

Sistemas Reconfiguráveis


Eng. de Computação

Profs. Francisco Garcia e Antônio Hamilton Magalhães

Aula 3 – Linguagem de descrição de *hardware* (HDL)

Unidades fundamentais, classes, tipos e operadores


Linguagem de descrição de *hardware* (HDL)



Uma **linguagem de descrição de *hardware*** (em inglês: *hardware description language* – HDL) é uma linguagem usada para uma **descrição formal e projetos de circuitos eletrônicos**, sendo mais comumente usada para circuitos **digitais**, podendo, no entanto, ser usada também para circuitos **analógicos** e **mistos**.

- Pode **descrever o funcionamento** de um circuito, a sua **concepção** e **organização** e, ainda, **testá-lo** para verificar seu funcionamento por meio de **simulação**.
- Usando **expressões, declarações e estruturas de controle**, é uma descrição textual da **estrutura espacial, temporal e comportamental** dos **sistemas eletrônicos**.

Linguagem de descrição de *hardware* (HDL)




Uma linguagem de descrição de hardware é parte integral de um **sistema de automação de projeto eletrônico** (*electronics design automation* – EDA), usada sobretudo em **circuitos complexos**, tais como circuitos integrados de aplicações específicas (*application-specific integrated circuits* – **ASICs**), **microprocessadores** e dispositivos lógicos programáveis (*programmable logic devices* – **PLDs**).

A linguagem pode ser usada simplesmente para **descrição** do **funcionamento**, para **simulação**, como também visando a **síntese**, onde, depois da compilação, é gerado um conjunto de **máscaras para a fabricação** de um circuito integrado ou um arquivo para a configuração de um PLD.

Em engenharia de computação, **síntese lógica** é um processo pelo qual uma **especificação abstrata da estrutura ou do comportamento desejado de um circuito é convertido para a implementação** do projeto em termos de portas lógicas, tipicamente por um programa de computador chamado ferramenta de síntese

Linguagem de descrição de *hardware* (HDL)



As primeiras linguagens de descrição de *hardware* apareceram ainda na década de 1960, mas sua utilização se deu mais intensamente a partir da década de 1970 e, principalmente, depois de 1980.

Algumas HDLs:

- ABEL
- AHDL
- PALASM
- Verilog
- VHDL

As HDLs mais usadas atualmente são Verilog e VHDL.

Nessa disciplina de Sistemas Reconfiguráveis, trabalharemos com a linguagem **VHDL**

Linguagem de descrição de *hardware* (HDL)

Algumas linguagens de descrição de hardware são **padrões internacionais** e, dessa forma, são usadas por vários fornecedores de maneira uniforme. Outras são exclusivas de um determinado fornecedor (**proprietárias**)

Dentre as vantagens de uma **linguagem padronizada** temos:

- **Independência do fornecedor**
- **Portabilidade**
- **Reusabilidade**

Linguagem de descrição de *hardware* (HDL)

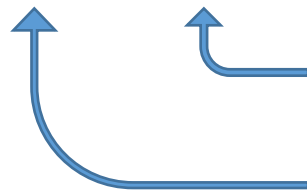
Uma outra forma de se fazer a descrição de um circuito é através de um **diagrama esquemático**. Fazer a descrição através de uma linguagem de descrição de hardware tem, entretanto, as seguintes vantagens:

- Projeto **independente** da **tecnologia**;
- Facilidade de **atualização** dos projetos;
- **Redução** do **tempo** de projeto e custos;
- Redução do **volume** da **documentação**.

VHDL



VHDL = VHSIC HDL



*Hardware **description language***

Very high speed integrated circuit

- A **VHDL** foi desenvolvida em **1983** como um projeto do **Departamento de Defesa dos Estados Unidos**, com a finalidade de documentar o comportamento de **ASICs** que as companhias **fornecedoras** estavam incluindo em seus equipamentos.
- Foi baseada na linguagem de programação **Ada**, tanto nos **conceitos** quanto na **sintaxe**, para evitar reinventar conceitos que já haviam sido exhaustivamente testados no desenvolvimento de Ada.

VHDL

Ada Lovelace, foi uma matemática e escritora inglesa. Hoje é reconhecida principalmente por ter escrito o primeiro algoritmo para ser processado por uma máquina, a máquina analítica de Charles Babbage (1842).

Por esse trabalho é considerada a primeira programadora de toda a história.



VHDL

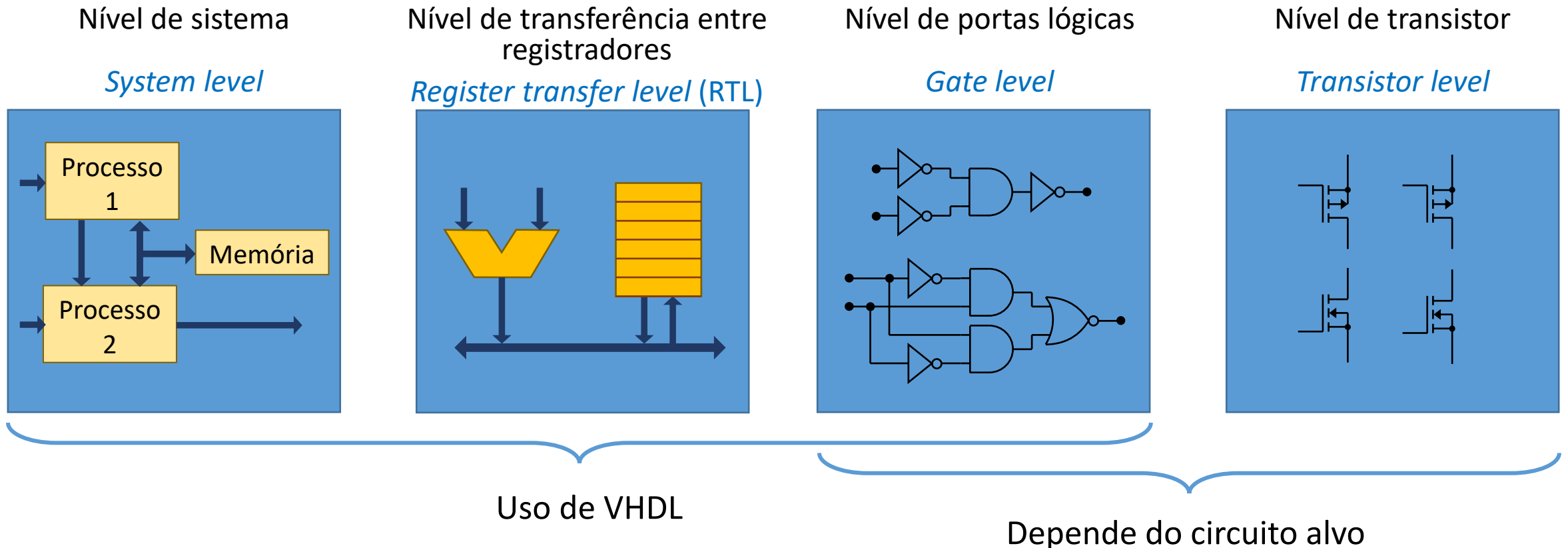


- Foi oficializada como um padrão internacional pelo Instituto de Engenheiros Eletricistas e Eletrônicos (*Institute of Electrical and Electronics Engineers* – **IEEE**) em **1987**, como padrão **IEEE 1076-1987**.
- Esse padrão sofreu uma revisão em **1993**, com poucas alterações. Outras revisões em **2000, 2002, 2008 e 2019**.
- O padrão original incluía uma ampla gama de tipos de dados, incluindo **numérico** (inteiro e real), **lógico** (bit e booleano), **caractere** e **tempo**, além de matrizes de bits chamadas **bit_vector** e de caracteres denominada **string**.

