

# Tipos de Dados em VHDL

---

Prof. MSc. André Macário Barros

01/04/2019

# Objetos em VHDL

- Um objeto em VHDL é um item batizado com um identificador que tem um valor.
- Logo, os objetos representam o meio através do qual os valores são passados, trocados e movimentados. São eles:
  - CONSTANT
  - SIGNAL
  - VARIABLE
  - FILE

# CONSTANT

- Como o nome dá a entender, constant é um objeto cujo valor não possa ser alterado.
- A declaração de uma constant é:
  - `CONSTANT constant_name: constant_type := constant_value;`
- Exemplos:
  - `CONSTANT bits: INTEGER := 16;`
  - `CONSTANT words: INTEGER := 2**bits;`
  - `CONSTANT mask: BIT_VECTOR(1 TO 8) := "00001111";`

# CONSTANT

- As constants podem ser declaradas nas partes declarativas da ARCHITECTURE (onde declaramos os sinais)
- Como veremos futuramente, as constants também podem ser declaradas em outras partes declarativas de outras seções do código, como em ENTITY, PACKAGE, PACKAGE BODY, BLOCK, GENERATE, PROCESS e FUNCTION

# SIGNAL

- Servem, como já vimos, para passar valores entre as unidades internas de um circuito.
- A declaração de um sinal é (entre colchetes é opcional):
  - `SIGNAL signal_name: signal_type [range] [:= default_value];`
- Exemplos:
  - `SIGNAL enable: BIT := '0';`
  - `SIGNAL temp: BIT_VECTOR(3 DOWNTO 0);`
  - `SIGNAL byte: STD_LOGIC_VECTOR(7 DOWNTO 0);`
  - `SIGNAL count: NATURAL RANGE 0 TO 255;`

# SIGNAL

- Da mesma forma que as constants, os sinais devem ser declarados, como já vimos, nas partes declarativas das seções do código em: ARCHITECTURE, ENTITY, PACKAGE, BLOCK, GENERATE

# SIGNAL

- Durante a declaração de um sinal, é possível atribuir-se ao mesmo um determinado valor. Quando isso é feito, utiliza-se o “`:=`” para tal, como mostrado no exemplo específico do sinal `enable`
- Durante o escopo restante do código, utiliza-se, como já vimos, o operador de atribuição “`<=`”, quando se deseja atribuir um dado valor ao sinal

# SIGNAL

- Quando se atribui mais de um valor ao longo do código a um sinal:
  - Se for em uma parte concorrente, ocorrerá erro de compilação
  - Se for em uma parte sequencial, isto é, dentro de o escopo de um process (ou uma function como será visto futuramente), o sinal receberá a última atribuição a ele determinada

# VARIABLE

- As variáveis são objetos que representam informações locais a um escopo sequencial (PROCESS) e só podem ser modificadas dentro de tal escopo. Sua atualização é imediata, tão logo seja a ela atribuído um dado valor.
- A declaração de uma variável é (entre colchetes é opcional):
  - VARIABLE variable\_name:
  - variable\_type [range] [:= default\_value];
- Exemplos:
  - VARIABLE flip: STD\_LOGIC := '1';
  - VARIABLE address: STD\_LOGIC\_VECTOR(0 TO 15);
  - VARIABLE counter: INTEGER RANGE 0 TO 127;

# VARIABLE

- Durante a declaração de uma variável e ao longo do restante do escopo do código, utiliza-se o “:=” para atribuir-lhe valores.

# SIGNAL *versus* VARIABLE

- Sinais são, portanto, diferentes de variáveis e não devem ser usados de maneira intercambiável
- Cada objeto, como será visto futuramente, comportar-se de maneira diferente ao longo do código

# FILE

- É o quarto e último tipo de objeto
- A declaração de um file desdobra-se em duas linhas:
  - `TYPE type_name IS FILE OF type_in_file;`
  - `FILE file_identifier: type_name [ [OPEN open_mode] IS expression];`
- Exemplo:
  - `TYPE bit_file IS FILE OF BIT;`
  - `FILE file01: bit_file IS "my_file.txt";`

