Sistemas Digitais I
LESI :: 2º ano

VHDL

António Joaquim Esteves
João Miguel Fernandes

www.di.uminho.pt/~aje

Bibliografia: secção 4.7, DDPP, Wakerly

DEP. DE INFORMÁTICA
ESCOLA DE ENGENHARIA
UNIVERSIDADE DO MINHO

4. VHDL

- Sumário -

- Fluxo de projecto
- *Entidades* e arquitecturas
- Tipos
- Funções e procedimentos
- Bibliotecas e *packages*
- Projecto estrutural
- Projecto fluxo de dados (*dataflow*)
- Projecto comportamental (ou funcional)
- Dimensão temporal
- Simulação
- Síntese

4. VHDL

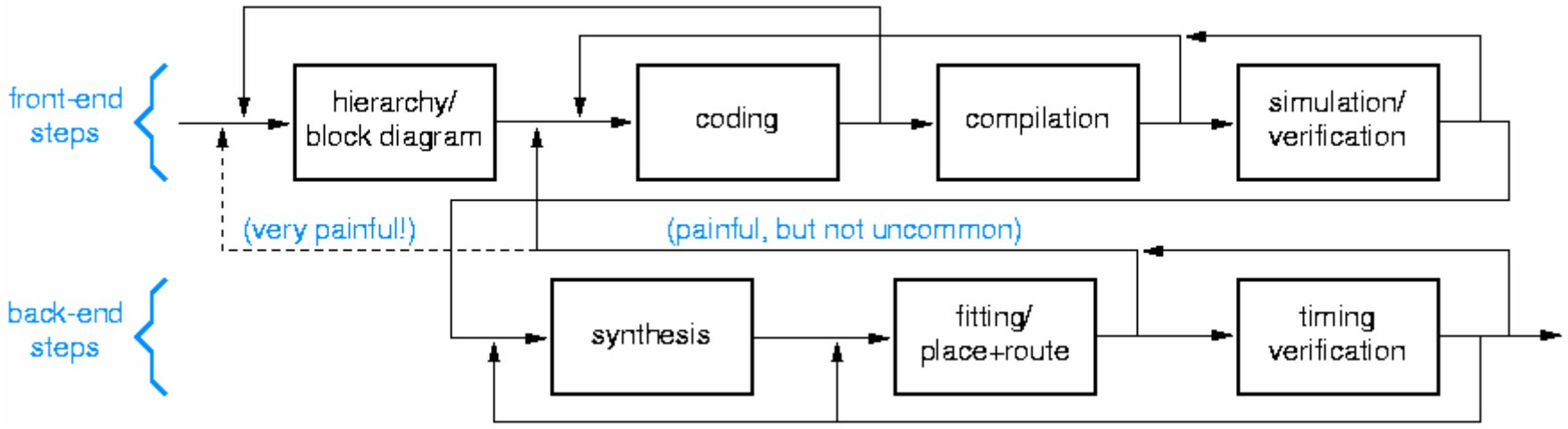
- Introdução -

- O VHDL foi desenvolvido na década de 80 pelo DoD e pelo IEEE.
- VHDL é um acrónimo de VHSIC Hardware Description Language;
  VHSIC é um acrónimo de Very High Speed Integrated Circuit.
- O VHDL possui as seguintes características:
  - Os projectos devem ser decompostos de forma **hierarquica**.
  - Cada elemento dum projecto possui uma **interface** e uma especificação do seu **comportamento**.
  - A especificação dum comportamento pode usar **um algoritmo ou uma estrutura** para definir o modo de operação do elemento.
  - Pode modelar **concorrência**, **temporização** e o conceito de **relógio** *(clocking)*.
  - Permite **simular** a operação lógica e o comportamento temporal dum projecto.

4. VHDL

- Fluxo de projecto -

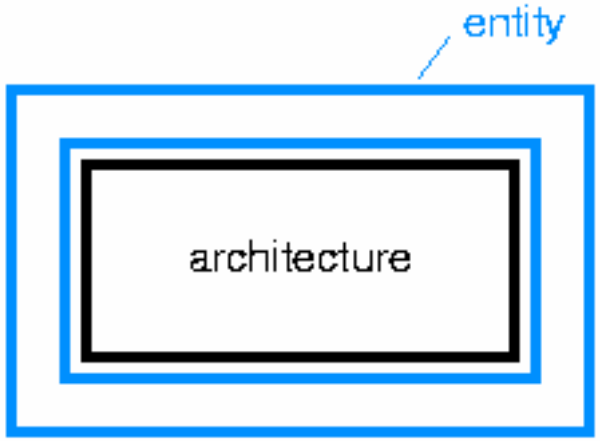
- O VHDL começou por ser uma linguagem de **documentação** e **modelação**, que permitia especificar e simular o comportamento dos projectos.
- Actualmente existem **ferramentas de síntese** comerciais que geram a estrutura dos circuitos lógicos directamente a partir de especificações em VHDL.



4. VHDL

- Entidades e Arquitecturas (1) -

- O VHDL foi desenvolvido tendo em consideração muitos dos princípios da programação estruturada.
- Muitas ideias do VHDL foram importadas do Pascal e do Ada.
- Uma das ideias chave do VHDL é a utilização de interfaces que definem a fronteira dos módulos (de *hardware),* ao mesmo tempo que escondem os pormenores relativos ao seu interior.
- Em VHDL, a **entidade** (*entity*) é uma declaração das entradas e saídas dum módulo.
- Em VHDL, a **arquitectura** (*architecture*) é uma descrição detalhada da estrutura ou comportamento interno dum módulo.

