Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

Relatório do Experimento 5

Autor: Henrique Morcelles Salum

Matrícula: 232003008

ENE0040 - Laboratório de Sistemas Digitais - Turma 07

Sumário

1 Introdução

1.1 Sobre o Experimento

Este experimento é separado em três tarefas, mas podemos tratá-las como apenas uma, pois todas são profundamente interligadas. Em suma, projetaremos um somador de 4 bits (*nibble*), seu *testbench* e um *Golden Model*, para analisar os resultados obtidos. Tudo será projetado em VHDL e simulado no ModelSim.

O somador de *nibble* tem dois vetores de entrada A e B, de quatro *bits* e um vetor de saída S, de cinco *bits*. O circuito principal é implementado por meio de instâncias de somadores completos; o *Golden Model* é implementado por meio de abstrações fornecidas pela biblioteca *std_logic_arith*, que permite que somemos *nibbles* (vetores de *std_logic*, em VHDL) apenas utilizando o operador '+' após convertê-los para *unsigned*; o *testbench* injeta estímulos em ambos os circuitos mencionados e compara as saídas, exibindo uma mensagem no terminal em caso de erro.

1.2 Introdução Teórica

O somador completo foi devidamente explicado no experimento 2, em que o implementamos por meio do código a seguir:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity SomadorCompleto is
    port (
        A, B, Cin: in std_logic;
        S, Cout: out std_logic
    );
end entity SomadorCompleto;