# Bibliotecas

- Como já vimos, a VHDL conta com a biblioteca **std\_logic\_1164**
- Porém há outras também padronizadas pelo IEEE: **standard**, **numeric\_bit**, **numeric\_std**, **textio**, e **numeric\_std\_unsigned(signed)**
- Bibliotecas como a **numeric\_std** permitirão a chamada a funções que em muito nos livrarão de diversos passos que até então eram necessários serem feitos via mapa de Karnaugh, como veremos adiante

# Tipos de dados pré-definidos e definidos pelo usuário

- Pré-definido (não precisa criar):
  - `TYPE INTEGER IS RANGE -2147483647 TO 2147483647;`
  - `TYPE BIT IS ('0', '1');`
- Definido pelo usuário:
  - `TYPE grade IS RANGE 0 TO 10;`
  - `TYPE color IS (red, green, blue);`

# Vetores multidimensionais

- `TYPE type1 IS ARRAY (POSITIVE RANGE <>) OF INTEGER;`
- `CONSTANT const1: type1(1 TO 4) := (5, -5, 3, 0);`
- -----
- `TYPE type2 IS ARRAY (0 TO 3) OF NATURAL;`
- `CONSTANT const2: type2 := (2, 0, 9, 4);`
- -----
- `TYPE type3 IS ARRAY (1 TO 2) OF type2;`
- `CONSTANT const3: type3 := ((5, 5, 7, 99), (33, 4, 0, 0));`
- -----
- `TYPE type3 IS ARRAY (NATURAL RANGE <>) OF BIT_VECTOR(2 DOWNTO 0);`
- `CONSTANT const3: type3(1 DOWNTO 0) := ("000", "111");`
- `CONSTANT const3: type3(1 DOWNTO 0) := (('0','0','0'), ('1','1','1'));`

# Vetores em VHDL

- Prosseguindo com nossos estudos, você irá aprender hoje mais um tipo de dado que existe dentro do pacote **std\_logic\_1164** da biblioteca **ieee**
- É o tipo de dado denominado **std\_logic\_vector**
- Este tipo de dado é apropriado para os casos em que trabalhamos com estruturas que estão organizadas em agrupamentos, por exemplo, uma via de dados de 32 *bits*, uma porta de 4 *bits*.

# Como declaramos um vetor?

- Um vetor é declarado como no exemplo a seguir:

```
1-entity ccto is  
2-port (  
3-    a, b: in std_logic_vector(3 downto 0);  
4-    y: out std_logic);  
5-end ccto;
```

- Observe a declaração da linha 3:

- Ali estamos declarando dois vetores,  $a$  e  $b$ , ambos com 4 bits, dispostos da seguinte forma:  $(a_3, a_2, a_1, a_0)$  e  $(b_3, b_2, b_1, b_0)$

# Como declaramos um vetor?

```
1-entity ccto is
2-port(
3-    a, b: in std_logic_vector(3 downto 0);
4-    y: out std_logic);
5-end ccto;
```

- Linha 4:
  - Estamos declarando nesta linha que o circuito possui uma saída simples, *y*, um tipo escalar, o **std\_logic**, que é o tipo de dado do pacote *std\_logic\_1164* com o qual estávamos trabalhando aqui.

# Como declaramos um vetor?

```
1-entity ccto is
2-port(
3-    a, b: in std_logic_vector(3 downto 0);
4-    y: out std_logic);
5-end ccto;
```

- O circuito é simbolizado pelo seguinte desenho:



# Como referenciamos os elementos de um vetor?

- Vamos supor que este circuito deva associar as entradas  $a$  e  $b$  para produzir uma saída  $y$  de acordo com a seguinte operação:
  - $y = (a_1 \oplus b_3) \cdot (a_0 + b_2)$
- Esta operação deverá aparecer no escopo da *architecture* do código VHDL correspondente da seguinte forma:

```
y <= (a(1) xor b(3)) and (a(0) or b(2));
```

# Operações diversas com vetores

- **SIGNAL a, b: BIT\_VECTOR(7 DOWNTO 0);**
- **SIGNAL x, y: BIT\_VECTOR(7 DOWNTO 0);**
- **SIGNAL v: BIT\_VECTOR(1 TO 8);**
- **SIGNAL w: BIT;**
- **x <= "11110000";**
- **y <= a XOR b;**
- **v <= a SLL 2; --deslocar à esq 2 un**
- **w <= '1' WHEN a > b ELSE '0';**

