

Departamento de Engenharia Elétrica e de Computação

“Apostila de Introdução a VHDL”

Profa. Luiza Maria Romeiro Codá

1. Introdução:

1.1 O QUE É HDL?

Linguagem de Descrição de Hardware (**HDL**-Hardware Descriptive Language) descreve, através de um arquivo de texto, o que um sistema digital faz e como. HDL consiste em um tipo de programação de alto nível que permite descrever todas as características importantes de um sistema lógico com qualquer complexidade, descrevendo o comportamento do circuito digital em diversos níveis de abstração (Ex: comportamental, transferência de registros e portas lógicas). Esta descrição é um modelo do sistema hardware, que será executado através da utilização de uma ferramenta (software) EDA (Electronic Design Automation) em um software chamado simulador da seguinte maneira: o projetista cria um arquivo de texto, seguindo certo conjunto de regras, conhecido como sintaxe da linguagem, e usa um compilador para criar dados de programação do dispositivo lógico programável(ou PLD). Um sistema descrito em linguagem de hardware pode ser implementado em um dispositivo programável permitindo assim o uso em campo do seu sistema, tendo a grande vantagem da alteração do código a qualquer momento.

As HDLs têm uma grande semelhança às linguagens de programação, mas são especificamente orientadas à descrição das estruturas e do comportamento do hardware. Uma grande vantagem das HDLs em relação à entrada esquemática é que elas podem representar diretamente equações booleanas, tabelas verdade e operações complexas (p.ex. operações aritméticas).

A HDL possui as seguintes características:

- Possui construção de linguagem para escrever códigos sucintos de descrição de lógica complexa.
- Suporta bibliotecas de projeto e criações de componentes re-utilizáveis.
- Projeto Independente do Dispositivo, ou seja, um projeto pode ser criado sem antes ter que se escolher o dispositivo.
- Possibilita a metodologia de projeto *top-down*, essencial para projetos de circuitos complexos, em substituição a antiga metodologia *bottom-up*.

Existem dezenas de linguagens de HDLs: VERILOG, Handel-C, SDL, ISP, ABEL, etc. As mais populares e padronizadas pelo IEEE (Institute of Electrical and Electronic Engineers) são a Verilog HDL e VHDL. Cada uma das linguagens HDL têm seu próprio estilo, uma estrutura de hardware pode ser modelada igualmente por ambas, a escolha de uma delas pode ser feita de acordo com preferência pessoal, porém a VHDL têm se mostrado mais aceita entre os projetistas.

1.2 O QUE SIGNIFICA VHDL?

Very **H**igh **S**peed **I**ntegrated **C**ircuit (circuitos integrados de altíssima velocidade)

Hardware
Description
Language

‘VHDL é uma Linguagem Descrição de Hardware específica, foi desenvolvida pelo Depto. De Defesa dos Estados Unidos por volta 1980. É uma linguagem usada para facilitar o projeto (concepção) de circuitos digitais programáveis. Apresenta uma descrição textual, um

algoritmo, para desenvolver o circuito, sem necessidade de especificar explicitamente as ligações entre componentes. VHDL é utilizada para as tarefas de documentação, descrição, síntese, simulação, teste e verificação formal. É padronizada pelo IEEE (Institute of Electrical and Electronic Engineers).

Dispositivos de Hardware Digital operam em paralelo, portanto uma linguagem de programação convencional não pode precisamente descrever ou modelar a operação de Hardware digital porque são baseados na execução seqüencial das instruções. Neste caso, VHDL é apropriado, pois seu processo opera em paralelo. Lembrando também que em VHDL, as variáveis mudam sem atraso e os sinais mudam com um pequeno atraso.

1.3 BREVE HISTÓRICO:

As primeiras linguagens de descrição de hardware foram desenvolvidas no final dos anos 60, como alternativa às linguagens de programação para descrever e simular dispositivos hardware. Durante dez anos, inúmeras linguagens foram desenvolvidas com sintaxe e semânticas incompatíveis, permitindo descrições a diferentes níveis de modelização.

No final dos anos 70, o Departamento de Defesa dos Estados Unidos definiu um programa chamado VHSIC (Very High Speed Integrated Circuit) que visava a descrição técnica e projeto de uma nova linha de circuitos integrados. Com o avanço acelerado dos dispositivos eletrônicos, entretanto este programa apresentou-se ineficiente, principalmente na representação de grandes e complexos projetos.

Em 1981, aprimorando-se as idéias do VHSIC, foi proposta uma linguagem de descrição de hardware mais genérica e flexível. Esta linguagem chamada VHDL (VHSIC Hardware Description Language) foi bem aceita pela comunidade de desenvolvedores de hardware e em 1987 se tornou um padrão pela organização internacional IEEE, Padrão IEEE: IEEE Std 1076-1987.

Em 1992 foram propostas várias alterações para a norma, 1993 o IEEE publicou uma versão revisada: IEEE Std 1076-1993 (VHDL-

93), mais flexível e com novos recursos. A qual até hoje é a mais amplamente utilizada.

Atualmente o VHDL é padrão obrigatório nos meios industrial e acadêmico, para: Documentar, Especificar, Simular, e Sintetizar circuitos digitais.

1.4 VANTAGEM DO PROJETO EM VHDL E RELAÇÃO AO ESQUEMÁTICO:

O uso de uma linguagem formal de descrição de hardware como o VHDL, ao invés da descrição por diagramas esquemáticos, apresenta as seguintes vantagens:

- Projeto independente da tecnologia;
- Pode-se utilizar a descrição do projeto em vários tipos de plataforma, de um simulador para outro.
- Pode-se utilizar um projeto em VHDL em diferentes projetos.
- Permite, através de simulação, verificar o comportamento do sistema digital;
- Facilidade na atualização dos projetos;
- Reduz tempo de projeto e custo;
- O objetivo do projeto fica mais claro do que na representação por esquemáticos;
- O volume de documentação diminui, já que um código bem comentado em VHDL substitui com vantagens o esquemático e a descrição funcional do sistema;

1.5 DESVANTAGEM DO PROJETO EM VHDL E RELAÇÃO AO ESQUEMÁTICO:

- O hardware gerado pela descrição VHDL não é otimizado.