VHDL

Níveis de abstração

Um determinado projeto eletrônico pode ser descrito em diferentes níveis:



VHDL



Objetivo da linguagem

- Descrever a estrutura
- Descrever o comportamento



de circuitos digitais



Preferível



VHDL



Uso da linguagem

- Documentação
- Descrição
- Síntese
- Simulação
- Testes

de circuitos digitais baseados em PLDs ou ASICs

Obs.: A linguagem VHDL foi desenvolvida originalmente para **descrição de circuitos digitais** e **não** para a **síntese**. Deste modo, **nem todas** as construções possíveis da linguagem são **sintetizáveis**

VHDL

Unidades fundamentais

A linguagem VHDL possui três **unidades fundamentais**:

- LIBRARY
- ENTITY
- ARCHITECTURE

VHDL - LIBRARY

LIBRARY

- Contém uma lista de todos os pacotes e bibliotecas usados no projeto

- Sintaxe:

`LIBRARY library_name;`

`USE library_name.package_name.package_parts;`

Termina com
ponto e
vírgula

- Exemplo:

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;
```

VHDL não é sensível
a maiúsculas /
minúsculas

A unidade **LIBRARY** não é obrigatória.

VHDL - ENTITY

ENTITY

- Especifica as entrada e saídas de um circuito ou sub-circuito (componente)

- Sintaxe:

```
ENTITY entity_name IS
    PORT (
        port_name : signal_mode signal_type;
        ...      ....      ...
        port_name : signal_mode signal_type
    );
END entity_name;
```

Deve ser o mesmo nome do arquivo (.vhd)

O último não tem ponto e vírgula

ou então:
END ENTITY;

VHDL - ENTITY



ENTITY

A declaração PORT é uma lista, entre parênteses, de itens separados por ponto e vírgula. Assim, o último não usa ponto e vírgula. Apenas o ponto e vírgula após o parênteses para fechar a declaração.

```
(~~~~~ ; ~~~~~ ; ~~~~~);.
```


VHDL - ENTITY

ENTITY

- Signal_mode: IN, OUT, BUFFER, INOUT
 - IN: entrada
 - OUT: exclusivamente saída (não pode ser lida)
 - BUFFER: saída que pode ser lida
 - INOUT: entrada / saída (bidirecional)
- Signal_type: BIT, BOOLEAN, INTEGER, NATURAL, POSITIVE, REAL, TIME, CHARACTER

Obs.: Se **x** é um objeto do modo OUT, então a segunda expressão abaixo é inválida:

```
x <= a OR b;  
y <= c AND x;
```

x não poderia
estar do lado
direito de uma
expressão

Se **x** for do modo BUFFER, é válida.

Outros tipos podem ser utilizados, dependendo do **PACKAGE** declarado na **LIBRARY**.

VHDL - ENTITY

ENTITY

- Exemplo:

```
ENTITY Cap3ex1 IS
  PORT (
    a      : IN BIT;
    b      : IN BIT;
    x,y,z  : OUT BIT
  );
END Cap3ex1;
```

O arquivo deve ser salvo com o nome de **Cap3ex1.vhd**

Pode ser também:
a, b : IN BIT;

Mesmo modo, mesmo tipo.

VHDL - ARCHITECTURE

ARCHITECTURE

- Contém o código VHDL propriamente dito, que descreve como o circuito deve funcionar.
- Sintaxe:

ARCHITECTURE architecture_name **OF** entity_name **IS**

[declarações;

.....

BEGIN

código;

.....

END architecture_name;

O texto
entre [] é
opcional

ou então:

END ARCHITECTURE;

VHDL - ARCHITECTURE

ARCHITECTURE

- Exemplo:

```
ARCHITECTURE arch OF Cap3ex1 IS
BEGIN
    x <= a AND b;      -- saída x
    y <= a OR b;       -- saída y
    z <= a XOR b;      -- saída z
END arch;
```

Nome da
ENTITY

Essa ARCHITECTURE,
não tem declarações

Esse código descreve o
mesmo circuito que, na
aula 2, foi descrito através
de diagrama esquemático.

Em VHDL, os comentários
começam com -- e vão
até o fim da linha

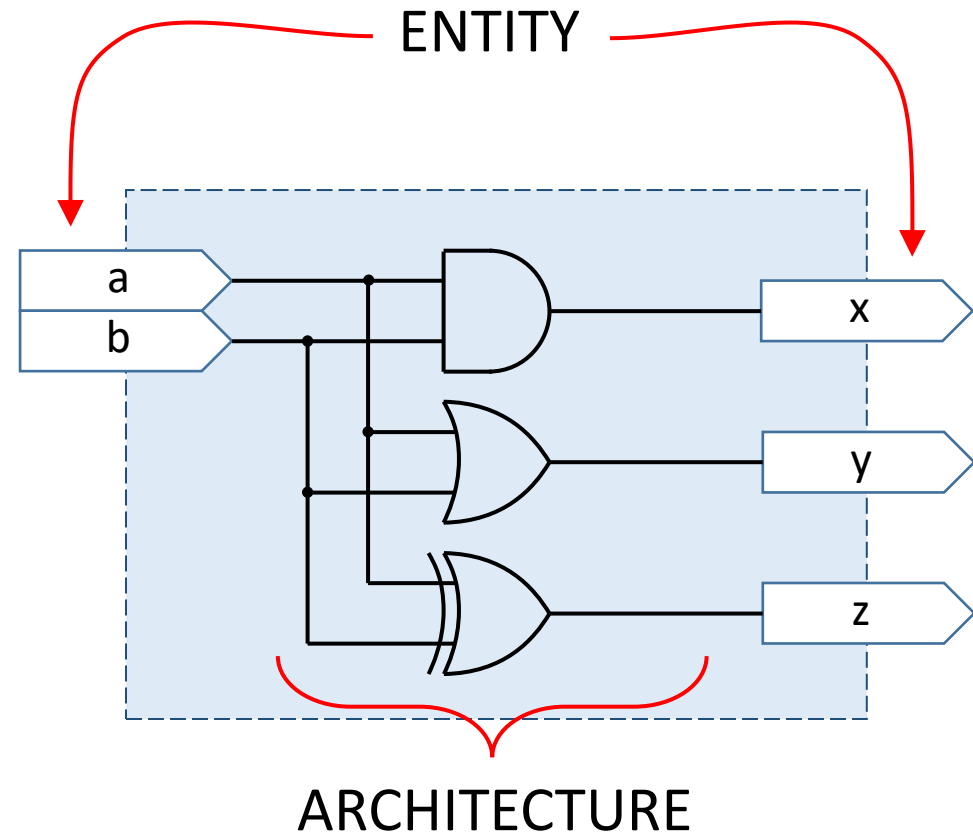
O operador "<=" é
usado para
atribuição de valor

VHDL – Código completo

Código completo:

```
ENTITY Cap3ex1 IS
  PORT (
    a      : IN BIT;
    b      : IN BIT;
    x,y,z : OUT BIT
  );
END Cap3ex1;

-----
ARCHITECTURE arch OF Cap3ex1 IS
BEGIN
  x <= a AND b;
  y <= a OR b;
  z <= a XOR b;
END arch;
```



VHDL – Código completo

Código completo:

```
ENTITY Cap3ex1 IS
    PORT (
        a      : IN BIT;
        b      : IN BIT;
        x,y,z : OUT BIT
    );
END Cap3ex1;

-----

ARCHITECTURE arch OF Cap3ex1 IS
BEGIN
    x <= a AND b;
    y <= a OR b;
    z <= a XOR b;
END arch;
```

Observação quanto ao uso de maiúsculas e minúsculas:

A linguagem VHDL **não faz distinção** entre letras maiúsculas e minúsculas.

No entanto, existem alguns guias de estilo que recomendam que os comandos e palavras reservadas sejam colocadas em MAIÚSCULAS, com os nomes dos sinais e variáveis em letras minúsculas.

O código ao lado foi escrito usando essa recomendação.

No entanto, como foi dito, é apenas uma recomendação de estilo, não sendo, de maneira alguma, obrigatória.

O editor de texto do Quartus II muda automaticamente a cor da letra de todas as palavras reservadas para azul.

VHDL Classes

Classe de objetos:

- CONSTANT
- SIGNAL
- VARIABLE
- FILE

VHDL Classes

CONSTANT


- Sintaxe:

CONSTANT constant_name: type := default_value;

SIGNAL

- Sintaxe:

SIGNAL signal_name: type [:= default_value];



Obs.: Não é
o valor
inicial

VHDL Classes

VARIABLE

- Sintaxe:

`VARIABLE variable_name: type [:= default_value];`

FILE

- Não vamos usar

Obs.: CONSTANT e SIGNAL são declarados na área de declaração da ARCHITECTURE (antes de BEGIN) e são válidos dentro de toda a ARCHITECTURE.

VARIABLE é declarada na área de declaração de um PROCESS (antes de BEGIN) e só é válida (visível) dentro do PROCESS.

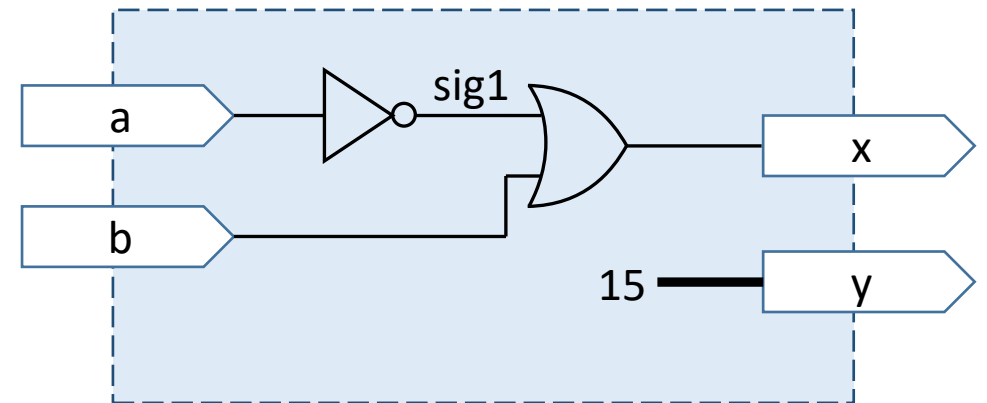
VHDL Classes

- Exemplos:

```
ENTITY Cap3ex2 IS
  PORT (
    a,b: IN BIT;
    x  : OUT BIT;
    y  : OUT INTEGER
  );
END Cap3ex2;

ARCHITECTURE arch OF Cap3ex2 IS
  CONSTANT qz : INTEGER := 15;
  SIGNAL sig1 : BIT;
BEGIN
  sig1 <= NOT a;
  x <= sig1 OR b;
  y <= qz;
END arch;
```

Nesse circuito, **sig1** não é uma entrada ou saída (não foi declarado na ENTITY), mas um nó interno do circuito.



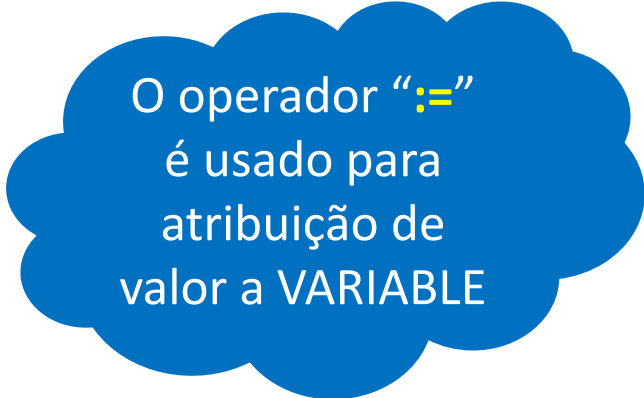
O operador "**<=**" é usado para atribuição de valor a SIGNAL e PORT

VHDL Classes

- Exemplos:

```
ENTITY Cap3ex3 IS
    .....
END Cap3ex3;

ARCHITECTURE arch OF Cap3ex3 IS
BEGIN
    .....
    PROCESS(...)
        VARIABLE var1 : BIT;
    BEGIN
        var1 := a;
        .....
    END PROCESS;
    .....
END arch;
```



O operador “:=”
é usado para
atribuição de
valor a VARIABLE

VHDL

Tipos

Tipos

Caracterizado por um conjunto de valores e um conjunto de operações

Tipos pré-definidos (IEEE 1076):

- BIT / BIT_VECTOR
- BOOLEAN
- INTEGER
- NATURAL
- POSITIVE
- REAL
- TIME
- CHARACTER

VHDL

Tipo BIT

BIT → 2 valores: '0' e '1'

BIT_VECTOR → Conjunto unidimensional de BITS

- Exemplos:

.....