architecture behavioral of SomadorCompleto is
begin
    S <= A xor B xor Cin;
    Cout <= (A and B) or (A and Cin) or (B and Cin);
end architecture behavioral;
```

Código 1: Implementação do somador completo em VHDL

Agora, nos dedicaremos a entender como utilizá-lo para projetar um somador de *nibble*. A ideia consiste em cascatear quatro somadores completos. O C_{out} de cada somador, que aqui representa uma posição (casa) do número que somamos (que, na representação binária, tem 4 posições), é o C_{in} do somador seguinte, que representa a próxima casa da soma. Note que é exatamente assim que funciona uma soma normal.

$$\begin{array}{ccccccc} {\scriptstyle 1} & & {\scriptstyle 1}0 & {\scriptstyle 1}1 & {\scriptstyle 1}0 & 1 \\ + & & 1 & 0 & 1 & 1 \\ \hline 1 & & 0 & 0 & 0 & 0 \end{array}$$

Podemos visualizar isso a nível de circuito pelo esquemático que segue.

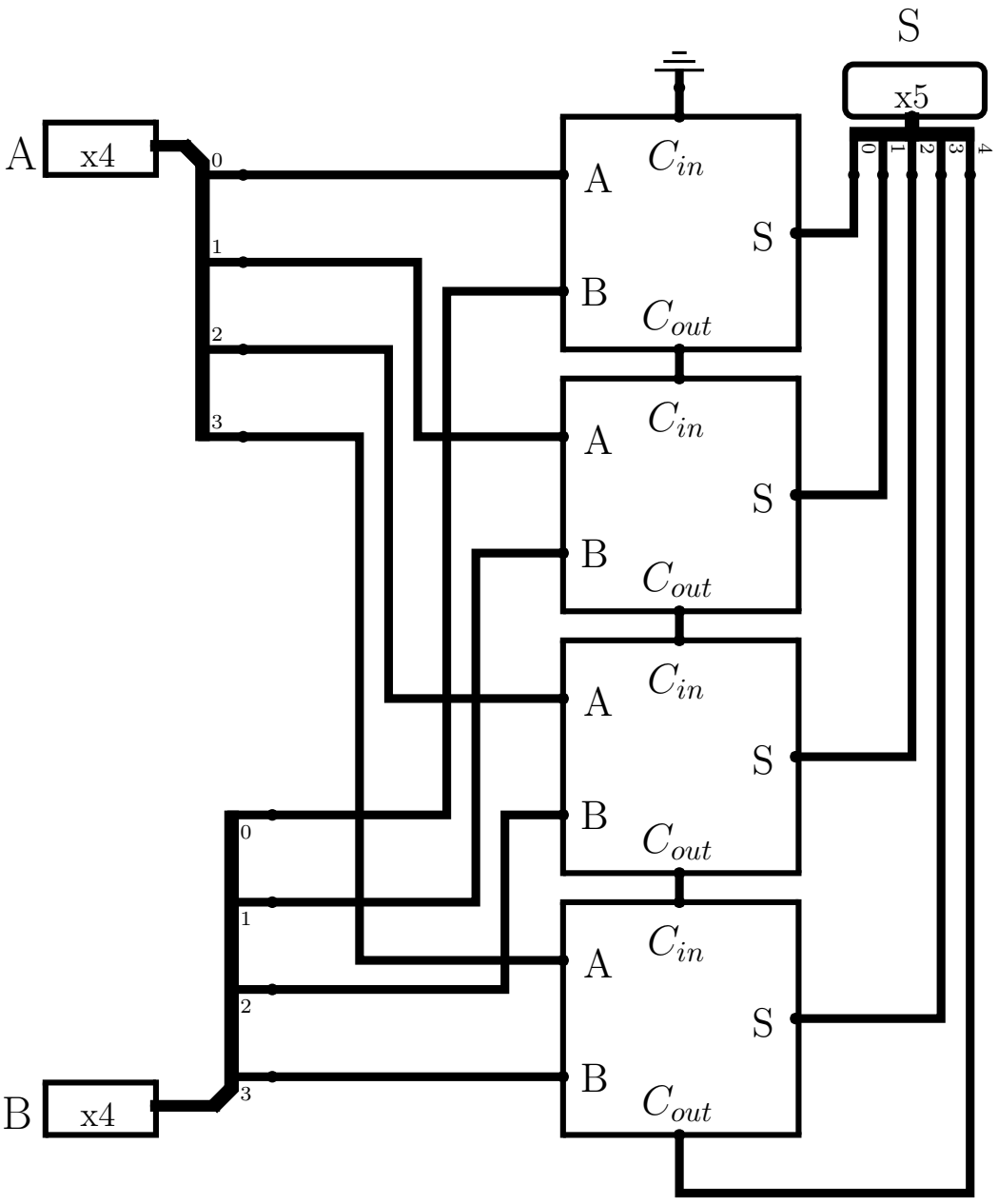

Figura 1: Esquemático do somador de *nibble*

2 Códigos

Nessa seção, apresentamos os códigos utilizados para implementar o sistema apresentado na seção anterior. O código que implementa o somador completo já foi apresentado na introdução, pois não é objeto deste experimento.

2.1 Somador de *Nibble*

Aqui, implementamos o somador instanciando somadores completos. São necessários sinais internos que servem como fios conectando os somadores nos casos em que uma saída de um vira uma entrada de outro (nos C_{out} e C_{in}) - vemos isso nos *port_map*.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity SomadorNibble is
5      port (
6          A, B: in std_logic_vector(3 downto 0);
7          S: out std_logic_vector(4 downto 0)
8      );
9  end entity SomadorNibble;
10
11 architecture structural of SomadorNibble is
12     component SomadorCompleto is
13         port (
14             A, B, Cin: in std_logic;
15             S, Cout: out std_logic
16         );
17     end component SomadorCompleto;
18
19     signal Cout_0, Cout_1, Cout_2: std_logic;
20 begin
21     somador_1: component SomadorCompleto
22         port map (
23             A => A(0),
24             B => B(0),
25             Cin => '0',
26             S => S(0),
27             Cout => Cout_0
28         );
29     somador_2: component SomadorCompleto
30         port map (
31             A => A(1),
32             B => B(1),
33             Cin => Cout_0,
34             S => S(1),
35             Cout => Cout_1
```

```
        );
    somador_3: component SomadorCompleto
        port map (
            A => A(2),
            B => B(2),
            Cin => Cout_1,
            S => S(2),
            Cout => Cout_2
        );
    somador_4: component SomadorCompleto
        port map (
            A => A(3),
            B => B(3),
            Cin => Cout_2,
            S => S(3),
            Cout => S(4)
        );
end architecture structural;
```

Código 2: Descrição de Hardware do somador de *nibble*

2.2 Golden Model do Somador de *Nibble*

Aqui implementamos o mesmo somador, mas abstraindo as complicações relativas aos somadores completos. Assumindo o funcionamento correto da biblioteca utilizada (*std_logic_arith*), podemos utilizar esse somador para garantir a correta implementação do outro, comparando as saídas.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity SomadorNibbleArith is
    port (
        A, B: in std_logic_vector(3 downto 0);
        S: out std_logic_vector(4 downto 0)
    );
end entity SomadorNibbleArith;