# Exemplo 1

- Desenvolva o circuito e o código VHDL correspondente a um circuito GT (*greater than*) de dois bits
- Solução: este é o Exercício 3 da aula de “Introdução à VHDL”
- Procuramos por um circuito que retorne ‘1’ toda vez que o vetor  $a$  for maior que o vetor  $b$ .
- Exemplo:
  - Se  $a = "10"$  (2) e  $b = "01"$  (1), então  $y$  retornará ‘1’
  - Se  $a = "00"$  (0) e  $b = "11"$  (3), então  $y$  retornará ‘0’

# Exemplo 1

- Não sabemos ainda como o circuito é implementado. Porém já podemos representar suas entradas e saída de acordo com o desenho a seguir:



- Veja a seguir a tabela-verdade do circuito GT

# Exemplo 1

- Tabela-verdade do circuito GT:
- Veja o mapa-K correspondente abaixo. Por exemplo, dois de seus seis mintermos estão aqui destacados.

$a_1 a_o / b_1 b_o$	00	01	11	10
00	1	1	1	
01		1	1	
11				
10			1	

$a_1$	$a_o$	$b_1$	$b_o$	$y$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

# Exemplo 1

- Uma combinação possível de agrupamentos dos mintermos para a obtenção da função lógica mínima –  $y_{mín}$  – podem ser:

$a_1 a_o /$ $b_1 b_o$	00	01	11	10
00		1	1	1
01			1	1
11				
10			1	

- Portanto,  $y_{mín}$  vale:
- $y = (m_{0100} + m_{1100}) + (m_{1100} + m_{1110}) + (m_{1100} + m_{1101} + m_{1000} + m_{1001})$
- $y = a_o b_1' b_o' + a_1 a_o b_o' + a_1 b_1'$

# Exemplo 1

- A saída  $y_{mín}$  é, portanto:
- $y = a_o b_1' b_o' + a_1 a_o b_o' + a_1 b_1'$
- E, a parte do código VHDL correspondente será:
  - $y \leq (a(0) \text{ and } \text{not}(b(1)) \text{ and } \text{not}(b(0)))$   
or  
 $(a(1) \text{ and } a(0) \text{ and } \text{not}(b(0)))$   
or  
 $(a(1) \text{ and } \text{not}(b(1))));$

# Exemplo 1

- Podemos, como já vimos anteriormente, utilizar sinais para diminuirmos a extensão da função lógica (e diminuirmos as chances de erro)
- $y = a_o b_i' b_o' + a_i a_o b_o' + a_i b_i'$
- $s_0 \leq a(0) \text{ and } \text{not}(b(1)) \text{ and } \text{not}(b(0));$
- $s_1 \leq a(1) \text{ and } a(0) \text{ and } \text{not}(b(0));$
- $s_2 \leq a(1) \text{ and } \text{not}(b(1));$
- $y \leq s_0 \text{ or } s_1 \text{ or } s_2;$

# Exemplo 1 – Código VHDL final

```
• library ieee;
• use ieee.std_logic_1164.all;
• entity gt_2b is
• Port(a, b:  in std_logic_vector(1 downto 0);
•          y: out std_logic);
• end gt_2b;
• architecture arq of gt_2b is
•     signal s0, s1, s2: std_logic;
• begin
•     s0 <= a(1) and not(b(1));
•     s1 <= a(0) and not(b(1)) and not(b(0));
•     s2 <= a(1) and      a(0)      and not(b(0));
•     y   <= s0 or s1 or s2;
• end arq;
```

# Exemplo 1

- Agora precisamos saber se nosso código de fato reproduz o que a tabela-verdade do circuito GT exige
- Para tal, efetuaremos a simulação através do código de *testbench* apresentado a seguir