1.6 CARACTERÍSTICAS DO VHDL:

- Suporta projetos com múltiplos níveis de hierarquias. Portanto, a descrição geral do circuito pode consistir na interligação de outras descrições menores.
- Favorece projeto “top-down”, onde projetos complexos partem de um nível de especificação mais elevado para um mais baixo.
- Permite, através de simulação, verificar o comportamento do sistema digital;
- Permite descrever hardware em diversos níveis de abstração, por exemplo:
 - Algorítmico ou comportamental;
 - Transferência entre registradores (RTL);
 - Nível de portas lógicas (Gate Level);
- Pode mesclar diferentes níveis de abstração em um mesmo código.
- Comandos executados Concorrentemente (com exceção de regiões específicas no código), assim como os elementos de um sistema digital executam tarefas simultaneamente:
 - Ordem dos comandos é irrelevante;
 - Mudança de valor em um sinal acarreta a execução de todos os comandos envolvidos;Ex: Uma alteração no valor de b, mostrado na Figura 1.1, leva à execução de todos os comandos sensíveis à ele, comandos das linhas 8 e 9.

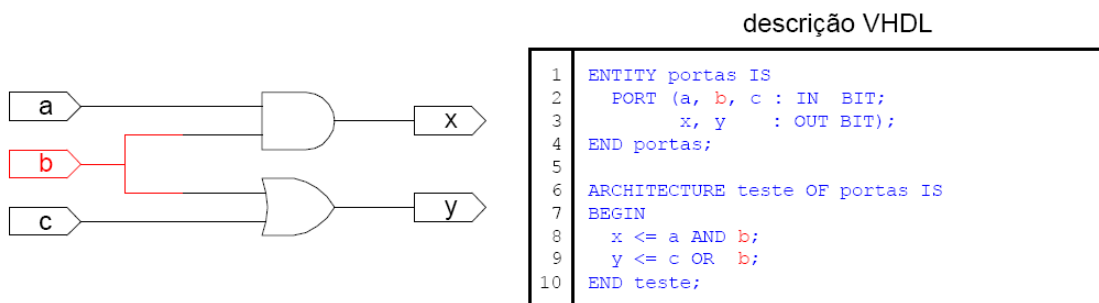


Figura 1.1 Concorrência em uma descrição.

- Possibilita delimitar regiões de código seqüencial (subprogramas e processos) onde a execução dos comandos segue a ordem de sua apresentação no código, como mostra a Figura 1.2. Dentro de cada região os comandos são executados concorrentemente.

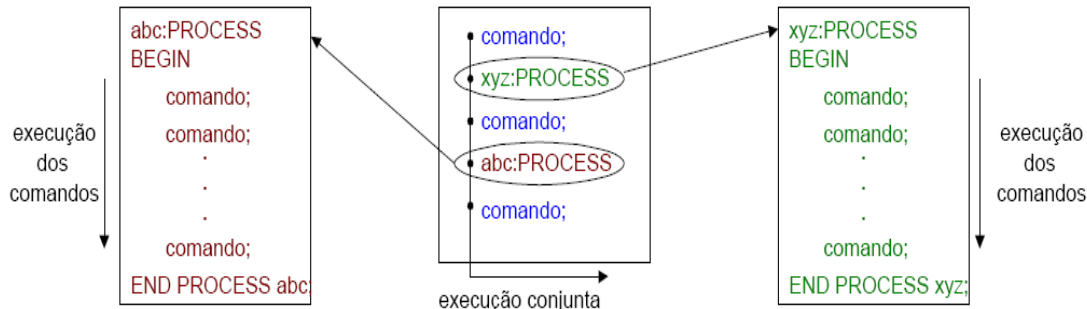


Figura 1.2 Regiões de código contendo comandos seqüenciais.

- Possibilita a definição de Biblioteca e Pacote (“Library” e “Package”, respectivamente).
 - **Pacotes**(“Package”,): armazenam subprogramas, constantes ou novos tipos definidos, evitando a repetição de uma definição em todas as descrições.
 - **Bibliotecas** (“Library”): armazenam informações compiladas, sendo a biblioteca corrente denominada “Work”.

1.7 FERRAMENTAS:

- ghdl (Linux):
 - Front-end do gcc para VHDL
- Altera Quartus II (Windows)
 - Síntese para FPGA da Altera
- Xilinx ISE (Windows)
 - Síntese para FPGA da Xilinx
- Modelsim (Windows/Linux)
 - Simulador
- Mentor Leonardo ou Precision (Windows/Linux)
 - Síntese para diversos fabricantes

2. Ciclo do Projeto em VHDL:

O projeto de um sistema digital auxiliado por ferramentas computadorizadas segue normalmente as 3 etapas mostradas na Figura 2.1 e

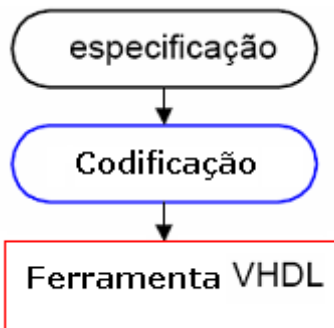


Figura 2.1 Etapas de projeto em VHDL.

- **Especificação:** determinar requisitos e funcionalidade do projeto.
- **Codificação:** com base nas especificações, descrever em VHDL todo o projeto, segundo padrões de sintaxe.
- **Ferramenta:** submeter a descrição em VHDL à um software para verificar a correspondência entre especificação e código e sintetizar o circuito. Essa etapa pode ser dividida em duas sub-etapas como mostra a Figura 2.2, a etapa de elaboração da descrição VHDL e etapa de síntese da descrição VHDL.

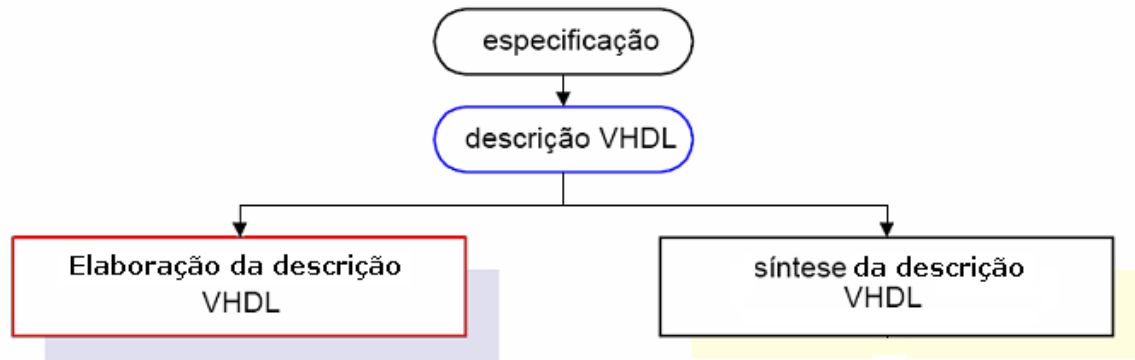


Figura 2.1 Etapas da Ferramenta VHDL.

