Universidade de Brasília
Departamento da Engenharia Elétrica
Laboratório de Sistemas Digitais

Relatório 05

Testbenches

José Antônio Alcântara da Silva de Andrade Mat: 232013031

Professor:
Eduardo B. R. F. Paiva
Turma 08

Brasília, DF

2 de janeiro de 2025

1 Objetivos

1. Implementar um somador de palavras binárias usando somadores completos em cascata.
2. Usar o pacote `STD_LOGIC_ARITH`.
3. Desenvolver um testbench para simulação e teste de circuitos em VHDL.

2 Atividades

Essa sessão do laboratório tinha como objetivo a simulação de um sistema de teste, criando-se o Modelo de Ouro (*Golden Model, GM*) que garante veracidade dos resultados, a Unidade Sobre Teste (*Device Under Testing, DUT*) que implementa o sistema em análise e a Banca de Testes (*Testbench*) que testa todas as possíveis entradas e verifica se o GM e DUT possuem a mesma resposta.

2.1 Exercício 1

O exercício 1 consiste na modelagem do DUT, um somador de palavras de 4 bits sem *overflow*, ou seja, que possue uma saída de 5 bits.

2.1.1 Modelagem do Problema

Deverão ser implementadas duas entradas, A e B, ambas de 4 bits, e uma saída S de 5 bits. As entradas serão as palavras a serem somadas, enquanto a saída será o resultado da soma.

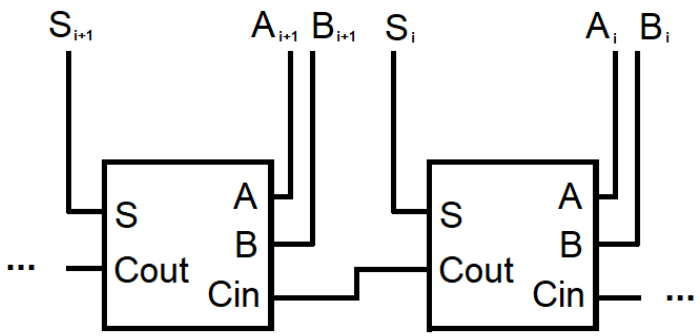
O problema impões dois requisitos que deverão ser seguidos para a compleção do exercício:

1. O somador de palavras deve ser composto apenas por somadores completos.
2. A arquitetura principal pode apenas realizar conexões entre componentes, nenhuma operação.

Utiliza-se, então, da arquitetura de somadores completos previamente desenvolvida no laboratório 2, aqui incluída como a Listagem 9.

Na lógica a ser incluída, realiza-se um cascateamento de somadores completos até atingir o tamanho desejado para o somador (nesse caso, 4 bits). Para tal, conecta-se a saída C_{out} de um somador na C_{in} do próximo, como visto na Figura 1.

Figura 1: Modelo de cascateamento de somadores completos.



Esse cascateamento apresenta um problema de simples solução na implementação: o primeiro C_{in} e o último C_{out}, como devem ser conectados? Usualmente, essas entradas ficam livres em circuitos físicos, a fim de permitir o cascateamento, mas no contexto do experimento, o C_{in} inicial será equivalente a constante 0, enquanto o C_{out} final será atribuído ao bit de maior significância da saída S.

2.1.2 Implementação

Começa-se, então, a implementação do DUT, usando-se apenas dos somadores descritos na Listagem 9. Logo, declara-se todos os requisitos para a funcionalidade do código, como demonstrado na Listagem 1

Listagem 1: Requisitos para o DUT.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity DUT 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 DUT;
10
11 architecture DUT_ARCH of DUT is
12     -- ...
13 end DUT_ARCH;
```

Como já descrito, declara-se, nas linhas 6 e 7 da Listagem 1, as entradas A e B como vetores de 4 bits e a saída S como vetor de 5 bits, respectivamente.

Posteriormente, nas linhas 11 e 13, realiza-se a declaração da arquitetura do DUT como `DUT_ARCH`, e adiciona-se os circuitos incluídos, sinais e associações a partir da linha 12.

Listagem 2: Arquitetura do DUT.

