Laboratório de Sistemas Digitais – 2025.1

Relatório 5 – Operações Aritméticas e Testbenches

João Pedro Fernandes Santos (222025342)

Turma 09

Introdução/Objetivos

Neste Experimento 5 de Sistemas Digitais, implementamos circuitos aritméticos em VHDL para explorar duas abordagens de soma de palavras de 4 bits e criamos um ambiente de verificação (testbench) para comparar automaticamente seus resultados. Na primeira parte, desenvolvemos um somador de 4 bits a partir de quatro somadores completos em cascata; na segunda, usamos o operador "+" do pacote STD_LOGIC_ARITH para realizar a mesma soma de forma vetorial, e por fim, construímos um testbench que gera todas as 256 combinações possíveis de entradas, compara as saídas dos dois somadores e alertar qualquer diferença com o resultado obtido. A atividade enfatiza a codificação modular, a conversão entre tipos (STD_LOGIC_VECTOR ↔ UNSIGNED), a elaboração de um testbench que cubra todas as entradas e saídas por meio de simulações no ModelSim. Ao finalizar, espera-se desenvolver a habilidade de projetar circuitos aritméticos em VHDL, manipular operadores de grandezas vetoriais, criar testbenches automatizados que garantam cobertura completa e interpretar criticamente os resultados de simulação.

1. Atividade 1

1.1 Descrição da atividade

A primeira atividade propôs a implementação, em VHDL e simulação no ModelSim, de um somador de palavra de 4 bits utilizando exclusivamente quatro somadores completos (1 bit) em cascata, sem recorrer a operadores aritméticos de alto nível. O componente somador completo de 1 bit já havia sido desenvolvido no Experimento 2.1 e encapsulado como entidade separada de nome **SOMADOR_COMPLETO**, contendo as portas de entrada A, B, Cin e as saídas Sum e Cout. O somador completo implementa as funções lógicas:

$$Sum = A \oplus B \oplus Cin$$

$$Cout = (A \wedge B) \vee (B \wedge Cin) \vee (A \wedge Cin)$$

Tabela 1: Tabela-verdade do Somador Completo (1 bit):

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1.2 Implementação em VHDL

A Listagem 1 apresenta o código criado para a execução desta atividade.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY SOMADOR4BIT IS

 PORT (

  A  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);

  B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);

  S  : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)

 );

END ENTITY;

ARCHITECTURE SOMADOR4BIT_ARCH OF
SOMADOR4BIT  IS

 COMPONENT SOMADOR_COMPLETO

  PORT (

   A, B  : IN  STD_LOGIC;

   Cin   : IN  STD_LOGIC;

   S, Cout: OUT STD_LOGIC

  );

 END COMPONENT;
```

```
SIGNAL C1, C2, C3, C4 : STD_LOGIC;

BEGIN

 U0: somador_completo PORT MAP(A => A(0), B =>
B(0), Cin => '0',  S => S(0), Cout => C1);

 U1: somador_completo PORT MAP(A => A(1), B =>
B(1), Cin => C1, S => S(1), Cout => C2);

 U2: somador_completo PORT MAP(A => A(2), B =>
B(2), Cin => C2, S => S(2), Cout => C3);

 U3: somador_completo PORT MAP(A => A(3), B =>
B(3), Cin => C3, S => S(3), Cout => C4);

S(4) <= C4;