A etapa de elaboração da descrição VHDL é ilustrada na Figura 2.3. Como a linguagem VHDL possibilita descrever o mesmo circuito de diversas maneiras, ou seja, com diferentes níveis de abstração, o código gerado pode não poder ser sintetizado. Portanto, são necessários vários processos iterativos de simulações até ser atingida uma descrição que coincida com as especificações do projeto e que também possa ser sintetizada.

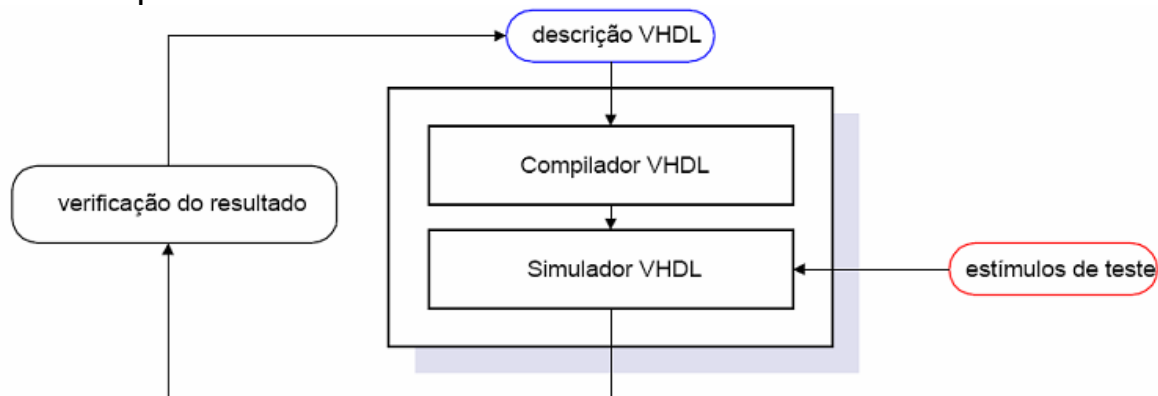


Figura 2.3 Etapa de elaboração da descrição VHDL.

Figura 2.2

A etapas da elaboração da descrição VHDL estão explicadas a seguir:

]

- **Compilação:** transforma o arquivo texto (descrição VHDL) em informações sobre o circuito.

- **Simulação do Código-Fonte:** simular o código em ferramenta confiável a fim de verificar preliminarmente cumprimento da especificação;

Uma vez completada a etapa de elaboração da descrição VHDL inicia-se a etapa de Síntese que pode ser vista na Figura 2.4, a qual consiste nos passos:

- **Geração de circuito nível RTL(Register Transfer Level):** circuito esse que emprega primitivas disponíveis na ferramenta (comparadores, somadores, registradores e portas lógicas). Esse circuito é gerado após a verificação de erro de sintaxe e consiste em uma interligação das estruturas necessárias para obter o circuito gerado a partir da descrição. Nessa etapa o circuito gerado ainda não está associado a nenhuma tecnologia de fabricação.
- **Geração de circuito específico para a tecnologia escolhida:** dessa etapa obtém-se um arquivo contendo uma rede de ligações entre os elementos disponíveis na tecnologia empregada.

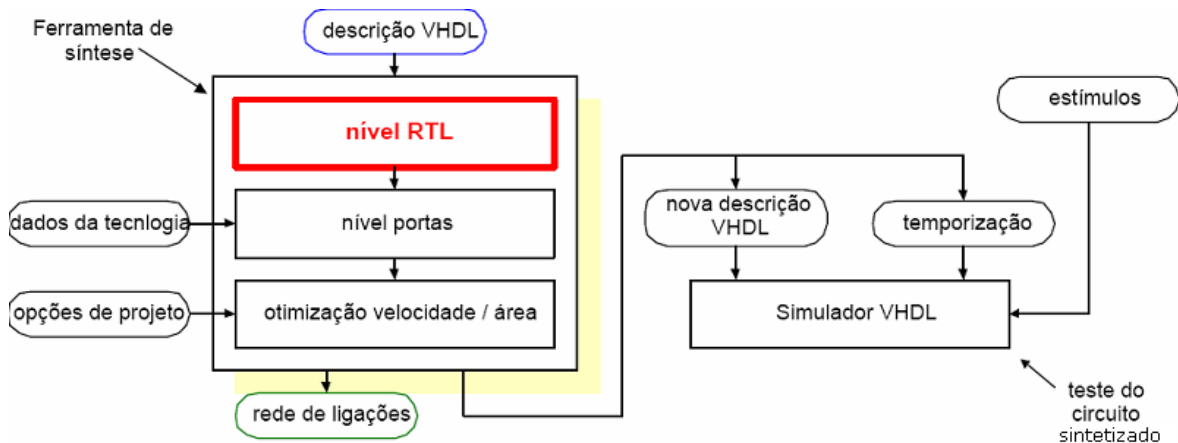


Figura 2.4 Síntese da descrição VHDL.

A Figura 2.5 ilustra um processo de síntese partindo da descrição VHDL, a qual executa a soma de valores entre zero e sete. A ferramenta gera o circuito RTL que sugere utilizar um

somador com três bits e na etapa seguinte elabora a primitiva RTL utilizando os elementos disponíveis na tecnologia escolhida.