```
1  component SUMM is
2      port (
3          A, B, Cin: in std_logic;
4          S, Cout: out std_logic
5      );
6  end component;
7
8  signal w1, w2, w3, w4: std_logic;
9
10 begin
11     INT1 : SUMM port map(A(0), B(0), '0', S(0), w1);
12     INT2 : SUMM port map(A(1), B(1), w1, S(1), w2);
13     INT3 : SUMM port map(A(2), B(2), w2, S(2), w3);
14     INT4 : SUMM port map(A(3), B(3), w3, S(3), w4);
15     S(4) <= w4;
```

O único componente incluído na arquitetura do DUT é o somador completo, definido na Listagem 9. Para sua inclusão, segue a definição das linhas 1–6 da Listagem 2. Com quatro desses, realizar-se-á o cascateamento necessário, justificando, então, a necessidade dos 4 fios simples declarados na linha 8.

Por fim, ainda na Listagem 2, a partir da linha 10, inicia-se as conexões e operações do DUT. Como visto na Figura 1, as entradas A_0 e B_0 são conectadas ao primeiro somador, e sua saída é conectada ao S_0, enquanto o C_{in} é constante nulo e C_{out} é enviado ao fio `w1`. Em seguida, as entradas A_1 e B_1 são devidamente conectadas, juntamente da saída S_1. Contudo, agora, C_{in} recebe o sinal de `w1`, o C_{out} anterior, enquanto o novo C_{out} é enviado ao fio `w2`. Isso continua-se até o último somador, onde seu C_{out}, o fio `w4`, é conectado diretamente na saída S_4.

Assim, obtém-se um somador de palavras de 4 bits, o DUT de estudo do laboratório. O código final está listado na Listagem 3.

Listagem 3: Código final do DUT.

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

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

architecture DUT_ARCH of DUT is
component SUMM is
    port (
        A, B, Cin: in std_logic;
        S, Cout: out std_logic
    );
end component;

signal w1, w2, w3, w4: std_logic;

begin
    INT1 : SUMM port map(A(0), B(0), '0', S(0), w1);
    INT2 : SUMM port map(A(1), B(1), w1, S(1), w2);
    INT3 : SUMM port map(A(2), B(2), w2, S(2), w3);
    INT4 : SUMM port map(A(3), B(3), w3, S(3), w4);
    S(4) <= w4;
end DUT_ARCH;
```

2.2 Exercício 2

O exercício 2 consiste na modelagem do GM, um somador de palavras que garantidamente sempre providenciará o resultado correto. O *Golden Model* será utilizado para determinar se o DUT, produzido no exercício anterior, apresenta a funcionalidade desejada.

2.2.1 Modelagem do Problema

A fim de garantir a construção de um *Golden Model* perfeitamente funcional, usa-se da biblioteca padrão `STD_LOGIC_ARITH`, que inclui o operador `+`. Com ele, é possível realizar somas com números em decimais, e então convertê-los para binários.

Dito isso, a montagem do GM é demasiadamente simples. Basta aceitar duas entradas, as palavras de 4 bits A e B, concatená-las à esquerda com uma constante 0, e, enfim, somá-las. O resultado será a saída S de 5 bits.

2.2.2 Implementação

Listagem 4: Código completo do GM.

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

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

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

Para a funcionalidade do código, é necessário a importação da biblioteca `STD_LOGIC_1164`, declarada na linha 2 da Listagem 4, mas, como dito anteriormente, também importa-se `STD_LOGIC_ARITH` na linha 3 para a codificação do GM.

Como o modelo GM é apenas uma versão garantidamente funcional do somador, suas entradas e saídas são idênticas as do DUT, como observa-se nas linhas 5–10 da Listagem 4. São declaradas duas entradas de 4 bits A e B, e uma saída de 5 bits S.

Por fim, a arquitetura do GM é simples e concisa, definida nas linhas 12–15 da Listagem 4. Ocorre apenas uma declaração em diversas etapas. Primeiramente os valores de A e B são convertidos para inteiros não-assignalados (ou seja, positivos) e, então, recebem uma concatenação de um zero à esquerda. Esse zero nada altera na soma, apenas permite que o resultado seja expresso em 5 bits ao invés de ocorrer um *overflow*. Por fim, soma-se A e B diretamente com o operador `+`, atribuido-se tal resultado à saída S.

2.3 Exercício 3

O último exercício, exercício 3, consiste na construção do *testbench* que irá coordenar as entradas no DUT e GM, e checar a equalidade das saídas, e o Top Module que apenas realizará as conexões necessárias entre os três circuitos.