END ARCHITECTURE;
```

Listagem 1: Implementação do circuito do somador de palavras de 4 bits.

O código define uma entidade VHDL chamada SOMADOR4BIT com duas entradas vetoriais de 4 bits (A e B) e uma saída vetorial de 5 bits (S). Na arquitetura SOMADOR4BIT_ARCH, declara-se o componente SOMADOR_COMPLETO (1 bit), cuja lógica interna foi desenvolvida no Experimento 2.1 e está modularizada em outro arquivo. Em seguida, são criados quatro sinais internos (C1, C2, C3 e C4) para propagar os bits de carry de menor para maior peso.

Na seção de implementação, há quatro instâncias de SOMADOR_COMPLETO, nomeadas **U0** a **U3**, cada uma conectada via PORT MAP aos bits correspondentes de A e B e ao sinal de carry apropriado:

- **U0**: recebe A(0), B(0) e Cin = '0', gerando S(0) e Cout → C1.
- **U1**: recebe A(1), B(1) e Cin = C1, gerando S(1) e Cout → C2.
- **U2**: recebe A(2), B(2) e Cin = C2, gerando S(2) e Cout → C3.
- **U3**: recebe A(3), B(3) e Cin = C3, gerando S(3) e Cout → C4.

Após as quatro instâncias, o bit de overflow é atribuído diretamente a S(4) por meio da atribuição S(4) <= C4;. Dessa forma, cada instância de SOMADOR_COMPLETO funciona como um bloco modular que realiza a soma de um único bit, propagando o carry para o próximo estágio sem qualquer lógica adicional na entidade principal. O vetor de saída S(4 downto 0) reúne as somas de cada estágio em S(3 downto 0) e o carry final em S(4). Esse design enfatiza a codificação modular e o encadeamento hierárquico de componentes, aproveitando o bloco SOMADOR_COMPLETO já testado anteriormente em ModelSim.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;

ENTITY SOMADOR_COMPLETO IS

PORT (A, B, Cin: IN STD_LOGIC;

S, Cout: OUT STD_LOGIC);

END SOMADOR_COMPLETO;


ARCHITECTURE SOMADOR_COMPLETO_ARCH OF
SOMADOR_COMPLETO IS

BEGIN

S <= (A XOR B) XOR Cin;

Cout <= (A AND B) OR (A AND Cin) OR (B AND Cin);

END SOMADOR_COMPLETO_ARCH;
```

Listagem 2: Código do componente **SOMADOR_COMPLETO** do experimento 2.1

1.3 Simulação

Para verificar o código do SOMADOR4BIT, as entradas A e B foram configuradas como ondas quadradas em ModelSim com períodos distintos para cada bit. Em A, o bit menos significativo (A(0)) alterna a cada 50 ms, A(1) a cada 100 ms, A(2) a cada 200 ms e A(3) a cada 400 ms. Em B, o bit menos significativo (B(0)) alterna a cada 800 ms, B(1) a cada 1 600 ms, B(2) a cada 3 200 ms e B(3) a cada 6 400 ms. Assim, dentro de cada ciclo de 6400 ms, A percorre suas 16 combinações de 4 bits 16 vezes, enquanto B muda apenas de estado uma vez a cada 800 ms. Esse desacoplamento de tempos garante que todas as 256 combinações possíveis de (A, B) sejam exercitadas de forma sistemática ao longo de 6 400 ms, permitindo observar como o somador responde a cada par de vetores de entrada em diferentes instantes.

Nas formas de onda, nota-se que, a cada mudança em A(0–3), o **somador completo 0** processa imediatamente A(0)+B(0), atualizando S(0) e gerando C1 quando necessário; esse C1 propaga-se sequencialmente pelos estágios U1, U2 e U3, com atrasos de aproximadamente 10–30 ns em cada bloco, refletindo o comportamento ripple carry ao longo dos bits. Sempre que B muda (por exemplo, em 800 ms, 1 600 ms, etc.), A já percorreu várias combinações, resultando em distintas somas parciais que evidenciam a robustez da propagação de carry em todos os casos. Quando A + B excede 15 (por exemplo, em t ≈ 4 000 ms, se A = “1001” e B = “0111”), U3 gera C4 = ‘1’, e a saída final S apresenta “0000” em S(3..0) com S(4) = ‘1’, indicando overflow. Em demais transições, S(4) permanece ‘0’, enquanto S(3..0) assume corretamente o valor binário de A + B (módulo 16) após o atraso de propagação. Dessa forma, a simulação confirma que, para cada combinação gerada pelos ciclos de 50 ms a 6 400 ms, o circuito executa corretamente a soma de 4 bits e propaga adequadamente o carry até o bit de overflow.