```
ARCHITECTURE arch OF Cap3ex4 IS
  SIGNAL a : BIT;
  SIGNAL b : BIT_VECTOR(0 TO 3);
  SIGNAL c : BIT_VECTOR(3 DOWNT0 0);
  SIGNAL d : BIT_VECTOR(7 DOWNT0 0);
BEGIN
  a <= '1';
  b <= "0011";
```

Aspas simples ''

Aspas duplas ""

O elemento dentro de um VECTOR é identificado por números entre ()

TO : ordem crescente
DOWNT0 : ordem decrescente

Equivale a:

```
b(0) <= '0';
b(1) <= '0';
b(2) <= '1';
b(3) <= '1';
(ordem crescente)
```

Continua

VHDL

Tipo BIT

```
c <= "1010";
```

```
d(6 DOWNTO 2) <= "10110";  
d(3 DOWNTO 0) <= c;
```

```
c <= x"A";  
d <= x"E9";
```

```
c <= d;  
c <= "101";
```

.....

Equivale a:

```
c(3) <= '1';  
c(2) <= '0';  
c(1) <= '1';  
c(0) <= '0';  
(ordem decrescente)
```

Legal, mesmo
comprimento

X indica hexadecimal
Obs.: o número de elementos do
VECTOR tem de ser múltiplo de 4

Illegal, comprimentos
diferentes

VHDL

Tipo BOOLEAN

BOOLEAN → 2 estados (false / true)

- Exemplo:

```
.....  
    SIGNAL eq : BOOLEAN;  
.....  
    eq <= a = b;  
.....
```

eq será **true** se a igual a b. Caso contrário, será **false**.
a e b devem ser do mesmo tipo

VHDL

Tipo INTEGER

INTEGER → Inteiro de 32 bits ($-(2^{31} - 1)$ a $2^{31} - 1$)
(-2.147.483.647 a 2.147.483.647)

- Exemplos:

```
.....  
    SIGNAL g : INTEGER;  
    SIGNAL h : INTEGER RANGE -16 TO 15;
```

```
.....  
    g <= 12345;
```

```
    h <= 11;
```

```
    h <= 21;
```

```
.....
```

Limitação de valores

Não usa aspas

Illegal, fora da faixa

Obs.: A limitação de valores em INTEGER serve apenas para o compilador determinar o número de bits usados para o sinal. Operações aritméticas **não** observam esse limite.

VHDL

Tipos NATURAL e POSITIVE

NATURAL → Sub tipo de INTEGER

Inteiro não negativo (0 a $2^{31} - 1$)

(0 a 2.147.483.647)

POSITIVE → Sub tipo de INTEGER

Inteiro positivo (1 a $2^{31} - 1$)

(1 a 2.147.483.647)

A mesma observação sobre a limitação de valores feita para INTEGER se aplica também para NATURAL e POSITIVE.

VHDL

Tipos REAL, TIME e CHARACTER

REAL → Número real ($-1,0 \times 10^{38}$ a $1,0 \times 10^{38}$)

Geralmente não sintetizável

TIME → não sintetizável

CHARACTER → não sintetizável

VHDL

Operadores

VHDL prove vários tipos de operadores pré-definidos:

- Operadores de atribuição
- Operadores lógicos
- Operadores aritméticos
- Operadores de comparação
- Operadores de deslocamento
- Operadores de concatenação

Cada tipo de operador é válido para tipos específicos

VHDL

Operadores de atribuição

`<=` atribui valor a **SIGNAL**

`:=` atribui valor a **VARIABLE**

- Exemplos:

```
.....
```

```
.....
```

```
s <= 123;
```

```
.....
```

```
.....
```

```
v := "1010";
```

```
.....
```

```
.....
```

s é SIGNAL

v é VARIABLE

Podemos inferir que **s** é um
INTEGER, pois o valor
atribuído está sem aspas.

Podemos inferir que **v** é um
VECTOR, pois o valor atribuído
está com aspas duplas.

VHDL

Operadores de atribuição

=> Atribui valor a elementos individuais de um **VECTOR** ou com **OTHERS**

- Exemplos:

```
.....  
    SIGNAL e : BIT_VECTOR(15 DOWNT0 0);  
    SIGNAL f : BIT_VECTOR(15 DOWNT0 0);  
.....  
    e <= (0 => '1', 2 => '1', OTHERS => '0');  
  
    f <= (OTHERS => '1');  
.....
```

Separado por vírgula.
A ordem não importa,
exceto **OTHERS** que
deve vir por último

Equivale a:

e <= "0000000000000101";

Equivale a:

f <= "1111111111111111";

VHDL

Operadores lógicos

Tipos: `BIT`, `STD_LOGIC`, `STD_ULOGIC` e seus vetores

`NOT`
`AND`
`OR`
`XOR`
`NAND`
`NOR`
`XNOR`

O operador `NOT` tem precedência sobre os demais.

Esses operadores **não tem precedência** de um sobre o outro. Assim, é obrigatório o uso de parênteses sempre que for usado mais de um tipo de operador

VHDL

Operadores lógicos

- Exemplos:

```
.....  
    SIGNAL a, b, c, w, x, y : BIT;  
    SIGNAL d, e, z : BIT_VECTOR(7 DOWNT0 0);  
.....
```

```
.....
```

```
w <= NOT a OR b;
```

```
.....
```

```
x <= A OR (B AND C);
```

```
.....
```

```
y <= (A OR B) AND C;
```

```
.....
```

```
z <= d XOR e;
```

```
.....
```

$$w = \bar{a} + b$$

$$x = a + bc$$

$$y = (a + b)c$$

$$z(7) = d(7) \oplus e(7)$$

$$z(6) = d(6) \oplus e(6)$$

....

$$z(0) = d(0) \oplus e(0)$$

As operações lógicas sobre vetores são feitas bit a bit. Todos os operandos devem ter o mesmo comprimento

VHDL

Operadores aritméticos

Tipos: **INTEGER** (e seus subtipos), **REAL** (geralmente não sintetizável), **SIGNED** ou **UNSIGNED** (pacote `std_logic_arith` ou `numeric_std`) ou **STD_LOGIC_VECTOR** (pacote `std_logic_signed` ou `std_logic_unsigned`)

+	soma
-	subtração
*	multiplicação
/	divisão (apenas 2^N = deslocamento)
**	exponenciação
MOD	módulo
REM	resto
ABS	valor absoluto

Os tipos **SIGNED**, **UNSIGNED** e **STD_LOGIC_VECTOR** serão estudados futuramente

x **MOD** y é igual ao resto de x/y com o sinal de y

x **REM** y é igual ao resto de x/y com o sinal de x

VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a MOD b;
```

$16/5 = 3,2$
Arredondando para baixo: 3
 $3 * 5 = 15$
 $16 - 15 = 1$

x MOD y é igual ao resto de x/y com o sinal de y (divisor)

O operador MOD dá o resto da divisão que arredonda para baixo (*floor*). Portanto:

Se a = 16 e b = 5, então z <= 1

Se a = -16 e b = 5, então z <= -4

Se a = 16 e b = -5, então z <= -4

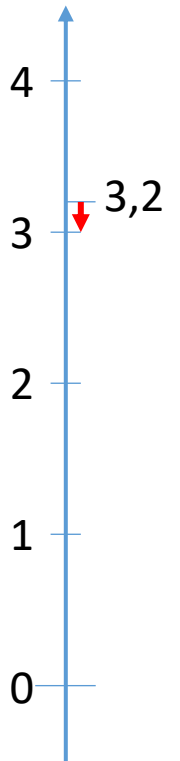
Se a = -16 e b = -5, então z <= -1

$$16 = 5 * 3 + 1$$

$$-16 = 5 * (-4) + 4$$

$$16 = (-5) * (-4) - 4$$

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a MOD b;
```

$-16/5 = -3,2$
Arredondando para baixo: -4
 $-4 * 5 = -20$
 $-16 - (-20) = 4$

x MOD y é igual ao resto de x/y com o sinal de y (divisor)

O operador MOD dá o resto da divisão que arredonda para baixo (*floor*). Portanto:

Se a = 16 e b = 5, então z <= 1

Se a = -16 e b = 5, então z <= 4

Se a = 16 e b = -5, então z <= -4

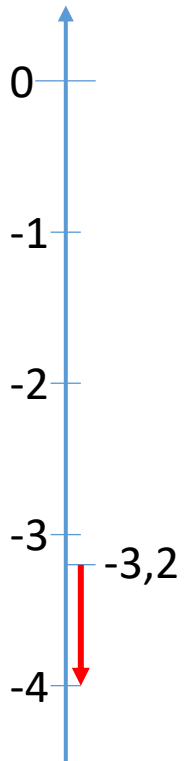
Se a = -16 e b = -5, então z <= -1

$$16 = 5 * 3 + 1$$

$$-16 = 5 * (-4) + 4$$

$$16 = (-5) * (-4) - 4$$

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a MOD b;
```

$16 / -5 = -3,2$
Arredondando para baixo: -4
 $-4 * (-5) = 20$
 $16 - 20 = -4$

x MOD y é igual ao resto de x/y com o sinal de y (divisor)

O operador MOD dá o resto da divisão que arredonda para baixo (*floor*). Portanto:

Se a = 16 e b = 5, então z <= 1

Se a = -16 e b = 5, então z <= -4

Se a = 16 e b = -5, então z <= -4

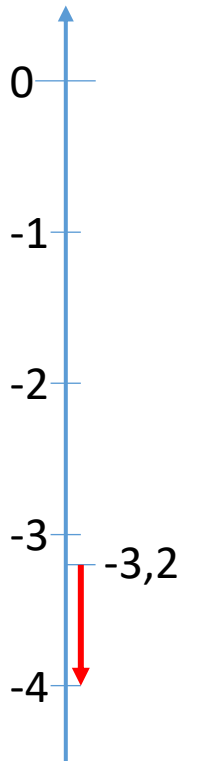
Se a = -16 e b = -5, então z <= -1

$$16 = 5 * 3 + 1$$

$$-16 = 5 * (-4) + 4$$

$$16 = (-5) * (-4) - 4$$

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a MOD b;
```

$-16/(-5) = 3,2$
Arredondando para baixo: 3
 $3 * (-5) = -15$
 $-16 - (-15) = -1$

x MOD y é igual ao resto de x/y com o sinal de y (divisor)

O operador MOD dá o resto da divisão que arredonda para baixo (*floor*). Portanto:

Se a = 16 e b = 5, então z <= 1

$$16 = 5 * 3 + 1$$

Se a = -16 e b = 5, então z <= -4

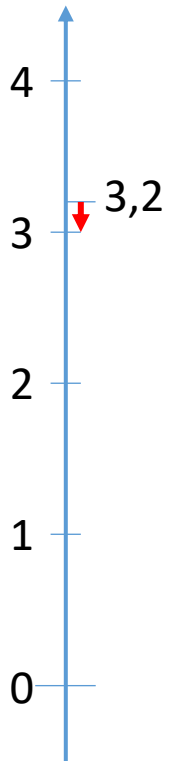
$$-16 = 5 * (-4) + 4$$

Se a = 16 e b = -5, então z <= -4

$$16 = (-5) * (-4) - 4$$

Se a = -16 e b = -5, então z <= -1

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a REM b;
```

$$16/5 = 3,2$$

Desprezando a parte fracionária: 3

$$3 * 5 = 15$$

$$16 - 15 = 1$$

x REM y é igual ao resto de x/y com o sinal de x (dividendo)

O operador REM dá o resto da divisão que arredonda em direção ao zero (desprezando a parte fracionária). Portanto:

Se a = 16 e b = 5, então z <= 1

Se a = -16 e b = 5, então z <= -1

Se a = 16 e b = -5, então z <= 1

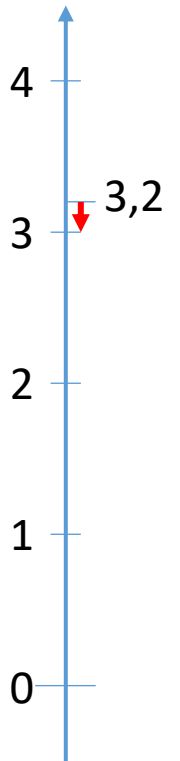
Se a = -16 e b = -5, então z <= -1

$$16 = 5 * 3 + 1$$

$$-16 = 5 * (-3) + 1$$

$$16 = (-5) * (-3) + 1$$

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a REM b;
```

$$-16/5 = -3,2$$

Desprezando a parte fracionária: -3

$$-3 * 5 = -15$$

$$-16 - (-15) = -1$$

x REM y é igual ao resto de x/y com o sinal de x (dividendo)

O operador REM dá o resto da divisão que arredonda em direção ao zero (desprezando a parte fracionária). Portanto:

Se a = 16 e b = 5, então z <= 1

$$16 = 5 * 3 + 1$$

Se a = -16 e b = 5, então z <= -1

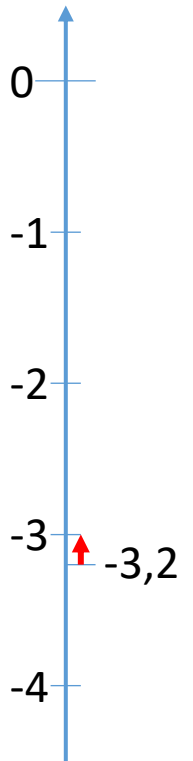
$$-16 = 5 * (-3) + 1$$

Se a = 16 e b = -5, então z <= 1

$$16 = (-5) * (-3) + 1$$

Se a = -16 e b = -5, então z <= -1

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a REM b;
```

$16/(-5) = -3,2$
Desprezando a parte fracionária: -3
 $-3 * (-5) = 15$
 $16 - 15 = 1$

x REM y é igual ao resto de x/y com o sinal de x (dividendo)

O operador REM dá o resto da divisão que arredonda em direção ao zero (desprezando a parte fracionária). Portanto:

Se a = 16 e b = 5, então z <= 1

$$16 = 5 * 3 + 1$$

Se a = -16 e b = 5, então z <= -1

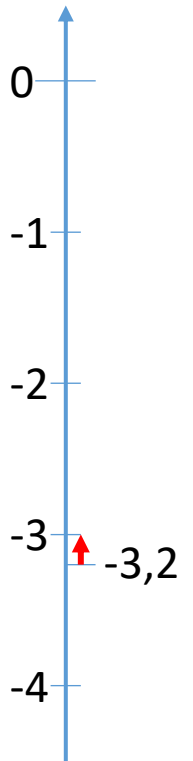
$$-16 = 5 * (-3) + 1$$

Se a = 16 e b = -5, então z <= 1

$$16 = (-5) * (-3) + 1$$

Se a = -16 e b = -5, então z <= -1

$$-16 = (-5) * 3 - 1$$



VHDL

Operadores aritméticos

- Exemplos:

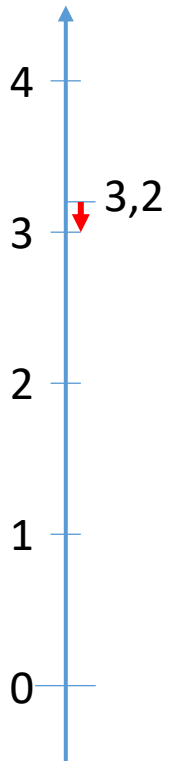
```
.....  
    SIGNAL a, b, z : INTEGER;  
.....  
    z <= a REM b;
```

$-16/(-5) = 3,2$
Desprezando a parte fracionária: 3
 $3 * (-5) = -15$
 $-16 - (-15) = -1$

x REM y é igual ao resto de x/y com o sinal de x (dividendo)

O operador REM dá o resto da divisão que arredonda em direção ao zero (desprezando a parte fracionária). Portanto:

Se a = 16 e b = 5, então z <= 1	$16 = 5 * 3 + 1$
Se a = -16 e b = 5, então z <= -1	$-16 = 5 * (-3) + 1$
Se a = 16 e b = -5, então z <= 1	$16 = (-5) * (-3) + 1$
Se a = -16 e b = -5, então z <= -1	$-16 = (-5) * 3 - 1$

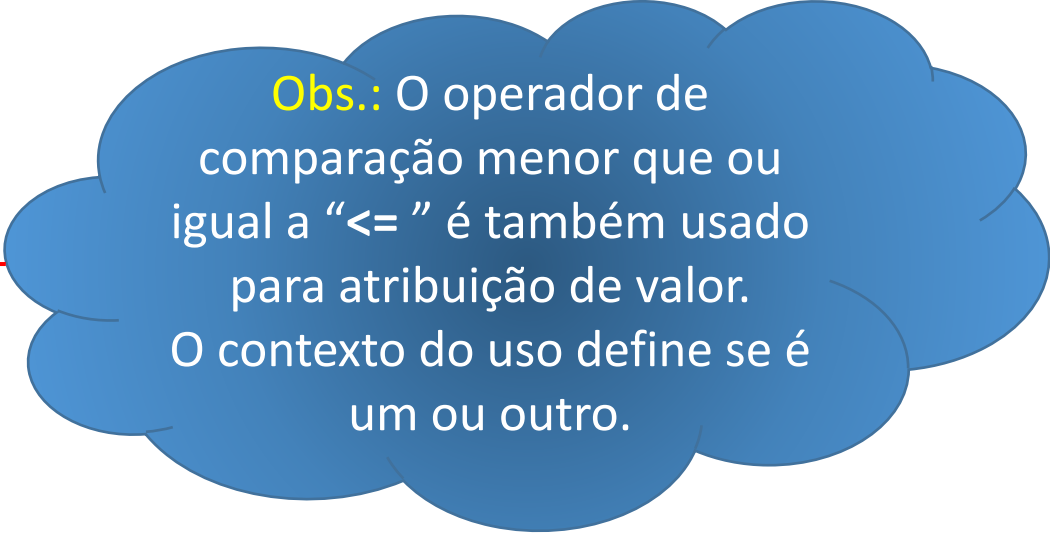


VHDL

Operadores de comparação

Todos os tipos. Mesmos tipos.

=	igual a
/=	não igual a (diferente de)
<	menor que
>	maior que
<=	menor que ou igual a
>=	maior que ou igual a



Obs.: O operador de comparação menor que ou igual a "<=" é também usado para atribuição de valor. O contexto do uso define se é um ou outro.

VHDL

Operadores de deslocamento

Tipos:

- Operando da esquerda: **BIT_VECTOR**
- Operando da direita: **INTEGER**



Determina o número de deslocamentos

SLL	desloca para a esquerda lógico (preenche com '0' a direita)
SRL	desloca para a direita lógico (preenche com '0' a esquerda)
SLA	desloca para a esquerda aritmético (repete bit à direita)
SRA	desloca para a direita aritmético (repete bit à esquerda)
ROL	rotaciona para a esquerda
ROR	rotaciona para a direita

VHDL

Operadores de deslocamento

- Exemplos:

```
.....  
    SIGNAL a, x, y, z : BIT_VECTOR(7 DOWNT0 0);  
.....
```

```
x <= a SLL 3;
```

Se a = "10000101",
então x <= "00101**000**"

```
y <= a SRA 2;
```

Se a = "**1**0001100",
então y <= "**11**100011"

```
z <= a ROL 2;
```

Se a = "**1**0001100",
então z <= "001100**10**"

```
.....
```

VHDL

Operadores de concatenação

Tipos: mesmos tipos permitidos nas operações lógicas, além de **SIGNED** e **UNSIGNED** (pacote `std_logic_arith` ou `numeric_std`)

&

- Exemplos:

```
.....  
    SIGNAL a, b, c : BIT;  
    SIGNAL d : BIT_VECTOR(2 DOWNT0 0);  
    SIGNAL z : BIT_VECTOR(4 DOWNT0 0);  
  
.....  
    d <= a & b & c;  
  
    z <= d & "01";  
  
.....
```

O operador "&" produz um vetor cujo comprimento é a soma do comprimento dos operandos.

Os operandos podem ser vetores ou escalares.

Se a = "0", b = '1' e c = '1',
então d <= "011"

z <= "01101"

VHDL

Operadores de concatenação

Tipos: mesmos tipos permitidos nas operações lógicas, além de **SIGNED** e **UNSIGNED** (pacote `std_logic_arith` ou `numeric_std`)

(, ,)

- Exemplos:

```
.....  
    SIGNAL a, b, c : BIT;  
    SIGNAL d : BIT_VECTOR(2 DOWNT0 0);  
    SIGNAL z : BIT_VECTOR(4 DOWNT0 0);  
.....  
    d <= (a, '0', b);  
  
    z <= (a, b, c, "01");  
.....
```

A construção (, ,) só pode ser usada com escalares .