2.3.1 Modelagem do Problema

Para o Top Module, não há muitos requisitos funcionais, será necessário apenas realizar conexões de tal forma que as entradas do DUT e GM são as saídas do Testbench, e vice-versa.

O Testbench apresenta uma engenharia mais complexa. Como será realizada o teste de um somador de palavras de 4 bits, existem 16 diferentes possibilidades para cada palavra, o intervalo de inteiros de 0 até 15. São duas entradas de palavras, então serão 256 testes no total.

Para realizar todas as possíveis combinações para a palavra A e a palavra B, faz-se dois loops, um interno ao outro, a fim de obter todas as 256 combinações. Em cada uma, será checado se o valor obtido pelo DUT é equivalente ao do GM. Se não, o DUT está incorretamente implementado.

Além disso, deve-se, por requisito do exercício, implementar mensagens que serão enviadas ao console durante a simulação. Uma mensagem ao iniciar e terminar a simulação, e mensagens indicando erros no teste, caso ocorra algum.

2.3.2 Implementação do Testbench

O Testbench, circuito responsável pela análise da funcionalidade do DUT, naturalmente possui uma complexidade mais elevada se comparado aos outros circuitos presentes nesse experimento.

Listagem 5: Requisitos do Testbench.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_arith.all;
4  use IEEE.std_logic_unsigned.all;
5
6  entity TESTBENCH is
7      port (
8          DUT_out, GM_out: in std_logic_vector(4 downto 0);
9          A, B: out std_logic_vector(3 downto 0)
10     );
11 end TESTBENCH;
```

Além da biblioteca normal de VHDL, nota-se a presença de duas outras bibliotecas nas linhas 3 e 4 da Listagem 5: a previamente introduzida `STD_LOGIC_ARITH` e uma nova, `STD_LOGIC_UNSIGNED`. A `ARITH` será utilizada para o mesmo propósito que fora anteriormente, o operador `+` permitirá a permutação de todas as possíveis entradas. Já a biblioteca `UNSIGNED` permite a fácil manipulação de vetores para realizar operações aritméticas em decimal.

Em relação às suas entradas e saídas, o Testbench recebe as duas entradas atuais do DUT e GM, na linha 8 da Listagem 5, e tem como saída as próximas entradas deles, A e B respectivamente, aqui declaradas na linha 9.

Listagem 6: Arquitetura do Testbench.

```
1  architecture TESTBENCH_ARCH of TESTBENCH is
2  begin
3      process
4          variable currA : std_logic_vector(3 downto 0);
5          variable currB : std_logic_vector(3 downto 0);
6      begin
7          report "Iniciando teste..." severity NOTE;
8
9          currA := "0000";
10         for i in 0 to 15 loop
11             A <= currA;
12             currB := "0000";
13             for j in 0 to 15 loop
14                 B <= currB;
15                 wait for 500 ns;
16
17                 assert (DUT_out = GM_out) report "Falhou em A = " & integer'image(i) & "
       e B = " & integer'image(j) severity ERROR;
18
19                 currB := currB + 1;
20             end loop;
21             currA := currA + 1;
22         end loop;
23
24         report "Teste finalizado." severity NOTE;
25
26         wait;
27     end process;
28 end TESTBENCH_ARCH;
```

Como discutido anteriormente, para que todas as combinações de A e B sejam atingidas, será necessário o uso de um loop com um outro loop interno, a fim de realizar todas as possíveis permutações.

Primeiramente, na Listagem 6, após declarar a arquitetura, deve-se adicionar-se duas variáveis para a realização de operações. Assim, nas linhas 4 e 5 define-se uma variável para manter salvo o valor atual de A, e outra de B, respectivamente.

Em seguida, começa-se os loops. A vairável `currA` é iniciada como 0 na linha 9, e então atribuída à saída A na linha 11. O mesmo é feito para `currB` nas linhas 12 e 14, respectivamente. Então, espera-se 500ns, na linha 15, para permitir que os resultados nos circuitos do DUT e GM sejam corretamente processados.

Finalmente, realiza-se o teste na linha 17, onde compara-se a saída do DUT com a saída do GM, e envia um erro ao console indicando a falha caso tais sejam diferentes. Para partir para a próxima combinação, soma-se 1 ao valor atual de `currB` na linha 19, e repete-se o loop interno. Quando o valor de `currB` atinge 15, soma-se 1 ao valor de A e retorna-se B para 0, e assim por diante até A atingir 15.

Durante a realização do teste, também são enviadas uma mensagem de aviso de início, na linha 7, e uma mensagem de término, na linha 24. Para evitar que o processo rode continuamente, sem parar, adiciona-se também a linha 26. Assim, o Testbench é completamente implementado. O código final está na Listagem 7.

Listagem 7: Código final do Testbench.

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

entity TESTBENCH is
    port (
        DUT_out, GM_out: in std_logic_vector(4 downto 0);
        A, B: out std_logic_vector(3 downto 0)
    );
end TESTBENCH;

architecture TESTBENCH_ARCH of TESTBENCH is
begin
    process
        variable currA : std_logic_vector(3 downto 0);
        variable currB : std_logic_vector(3 downto 0);
    begin
        report "Iniciando teste..." severity NOTE;

        currA := "0000";
        for i in 0 to 15 loop
            A <= currA;
            currB := "0000";
            for j in 0 to 15 loop
                B <= currB;
                wait for 500 ns;

                assert (DUT_out = GM_out) report "Falhou em A = " & integer'image(i) & "
    e B = " & integer'image(j) severity ERROR;

                currB := currB + 1;
            end loop;
            currA := currA + 1;
        end loop;

        report "Teste finalizado." severity NOTE;

        wait;
    end process;
end TESTBENCH_ARCH;
```