# Exemplo 1

## código

## VHDL

### *testbench*

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-entity tb_gt_2b is
• 04-end tb_gt_2b;
• 05-architecture arq of tb_gt_2b is
• 06-    signal ta, tb: std_logic_vector(1 downto 0);
• 07-    signal      ty: std_logic;
• 08-begin
• 09-    uut: entity work.gt_2b(arq)
• 10-    port map(a => ta, b => tb, y => ty);
• 11-    process begin
• 12-        ta <= "00"; tb <= "00"; wait for 10 ns;
• 13-        ta <= "00"; tb <= "01"; wait for 10 ns;
• 14-        ...
• 25-        ta <= "11"; tb <= "10"; wait for 10 ns;
• 26-        ta <= "11"; tb <= "11"; wait for 10 ns;
• 27-    end process;
• 28-    end arq;
```

# Exemplo 1 – código VHDL *testbench*

- ...
- 05-architecture arq of tb\_gt\_2b is
- 06- signal ta, tb: std\_logic\_vector(1 downto 0);
- 07- signal ty: std\_logic;
- ...
- Linha 06:
  - Esta linha apresenta a declaração dos dois sinais que serão mapeados para as entradas *a* e *b*
  - Observem que a notação utilizada é a que você aprendeu para declarar sinais, no entanto é um vetor para que o mapeamento ocorra de vetor (*a*, *b*) para vetor (*ta*, *tb*)

# Exemplo 1 – código VHDL *testbench*

- ...
- 09- uut: entity work.gt\_2b(arq)
- 10- port map(a => ta, b => tb, y => ty);
- ...
- Linha 10:
  - Observe que este mapeamento aplica-se às componentes do vetor. A declaração da linha 10 poupa esforço e tem o mesmo efeito que esta:

```
port map(a(1) => ta(1), a(0) => ta(0),  
         b(1) => tb(1), b(0) => tb(0),  
         y => ty);
```

# Exemplo 1 – código VHDL *testbench*

- ...
- 12-      ta <= "00"; tb <= "00"; wait for 10 ns;
- 13-      ta <= "00"; tb <= "01"; wait for 10 ns;
- ...
- 26-      ta <= "11"; tb <= "10"; wait for 10 ns;
- 27-      ta <= "11"; tb <= "11"; wait for 10 ns;
- ...
- Linha 12: observe a diferença
  - Sempre que queremos atribuir um valor a um vetor, usamos as aspas duplas: ta <= "00"; se quiséssemos atribuir um escalar a um componente do vetor, as declarações seriam com aspas simples e gerariam o mesmo efeito: ta(1) <= '0'; ta(0) <= '0';

# Exemplo 1 – código VHDL *testbench*

- ...
- 12-      ta <= "00"; tb <= "00"; wait for 10 ns;
- 13-      ta <= "00"; tb <= "01"; wait for 10 ns;
- ...
- 26-      ta <= "11"; tb <= "10"; wait for 10 ns;
- 27-      ta <= "11"; tb <= "11"; wait for 10 ns;
- ...
- Linhas 12 à 27:
  - Para que saibamos se nosso circuito GT irá funcionar, deveremos processar todos os estímulos possíveis aos sinais *ta* e *tb* (iguais aos da tabela-verdade), precisando portanto de 16 combinações diferentes ("00" "00" até "11" "11"). Por este motivo há o símbolo de reticências "..." entre as linhas 13 e 26

# Exemplo 1 – código de *testbench*

- O princípio adotado no código de *testbench* apresentado é o mesmo que foi utilizado nas simulações com entradas escalares apresentado anteriormente.
- Caso queiramos prosseguir com este mesmo princípio para gerarmos os estímulos em um arquivo de *testbench*, teremos de criar, por exemplo, 256 linhas de estímulos para um circuito com oito entradas, 1024 linhas para dez entradas e assim por diante.
- Para que tal dificuldade seja resolvida, observe o código a seguir, que utiliza uma estrutura de repetição e por isto é diminui o esforço.

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all;
• 04-entity tb_gt_2b is
• 05-end tb_gt_2b;
• 06-architecture arq of tb_gt_2b is
• 07-    signal ta, tb: std_logic_vector(1 downto 0);
• 08-    signal      ty: std_logic;
• 09-begin
• 10-    uut: entity work.gt_2b(arq)
• 11-    port map(a => ta, b => tb, y => ty);
• 12-    process
• 13-        variable i, j: integer := 0;
• 14-        begin
• 15-            for i in 0 to 3 loop
• 16-                for j in 0 to 3 loop
• 17-                    ta <= std_logic_vector(to_unsigned(i,2));
• 18-                    tb <= std_logic_vector(to_unsigned(j,2));
• 19-                    wait for 10 ns;
• 20-                end loop;
• 21-            end loop;
• 22-        end process;
• 23-end arq;
```