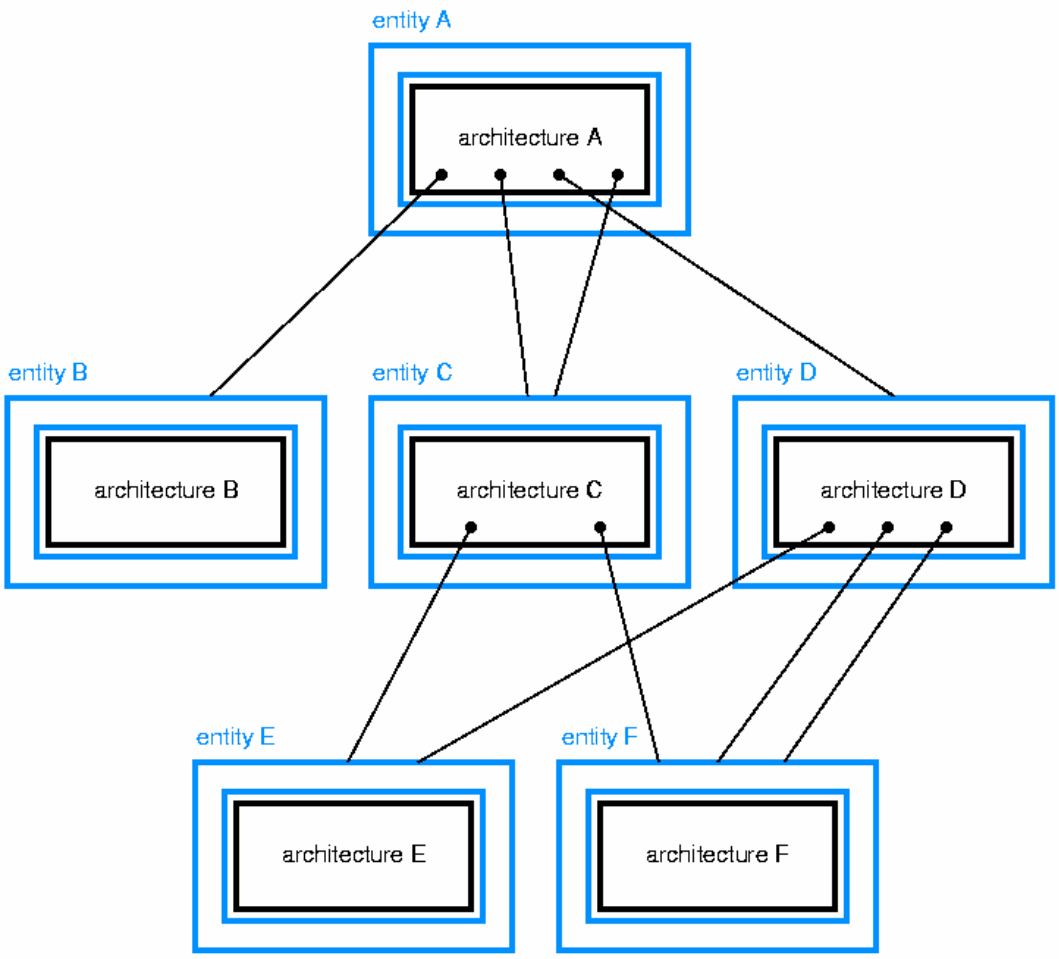


4. VHDL

- Entidades e Arquitecturas (2) -

- Um arquitectura pode usar outras entidades.
- Uma arquitectura de nível superior pode usar uma entidade de nível inferior várias vezes.
- Várias arquitecturas de nível superior podem usar a mesma entidade de nível inferior.
- Estas facilidades são o suporte básico para projectar sistemas de forma hierárquica.

→ Configurações - definem qual a arquitectura a usar em cada ocorrência duma entidade.



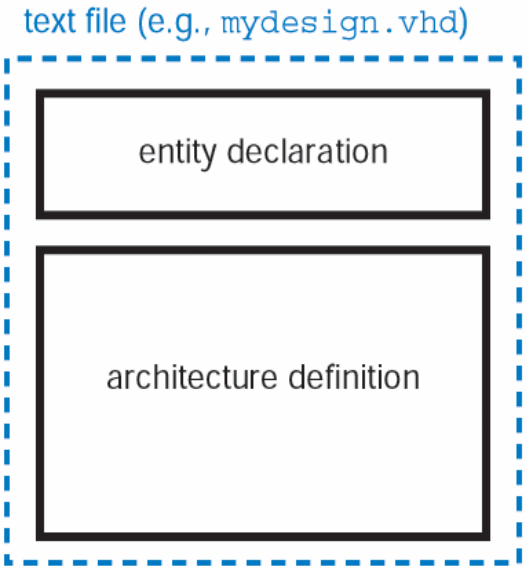
4. VHDL

- Entidades e Arquitecturas (3) -

- Num ficheiro texto, contendo código VHDL, a declaração da entidade e a definição da arquitectura estão separadas.

```
entity Inhibit is
  port (X,Y: in BIT;
        Z:   out BIT);
end Inhibit;

architecture Inhibit_arch of Inhibit is
begin
  Z <= '1' when X='1' and Y='0' else '0';
end Inhibit_arch;
```



- A linguagem não distingue maiúsculas de minúsculas.
- Os comentários começam com 2 hifens "--" e terminam no fim da linha.
- O VHDL possui palavras (chave) reservadas: `port`, `is`, `in`, `out`, `begin`, `end`, `entity`, `architecture`, `if`, `case`, ...

4. VHDL

- Entidades e Arquitecturas (4) -

- Sintaxe da declaração duma entidade:

```
entity entity-name is
  port(signal-names  : mode signal-type;
       signal-names  : mode signal-type;
         ...
       signal-names  : mode signal-type);
end entity-name;
```

- mode indica qual a direcção dum porto da interface:
  - `in`: entrada da entidade
  - `out`: saída da entidade
  - `buffer`: saída da entidade (o seu valor pode ser lido dentro da arquitectura)
  - `inout`: entrada e saída da entidade.
- Um porto é uma entrada, saída ou entrada/saída dum módulo.
- signal-type é um tipo de sinal pré-definido ou um tipo definido pelo utilizador.

4. VHDL

- Entidades e Arquitecturas (5) -

- **Sinal** é o objecto primário utilizado para descrever sistemas, equivale a um "fio" físico e tem um historial de valores passados associado.
- Os sinais funcionam como canais de comunicação entre instruções concorrentes.
- **Sintaxe** da declaração dum **sinal**:

  signal *nome_sinal* : *tipo_sinal* := *valor_inicial_opcional* ;
- Os sinais podem ser declarados explicitamente em:
  - declaração dum *package*
  - arquitectura
  - bloco
  - sub-programa
- A declaração dum porto duma entidade é uma declaração implicita dum sinal.
- **Variável** é similar a um sinal, mas sem o equivalente físico e sem o historial de valores.