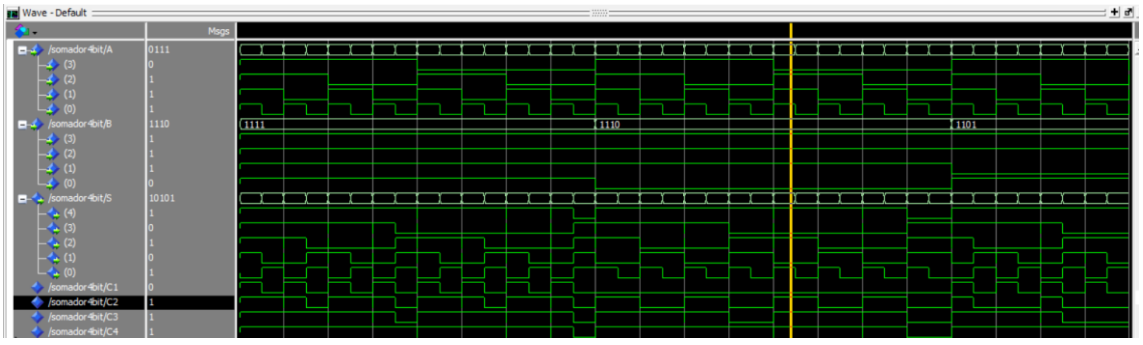

Figura 1: Simulação do SOMADOR4BIT, caso A = 0111 e B = 1110, ou em decimal, A=7 e B=14. É possível notar o funcionamento dos sinais e da saída do sistema corresponde ao esperado pois C = 10101, ou em decimal, C =21.

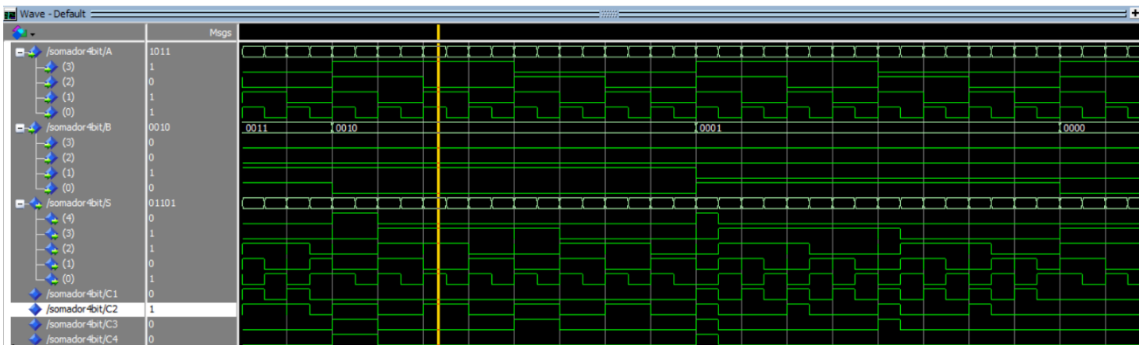

Figura 2: Simulação do SOMADOR4BIT, caso A = 1011 e B = 0010, ou em decimal, A=11 e B=2. É possível notar o funcionamento dos sinais e da saída do sistema corresponde ao esperado pois C = 01101, ou em decimal, C =13.

2. Atividade 2

2.1 Descrição da atividade

A segunda atividade propôs a implementação, em VHDL e simulação no ModelSim, de um somador de palavras de 4 bits utilizando exclusivamente o operador "+" do pacote STD_LOGIC_ARITH. A nova entidade, chamada **SOMADOR4BIT_ARITH**, deve receber como entradas dois vetores de 4 bits (A e B) e gerar um vetor de 5 bits (S), de modo a acomodar o bit de overflow. O roteiro define que, para poder usar "+" em vetores do tipo STD_LOGIC_VECTOR, é necessário converter primeiro A e B para UNSIGNED com o comando unsigned(.); após realizar a soma, o resultado ,potencialmente com carry no quinto bit, é então convertido de volta para STD_LOGIC_VECTOR. Não há instância de componentes externos nem lógica de propagação de carry manual: toda a aritmética é unicamente expressa por essa soma vetorial, tornando o código conciso e modular.