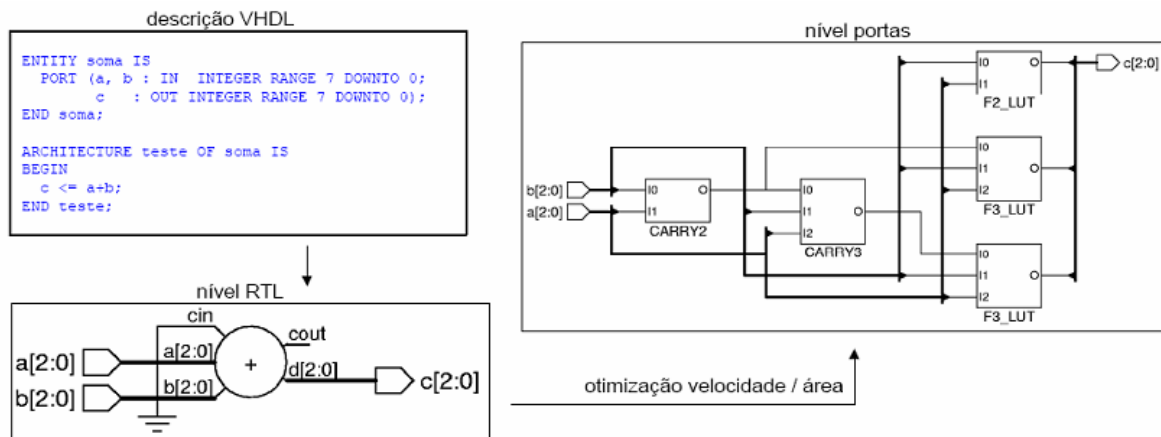


Figura 2.5 Passos executados na ferramenta de síntese VHDL.

A partir da rede de ligações obtida da etapa anterior, a ferramenta define o posicionamento e as interligações dos componentes no dispositivo, antes da construção, ou implementação no dispositivo. Essa etapa é ilustrada na Figura 2.6.



Figura 2.6 Etapa Final do projeto

■ **Implementação:** configuração das lógicas programáveis ou de fabricação de ASICs

A figura 2.7 esquematiza todo o processo, desde a especificação até a implementação do circuito.

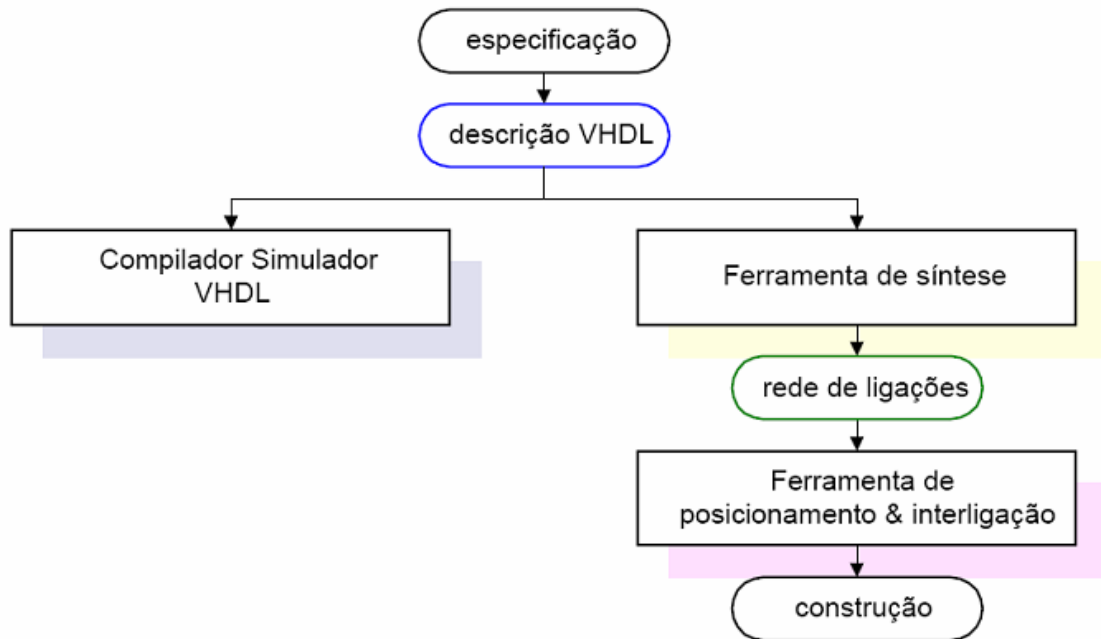


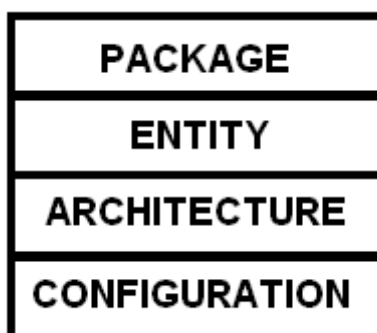
Figura 2.7 Etapas de um projeto VHDL.

3. Conceitos Básicos da Linguagem VHDL:

Devido à sua potencialidade, a linguagem VHDL é complexa, e muitas vezes de difícil entendimento, dada as inúmeras opções para modelar o comportamento de um circuito. Entretanto, o entendimento de um pequeno número de comandos, suficiente para o modelamento de estruturas medianamente complexas, pode ser rapidamente atingido. A necessidade de projetos mais complexos encaminha à procura por novos comandos levando a uma maior compreensão das opções da linguagem.

Considerando o bloco de função lógica da Figura 3.1, ele é composto de portas de entradas as quais são as variáveis lógicas **I0**, **I1** e **I2** e de duas portas de saída para a função $f(I0, I1, I2)$ que são **S1** e **S2**. Uma Linguagem de Descrição de Hardware, como a VHDL, possibilita que a operação lógica que descreve internamente o bloco por exemplo, da Figura 3.1, seja descrita usando enunciados bem definidos tal como uma linguagem de programação de alto nível para computadores. Para utilizar a linguagem VHDL, inicialmente faz-se a descrição da unidade lógica usando a sintaxe e formato específicos da linguagem. Essas informações são as entradas de um arquivo texto o qual será utilizado como entrada de um compilador VHDL. O arquivo de saída é então utilizado na etapa de simulação gráfica, para verificação de resultados. Estando o projeto funcionando como esperado, é só programar o chip com o projeto.

A estrutura de um programa em VHDL baseia-se em 4 blocos:



- **PACKAGE ou LIBRARY:** conjunto de sub-programas que descrevem, elementos e componentes já programados para serem reutilizados.
- **ENTITY (Entidade):** define as portas de entradas e saídas dos circuitos na descrição.
- **ARCHITECTURE (Arquitetura):** implementações do projeto; descreve as relações entre as portas.
- **CONFIGURATION (Configuração):** define as arquiteturas que serão utilizadas.