# Exemplo 1

## código de *testbench* com *loop*

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all; (line circled)
• 04-entity tb_gt_2b is
• 05-end tb_gt_2b;
• 06-architecture arq of tb_gt_2b is
• 07-    signal ta, tb: std_logic_vector(1 downto 0);
• 08-    signal      ty: std_logic;
• 09-begin
• 10-    uut: entity work.gt_2b(arq)
• 11-    port map(a => ta, b => tb, y => ty);
• 12-    process
• 13-        variable i, j: integer := 0;
• 14-        begin
• 15-            for i in 0 to 3 loop
• 16-                for j in 0 to 3 loop
• 17-                    ta <= std_logic_vector(to_unsigned(i,2));
• 18-                    tb <= std_logic_vector(to_unsigned(j,2));
• 19-                    wait for 10 ns;
• 20-                end loop;
• 21-            end loop;
• 22-        end process;
• 23-end arq;
```

# Exemplo 1

## código de *testbench* com *loop*

# Exemplo 1 – código de *testbench* com *loop*

- 01-library ieee;
- 02-use ieee.std\_logic\_1164.all;
- 03-use ieee.numeric\_std.all;
- ...
- Linha 03:
  - Precisaremos utilizar um novo pacote, chamado **numeric\_std**, também pertencente à biblioteca **ieee**
  - Sempre precisaremos utilizar este pacote quando tivermos em nosso código o tipo signed ou unsigned ou funções que utilizem estes tipos.
  - Explicaremos onde tipo de dado será necessário na sequência

# Exemplo 1

## código de *testbench* com loop

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all;
• 04-entity tb_gt_2b is
• 05-end tb_gt_2b;
• 06-architecture arq of tb_gt_2b is
• 07-    signal ta, tb: std_logic_vector(1 downto 0);
• 08-    signal      ty: std_logic;
• 09-begin
• 10-    uut: entity work.gt_2b(arq)
• 11-    port map(a => ta, b => tb, y => ty);
• 12-    process
• 13-        variable i, j: integer := 0;
• 14-    begin
• 15-        for i in 0 to 3 loop
• 16-            for j in 0 to 3 loop
• 17-                ta <= std_logic_vector(to_unsigned(i,2));
• 18-                tb <= std_logic_vector(to_unsigned(j,2));
• 19-                wait for 10 ns;
• 20-            end loop;
• 21-        end loop;
• 22-    end process;
• 23-end arq;
```

# Exemplo 1 – código de *testbench* com *loop*

- ...
- 12-process
- 13- variable i, j: integer := 0;
- 14-begin
- ...
- Linha 13:
  - Esta linha declara duas variáveis do tipo inteiro. Um inteiro pode receber valores de 32 bits entre -2147483647 e +2147483647
  - Observem que de fato *i* e *j* são **variáveis**, ao contrário dos **sinais** *ta*, *tb* e *ty*.
  - *i* e *j* são declaradas, ao mesmo tempo iniciadas em zero. Declarações de variáveis precisam estar compreendidas dentro do escopo de um PROCESS.