Na arquitetura estrutural de somador4b_arith, a conversão de tipos e a soma são feitas em três etapas principais: primeiro, unsigned(A) + unsigned(B) produz um sinal intermediário de 4 bits que corresponde à soma sem sinal; segundo, concatena-se um bit '0' à esquerda desse sinal para formar um vetor de 5 bits que contém o carry como MSB; por fim, converte-se esse vetor de volta para STD_LOGIC_VECTOR(4 downto 0) e atribui-se à saída S. Todo o gerenciamento do bit de overflow fica a cargo do operador "+" e das conversões entre STD_LOGIC_VECTOR e UNSIGNED, sem necessidade de componentes de somador completo. Na simulação no ModelSim, as 16 combinações de A e B variando de "0000" a "1111" foram testadas, confirmando que S(3 downto 0) assume corretamente o valor binário de A + B (módulo 16) e que S(4) torna-se '1' sempre que ocorre overflow (ou seja, A + B ≥ 16), validando funcionalmente a implementação proposta.

2.2 Implementação em VHDL

A Listagem 3 apresenta o código criado para esta atividade.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;

USE IEEE.STD_LOGIC_ARITH.ALL;


ENTITY SOMADOR4BIT_ARITH IS

 PORT (

  A : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);

  B : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);

  S : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)

 );

END ENTITY;


ARCHITECTURE SOMADOR4BIT_ARITH_ARCH OF
SOMADOR4BIT_ARITH IS

BEGIN

 S <= conv_std_logic_vector(

     unsigned('0' & A) + unsigned('0' & B),

     5

    );