4. VHDL

- Entidades e Arquitecturas (6) -

- Sintaxe da definição duma arquitectura:

```
architecture architecture-name of entity-name is
   type declarations
   signal declarations
   constant declarations
   function definitions
   procedure definitions
   component declarations
begin
   concurrent-statement
    ...
   concurrent-statement
end architecture-name;
```

- As declarações (tipos, sinais,…,componentes) podem surgir por qualquer ordem.
- Em "*signal declarations*" definem-se os sinais internos à arquitectura.

4. VHDL

- Tipos (1) -

- Qualquer sinal, variável e constante tem um tipo associado.
- O **tipo** especifica o conjunto de valores permitidos a um objecto e os operadores que podem ser aplicados a esse objecto $\Rightarrow$ tipo de dados abstracto, um conceito similar ao de classe em OO.
- VHDL é uma linguagem fortemente "tipada" mas que possui apenas os seguintes tipos pré-definidos:

`bit`	`character`	`severity_level`
`bit_vector`	`integer`	`string`
`boolean`	`real`	`time`

- **`integer`** inclui os inteiros no intervalo -2.147.483.647 a +2.147.483.647
- **`boolean`** possui 2 valores: *true* e *false*
- **`character`** inclui os caracteres do conjunto ISO 8-bit.

4. VHDL

- Tipos (2) -

- Operadores pré-definidos para os tipos **`integer`** e **`boolean`**.

integer Operators		*boolean* Operators	
+	addition	and	AND
–	subtraction	or	OR
*	multiplication	nand	NAND
/	division	nor	NOR
mod	modulo division	xor	Exclusive OR
rem	modulo remainder	xnor	Exclusive NOR
abs	absolute value	not	complementation
**	exponentiation		

A rem B → módulo = A – int(A/B)*B e sinal = sinal de A

A mod B → módulo = A – int(A/B)*B [quando A e B são do mesmo sinal]

módulo = A + ceil(|A/B|)*B [quando A e B são de sinais diferentes]

sinal = sinal de B

4. VHDL

- Tipos (3) -

- Os tipos definidos pelo utilizador são comuns em VHDL.
- Um tipo enumerado é definido através duma lista de valores permitidos.

```
type type-name is (value-list);

subtype subtype-name is type-name start to end;
subtype subtype-name is type-name start downto end;

constant constant-name : type-name := value;
```

- Tipo STD_LOGIC com 9 valores →

```
type STD_ULOGIC is (
   'U',   -- Uninitialized
   'X',   -- Forcing  Unknown
   '0',   -- Forcing  0
   '1',   -- Forcing  1
   'Z',   -- High Impedance
   'W',   -- Weak     Unknown
   'L',   -- Weak     0
   'H',   -- Weak     1
   '-');  -- Don't care
subtype STD_LOGIC is resolved STD_ULOGIC;
```

- **Outros exemplos:**
  - `type traffic_light_state is (reset, stop, start, go);`
  - `subtype int32 is integer range 31 downto 0;`
  - `constant BUS_SIZE: integer := 32;`

4. VHDL

- *Tipos (4)* -

- O utilizador também pode definir **arrays** usando as palavras chave **to**, **downto** e **range** para definir a dimensão. Um array é um conjunto ordenado de elementos do mesmo tipo.

```
type type-name is array(start to end) of element-type;

type type-name is array(start downto end) of element-type;

type type-name is array(range-type) of element-type;

type type-name is array(range-type range start to end) of element-type;

type type-name is array(range-type range start downto end) of element-type;
```

```
type monthly_count is array (1 to 12) of integer;
type byte is array (7 downto 0) of STD_LOGIC;

constant WORD_LEN: integer := 32;
type word is array (WORD_LEN-1 downto 0) of STD_LOGIC;

constant NUM_REGS: integer := 8;
type reg_file is array (1 to NUM_REGS) of word;

type statecount is array (traffic_light_state) of integer;
```

4. VHDL

- Tipos (5) -

- O conteúdo dos elementos dum *array* pode ser especificado por posição, colocando a lista de valores a atribuir entre parenteses:
  `B := ('1','1','0','1','1','0','0','1'); -- tipo byte`
- O conteúdo do *array* também pode ser especificado usando índices:
  `W := (0=>'0', 3=>'0', 9=>'0',others=>'1'); -- tipo word`
- O conteúdo dum *array* STD_LOGIC pode ser especificado usando strings:
  `B := "11011001";`
  `W := "0110111110111111";`
- Também se pode especificar uma parcela dum *array*:
  `B(2 to 4):="101";  W(9 downto 0):="0101011010";`
- Pode juntar-se *arrays* usando o operador de concatenação (`&`):
  `'0'&'1'&"1Z"` é equivalente a `"011Z"`.
  `B(6 downto 0)& B(7)` equivale a rodar o array B um 1-bit à esquerda.

4. VHDL

- Funções e Procedimentos (1) -

- Uma função aceita um conjunto de argumentos e devolve um resultado.
- Tanto os argumentos como o resultado devem ter um tipo.
- O corpo da função é 1 conjunto de instruções executadas em sequência.
- A sintaxe da definição de função é:

```
function function-name (
      signal-names  : signal-type;
      signal-names  : signal-type;
        ...
      signal-names  : signal-type
)  return return-type is
   type declarations
   constant declarations
   variable declarations
   function definitions
   procedure definitions
begin
   sequential-statement
    ...
   sequential-statement
end function-name;
```

- Exemplo de definição e utilização duma função `ButNot`:

```
architecture Inhibit_archf of Inhibit is

function ButNot (A, B: bit) return bit is
begin
  if B = '0' then return A;
  else return '0';
  end if;
end ButNot;

begin
  Z <= ButNot(X,Y);
end Inhibit_archf;
```

4. VHDL

- Funções e Procedimentos (2) -

- Normalmente é necessário converter um sinal de um tipo para outro.
- Usando o tipo de array

  `type STD_LOGIC_VECTOR is array (natural range <>) of STD_LOGIC;`

  apresentam-se 2 funções de conversão de tipos:
  - std_logic → inteiro
  - inteiro → std_logic

```
function CONV_INTEGER (X: STD_LOGIC_VECTOR) return INTEGER is
  variable RESULT: INTEGER;
begin
  RESULT := 0;
  for i in X'range loop
    RESULT := RESULT * 2;
    case X(i) is
      when '0' | 'L'  => null;
      when '1' | 'H'  => RESULT := RESULT + 1;
      when others     => null;
    end case;
  end loop;
  return RESULT;
end CONV_INTEGER;
```

```
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
    return STD_LOGIC_VECTOR is
  variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
  variable temp: integer;
begin
  temp := ARG;
  for i in 0 to SIZE-1 loop
    if (temp mod 2) = 1 then result(i) := '1';
    else result(i) := '0';
    end if;
    temp := temp / 2;
  end loop;
  return result;
end;
```

4. VHDL

- Funções e Procedimentos (3) -

- Um **procedimento** é idêntico a uma função, mas não devolve um resultado.