2.3.3 Implementação do Top Module

Não há muito há se dizer sobre o Top Module, uma vez que a funcionalidade geral dos testes é implementada no Testbench, na Listagem 7. A implementação completa do Top Module está representada na Listagem 8

Listagem 8: Código completo do Top Module.

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

entity TOPMODULE is
end TOPMODULE;

architecture TOPMODULE_ARCH of TOPMODULE is
component DUT is
    port (
        A, B: in std_logic_vector(3 downto 0);
        S: out std_logic_vector(4 downto 0)
    );
end component;

component GM is
    port (
        A, B: in std_logic_vector(3 downto 0);
        S: out std_logic_vector(4 downto 0)
    );
end component;
```

```
21
22 component TESTBENCH is
23     port (
24         DUT_out, GM_out: in std_logic_vector(4 downto 0);
25         A, B: out std_logic_vector(3 downto 0)
26     );
27 end component;
28
29 signal A, B : std_logic_vector(3 downto 0);
30 signal DUT_out, GM_out : std_logic_vector(4 downto 0);
31
32 begin
33     INT1 : DUT port map(A, B, DUT_out);
34     INT2 : GM port map(A, B, GM_out);
35     INT3 : TESTBENCH port map(DUT_out, GM_out, A, B);
36 end TOPMODULE_ARCH;
```

Note que o código da listagem 8 não possui nem entradas nem saídas, uma vez que a entidade `TOPMODULE` é definida sem porta, nas linhas 4 e 5. Ainda mais, sua arquitetura, representada da linha 7 a 36, consiste apenas de circuitos previamente introduzidos, os quais são conectados entre si.

Mais detalhadamente, os sinais definidos nas linhas 29 e 30 são mapeados para as entradas e saídas de mesmo nome nos componentes. Na linha 33, conecta-se A e B na entrada do DUT, enquanto coloca-se sua saída no propriamente denominado DUT_{out}. Faz-se similar para o GM na linha 34, apenas altera-se o nome da saída para GM_{out}. Já para o Testbench, na linha 35, usa-se as saídas dos outros dois componentes, DUT_{out} e GM_{out}, como entradas; inversamente, as saídas serão as entradas dos outros componentes: A e B.

2.4 Simulação

O Top Module será o circuito utilizado para realizar as simulações, uma vez que sua estrutura é a que conecta o GM, DUT e Testbench. Como todas as operações necessárias para a visualização da simulação já foram implementadas, não é preciso realizar operações no software ModelSim. Basta apenas iniciar a simulação.

Obtém-se 256 resultados para as 256 possibilidades. Notavelmente, o console não apresentou nenhum erro, apenas as mensagens de início e término, além de avisos advindos da natureza assíncrona da linguagem VHDL, como visto na Figura 2.

Figura 2: Console do ModelSim durante a simulação.