# Exemplo 1

## código de *testbench* com loop

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all;
• 04-entity tb_gt_2b is
• 05-end tb_gt_2b;
• 06-architecture arq of tb_gt_2b is
• 07-    signal ta, tb: std_logic_vector(1 downto 0);
• 08-    signal      ty: std_logic;
• 09-begin
• 10-    uut: entity work.gt_2b(arq)
• 11-    port map(a => ta, b => tb, y => ty);
• 12-    process
• 13-        variable i, j: integer := 0,
• 14-    begin
• 15-        for i in 0 to 3 loop
• 16-            for j in 0 to 3 loop
• 17-                ta <= std_logic_vector(to_unsigned(i,2));
• 18-                tb <= std_logic_vector(to_unsigned(j,2));
• 19-                wait for 10 ns;
• 20-            end loop;
• 21-        end loop;
• 22-    end process;
• 23-end arq;
```

# Exemplo 1 – código de *testbench* com *loop*

- ...
- 15-for i in 0 to 3 loop
- 16- for j in 0 to 3 loop
- 17- ta <= std\_logic\_vector(to\_unsigned(i,2));
- 18- tb <= std\_logic\_vector(to\_unsigned(j,2));
- 19- wait for 10 ns;
- 20- end loop;
- 21-end loop;
- ...
- Linhas 15 a 21:
  - Este *loop* faz os sinais ta e tb receberem os 16 valores necessários, sem que haja a necessidade de defini-los um a um, como no *testbench* anterior.

# Os tipos signed e unsigned

- Vamos explicar inicialmente o que são os tipos de dados signed e unsigned para prosseguirmos com a descrição deste código de *testbench*
- Estes tipos de dados têm a mesma “aparência” que um vetor do tipo std\_logic\_vector. A diferença é que admitem operações aritméticas, sendo o tipo signed com sinal e o unsigned sem sinal
- Portanto, nós podemos por exemplo, somar dois vetores signed ou unsigned.

# A função `to_unsigned`

- Agora vamos entender o que faz a função `to_unsigned(int, numbits)`
- Esta função transforma um inteiro em um vetor `unsigned`
- A conversão ocorre da seguinte forma: o inteiro declarado no argumento `int` será convertido em um vetor `unsigned` com um número de *bits* igual ao argumento `numbits`. Logo:
  - `to_unsigned(4, 3)` retorna “`(100)unsigned`”;
  - `to_unsigned(11, 3)` retorna erro (não ocorre síntese do código, pois 11 precisa de quatro bits);
  - `to_unsigned(14, 4)` retorna “`(1110)unsigned`”;

# A função std\_logic\_vector

- E agora vamos entender o que faz a função `std_logic_vector(arg)`
- Esta função transforma um vetor `unsigned` ou `signed` (que tenha sido colocado como argumento em `arg`) em um vetor `std_logic_vector`
- Sempre que necessitarmos operar com tipos `signed/unsigned` ou funções que recebam ou gerem tais tipos, é necessária a inclusão do pacote `std_numeric` no cabeçalho
- Com base nestas explicações, voltemos agora ao código de *testbench* com *loop*

# Exemplo 1 – código de *testbench* com *loop*

- ...
- 15-for i in 0 to 3 loop
- 16- for j in 0 to 3 loop
- 17- ta <= std\_logic\_vector(to\_unsigned(i,2));
- 18- tb <= std\_logic\_vector(to\_unsigned(j,2));
- 19- wait for 10 ns;
- 20- end loop;
- 21-end loop;
- ...
- Linhas 17 e 18: Estas duas declarações convertem os inteiros gerados no *loop* (0, 1, 2 e 3) em vetores std\_logic\_vector (“00”, “01”, “10” e “11”) a serem atribuídos aos sinais ta e tb

# Exemplo 1 – código de *testbench* com *loop*

- ...
- 15-for i in 0 to 3 loop
- 16- for j in 0 to 3 loop
- 17- ta <= std\_logic\_vector(to\_unsigned(i,2));
- 18- tb <= std\_logic\_vector(to\_unsigned(j,2));
- 19- wait for 10 ns;
- 20- end loop;
- 21-end loop;
- ...
- Linhas 17 e 18: A conversão é feita em duas etapas. Primeiramente de inteiro para unsigned e depois de unsigned para std\_logic\_vector
- Não há função de conversão direta de inteiro para std\_logic\_vector

# Exemplo 1 – código de *testbench* com *loop*

- ...
- 15-for i in 0 to 3 loop
- 16- for j in 0 to 3 loop
- 17-       ta <= std\_logic\_vector(to\_unsigned(i,2));
- 18-       tb <= std\_logic\_vector(to\_unsigned(j,2));
- 19-       wait for 10 ns;
- 20- end loop;
- 21-end loop;
- ...
- Linhas 15 a 21: O *loop* fixa **ta** primeiramente e gera todas as combinações de **tb**. Em seguida incrementa **ta** de um valor e repete as combinações para **tb**. Isto se deve ao *loop* de **tb** ser interno (regulado pela variável **j**), da mesma forma que o *loop* de **ta** ser externo (regulado pela variável **i**)