A estrutura de uma descrição VHDL com os seus blocos definidos é mostrado a seguir:

LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.all; USE IEEE.STD_LOGIC_UNSIGNED.all;	PACKAGE (BIBLIOTECAS)
ENTITY exemplo IS PORT (<descrição dos pinos de I/O>); END exemplo;	ENTITY (PINOS DE I/O)
ARCHITECTURE teste OF exemplo IS BEGIN END teste;	ARCHITECTURE (ARQUITETURA)

A descrição em VHDL de qualquer módulo lógico é representada pela **Entidade de Projeto**, Figura 3.2 a qual deve ser composta de ao menos duas das estruturas, que consistem na declaração da **entidade** (“**entity**”) e na **arquitetura** (“**architecture**”). Para o módulo da Figura 3.1

■ **ENTITY:** No caso da entidade de Projeto da Figura 3.1 seria: I0, I1, I2, S0 e S1

■ **ARCHITECTURE:** No caso da entidade de Projeto da Figura 3.1 seria o circuito interno ao bloco referente ao módulo.

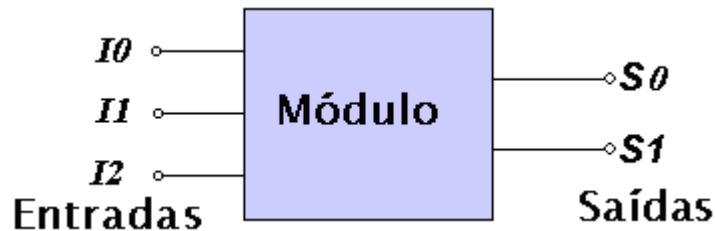


Figura 3.1 Módulo Lógico Genérico.

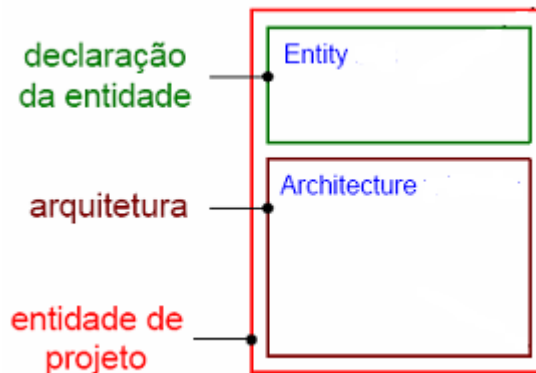


Figura 3.2 Entidade de projeto da descrição VHDL

3.1 DECLARAÇÃO DE PACKAGE ou LIBRARY:

PACKAGE ou **LIBRARY** são conjuntos de declarações usadas para um determinado fim. Pode ser um conjunto de sub-programas (contendo bibliotecas ou funções) que operem com um tipo de dado assim como pode ser um conjunto de declarações necessárias para modelar um determinado projeto.

Pacotes mais utilizados estão citados no Quadro 3.1

Quadro 3.1

Biblioteca	Pacote	TIPOS DE DADOS	DESCRIÇÃO
IEEE	STD_LOGIC_1164	STD_LOGIC e STD_LOGIC_VECTOR	Define o padrão para descrever a interconexão entre tipos de dados usados na linguagem VHDL
IEEE	STD_LOGIC_ARITH	Especifica tipos de dados sinalizados e não sinalizados	Funções de conversão, operações aritméticas e comparações para uso de dados sinalizados e não sinalizados
IEEE	STD_LOGIC_UNSIGNED	STD_LOGIC_VECTOR	Define funções que permitem usar tipos de dados STD_LOGIC_VECTOR, como se fossem tipo de dado não sinalizado
IEEE	STD_LOGIC_SIGNED	STD_LOGIC_VECTOR	Define funções que permitem usar tipos de dados STD_LOGIC_VECTOR, como se fossem tipo de dado sinalizado
IEEE	NUMERIC_STD		Define operações aritméticas seguindo o padrão IEEE. Substitui os pacotes usados juntos STD_LOGIC_ARITH, STD_LOGIC_UNSIGNED e STD_LOGIC_SIGNED
STD	STANDARD	BIT(0 ou 1) BIT_VECTOR(3 Downto 0); BIT_VECTOR(0 to 3); BOOLEAN(falso ou verdadeiro); INTEIRO(positivo ou negativo)	
STD	TEXTIO		Define os arquivos de operações.
WORK	<definido pelo usuário>		Biblioteca corrente de trabalho

OBS: As bibliotecas STD e WORK não necessitam ser referenciadas no projeto VHDL, pois são assumidas automaticamente pela linguagem.

A declaração do uso de pacotes e bibliotecas é mostrada nos Quadros 3.2 a e b

Quadro 3.2a

```
LIBRARY < nome_da biblioteca>
```

Quadro 3.2b

```
LIBRARY < nome_da biblioteca>
```

```
USE < nome_do_pacote_biblioteca> .ALL
```

No Quadro 3.2b o uso de **.all** implica que todos os elementos da biblioteca podem ser usados.

OBS: Sempre que for utilizar alguma biblioteca sempre usar .ALL

3.2 DECLARAÇÃO DA ENTIDADE:

Todo componente VHDL tem que ser definido como uma entidade (entity), o que nada mais é do que uma representação formal de uma simples porta lógica até um sistema lógico completo. Na declaração de uma entidade, descreve-se o conjunto de entradas e saídas que constituem o projeto, é equivalente ao símbolo de um bloco em captura esquemática, como está esquematizado na Figura 3.3.

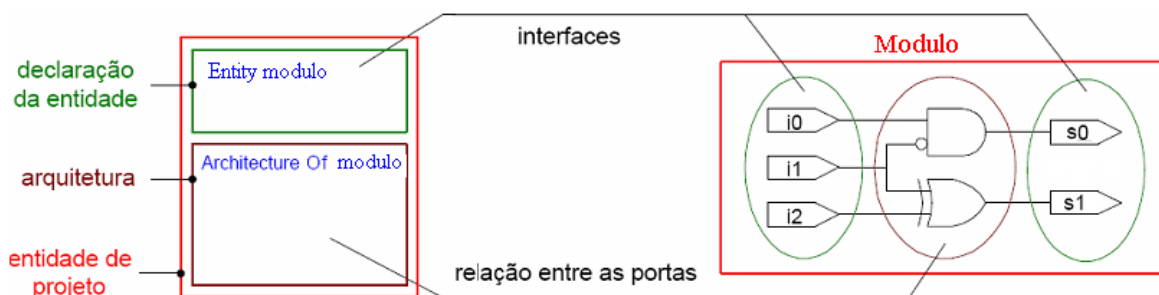


Figura 3.3 Exemplo de entidade de Projeto do bloco Modulo .