architecture behavioral of SomadorNibbleArith is
begin
    S <= ('0' & unsigned(A)) + ('0' & unsigned(B));
end architecture behavioral;
```

Código 3: *Golden Model* da questão 2

2.3 Testbench

Assim como no último experimento, esse *testbench* é diferente dos desenvolvidos nos primeiros três experimentos: a *entity* não está vazia, há sinal(is) de saída, e apenas geramos estímulos nesses sinais, não conectamos os "cabos" (sinais internos) às entradas do sistema que testamos. Isso ocorre porque instanciaremos esse *testbench* no *top module* e só lá conectaremos os "cabos" dele com os sinais de entrada do sistema.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity tb_SomadorNibble is
    port (
        A, B: out std_logic_vector(3 downto 0)
    );
end entity tb_SomadorNibble;

architecture testbench of tb_SomadorNibble is
begin
    estimulos: process
        variable I: std_logic_vector(7 downto 0) := (others => '0');
    begin
        A <= I(3 downto 0);
        B <= I(7 downto 4);
        I := std_logic_vector(unsigned(I) + 1);
        wait for 500 ns;
    end process estimulos;
end architecture testbench;
```

Código 4: Testbench do somador de *nibble*

2.4 Top Module

Aqui, no *top module*, instanciamos e conectamos tudo. É como se fosse onde ligamos o sistema ao resto do circuito. Perceba que há um processo chamado "comparacao", no qual definimos que deve ser exibida uma mensagem em caso de discrepâncias entre o sistema testado e o *Golden Model*.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity TopModule is
end entity TopModule;