```

• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all;
• 04-entity tb_gt_2b is
• 05-end tb_gt_2b;
• 06-architecture arq of tb_gt_2b is
• 07-    signal ta, tb: std_logic_vector(1 downto 0);
• 08-    signal      ty: std_logic;
• 09-begin
• 10-    uut: entity work.gt_2b(arq)
• 11-    port map(a => ta, b => tb, y => ty);
• 12-    process ←
• 13-        variable i, j: integer := 0;
• 14-    begin
• 15-        for i in 0 to 3 loop
• 16-            for j in 0 to 3 loop
• 17-                ta <= std_logic_vector(to_unsigned(i,2));
• 18-                tb <= std_logic_vector(to_unsigned(j,2));
• 19-                wait for 10 ns;
• 20-            end loop;
• 21-        end loop;
• 22-    end process;
• 23-end arq;

```

# Exemplo 1

## código de *testbench* com *loop*

escopo process para  
declarações sequenciais

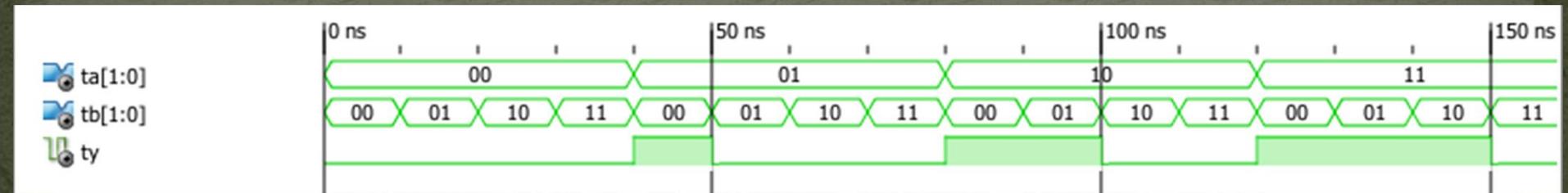
# Exemplo 1 – código de *testbench* com *loop*

- Linhas 15 a 21: Somente relembrando que este *loop*, por estar dentro do escopo de um process, executa cada uma destas combinações uma após a outra. Daí a existência do `wait for` na linha 19
- Portanto, a sequência de sinais gerados é:
  - `ta = "oo"; tb="oo";` espera de 10 ns;
  - `ta = "oo"; tb="01";` espera de 10 ns;
  - `ta = "oo"; tb="10";` espera de 10 ns;
  - ...
  - `ta = "11"; tb="10";` espera de 10 ns;
  - `ta = "11"; tb="11";` espera de 10 ns;
  - fim do *loop*

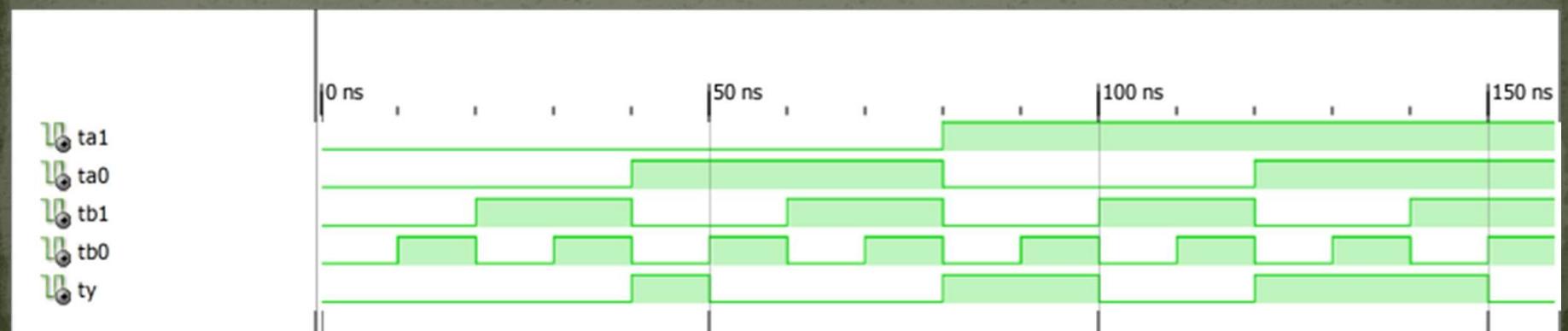
# Exemplo 1 – código de *testbench* com *loop*

- ...
- 15-for i in 0 to 3 loop
- 16- for j in 0 to 3 loop
- 17- ta <= std\_logic\_vector(to\_unsigned(i,2));
- 18- tb <= std\_logic\_vector(to\_unsigned(j,2));
- 19- wait for 10 ns;
- 20- end loop;
- 21-end loop;
- ...
- Linhas 15 a 21: se tivéssemos, por exemplo, dois vetores, cada um com três *bits* na entrada, nossas variáveis *i* e *j* precisariam variar nas linhas de 0 a 7 nas linhas 15 e 16, da mesma forma que o “2” das linhas 17 e 18 seria “3”

# Exemplo 1 – resultado da simulação



- Observe acima que nesta simulação a visibilidade e compreensão dos sinais é muito mais clara e lógica do que se escolhessemos trabalhar com quatro escalares na síntese e no *testbench* ( $a_1$ ,  $a_o$ ,  $b_1$  e  $b_o$  – veja abaixo)



## Exemplo 2

- Desenvolver, implementar e simular em VHDL um circuito conversor binário → gray de 3 bits, conforme tabela-verdade a seguir:

$b_2$	$b_1$	$b_o$	$g_2$	$g_1$	$g_o$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

## Exemplo 2

- Conversor binário → *gray* de 3 bits.
- É o exercício 4 da aula “Introdução à VHDL”
- Circuito “caixa preta”, somente com as entradas e saídas.



# Exemplo 2 – Conversor binário → gray

## Código VHDL