Para o módulo lógico da Figura 3.1, a declaração da entidade é que vai declarar que este módulo contém **i0**, **i1** e **i2** como entradas, e **s0** e **s1** como saída. Uma declaração genérica para o módulo da Figura 3.1 é mostrada no Quadro 3.3.

Quadro 3.3

```
ENTITY modulo IS
```

```
    PORT ( i0, i1, i2 : modo_1 tipo_a; --entradas  
          S0, S1 : modo_2 tipo_b ); -- saídas
```

```
END modulo;
```


A declaração de uma entidade deve conter três cláusulas como explicado a seguir:

■ **ENTITY** : palavra reservada “ENTITY” inicia a declaração seguida do nome que a identifica e da palavra reservada **IS**.

Ex: No caso da entidade de Projeto da Figura 3.1 seria:

ENTITY modulo **IS**

■ **PORT**: lista as entradas e as saídas da entidade de projeto. Define o modo e tipo das portas.

Modos de operação de uma PORT: existem quatro modos, representados na Figura 3.4, os quais são:

- Modo **IN** : entrada
- Modo **OUT**: saída
- Modo **BUFFER**: saída que pode ser realimentada internamente.
- Modo **INOUT**: bidirecional.

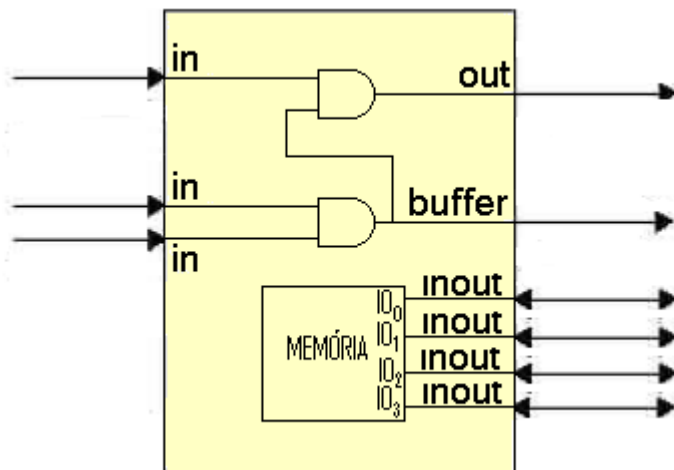


Figura 3.4 Modos de operação de uma PORT.

A declaração inicia com a palavra reservada **PORT** e em seguida abre-se um parêntesis (seguido do nome das entradas separados por vírgula, seguido de dois pontos : do modo, por exemplo : **IN** e

do tipo das entradas listadas na linha e de ponto e vírgula(;), quando trata-se de entradas. A última declaração da porta é finalizada com o parêntesis antes do ponto e vírgula (;).

Ex: No caso da entidade de Projeto da Figura 3.1 seria:

```
PORT ( I0, I1  : IN tipo_a; --entradas
        S1, S2 : OUT tipo_b ); -- saídas
```

OBS: comentários iniciam com --

Tipos de uma PORT: Serão explicados em um item posterior, mas os tipos de PORT mais utilizados estão descritos no Quadro 3.4 a seguir.

Quadro 3.4

bit	Assume valores ‘0’ ou ‘1’. x: in bit;
bit_vector	Vetor de bits. x: in bit_vector(7 downto 0); x: in bit_vector(0 to 7);
std_logic	x: in std_logic;
std_logic_vector	x: in std_logic_vector(7 downto 0); x: in std_logic_vector(0 to 7);
boolean	Assume valores TRUE ou FALSE
real	

Ex: No caso da entidade de Projeto da Figura 3.1 a atribuição do Tipo poderia ser:

```
PORT ( I0, I1  : IN BIT; --entradas
        S1, S2 : OUT BIT ); -- saídas
```

■ **END:** termina a declaração. Usa-se a palavra reservada END seguida do nome da Entidade de Projeto e de ponto e vírgula(;).

o projeto é composto por diversas entidades distintas

Ex: No caso da entidade de Projeto da Figura 3.1 seria:

END modulo;

Desta forma a declaração da Entidade completa da Entidade de Projeto da Figura 3.1 é mostrada no Quadro 3.5 a seguir:

Quadro 3.5

```
ENTITY modulo IS
    PORT ( I0, I1 : IN BIT; --entradas
           S1, S2 : OUT BIT ); -- saídas
END modulo;
```

Uma declaração genérica de uma entidade é mostrada no Quadro 3.6

Quadro 3.6

```
ENTITY nome IS
    PORT ( I0, I1,...In : IN tipo_a; --entradas
           S0, S1      : OUT tipo_b ); -- saídas
           Y0         : BUFFER tipo_c); -- saída
           Z0, Z1     : INOUT tipo_d ); -- entrada/saída
END nome;
```

Normalmente as portas de entrada de uma entidade são declaradas antes das de saída. Cada declaração de interface é seguida por ponto e vírgula ';'. Também é necessário colocar-se ponto e vírgula no final da definição de porta.

3.3 DECLARAÇÃO DA ARQUITETURA:

A declaração da arquitetura (“**architecture**”) descreve o comportamento da entidade, define o seu funcionamento interno, isto é, como as entradas e saídas influem no funcionamento e como se relacionam com outros sinais internos. Para tal, utiliza-se uma série de comandos de operação. A declaração de uma arquitetura pode conter comandos concorrentes ou seqüenciais. Sua organização pode conter declaração de sinais, constante, componentes, operadores lógicos, etc, assim como comandos(ex: BEGIN, END). VHDL permite ter mais de uma **architecture** para a mesma entidade. Uma **Arquitetura** consiste de duas partes:

- a seção de declaração da **arquitetura**.
- e o corpo da **arquitetura**.