- Enquanto a invocação duma função pode ser usada em vez duma expressão, a invocação dum procedimento pode ser usada em vez duma instrução.

- Como os argumentos dum procedimento podem ter uma direcção do tipo **`out`** ou **`inout`**, é possível um procedimento "devolver" resultado(s).

4. VHDL

- Bibliotecas e Packages (1) -

- Uma **biblioteca** é o local onde o compilador de VHDL guarda a informação relativa a um determinado projecto (intermédia, da simulação e da síntese).
- Para qualquer projecto, o compilador cria e utiliza a biblioteca **`work`**.
- Um projecto pode usar múltiplos ficheiros, cada um com unidades (entidades / arquitecturas) diferentes.
- Quando um ficheiro é compilado, os resultados são guardados na biblioteca `work`.
- Nem toda a informação necessária a um projecto deve estar na biblioteca `work`. O projectista pode recorrer a definições ou funções que são comuns a vários projectos (por exemplo, incluídas na **biblioteca** IEEE).
- Um projecto pode especificar que vai usar bibliotecas que contêm definições partilháveis.
  Exemplo: **`library ieee;`**

4. VHDL

- Bibliotecas e Packages (2) -

- A especificação duma biblioteca permite aceder a todas as entidades e arquitecturas guardadas nessa biblioteca , mas não dá acesso aos tipos, sub-tipos, funções, procedimentos,...
- Um *package* é um ficheiro com definições de objectos (sinais, tipos, constantes, funções, procedimentos, componentes) que podem ser utilizados nos projectos.
- A cláusula seguinte permite a um projecto usar todas as definições do package standard 1164 do IEEE:
  use ieee.std_logic_1164.all;

- Sintaxe da definição dum *package*

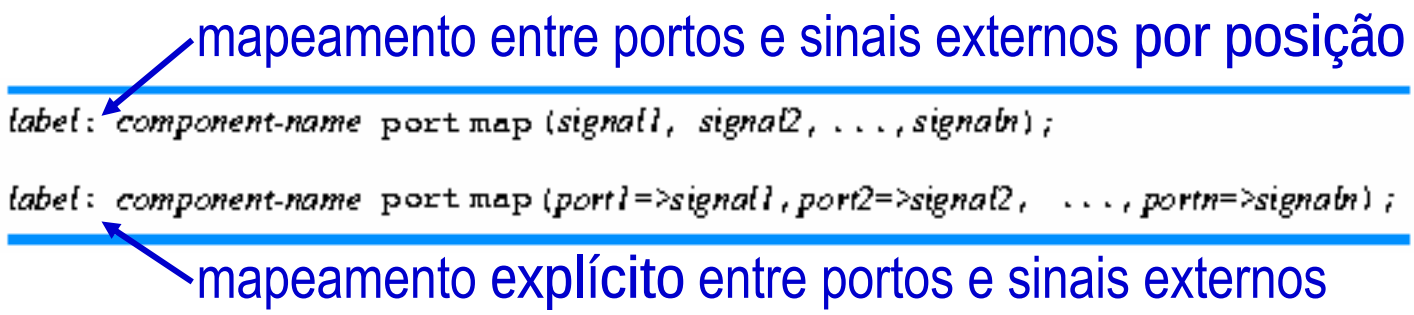
```
package package-name is
   type declarations
   signal declarations
   constant declarations
   component declarations
   function declarations
   procedure declarations
end package-name ;
package body package-name is
   type declarations
   constant declarations
   function definitions
   procedure definitions
end package-name ;
```

4. VHDL

- Projecto estrutural (1) -

- O corpo duma arquitectura é uma série de **instruções concorrentes**.
- Cada instrução concorrente é executada em simultâneo com as outras instruções concorrentes incluídas no mesmo corpo de arquitectura.
- As instruções concorrentes são necessárias para simular o modo paralelo em que os elementos de *hardware* funcionam.
- A instrução concorrente mais elementar é a instanciação dum **componente**.

- Sintaxe da instanciação dum componente



- `component-name` é o nome duma entidade definida anteriormente.
- Por cada instanciação dum componente é criada uma instância da entidade respectiva.

4. VHDL

- Projecto estrutural (2) -

- Antes de instanciar um componente ele tem que ser declarado na parte declarativa da arquitectura usando o construtor component.
- A declaração dum componente é essencialmente o mesmo que a declaração da interfce da entidade correspondente.
- Sintaxe da declaração dum componente ►

```
component component-name
   port (signal-names  : mode signal-type;
         signal-names  : mode signal-type;
           ...
         signal-names  : mode signal-type);
end component;
```

- Os componentes usados numa arquitectura podem ter sido definidos anteriormente no projecto em causa, ou podem estar definidos numa biblioteca.

4. VHDL

- Projecto estrutural (3) -

- Exemplo: descrição estrutural do detector de números primos

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prime is
    port ( N: in STD_LOGIC_VECTOR (3 downto 0); F: out STD_LOGIC );
end prime;
architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L: STD_LOGIC;
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port(I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port(I0,I1,I2,I3: in STD_LOGIC; O:out STD_LOGIC);end component;
begin
  U1: INV port map (N(3), N3_L);
  U2: INV port map (N(2), N2_L);
  U3: INV port map (N(1), N1_L);
  U4: AND2 port map (N3_L, N(0), N3L_N0);
  U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
  U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
  U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
  U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prime1_arch;
```

sinais internos

declaração de componentes

instanciação de componentes

4. VHDL

- Projecto estrutural (4) -

- Uma arquitectura que utiliza componentes é uma **descrição estrutural**, uma vez que descreve a estrutura de interligação entre os sinais e as entidades que concretizam essa entidade.
- A instrução generate permite criar estruturas repetitivas.
- Sintaxe do ciclo **`for…generate`**

variável implicitamente declarada

