsável pela realização de uma multiplicação de dois números binários, ele é considerado uma das operações mais custosas em um computador digital.Basicamente, O algoritmo tradicional de multiplicação de dois números binários sem sinal écomposto por sucessivos deslocamentos do multiplicando à esquerda (que constituem as parcelas do cálculo dos produtos parciais) e uma soma´, conforme a figura 1:

Figura 1: Funcionamento do multiplicador binário.



Midorikawa (2001) explica a execução do algoritmo que leva em consideração um bit domultiplicador de cada vez, com o bit menos significativo em primeiro lugar. Se o bit domultiplicador for 1, o multiplicando é copiado para ser somado posteriormente. Em casocontrário, o bit do multiplicador for 0, um valor nulo é copiado em seu lugar. Os númeroscopiados em linhas sucessivas são deslocados à esquerda de uma posição em relação à linhaanterior. Finalmente, os números são somados gerando o produto final.

1.1 Código de Multiplicação Binaria de Ordonez

```
library ieee;
use ieee.std_logic_1164.all;
entity mult4bits is
        port( a,b: in std_logic_vector(3 downto 0);
        s: out std_logic_vector(3 downto 0));
end mult4bits;
architecture arch_multi4bits of mult4bits is
-- função deslocamento de 1 bit para a esquerda, zerando o bit menos significativo;
        function deslocador(x: std_logic_vector(3 downto 0))
        return std_logic_vector is
        variable y: std_logic_vector(3 downto 0);
        begin
                for i in 3 downto 1 loop
                        y(i):=x(i-1);
                end loop;
                y(0):='0';
                return y;
        end;
-- Somador de 4 bits
                function somador4bits(a: std_logic_vector(3 downto 0);
```

```
                        b: std_logic_vector(3 downto 0))
        return std_logic_vector is
        variable vaium: std_logic;
        variable soma: std_logic_vector(3 downto 0);
        begin
                vaium:='0';
                for i in 0 to 3 loop
                soma(i):=a(i) xor b(i) xor vaium;
                vaium := (a(i) and b(i)) or (b(i) and vaium) or (vaium and a(i));
        end loop;
        return soma;
        end;
        begin
                process(a,b)
                variable aux1: std_logic_vector(3 downto 0);
                variable aux2: std_logic_vector(3 downto 0);
                variable vaium: std_logic;
                begin
                        -- inicializações
                        aux1:= "0000";
                        aux2:= a;
                        vaium:='0';

                        -- implementação do algoritmo
                        for i in 0 to 3 loop
                        aux1:=deslocador(aux1);
                        vaium:=aux2(3);
                        if vaium = '1' then
                                aux1:=somador4bits(aux1,b);
                        end if;
                        aux2:=deslocador(aux2);
                end loop;
                s<=aux1;
        end process;
    end arch_multi4bits;
```

Descobrimos que o tamanho do vetor das entradas, saídas, variáveis auxiliares, da função deslocamento e somador não comportavam as operações realizadas na multiplicação, ou seja, as operações eram realizadas, mas, não obtêm o resultado completo por conta que a execução prioriza os 4 bits existentes do multiplicador, mas, não prove o overflow ocasionado pelo deslocamento das operações de multiplicação e a soma da mesma. Como Midorikawa(2001) afirma [3]Como são somadas n produtos parciais de n bits cada uma, é necessário um circuito somador com 2n bits. Isto representa uma questão que deve ser levado em conta durante o projeto de um circuito de multiplicação binária´. Além, dos deslocamentos ocorridos. Abaixo o código implementado está funcionando corretamente.

```
library ieee;
use ieee.std_logic_1164.all;
entity multi4bits is
        port( a,b: in std_logic_vector(7 downto 0);
        s: out std_logic_vector(7 downto 0));
end multi4bits;
architecture arch_multi4bits of multi4bits is

-- função deslocamento de 1 bit para a esquerda, zerando o bit menos significativo;
        function deslocador(x: std_logic_vector(7 downto 0))
        return std_logic_vector is
        variable y: std_logic_vector(7 downto 0);
        begin
                for i in 7 downto 1 loop
                        y(i):=x(i-1);
                end loop;
                        y(0):='0';
                return y;
        end;

-- Somador de 8 bits
```

```
function somador8bits(a: std_logic_vector(7 downto 0);
                      b: std_logic_vector(7 downto 0))
return std_logic_vector is
variable vaium: std_logic;
variable soma: std_logic_vector(7 downto 0);
begin
        vaium:='0';
        for i in 0 to 7 loop
        soma(i):=a(i) xor b(i) xor vaium;
        vaium := (a(i) and b(i)) or (b(i) and vaium) or (vaium and a(i));
end loop;
return soma;
end;
begin
        process(a,b)
        variable aux1: std_logic_vector(15 downto 0);
        variable aux2: std_logic_vector(7 downto 0);
        variable vaium: std_logic;
        begin
                -- inicializações
                aux1:="0000000000000000";
                aux2:= a;
                vaium:='0';

                -- implementação do algoritmo
                for i in 0 to 7 loop
                aux1:=deslocador(aux1);
                vaium:=aux2(3);
                if vaium = '1' then
                        aux1:=somador8bits(aux1,b);
                end if;
                aux2:=deslocador(aux2);
        end loop;
        s<=aux1;
end process;
end arch_multi4bits;
```