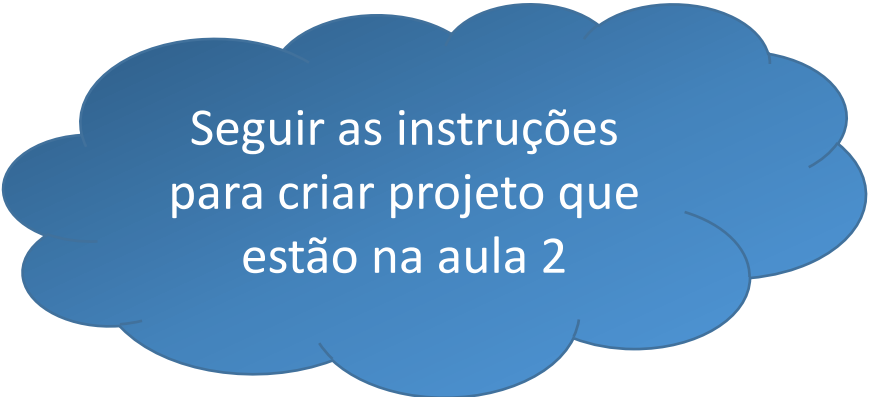
Se a = '0' e b = '1',
então d <= "001"

Illegal

Exercício - criação do projeto

Vamos fazer agora um exercício utilizando, para o *design entry*, a linguagem de descrição de *hardware* VHDL. Neste exemplo usaremos apenas operadores lógicos.

- Inicialmente, é necessário criar, em seu computador, uma pasta para o projeto. Vamos usar o nome “Cap3ex1” para a pasta desse projeto.
- Em seguida, deve ser criado o projeto (*project*), que também deve chamar “Cap3ex1”;
- No menu “File”, escolher a opção “New”;
- na janela aberta, selecionar a opção “VHDL File”;
- salvar esse arquivo usando o nome “Cap3ex1” (a extensão “.vhd” será adicionada automaticamente)



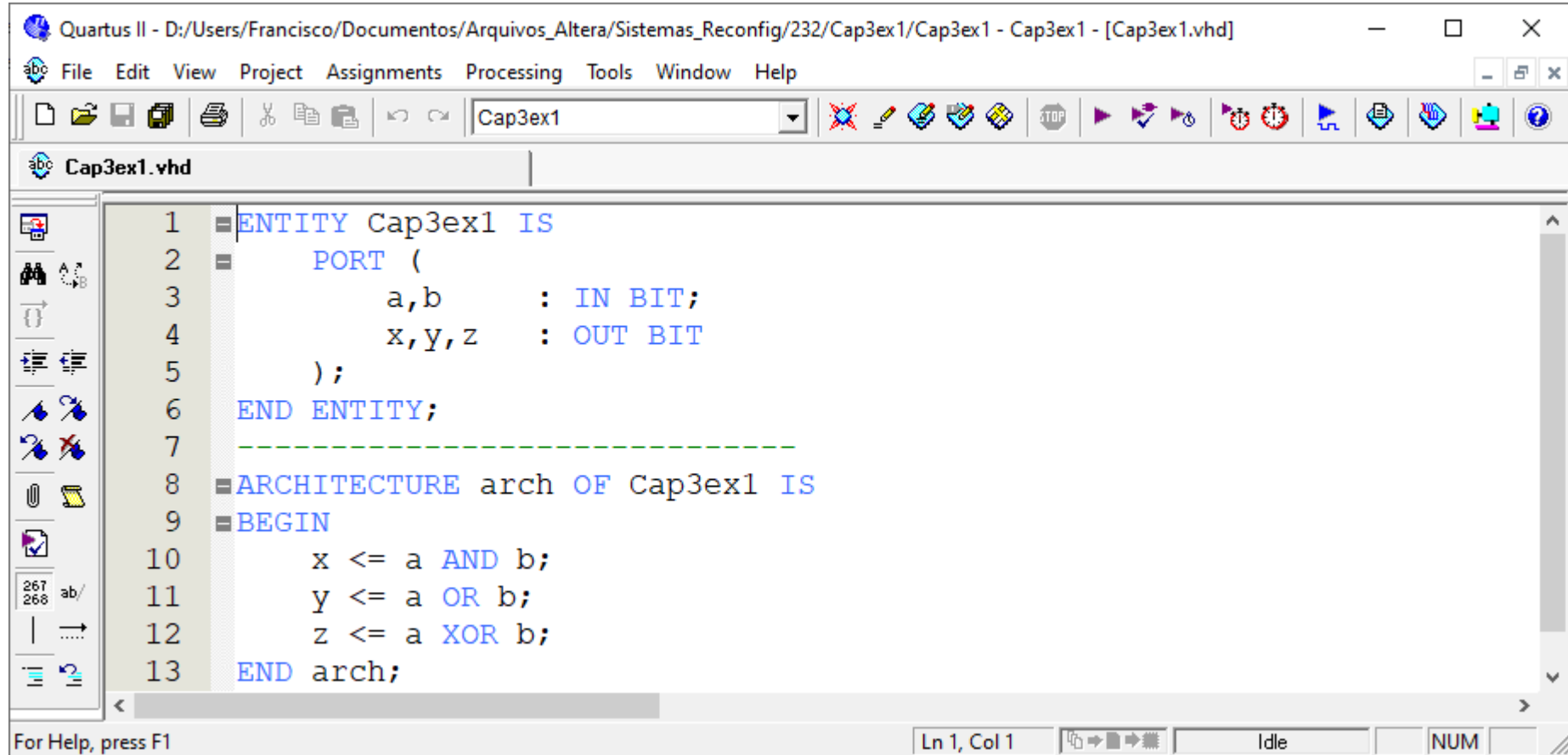
Seguir as instruções
para criar projeto que
estão na aula 2

Exercício - edição

- Digitar o código do exemplo e salvar o arquivo;
- realizar o processo de **análise e síntese**, para verificação. Não devem ocorrer erros ou *warnings*

Obs.:

- Para fazer a análise e síntese, seguir as instruções que estão na aula 2.