END ARCHITECTURE SOMADOR4BIT_ARITH_ARCH;
```

Listagem 3: Implementação do circuito do SOMADOR4BIT_ARITH.

O código define uma entidade VHDL chamada SOMADOR4BIT_ARITH com duas entradas de 4 bits (A e B) e uma saída de 5 bits (S).O termo USE IEEE.STD_LOGIC_ARITH.ALL, habilita o uso do tipo UNSIGNED e do operador “+” para vetores do tipo STD_LOGIC_VECTOR. Na arquitetura SOMADOR4BIT_ARITH_ARCH, não há instância de componentes externos; toda a lógica aritmética está concentrada em uma única função. Essa atribuição concatena um bit lógico ‘0’ à esquerda de cada vetor de 4 bits ('0' & A e '0' & B), produzindo dois vetores de 5 bits que, ao serem convertidos para UNSIGNED, garantem espaço para o eventual

carry de overflow. Em seguida, executa-se unsigned('0' & A) + unsigned('0' & B), resultando em um valor de 5 bits cuja soma já inclui o possível bit de overflow.

Para devolver o resultado ao tipo STD_LOGIC_VECTOR(4 downto 0), utiliza-se a função conv_std_logic_vector() passando como primeiro parâmetro a soma em tipo UNSIGNED e, como segundo, a largura desejada (5). Dessa forma, o bit mais significativo de S (S(4)) reflete automaticamente qualquer carry gerado na soma (se A + B ≥ 16), enquanto S(3 downto 0) contêm o valor binário de A + B módulo 16. Não há manipulação manual de sinais de carry nem necessidade de somadores completos em cascata; o pacote STD_LOGIC_ARITH abstrai a aritmética de vetor, simplificando o código. Essa implementação enfatiza conceitos de conversão de tipos (STD_LOGIC_VECTOR ↔ UNSIGNED), extensão de sinal via concatenação para acomodar overflow e uso direto de expressões aritméticas em VHDL para circuitos combinacionais de soma vetorial.

2.3 Simulação

Para verificar o SOMADOR4BIT_ARITH, as entradas A e B foram geradas como ondas quadradas em ModelSim com períodos que dobram a cada bit, iniciando em 50 ms para A(0) e estendendo até 6400 ms para B(3). Especificamente, A(0) alterna a cada 50 ms, A(1) a cada 100 ms, A(2) a cada 200 ms e A(3) a cada 400 ms, de modo que A percorre suas 16 combinações (“0000” a “1111”) em 800 ms. Em contraste, B(0) alterna a cada 800 ms, B(1) a cada 1 600 ms, B(2) a cada 3 200 ms e B(3) a cada 6 400 ms. Assim, ao longo de 6 400 ms, A completa oito ciclos de 800 ms para cada mudança lenta de B, garantindo que todas as 256 combinações possíveis de (A,B) sejam exercitadas.

Nas waveforms, cada transição de A (a cada 50 ms a 400 ms) dispara imediatamente a expressão unsigned('0' & A) + unsigned('0' & B), atualizando S(4 downto 0) em um único passo, sem ripple carry—o resultado aparece quase instantaneamente (com atraso de poucos nanosegundos). Enquanto A oscila em ciclos de 800 ms, B permanece constante até sua próxima borda lenta (800 ms, 1 600 ms, 3 200 ms, 6 400 ms). Sempre que A + B ≥ 16, o bit S(4) salta para ‘1’; por exemplo, em t = 1 600 ms, quando B muda de “0001” para “0010”, e A = “1111” (15), observa-se S = “10001” (17) ou S = “10000” (16) dependendo do instante exato, confirmando overflow. Nos demais eventos, S(3..0) reflete corretamente A + B (módulo 16) e S(4) permanece ‘0’. Essa simulação comprova que o operador “+” de STD_LOGIC_ARITH, aliado às conversões de tipo, produz soma vetorial correta e detecta overflow sem necessidade de cuidados manuais com carries.

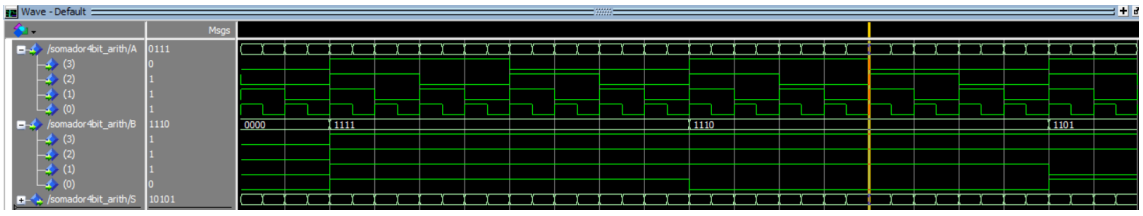