1.2 Código do Multiplicador de 4 bits diferenciado do livro de Ordonez

Esse código de multiplicação teve por base o algoritmo de Mano & Kime "Logic and Computer Design Fundamentals" e foi encontrado no artigo de Midorikawa (2001). Midorikawa explana o desenvolvimento do multiplicador binário, descrito a seguir. A operação do circuito deve seguir os seguintes passos:1. Acertar um valor binário na vida de dados de entrada (IN);2. Ativar o sinal ENTRA_MULTIPLICANDO;3. Colocar outro valor na via de dados de entrada;4. Ativar o sinal ENTRA_MULTIPLICADOR;5. Acionar o botão INICIAR para a execução da multiplicação binária;6. Verificar resultado na via de dados de saída (OUT).Primeiramente, são definidas as entradas e as saídas do multiplicador. Após, os sinais internos que gerarão registradores: estado e prox_estado para o controle, os registradores A, B, P e Q e o flip-flop C e Z. O primeiro comando de atribuição força o sinal Z a ter valor 1 quando P contiver valor 0. O segundo comando de atribuição atribui a saída concatenada dos registradores A e Q para a saída do multiplicador MULT_OUT. Isto é necessário para fazer com que os sinais A e Q possam ser usados internamente no circuito. O restante da descrição envolve três processos. O primeiro processo descreve o funcionamento do registrador de estado e inclui o processamento de um sinal RESET e de clock. O segundo processo descreve a determinação do próximo estado do circuito com os sinais INICIAR, ZERO e estado. O terceiro processo descreve a função do fluxo de dados. Abaixo está o código

```
-- Multiplicador binário com n=4: descrição VHDL
-- adaptado de Mano & Kime "Logic and Computer Design Fundamentals"
-- Prentice-Hall, 2nd edition, 2000
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;
entity multiplicador_binario is
		port(CLK, RESET, INICIAR, ENTRA_MULTIPLICANDO, ENTRA_MULTIPLICADOR: in std_logic;
				MULT_IN: in std_logic_vector(3 downto 0);
				MULT_OUT: out std_logic_vector(7 downto 0) );
end multiplicador_binario;

architecture comportamento_mult4 of multiplicador_binario is
		type tipo_estado is (PARADO, MUL0, MUL1);
		signal estado, prox_estado : tipo_estado;
		signal A, B, Q: std_logic_vector(3 downto 0);
		signal P: std_logic_vector(1 downto 0);
		signal C, ZERO: std_logic;
begin
		ZERO <= P(1) NOR P(0);
		MULT_OUT <= A & Q;
		registra_estado: process (CLK, RESET)
		begin
		 if (RESET = '1') then
				estado <= PARADO;
				elsif (CLK'event and CLK = '1') then
				estado <= prox_estado;
		 end if;
		end process;

func_prox_estado: process (INICIAR, ZERO, estado)
begin
		case estado is
		 when PARADO =>
				if INICIAR = '1' then
				 prox_estado <= MUL0;
				else
				 prox_estado <= PARADO;
				end if;
		 when MUL0 =>
				prox_estado <= MUL1;
		 when MUL1 =>
				if ZERO = '1' then
				 prox_estado <= PARADO;
				else
				 prox_estado <= MUL0;
				end if;
		end case;
end process;


func_fluxo_dados: process (CLK)
variable CA: std_logic_vector(4 downto 0);
begin
 if (CLK'event and CLK = '1') then
		if ENTRA_MULTIPLICANDO = '1' then
				B <= MULT_IN;
		end if;
		if ENTRA_MULTIPLICADOR = '1' then
				Q <= MULT_IN;
		end if;
		case estado is
		 when PARADO =>
		 if INICIAR = '1' then
				C <= '0';
				A <= "0000";
				P <= "11";
		 end if;
		 when MUL0 =>
		 if Q(0) = '1' then
		  CA := ('0'& A) + ('0' & B);
		  else
		  CA := C & A;
		 end if;
		  C <= CA(4);
		  A <= CA(3 downto 0);
		 when MUL1 =>
		  C <= '0';
		  A <= C & A(3 downto 1);
		  Q <= A(0) & Q(3 downto 1);
		  P <= P - "01";
		end case;
 end if;
 end process;
end comportamento_mult4;
```

Exercício:

Implementar os dois códigos dos multiplicadores, obter os netlists deles e enviar um arquivo .zip dos netlists em pdf e dos códigos vhdl implementados.

Bibliografia

MIDORIKAWA, E. T. Uma Introdução às Linguagens de Descrição de Hardware. Ed.EPUSP, 2001.

ORDONEZ, E. D. M; PEREIRA, F. D; PENTEADO, C. G.; PERICINI, R. A. Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGA) EdBless, 2003.