The screenshot shows the Quartus II IDE with the file 'Cap3ex1.vhd' open. The code is as follows:

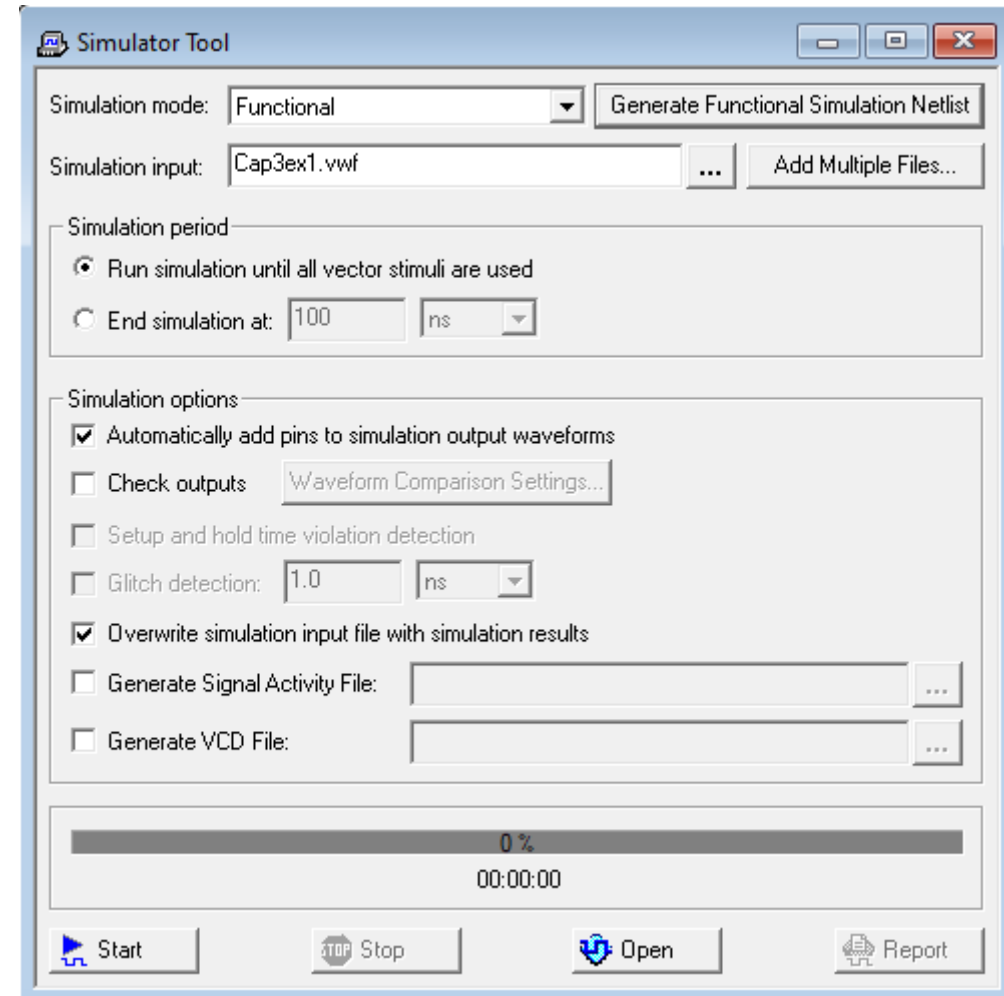
```
1 ENTITY Cap3ex1 IS
2     PORT (
3         a,b      : IN BIT;
4         x,y,z     : OUT BIT
5     );
6 END ENTITY;
7 -----
8 ARCHITECTURE arch OF Cap3ex1 IS
9 BEGIN
10     x <= a AND b;
11     y <= a OR b;
12     z <= a XOR b;
13 END arch;
```

The status bar at the bottom indicates 'For Help, press F1', 'Ln 1, Col 1', 'Idle', and 'NUM'.

Exercício - simulação funcional

- Para verificar se está tudo certo com o projeto, realizar a simulação funcional, seguindo os mesmos passos da aula 2.

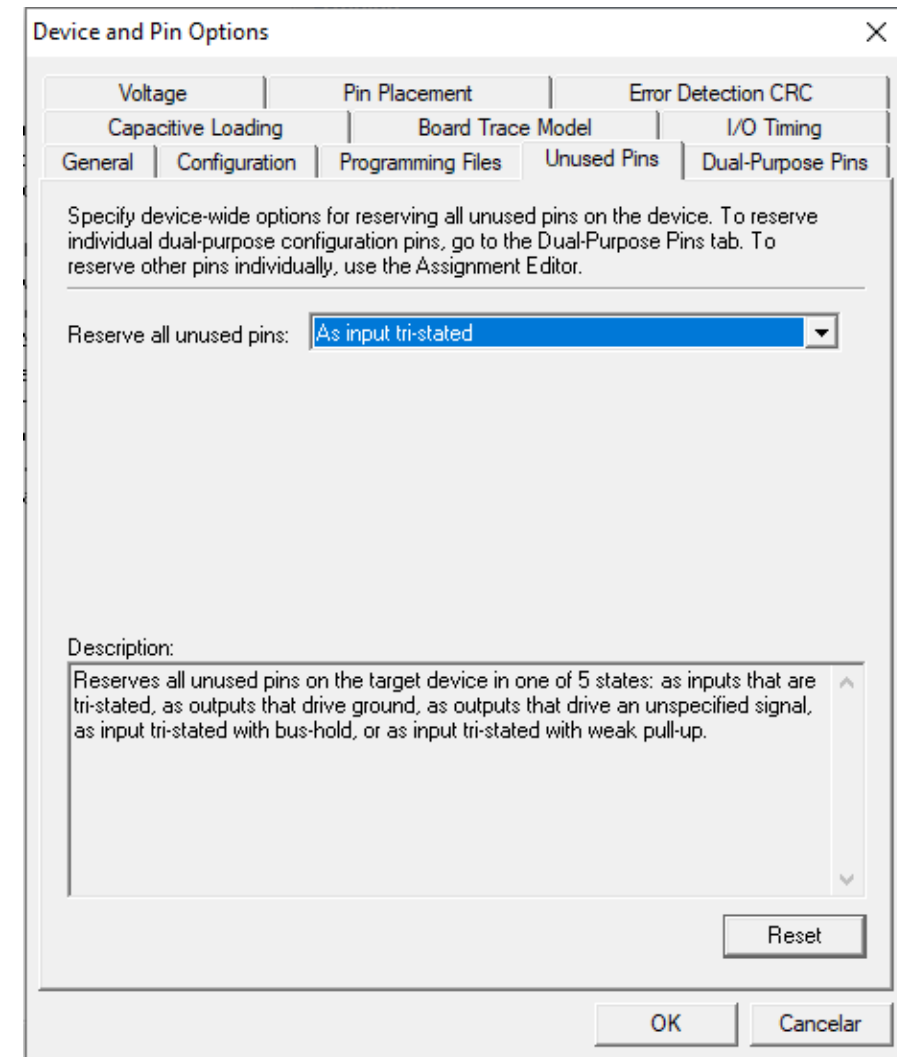
Os resultados obtidos correspondem aos resultados esperados?



Exercício - teste no módulo

- Antes de compilar o projeto para testá-lo no módulo DE2, configurar a função dos pinos não usados (que sempre deverão ser configurados para a opção “**As input tri-stated**”) e a pinagem das entradas e saídas do projeto, da mesma maneira que foi feito na aula 2.
- compilar o projeto, e verificar se ocorreram erros e/ou *warnings*. São esperadas duas mensagens de *warnings*, assim como aconteceu na aula 2.
- programar o dispositivo FPGA no módulo DE2, usando a interface USB e fazer o teste do projeto.

Os resultados obtidos correspondem aos resultados esperados?





Fim

Até a próxima aula!