architecture structural of TopModule is

    component tb_SomadorNibble is
```

```
10          port (
11              A, B: out std_logic_vector(3 downto 0)
12          );
13      end component tb_SomadorNibble;
14
15      component SomadorNibble is
16          port (
17              A, B: in std_logic_vector(3 downto 0);
18              S: out std_logic_vector(4 downto 0)
19          );
20      end component SomadorNibble;
21
22      component SomadorNibbleArith is
23          port (
24              A, B: in std_logic_vector(3 downto 0);
25              S: out std_logic_vector(4 downto 0)
26          );
27      end component SomadorNibbleArith;
28
29      signal A_tb, B_tb: std_logic_vector(3 downto 0);
30      signal S, S_arith: std_logic_vector(4 downto 0);
31  begin
32
33      testbench: component tb_SomadorNibble
34          port map (
35              A => A_tb,
36              B => B_tb
37          );
38
39      instancia_somadorNibble: component SomadorNibble
40          port map (
41              A => A_tb,
42              B => B_tb,
43              S => S
44          );
45
46      instancia_somadorNibbleArith: component SomadorNibbleArith
47          port map (
48              A => A_tb,
49              B => B_tb,
50              S => S_arith
51          );
52
53      comparacao: process (A_tb, B_tb)
54      begin
55          if S /= S_arith then
56              report "Diferente! S = " & to_string(S) & ", S_arith = " & to_string(S_arith);
57          end if;
58      end process comparacao;
59
60  end architecture structural;
```

Código 5: Top module da questão 1

Observação: Para utilizar o *to_string*(), foi necessário mudar a versão do VHDL utilizada pelo ModelSim para a "1076-2008". Caso isso não seja feito, o código acima gerará um erro de compilação.

3 Compilação

Após escrever os códigos, é necessário compilá-los pelo ModelSim para que se possa simular os sistemas digitais discutidos. Caso a compilação tenha sucesso, sabemos que não houve erros nos códigos apresentados, mas ainda não podemos afirmar que a lógica para implementar os circuitos está correta; isso será analisado nas próximas seções.

A seguir, está a mensagem de compilação dos códigos apresentados acima. Sem nenhum erro, como pode ser visto no terminal no canto inferior da figura.

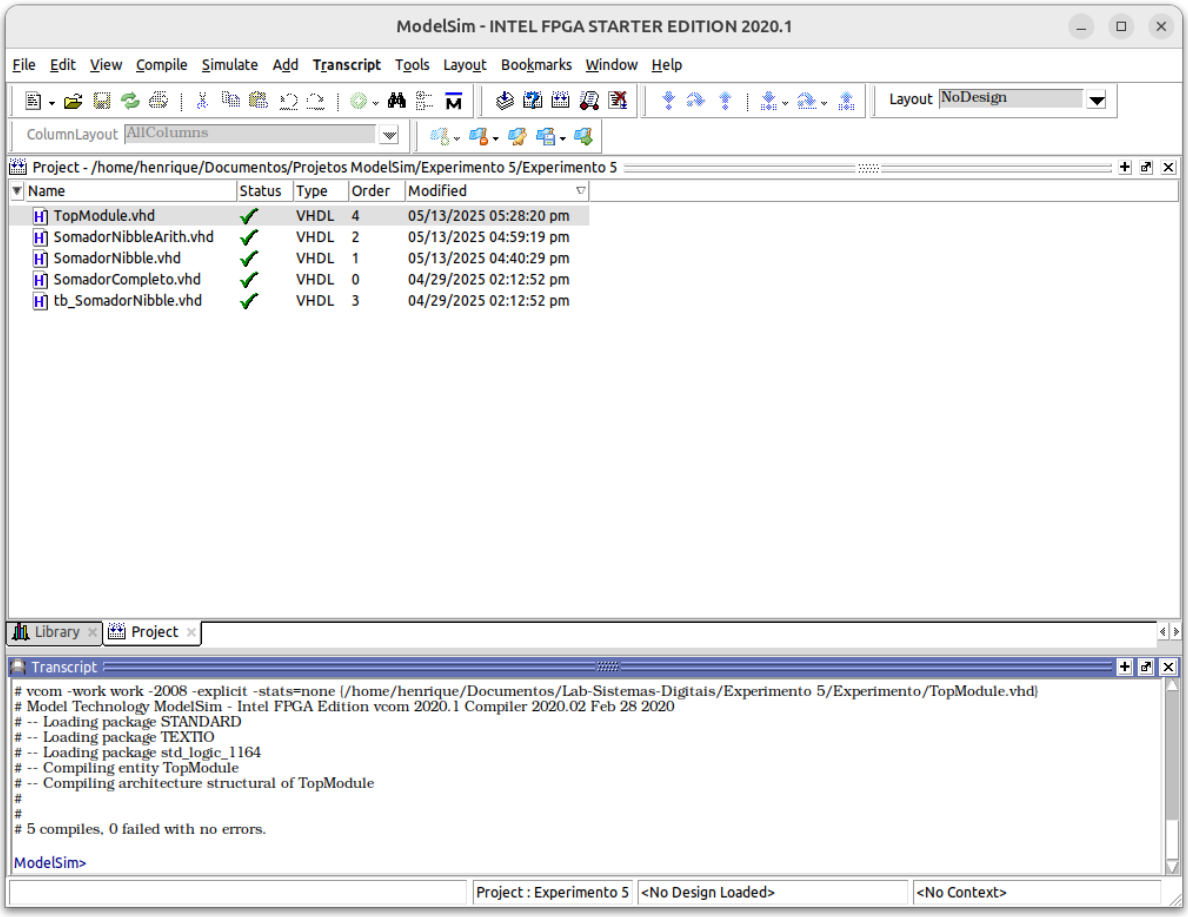