```
• library ieee;  
• use ieee.std_logic_1164.all;  
• entity conversor is  
• port(  
•     b: in std_logic_vector(2 downto 0);  
•     g: out std_logic_vector(2 downto 0));  
• end conversor;  
• architecture arq of conversor is  
• begin  
•     g(2) <= b(2);  
•     g(1) <= b(2) xor b(1);  
•     g(0) <= b(1) xor b(0);  
• end arq;
```

# Exemplo 2 – Conversor binário → gray

## Código VHDL

```
• 01-library ieee;
• 02-use ieee.std_logic_1164.all;
• 03-use ieee.numeric_std.all;
• 04-entity tb_conversor is
• 05-end tb_conversor;
• 06-architecture arq of tb_conversor is
• 07-    signal tb, tg: std_logic_vector(2 downto 0);
• 08-begin
• 09-    uut: entity work.conversor(arq)
• 10-        port map(b => tb, g => tg);
• 11-    process
• 12-        variable i: integer := 0;
• 13-    begin
• 14-        for i in 0 to 7 loop
• 15-            tb <= std_logic_vector(to_unsigned(i,3));
• 16-            wait for 10 ns;
• 17-        end loop;
• 18-    end process;
• 19-end arq;
```

# Operações com vetores

- Há operadores e operações que nos auxiliam na manipulação de vetores
- & é um operador de concatenação. Este operador junta dois escalares, dois vetores ou ambos em um vetor maior.
- Observe os exemplos a seguir.

# Operadores para vetores

- signal a1: std\_logic;
- signal a4: std\_logic\_vector(3 downto 0);
- signal b8, c8, d8: std\_logic\_vector(7 downto 0);
- signal a20: std\_logic\_vector(19 downto 0);
- a1 <= '1';
- a4 <= "1010";
- b8 <= a4 & a4; --b8 recebe "10101010"
- c8 <= a1 & a1 & a4 & "00"; --c8 recebe "11101000"
- --d8 recebe o nibble inferior de b8 (1010) concatenado
- --com o nibble superior de c8 (1110).
- --Portanto: o comando a seguir faz d8 receber "10101110"
- d8 <= b8(3 downto 0) & c8(7 downto 4);
- --o comando abaixo faz a20 receber "11101000111111111111"
- a20 <= d8(3 downto 0) & c8(3 downto 0) & "111111111111";
- a20 <= "000000000000000000000000";
- --o comando abaixo faz o mesmo que o da linha acima
- a20 <= (others => '0');

# Referências

- CHU, Pong P. **FPGA Prototyping by VHDL examples.** New Jersey (NJ): John Wiley & Sons, 2008. Há 8 exemplares disponíveis na biblioteca. Número de chamada: 621.395 C559f