```
label: for identifier in range generate
        concurrent-statement
       end generate;
```

- Inversor de 8 bits descrito com um ciclo `for … generate`

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inv8 is
    port ( X: in STD_LOGIC_VECTOR (1 to 8);
           Y: out STD_LOGIC_VECTOR (1 to 8) );
end inv8;
architecture inv8_arch of inv8 is
component INV port(I: in STD_LOGIC; O:out STD_LOGIC); end component;
begin
  g1: for b in 1 to 8 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end inv8_arch;
```

4. VHDL

- Projecto estrutural (5) -

- Pode definir-se **constantes genéricas** na declaração duma entidade.
- Sintaxe da declaração duma entidade usando constantes genéricas ▶
- As constantes genéricas permitem definir uma entidade parametrizada

```
entity entity-name is
  generic (constant-names : constant-type;
           constant-names : constant-type;
             ...
           constant-names : constant-type);
  port (signal-names  : mode signal-type;
        signal-names  : mode signal-type;
          ...
        signal-names  : mode signal-type);
end entity-name;
```

- Cada constante pode ser usada dentro da arquitectura em que é declarada e a atribuição dum valor a essa constante ocorre apenas quando a entidade for instanciada noutra arquitectura.
- Ao instanciar um componente, para atribuir valores às constantes genéricas utiliza-se uma cláusula **`generic map`**.

4. VHDL

- Projecto estrutural (6) -

- Circuito inversor dos bits dum barramento de largura parametrizável (WID) e descrito com um ciclo `for ... generate`

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

entity businv is
    generic (WID  : positive);
    port ( X: in STD_LOGIC_VECTOR (WID -1 downto 0);
           Y: out STD_LOGIC_VECTOR (WID -1 downto 0) );
end businv;

architecture businv_arch of businv is
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in WID-1 downto 0 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end businv_arch;
```

4. VHDL

- Projecto estrutural (7) -

- Entidade que instancia o inversor de barramento com 3 valores de WID: 8,16 e 32

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

entity businv_example is
    port ( IN8: in STD_LOGIC_VECTOR (7 downto 0);
           OUT8: out STD_LOGIC_VECTOR (7 downto 0);
           IN16: in STD_LOGIC_VECTOR (15 downto 0);
           OUT16: out STD_LOGIC_VECTOR (15 downto 0);
           IN32: in STD_LOGIC_VECTOR (31 downto 0);
           OUT32: out STD_LOGIC_VECTOR (31 downto 0) );
end businv_example;

architecture businv_ex_arch of businv_example is
component businv
    generic (WID : positive);
    port ( X: in STD_LOGIC_VECTOR (WID -1 downto 0);
           Y: out STD_LOGIC_VECTOR (WID -1 downto 0) );
end component;
begin
U1: businv generic map (WID =>8) port map (IN8, OUT8);
U2: businv generic map (WID =>16) port map (IN16, OUT16);
U3: businv generic map (WID =>32) port map (IN32, OUT32);
end businv_ex_arch;
```

4. VHDL

- Projecto fluxo de dados (1) -

- Se uma arquitectura descreve o fluxo de dados e as operações que são aplicadas a esses dados, utilizando apenas atribuições concorrentes, segue o estilo de descrição designado por **fluxo de dados**.
- Dois tipos de instrução concorrente que se usam numa descrição fluxo de dados são:
  - atribuição concorrente dum valor a um sinal – a largura e o tipo de ambos os lados da instrução têm que ser compatíveis;
  - atribuição concorrente e condicional dum valor a um sinal.

- Sintaxe dos 2 tipos de atribuição concorrente ▶

```
signal-name  <= expression;

signal-name  <= expression when boolean-expression else
                expression when boolean-expression else
                   ...
                expression when boolean-expression else
                expression;
```

4. VHDL

- Projecto fluxo de dados (2) -

- Exemplo: descrição fluxo de dados para o detector de números primos, usando operadores pré-definidos como and, or, not e atribuições simples

```
architecture prime2_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0      <= not N(3)                                  and N(0);
  N3L_N2L_N1  <= not N(3) and not N(2) and     N(1)             ;
  N2L_N1_N0   <=                 not N(2) and     N(1) and N(0);
  N2_N1L_N0   <=                     N(2) and not N(1) and N(0);
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime2_arch;
```

- Exemplo: descrição fluxo de dados usando atribuições condicionais

```
architecture prime3_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0      <= '1' when N(3)='0' and N(0)='1' else '0';
  N3L_N2L_N1  <= '1' when N(3)='0' and N(2)='0' and N(1)='1' else '0';
  N2L_N1_N0   <= '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0';
  N2_N1L_N0   <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime3_arch;
```

para cobrir todas as combinações restantes

4. VHDL

- Projecto fluxo de dados (3) -

- Outro tipo de instrução concorrente é a **atribuição selectiva a um sinal**, idêntica a um construtor CASE.

Sintaxe da atribuição selectiva

```
with expression select
   signal-name <= signal-value when choices,
                  signal-value when choices,
                     . . .
                  signal-value when choices;
```

- Exemplo: descrição fluxo de dados para o detector de números primos usando uma atribuição selectiva ao sinal F

```
architecture prime4_arch of prime is
begin
  with N select
    F <= '1' when "0001",
         '1' when "0010",
         '1' when "0011" | "0101" | "0111",
         '1' when "1011" | "1101",
         '0' when others;
end prime4_arch;
```

escolha simples

lista

```
architecture prime5_arch of prime is
begin
  with CONV_INTEGER(N) select
    F <= '1' when 1|2|3|5|7|11|13,
         '0' when others;
end prime5_arch;
```

- As várias escolhas devem ser mutuamente exclusivas e cobrir todos os casos.

4. VHDL

- Projecto comportamental (1) -

- O principal construtor usado em descrições comportamentais é o processo, o qual consiste numa série de instruções sequenciais que são executadas em paralelo com outras instruções e processos concorrentes.
- Um processo tem um tempo de simulação nulo.
- Um processo em VHDL é assim uma instrução concorrente, com sintaxe:

```
process (signal-name, signal-name, . . . , signal-name)
  type declarations
  variable declarations
  constant declarations
  function definitions
  procedure definitions .
begin
  sequential-statement
  . . .
  sequential-statement
end process;
```

lista de sensibilidade

4. VHDL

- Projecto comportamental (2) -

- Um processo não pode declarar sinais, apenas variáveis, utilizadas para guardar informação relativa ao estado do processo.
- A sintaxe da definição duma variável é:
  `variable` ***`nome_variavel`*** **`:`** ***`tipo_variavel`*****`;`**
- Um processo em VHDL está num de 2 estados: em execução ou suspenso.
- A lista de sinais incluída na definição dum processo (**lista de sensibilidade**) determina quando é que ele é executado.
- No início da simulação todos os processos são executados.
- Quando um processo estiver suspenso, ele retoma a execução quando um sinal da lista de sensibilidade muda de valor.
- Se um sinal da lista de sensibilidade mudar de valor durante a execução do processo, este será executado outra vez.