Figura 3: Replicando a entrada da figura 1, com A = 0111 e B = 1110, ou em decimal, A=7 e B=14. É possível notar o funcionamento dos sinais e da saída do sistema, como na atividade 1, corresponde ao esperado pois C = 10101, ou em decimal, C =21.

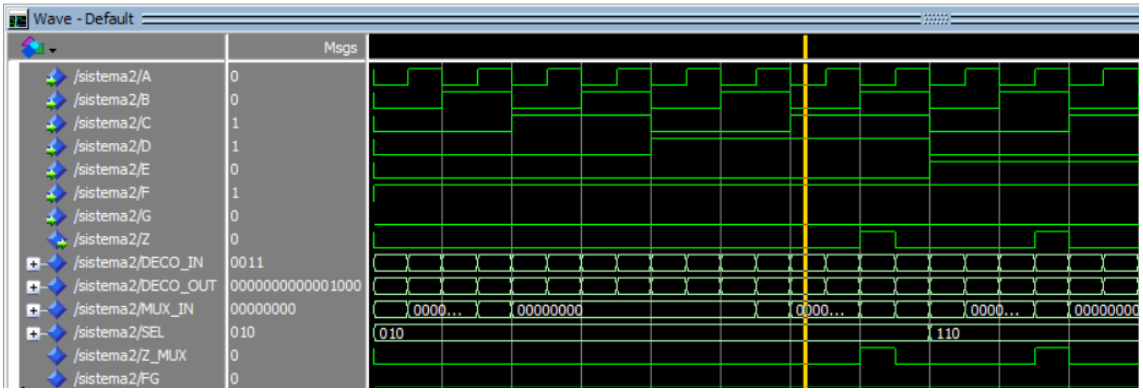

Figura 4: Replicando a entrada da figura 2, com A = 1011 e B = 0010, ou em decimal, A=11 e B=2. É possível notar o funcionamento dos sinais e da saída do sistema, como na atividade 2, corresponde ao esperado pois C = 01101, ou em decimal, C =13.

3. Atividade 3

3.1 Descrição da atividade

A terceira atividade propôs a elaboração de um testbench em VHDL e simulação no ModelSim para validar automaticamente o somador de 4 bits implementado no item 1 (DUT) contra o somador vetorial do item 2 (golden model). O testbench deve gerar todas as 256 combinações possíveis de A e B (cada um variando de “0000” a “1111”), aguardando 500 ns entre cada nova atribuição, de modo a exercitar todas as entradas. Para cada par (A, B), o DUT instanciado como **SOMADOR4BIT** é comparado com o golden model instanciado como **SOMADOR4BIT_ARITH** e, se as saídas diferirem, uma mensagem de erro é impressa no console, indicando o valor de A, B, a saída esperada (golden) e a saída obtida (DUT). Espera-se que, em condição correta, nunca haja divergência, confirmando a equivalência funcional entre as duas implementações.

Na arquitetura do testbench, declaram-se sinais vetoriais A e B de 4 bits, bem como S_fulladder e S_arith de 5 bits para receber as saídas de cada instância. Em seguida, instanciam-se DUT_fulladder: entity work.somador4b_fulladder port map (A => A, B => B, S => S_fulladder); e DUT_arith: entity work.somador4b_arith port map (A => A, B

=> B, S => S_arith);. Um processo stim_proc faz dois loops aninhados de 0 a 15 (i_int e j_int), convertendo cada inteiro em vetor de 4 bits via std_logic_vector(to_unsigned(...,4)), aplicando a A e B, aguardando wait for 500 ns;, e então comparando S_fulladder e S_arith. Caso haja diferença, o testbench usa report ... severity error; para imprimir detalhes e interromper a simulação. Ao final das 256 combinações sem erro, emite-se um relatório "Simulação concluída sem erros.". Na simulação no ModelSim, a saída indicada é apenas essa mensagem de nota, já que todas as comparações coincidem, confirmando que o somador por full adders e o somador aritmético produzem sempre o mesmo resultado.

2.2 Implementação em VHDL

A Listagem 4 apresenta o código criado para esta atividade.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;


ENTITY TB_SOMADOR4BIT IS
END ENTITY TB_SOMADOR4BIT;
ARCHITECTURE TB_SOMADOR4BIT_ARCH OF TB_SOMADOR4BIT IS
 SIGNAL A, B : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
 SIGNAL S   : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
 UUT: ENTITY work.SOMADOR4BIT
  PORT MAP (
   A => A, B => B,  S => S  );
stim_proc: PROCESS
  VARIABLE Ai, Bi     : INTEGER;
  VARIABLE S_int, exp_int : INTEGER;
```

```
BEGIN
  FOR Ai IN 0 TO 15 LOOP
   FOR Bi IN 0 TO 15 LOOP
    A <= std_logic_vector(to_unsigned(Ai, 4));
    B <= std_logic_vector(to_unsigned(Bi, 4));
    WAIT FOR 500 ns;
     S_int  := to_integer(unsigned(S));
    exp_int := Ai + Bi;


    ASSERT S_int = exp_int
     REPORT "Erro em A=" & INTEGER'IMAGE(Ai) &
          " B=" & INTEGER'IMAGE(Bi) &
          " : S=" & INTEGER'IMAGE(S_int) &
          " esperado=" & INTEGER'IMAGE(exp_int)
     SEVERITY ERROR;
   END LOOP;
  END LOOP;
  REPORT "Teste concluído sem erros." SEVERITY NOTE;
  WAIT;
 END PROCESS stim_proc;
```