```
VSIM 12> run -all
# GetModuleFileName: Não foi possível encontrar o módulo especificado.
#
#
# ** Note: Iniciando teste...
#    Time: 0 ps  Iteration: 0  Instance: /topmodule/INT3
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 0  Instance: /topmodule/INT2
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 0  Instance: /topmodule/INT2
# ** Note: Teste finalizado.
#    Time: 128 us  Iteration: 0  Instance: /topmodule/INT3
```

Em relação aos resultados obtidos, as Figuras 3, 4, 5, 6, 7, 8, 9 e 10 são as visualizações de onda de todas as possibilidades. Os números estão representados em decimal, para legibilidade.

Figura 3: Visualização de onda quando A = 0 ou 1.

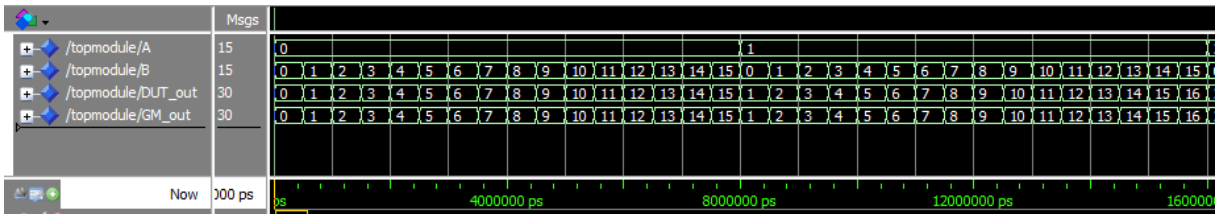


Figura 4: Visualização de onda quando A = 2 ou 3.

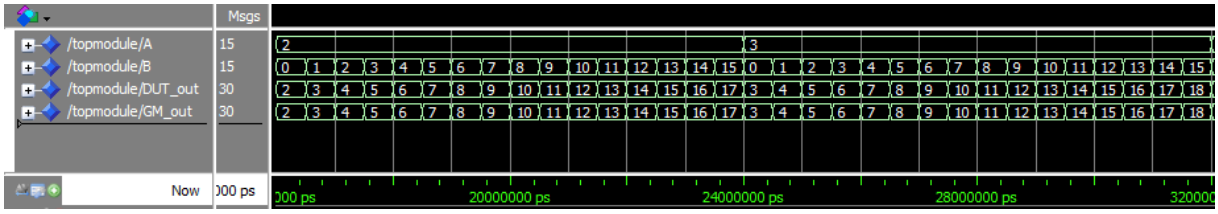


Figura 5: Visualização de onda quando A = 4 ou 5.

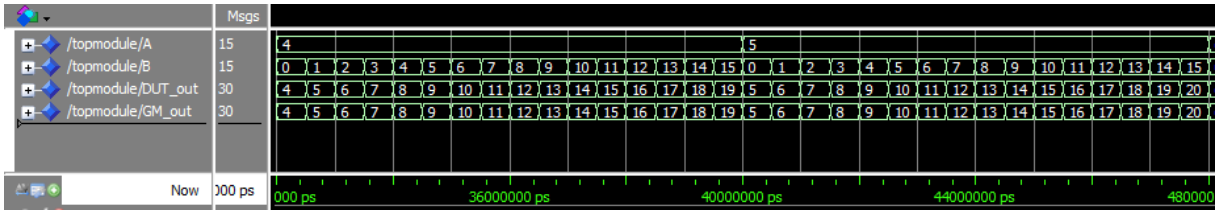


Figura 6: Visualização de onda quando A = 6 ou 7.

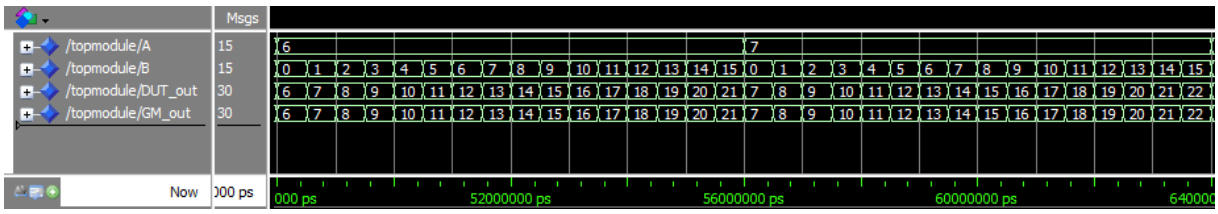


Figura 7: Visualização de onda quando A = 8 ou 9.