4. VHDL

- Projecto comportamental (3) -

- A execução continua até o processo terminar a execução sem que nenhum destes sinais tenha mudado de valor.
- Na simulação, a execução do corpo dum processo (até ele ser suspenso) decorre num tempo de simulação nulo.
- Após ter retomado a execução, um processo correctamente escrito será suspenso ao fim de algumas execuções.
- Contudo, é possível escrever (incorrectamente) um processo que nunca é suspenso.

  Um exemplo: um processo com uma única instrução "`X <= not X;`" e uma lista de sensibilidade igual a "`(X)`".

  Como X muda em cada execução, o processo executa indefinidamente num tempo de simulação nulo.
- Na prática, os simuladores conseguem detectar estas situações e terminar a simulação.

4. VHDL

- Projecto comportamental (4) -

- A instrução de atribuição sequencial dum valor a um sinal possui a mesma sintaxe que a versão concorrente, mas ocorre no corpo dum processo em vez de numa arquitectura:
  ***nome-sinal* <= expressão;**

- A instrução de atribuição dum valor a uma variável possui a seguinte sintaxe:
  ***nome-variavel* := expressão;**

- Exemplo: descrição comportamental para o detector de números primos, em que se usa um processo com atribuições a variáveis

```
architecture prime6_arch of prime6 is
begin
  process(N)
    variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  begin
    N3L_N0      := not N(3)                                  and N(0);
    N3L_N2L_N1  := not N(3) and not N(2) and     N(1)              ;
    N2L_N1_N0   :=              not N(2) and     N(1) and N(0);
    N2_N1L_N0   :=                  N(2) and not N(1) and N(0);
    F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
  end process;
end prime6_arch;
```

4. VHDL

- Projecto comportamental (5) -

- Além da atribuição, pode usar-se outras instruções sequenciais, descritas por alguns dos construtores mais populares, tais como: **`if`**, **`case`**, **`loop`**, **`for`** e **`while`**.
- O **`case`** é mais legível que um if com múltiplas cláusulas if/elsif e pode ser sintetizado de forma mais adequada

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
else sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
else sequential-statement
end if;
```

```
case expression is
  when choices => sequential-statements
  ...
  when choices => sequential-statements
end case;
```

```
loop
  sequential-statement
  ...
  sequential-statement
end loop;
```

```
for identifier in range loop
  sequential-statement
  ...
  sequential-statement
end loop;
```

variável implicitamente declarada

```
while boolean-expression loop
  sequential-statement
  ...
  sequential-statement
end loop;
```

ciclo infinito

- As várias escolhas dum `case` devem ser mutuamente exclusivas e cobrir todos os casos.

4. VHDL

- Projecto comportamental (6) -

- Exemplos: descrição comportamental para o detector de números primos usando um construtor **`if`** ou **`case`** (necessariamente incluídos num processo)

```
architecture prime7_arch of prime is
begin
  process(N)
    variable NI: INTEGER;
  begin
    NI := CONV_INTEGER(N);
    if NI=1 or NI=2 then P <= '1';
    elsif NI=3 or NI=5 or NI=7 or NI=11 or
          NI=13 then P <= '1';
    else P <= '0';
    end if;
  end process;
end prime7_arch;
```

```
architecture prime8_arch of prime is
begin
  process(N)
  begin
    case CONV_INTEGER(N) is
      when 1 => P <= '1';
      when 2 => P <= '1';
      when 3 | 5 | 7 | 11 | 13 => P <= '1';
      when others => P <= '0';
    end case;
  end process;
end prime8_arch;
```

4. VHDL

- Projecto comportamental (7) -

- Exemplo: descrição comportamental para o detector de números primos usando um ciclo **`for`** (o tamanho da entrada - N - passou para 8 bits)

```
architecture prime9_arch of prime9 is
begin
  process(N)
  variable NI: INTEGER;
  variable prime: boolean;
  begin
    NI := CONV_INTEGER(N);
    prime := true;
    if NI=1 or NI=2 then null; -- boundary cases
    else for i in 2 to NI-1 loop
           if NI mod i = 0 then
             prime := false; exit;
           end if;
         end loop;
    end if;
    if prime then P <= '1'; else P <= '0'; end if;
  end process;
end prime9_arch;
```

4. VHDL

- Dimensão temporal (1) -

- Nenhum dos exemplos anteriores aborda a dimensão temporal associada com o funcionamento dos circuitos: tudo se passa num tempo de simulação nulo.
- O VHDL modela adequadamente os aspectos relacionados com tempo.
- Em VHDL pode especificar-se um atraso usando a palavra chave **`after`** na instrução de atribuição dum valor a um sinal.
- ```
  Z <= '1' after 4ns when X='1' else
       '0' after 3ns;
  ```
- Esta instrução modela uma porta que apresenta um atraso de 4ns quando a saída Z muda de 0→1 e apenas 3ns quando muda de 1→0.
- Deste modo, ao simular o modelo dum circuito obtém-se uma boa aproximação para o seu comportamento temporal.
  ```

4. VHDL

- Dimensão temporal (2) -

- Outra forma de incorporar informação temporal numa descrição VHDL consiste em usar a instrução **`wait`**.
- Esta instrução sequencial pode ser usada para suspender um processo durante um intervalo determinado.
- É frequente usar a instrução `wait` para ajudar a descrever os padrões que se aplicam nas entradas dum circuito que está a ser simulado. Exemplo:

```
entity InhibitTestBench is
end InhibitTestBench;

architecture InhibitTB_arch of InhibitTestBench is
component Inhibit port (X,Y: in BIT; Z: out BIT); end component;
signal XT, YT, ZT: BIT;
begin
  U1: Inhibit port map (XT, YT, ZT);
  process
  begin
    XT <= '0'; YT <= '0';
    wait for 10 ns;
    XT <= '0'; YT <= '1';
    wait for 10 ns;
    XT <= '1'; YT <= '0';
    wait for 10 ns;
    XT <= '1'; YT <= '1';
    wait; -- this suspends the process indefinitely
  end process;
end InhibitTB_arch;
```