Listagem 3: Implementação do circuito do TB_SOMADOR4BIT.

O código define um testbench VHDL chamado TB_SOMADOR4BIT sem portas externas. Na arquitetura TB_SOMADOR4BIT_ARCH, são declarados dois sinais vetoriais de 4 bits (A e B), inicializados em "0000", e um sinal de 5 bits (S) para capturar a saída do somador. Em seguida, instancia-se a unidade sob teste (UUT) referenciando SOMADOR4BIT (o somador do item 1) via PORT MAP (A => A, B => B, S => S). Não há instâncias adicionais, pois o golden model é implícito na verificação por comparação numérica. A ideia é exercitar o DUT em todas as combinações possíveis de entradas e verificar se o valor resultante em S corresponde ao somatório inteiro de Ai + Bi.

O bloco principal de estímulos, "stim_proc", é um processo sequencial que utiliza duas variáveis inteiras (Ai e Bi) para iterar de 0 a 15 em loops aninhados, gerando 256 pares (A, B). Para cada par, converte-se Ai e Bi em vetores de 4 bits via std_logic_vector(to_unsigned()) e atribui-se a A e B. Em seguida, o processo aguarda WAIT FOR 500 ns antes de ler o sinal S e convertê-lo para inteiro com to_integer(unsigned(S)), armazenando em S_int. Ao mesmo tempo, calcula-se exp_int := Ai + Bi, que é o valor esperado da soma. A seguir, o ASSERT S_int = exp_int compara o valor real com o esperado; em caso de discrepância, emite-se uma mensagem de erro detalhando Ai, Bi, S_int e exp_int e aciona uma severidade ERROR que interrompe a simulação. Se todas as 256 combinações forem bem-sucedidas, o processo imprime "Teste concluído sem erros." com SEVERITY NOTE e então entra em modo WAIT, encerrando a verificação sem mais alterações.

2.3 Simulação

Durante a simulação do testbench, os sinais A e B foram atualizados sistematicamente a cada 500 ns, percorrendo todas as 256 combinações de 4 bits de 0 a 15. Concretamente, o processo stim_proc atribuiu a A e B valores crescentes de 0 a 15 (convertidos para STD_LOGIC_VECTOR) e, após cada atribuição, aguardou 500 ns para que o DUT — o somador de 4 bit — apresentasse sua saída em S. Em seguida, o valor de S foi convertido para inteiro (S_int) e comparado com a soma aritmética direta de Ai + Bi (exp_int). Essa comparação era feita pelo comando ASSERT; caso houvesse divergência, uma mensagem de erro seria disparada e a simulação interrompida imediatamente.

Como não houve discrepâncias em nenhuma das 256 combinações, o ASSERT nunca acionou erro, e ao final dos loops aninhados foi exibido no console do ModelSim o relatório **"Teste concluído sem erros".** Isso confirma que, para cada par de entrada (A, B), o SOMADOR4BIT produziu exatamente o mesmo resultado que a soma inteira esperada, validando a implementação funcional do somador de 4 bits.

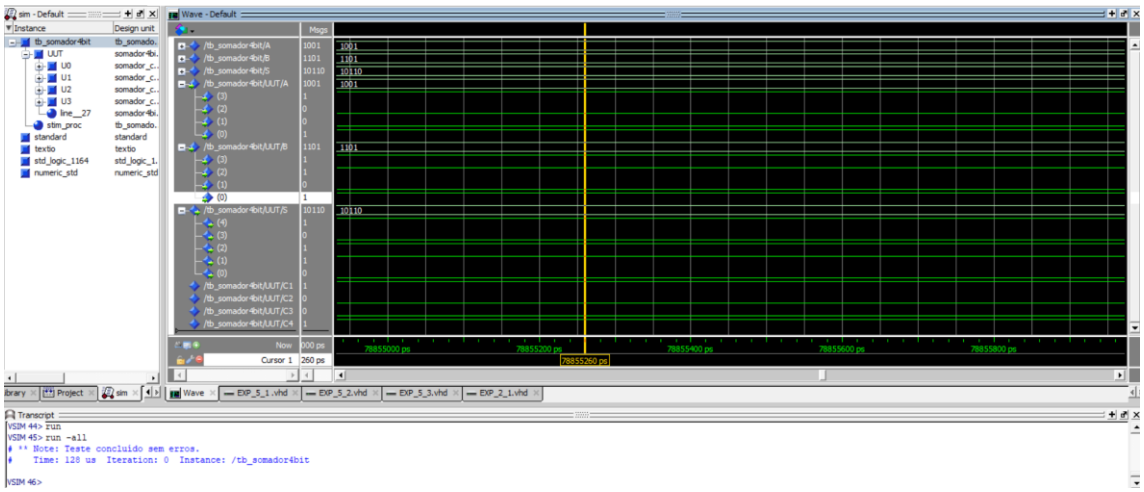

Figura 5: Janela de simulação do ModelSim para o tb_somador4bit. À esquerda, é exibida a hierarquia de instâncias (U0–U3 correspondem aos quatro somadores completos). À direita, os vetores de entrada A = “1001” e B = “1101”, e a saída S = “10110” (representando 9 + 13 = 22). Abaixo, os sinais internos de carry (C1, C2, C3 e C4) indicam a propagação por cada estágio, com C4 = 1 gerando o bit de overflow em S(4). No painel “Transcript”, aparece a mensagem “** Note: Teste concluído sem erros. **” confirmando que todas as 256 combinações foram verificadas com sucesso.