Um exemplo de declaração de arquitetura é mostrado no Quadro 3.7:

Quadro 3.7

```
architecture behavioral of ent is  
signal c_internal: small_int;  
begin  
c_internal <= a0 + b0;  
c0 <= c_internal;  
c1 <= c_internal + a1 + b1;  
end behavioral;
```

A seção de declaração da arquitetura (“**architecture**”) é a área entre a chave **architecture** e a chave **begin**. Nesse espaço pode-se declarar objetos que são localizados na **arquitetura**. Após a seção de declaração tem-se o corpo da **arquitetura** o qual especifica o comportamento da **arquitetura**.

A arquitetura de uma entidade pode ser descrita de três formas distintas de abstração, mas que, em geral, conduzem a uma mesma implementação.

- **Descrição estrutural**
- **Descrição por Fluxo de dados (“data-flow”)**
- **Descrição comportamental**

3.2.1 DESCRIÇÃO ESTRUTURAL:

Descreve todos os componentes e suas interconexões onde as atribuições de sinais são feitas através do mapeamento de entradas e saídas de *componentes*. Ou seja, é como se fosse uma lista de ligações entre componentes básicos pré-definidos, onde:

- **Component:** é exatamente a descrição de um componente
- **port map:** é um mapeamento deste componente em um sistema maior.

Exemplo 1: Um exemplo de arquitetura estrutural pode ser visto tomando o circuito da Figura 3.5., (Unidade_AOI), a saída é dada pela expressão:

$$f = a.b + c.d$$

Onde:

$$\begin{aligned} X_1 &= a.b \\ X_2 &= c.d \end{aligned}$$

Então:

$$f = X_1 + X_2$$

As três equações para X_1 , X_2 e f fornecem a estrutura básica do circuito lógico, pois fornecem a informação necessária para reconstruir o diagrama exatamente como ele foi originalmente apresentado. Modelos estruturais são escritos usando o mesmo conceito.

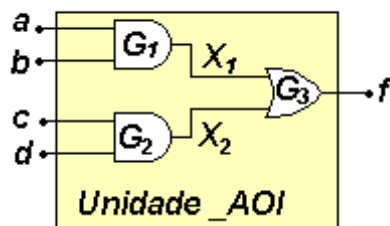


Figura 3.5 Exemplo de modelo estrutural

No caso do exemplo da Figura 3.5, os sinais X1 e X2 são sinais internos, portanto não foram declarados na “**ENTITY**”, então devem ser declarados no corpo da arquitetura antes entre as declarações “**ARCHITECTURE**” e “**BEGIN**”.

Modelos estruturais em VHDL são criados utilizando blocos construtivos chamados de **componentes**. Um componente é um módulo lógico definido em uma listagem normal de uma entidade com uma arquitetura, sendo utilizado para criar outros módulos lógicos. Um componente pode ser visto como uma simples porta lógica, ou representar um bloco mais complicado. Um componente será analisado como um simples bloco construtivo, independente da sua complexidade interna. Para criar um modelo estrutural, mostrado no Quadro 3.8 do circuito da Figura 3.5, primeiro se escolhe os componentes e depois se descreve como eles são interconectados. A conexão é especificada por uma listagem formal que mapeia as portas dos módulos. O mapeamento de portas (**port map**) especifica que portas da entidade são conectadas para quais sinais do sistema que se está montando.

Quadro 3.8: Descrição estrutural da Unidade_AOI,

```
--Descrição estrutural circuito da Figura 3.5
entity Unidade_AOI is
    port(a, b, c, d : in bit;
          f : out bit);
end Unidade_AOI;
-- A seguinte listagem constrói a unidade usando componentes
architecture Estrutural of Unidade_AOI is
--Define a porta AND como um componente
component AND2
    port(x, y : in bit;
          z : out bit);
end component;
--Define a porta OR como um componente
component OR2
    port(x, y : in bit;
          z : out bit);
end component;
--A próxima linha declara os seguintes sinais internos ao módulo
```

```

signal X1, X2 : bit;
-- O mapeamento das portas especifica a conexão interna
begin
    G1 : AND2 port map(a, b, X1);
    G2 : AND2 port map(c, d X2);
    G3 : OR2 port map(X1, X2, f);
end Estrutural;

```

Um dos conceitos novos apresentados na Quadro 3.8 foi a declaração de um componente a qual genericamente é mostrada no Quadro 3.9:

Quadro 3.9 Declaração de componente

```

component AND2
    port(x, y : in bit,
        z : out bit);
end component;

```

Para utilizar uma unidade como se fosse um componente, um módulo mais complexo, ele deve ser previamente definido como uma entidade que tem uma determinada arquitetura. Isto significa que já haviam sido feitas na listagem completa do programa em VHDL, as seguintes declarações dos Quadros 3.10 e 3.11:

Quadro 3.10 Declaração de componente AND2

```

entity AND2 is
    port(u, v, : in bit,
        q : out bit);
end AND2;
architecture Logica of AND2 is
begin
    q <= u and v;
end Logica;

```

e,

Quadro 3.11 Declaração de componente OR2

```
entity OR2 is
    port(u, v, : in bit,
          q : out bit);
end OR2;
architecture Logica of OR2 is
begin
    q <= u or v;
end Logica;
```

O conceito de componentes pode ser entendido usando o conceito de biblioteca do projeto, que é uma coleção de diferentes módulos, cada um definido por uma declaração de entidade e arquitetura. Uma vez que as células tenham sido definidas na biblioteca, pode-se usar cópias das células no projeto em questão usando o comando **component**. Isto é chamado de **instanciação** da célula e o componente é chamado de **instância** do original. Instanciar uma célula, significa copiar a célula que está na biblioteca. Uma célula pode ser instanciada quantas vezes for necessário.

Na linha **signal: X1, X2 : bit;**

A palavra **SIGNAL** é utilizada para definir os identificadores (variáveis) internos X1 e X2 que definem as saídas das portas G1 e G2 do diagrama lógico original. A definição de sinal é diferente da definição de identificadores utilizados na declaração de uma porta (**port**) que existe dentro do módulo. Identificadores internos, são portanto utilizados para conectar portas, formando as conexões do módulo.

Depois dos componentes declarados, o circuito é descrito com a utilização das palavras chave **port map**, as quais fornecem a informação de como os sinais (ligações interbas) dos componentes interagem.