- Variantes da instrução wait:
  - wait;
  - wait on lista_sinais;
  - wait until condição;
  - wait for tempo;

4. VHDL

- Simulação (1) -

- A partir do momento em que se dispõe duma descrição VHDL sem erros de sintaxe e de semântica, pode aplicar-se a descrição num simulador para verificar se o circuito descrito funciona como esperado.
- O processo de simulação arranca no instante zero de simulação.
- Neste instante, o simulador inicia todos os sinais com o seu valor por defeito.
- Também se iniciam os sinais e variáveis para os quais o código VHDL declare explicitamente valores iniciais.
- A seguir, o simulador começa a executar “todos” os processos (e instruções concorrentes) da descrição do circuito.
- Para simular a execução de todos os processos, o simulador usa (i) uma lista com eventos calendarizados (com ocorrência temporal escalonada) e (ii) uma matriz com os sinais a que cada processo é sensível.

4. VHDL

- Simulação (2) -

- No instante zero de simulação todos os processos são escalonados para serem executados.
- Selecciona-se um dos processos e executam-se todas as suas instruções sequenciais, incluindo os ciclos.
- Quando a execução deste processo terminar, escolhe-se outro para execução e repete-se este procedimento até que todos os processos tenham sido executados.
- Completa-se assim um ciclo de simulação.
- Durante a execução, um processo pode atribuir novos valores a sinais.
- Os novos valores não são atribuídos imediatamente. Antes são colocados na lista de eventos e a sua efectivação é escalonada para um tempo determinado.

4. VHDL

- Simulação (3) -

- Se uma atribuição tiver um tempo de simulação explícito (usando a cláusula `after`, por exemplo), então ela será colocada na lista de eventos e escalonada para que ocorra ao fim desse tempo.
- Caso contrário, a atribuição é concretizada “imediatamente”.
- Na realidade a atribuição é escalonada para ocorrer num instante que é dado pela soma do tempo actual com um atraso minúsculo (***delta delay***).
- O *delta delay* é um intervalo infinitamente curto, que mesmo somando ao tempo de simulação actual um número qualquer de *delta delay*’s ainda se obtém o tempo de simulação actual.
- O conceito de *delta delay* permite que um processo seja executado múltiplas vezes (se necessário) num tempo de simulação nulo.
- Quando se completa um ciclo de simulação, percorre-se a lista de eventos em busca dos sinais em que vai ocorrer a alteração + próxima.

4. VHDL

- Simulação (4) -

- A proximidade pode ser tão diminuta quanto um *delta delay*, ou então ser um atraso real. No 2º caso o tempo de simulação avança.
- Em qualquer dos casos, as alterações num sinal, escalonadas para esse instante, serão efectuadas.
- Os processos que forem sensíveis às alterações do sinal que acaba de mudar serão escalonados para execução no próximo ciclo de simulação.
- A simulação evoluirá indefinidamente até a lista de eventos estar vazia.
- O mecanismo da lista de eventos permite simular o funcionamento de processos concorrentes num sistema uni-processador.
- O mecanismo do *delta delay* garante um funcionamento correcto mesmo quando os processos exigem várias execuções antes de estabilizar.

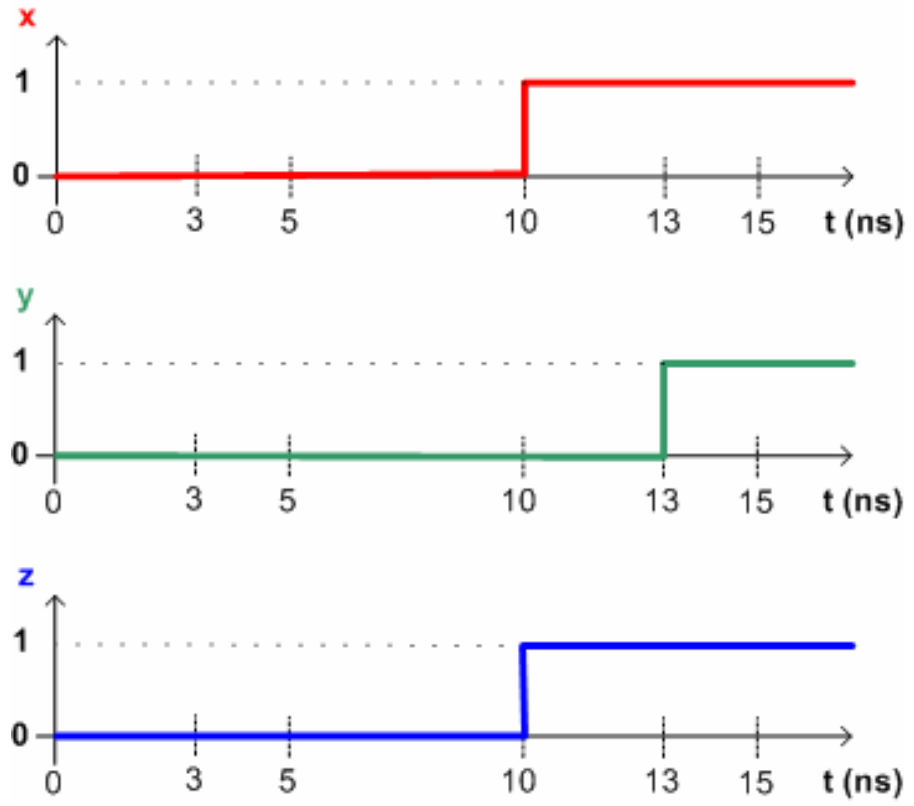
4. VHDL

- Simulação (5) -

- Exemplo com 2 processos:

```
signal x : std_logic := ‘0’;
signal y : std_logic := ‘0’;
signal z : std_logic := ‘0’;
…
process (x,y)
begin
   x <= '1' after 10 ns;
   y <= x   after  3 ns;
end process;

process
begin
   z <= x or y;
   wait for 5ns;
end process;
```