Figura 2: Compilação de todos os códigos apresentados

4 Simulação

As simulações do somador de *nibble* e do seu *Golden Model* estão exibidas a seguir. Os sinais de entrada estão em azul, a saída em verde e o *Golden Model* em amarelo. O tempo de simulação para gerar todas as combinações é de $2^8 \cdot 500ns = 128.000ns$.

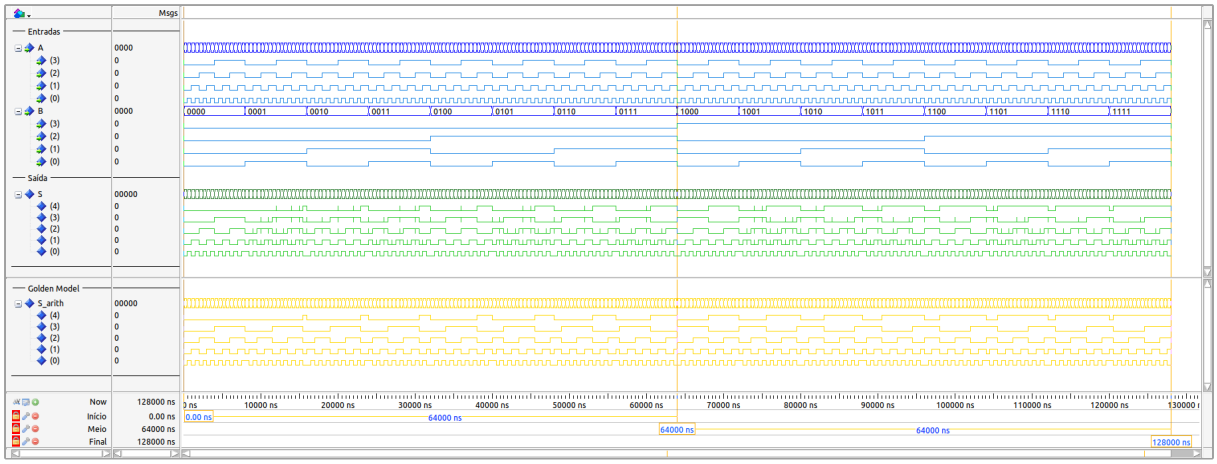

Figura 3: Simulação em forma de onda binária do dispositivo da questão 1

Perceba que, na saída do somador de *nibble* (não no *Golden Model*), há oscilações de duração, aparentemente, infinitesimais. Não foi encontrada explicação para esse fenômeno e ele não ocorreu todas as vezes que o código foi simulado (quando foi recebido o visto, por exemplo). Ver-se-á doravante que essas variações não foram sequer reconhecidas pelo processo “comparacoes”, mencionado anteriormente.

5 Análise

A análise nesse experimento consiste em notar que as ondas do *Golden Model* e do sistema testado são iguais, excetuando-se os picos e vales infinitesimais na saída do sistema. Isso é corroborado pela figura a seguir: notamos que no *Transcript* (parte inferior da imagem) só foi exibida a mensagem de erro, definida no *top module*, uma vez: no início da simulação, quando uma saída era "UUUUU" e a outra era "XXXXX" - caso limite, que não ilustra falha alguma na lógica de implementação utilizada.

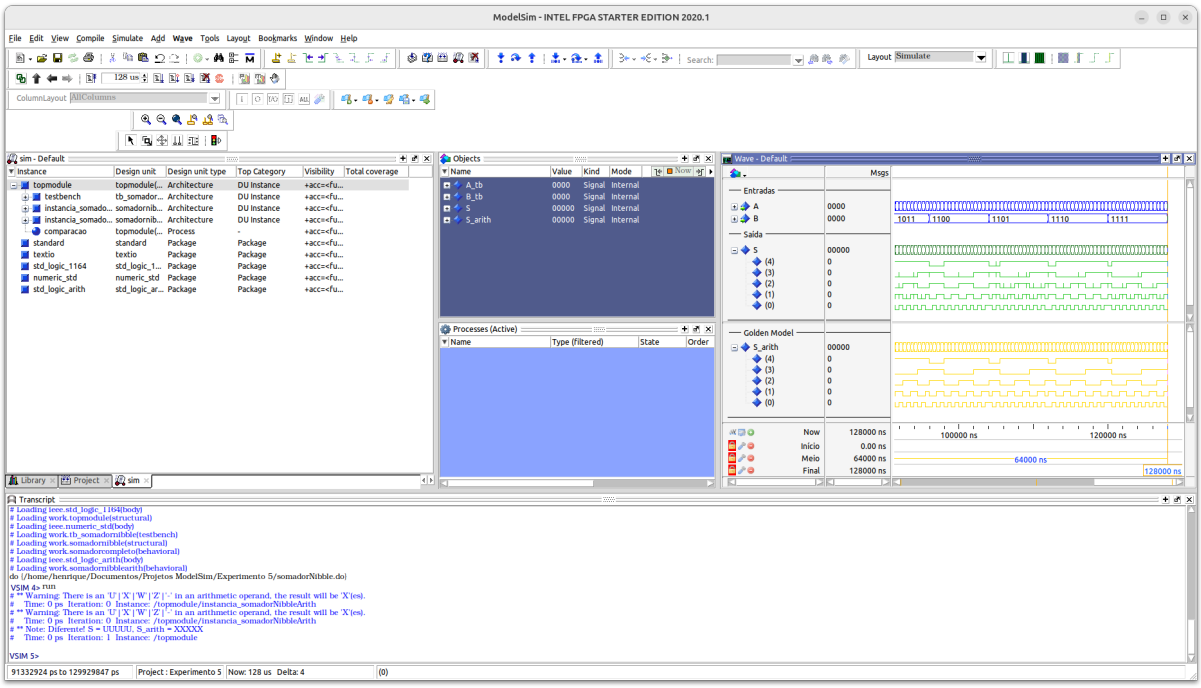

Figura 4: Tela do ModelSim após a simulação

6 Conclusão

Neste experimento, implementamos um somador de 4 *bits* utilizando somadores completos, consolidando um conceito fundamental para a computação moderna, já que operações aritméticas como essa são a base do funcionamento da ULA (Unidade Lógica Aritmética) em processadores.

Além disso, ampliamos nosso conhecimento em metodologias de verificação com a introdução formal do *Golden Model*, uma técnica essencial para validação de projetos digitais. Outro avanço significativo foi a utilização prática da biblioteca *std_logic_arith* da Synopsys, que simplifica operações aritméticas em VHDL.

Por fim, o experimento não apenas reforçou conceitos-chave da disciplina, mas também demonstrou a importância de ferramentas automatizadas, como o *Golden Model*, que eliminam a necessidade de verificações manuais exaustivas (como a comparação de todas as combinações de entradas), aumentando a eficiência e a confiabilidade do desenvolvimento.