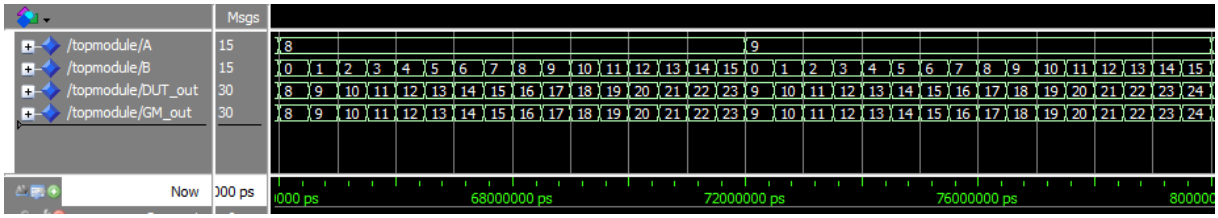


Figura 8: Visualização de onda quando A = 10 ou 11.

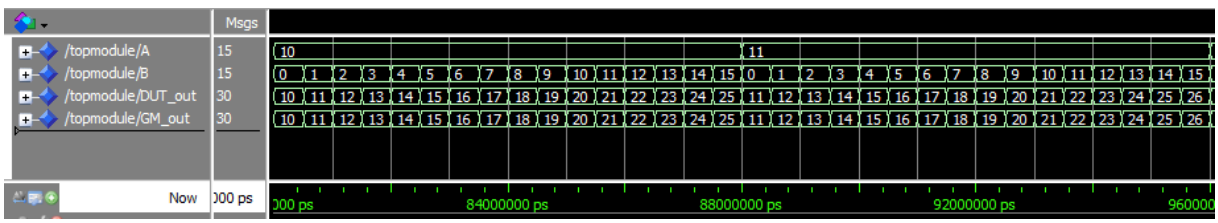


Figura 9: Visualização de onda quando A = 12 ou 13.

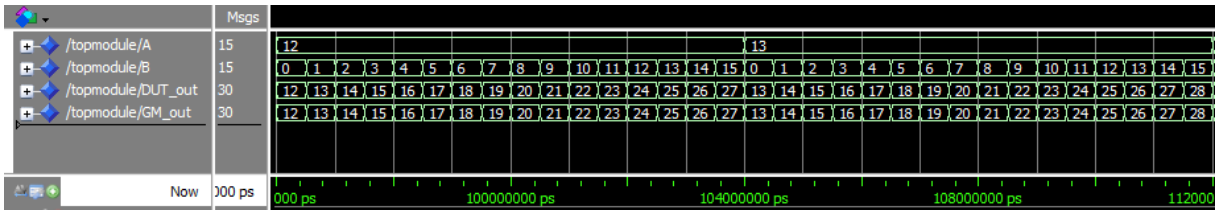
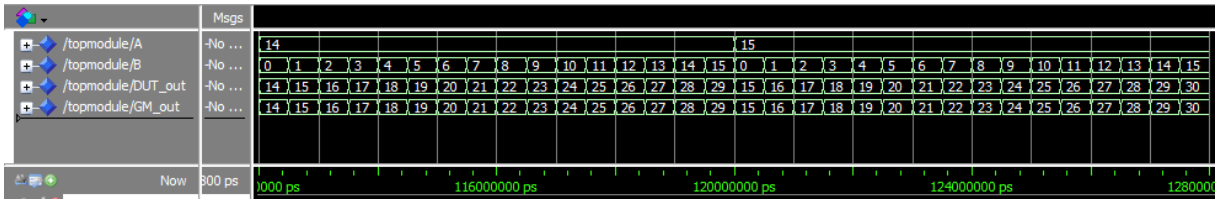


Figura 10: Visualização de onda quando A = 14 ou 15.



2.5 Extra

2.5.1 Códigos Importados

Segue as Listagens contendo códigos importados de sessões anteriores do laboratório.

Listagem 9: Somador completo.

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

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

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

2.5.2 Compilação

Como segue na Figura 11, nenhum dos códigos obteve erros de compilação.

Figura 11: Compilação dos códigos.

```
# Compile of e1.vhd was successful.
# Compile of e2.vhd was successful.
# Compile of e3.vhd was successful.
# Compile of summ.vhd was successful.
# Compile of testbench.vhd was successful.
VSIM 13> piles, 0 failed with no errors.
```