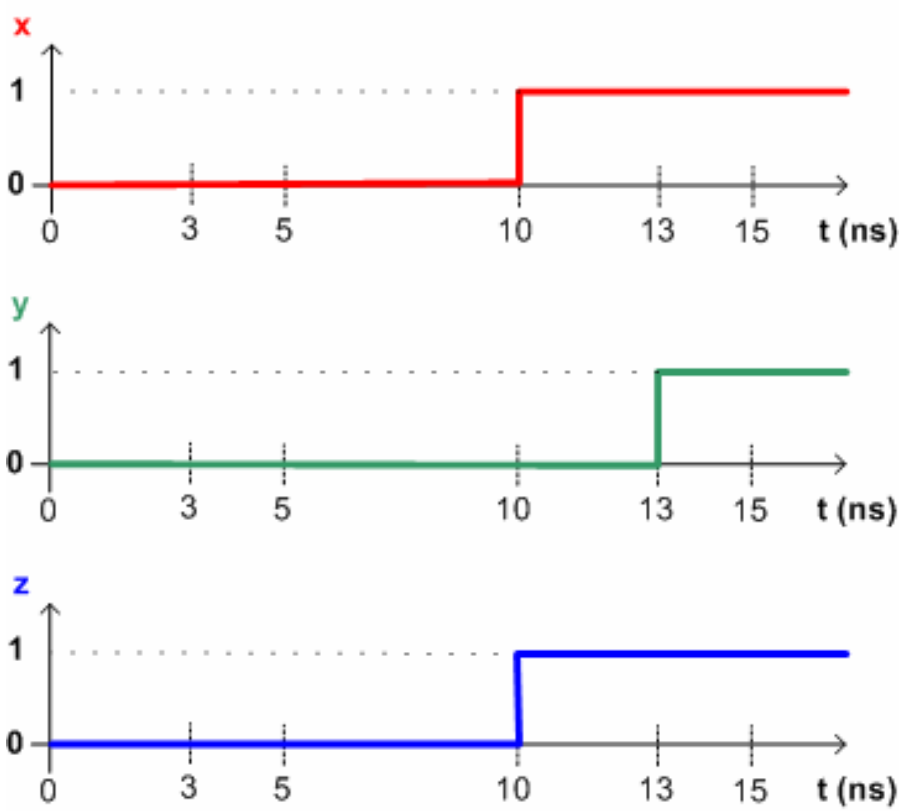


4. VHDL

- Simulação (6) -

- t=0 x,y,z = 0,0,0 (valores iniciais)
  (P1) escalona x=1 para t=10,
  escalona y=1 para t=3, suspende P1
  (P2) z=0+0=0, suspende P2 até t=5ns
- t=3 y = 0 (já era), não activa P1
- t=5 retoma-se P2
  (P2) z=0+0=0, suspende P2 até t=10ns
- t=10 x = 1 (evento em x que activa P1)
  (P1) escalona x=1 para t=20,
  escalona y=1 para t=13, suspende P1
  (P2) z=1+0=1, suspende P2 até t=15ns
- t=13 y = 1 (evento em y que activa P1)
  (P1) escalona x=1 para t=23,
  escalona y=1 para t=16, suspende P1
- t=15 retoma-se P2
  (P2) z=1+1=1, suspende P2 até t=20ns
- t=16 y = 1 (já era), não activa P1

[…] P1 nunca mais é activado e P2 continua a executar de 5 em 5 ns mas não provoca alterações



4. VHDL

- *Simulação (7)* -

- *Testbench* para ALU de 1 bit:

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

entity testAlu1bit is
end entity test_alu1bit;

architecture tst of testAlu1bit is

  component alu1bit is
    port (
    a, b, c  : in std_logic;
    sel : in std_logic_vector (1
    downto 0);
    res, f   : out std_logic);
  end component alu1bit;

  signal i1  : std_logic := '0';
  signal i2  : std_logic := '0';
  signal ci  : std_logic := '0';
  signal op  : std_logic_vector
       (1 downto 0) := "00";
  signal res : std_logic;
  signal co  : std_logic;
```

```
begin
  -- instanciar o sistema
  -- a testar
  ALU1: alu1bit
    port map (
      a      => i1  ,
      b      => i2  ,
      c      => ci  ,
      sel    => op  ,
      res    => res ,
      f      => co  );

  process (i1) is
  begin
    if i1='1' then
      i1 <= '0' after 10ns;
    elsif i1='0' then
      i1 <= '1' after 10ns;
    end if;
  end process;

  process (i2) i
  begin
    if i2='1' then
      i2 <= '0' after 20ns;
    elsif i2='0' then
      i2 <= '1' after 20ns;
    end if;
  end process;
```

```
  process (ci) is
  begin
    if ci='1' then
      ci <= '0' after 40ns;
    elsif ci='0' then
      ci <= '1' after 40ns;
    end if;
  end process;

  process (op) is
  begin
    if op="00" then
      op <= "01" after 80ns;
    elsif op="01" then
      op <= "10" after 80ns;
    elsif op="10" then
      op <= "11" after 80ns;
    elsif op="11" then
      op <= "00" after 80ns;
    end if;
  end process;
end architecture;
```

4. VHDL

- Síntese (1) -

- O VHDL foi pensado para ser uma linguagem de descrição e simulação.
- Mais tarde, foi também adoptado no processo de síntese.
- A linguagem possui várias características e construtores NÃO sintetizáveis.
- O sub-conjunto da linguagem e o estilo de programas apresentados até aqui são em boa medida sintetizáveis pela maior parte das ferramentas comerciais.
- O código que descreve um circuito pode ter um impacto enorme na qualidade do circuito que se consegue sintetizar. Exemplos:
- Estruturas de controlo encadeadas, como o `if-elsif-elsif-else`, podem originar uma série de portas lógicas em cadeia usadas para testar as várias condições.
- É preferível usar um `case` ou uma atribuição selectiva, desde que as condições de selecção (`choices`) sejam mutuamente exclusivas.

4. VHDL

- Síntese (2) -

- Os ciclos dentro de processos são geralmente desdobrados para criar várias cópias da lógica combinacional que executa as instruções do corpo do ciclo.
- Quando se pretende apenas uma cópia da lógica combinacional a executar as instruções do corpo do ciclo, é necessário especificar um circuito sequencial.
- Quando um processo inclui instruções condicionais que não cobrem todas as combinações das entradas, o compilador sintetiza uma *latch* para guardar o valor da(s) saída(s) nos casos não cobertos.
- Geralmente, estas *latches* não são desejadas.
- Algumas das características e construtores que não são sintetizáveis em todas as ferramentas são: memória dinâmica, ficheiros e apontadores.