A linha **G1 :AND2 port map (a, b, X1);**

É o primeiro mapeamento e define que a porta G1 é um componente AND2 e que está conectado aos identificadores de porta a, b e ao sinal X1. A ordem dos sinais na declaração do componente AND2 foi dada como (u, v, q) que corresponde a primeira entrada, segunda entrada e saída, respectivamente. O mapeamento da porta (a, b, X1) mantém essa mesma ordem. Portanto, a variável de entrada **a** da **Unidade_AOI** é mapeada na variável de entrada **u** da entidade **AND2**; a variável de entrada **b** da **Unidade_AOI** é mapeada na variável de entrada **v** da entidade **AND2**; a variável de entrada **X1** da **Unidade_AOI** é mapeada na variável de entrada q da entidade **AND2**; O mesmo raciocínio serve para as outras linhas de mapeamento.

Exemplo2: Descrição estrutural da entidade d_latch

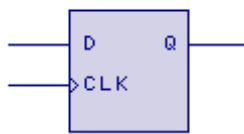


Figura 3.6 Representação da entidade *d_latch*

Tomando-se como base a entidade um flip-flop tipo D simples, mostrado na Figura 3.6, cuja declaração encontra-se a seguir no Quadro 3.12:

Quadro 3.12 Declaração de entidade do circuito da Figura 3.6

```
entity d_latch is
    port ( d, clk : in bit;
           q : out bit);
end d_latch;
```

Pode-se utilizar esta estrutura para se compor sistemas mais complexos, como registradores de um número qualquer de bits. Para ilustrar isto a seguir é apresentada a descrição de um registrador de 2 bits, chamado *reg_2bits*, construído a partir do flip_flop *d_latch*, como mostra a Figura 3.7:

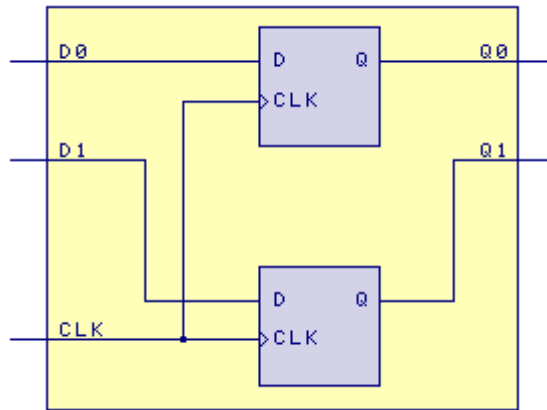


Figura 3.7 Representação de *reg_2bits* a partir de entidades *d_latch*.

A descrição VHDL da estrutura da Figura 3.7 é apresentada a seguir no Quadro 3.13:

Quadro 3.13 Declaração de entidade do circuito da Figura 3.7

```
entity reg_2bits is
    port ( d0, d1, clk : in bit;
           q0, q1 : out bit);
end reg_2bits;
architecture estrutura of reg_2bits is
--Define a porta d_latch como um componente
component d_latch is
    port ( d, clk : in bit;
           q : out bit);
end component;
begin
bit0: d_latch port map (d0,clk,q0);
bit1: d_latch port map(d1, clk, q1);
end architecture estrutura;
```

Pode-se observar neste caso, que na própria indicação dos componentes que constituem o sistema *reg_2bits* já é feito o mapeamento de pinos desejado, ou seja, *d0* é associado ao pino *d* do primeiro flip-flop, assim como o pino *clk* ao sinal de porta de mesmo nome e *q0* ao bit *q*, conforme a estruturação *bit0*. A estrutura *bit1* é descrita de forma análoga somente que uma nova entidade *d_latch* (uma cópia deste componente) será criada para a realização de suas ligações. Este formato traz por si só uma maior flexibilidade do projeto na manipulação de sinais.

Exemplo3: Descrição estrutural da entidade *componente_sistema*

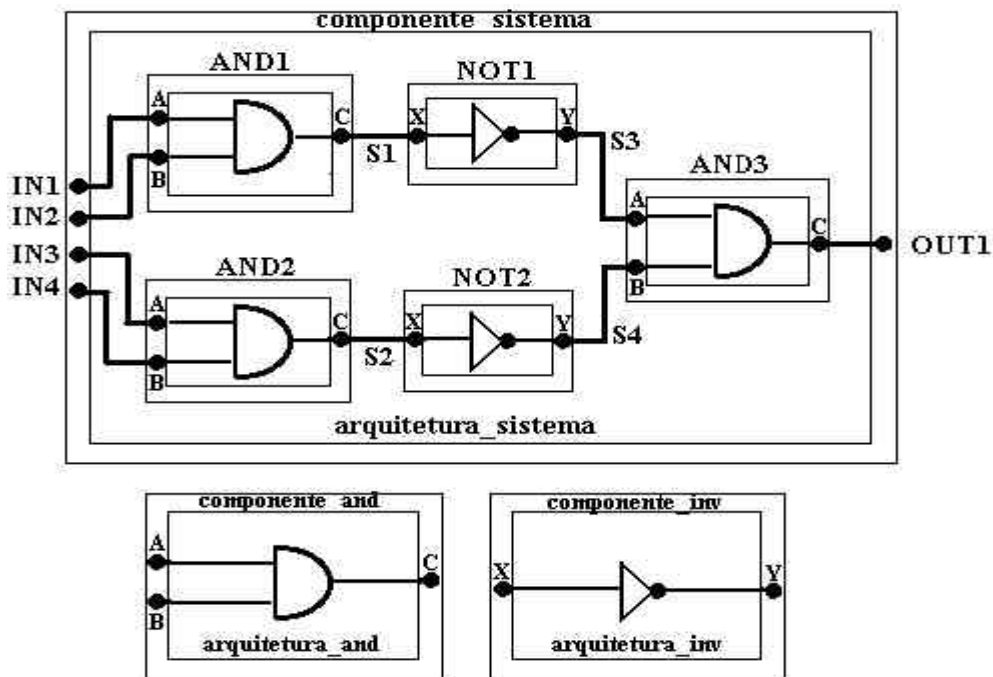


Figura 3.8 Circuito referente ao exemplo 3
componente_sistema

Quadro 3.14 Descrição VHDL dos componentes do exemplo 3

Programa 1	Programa 2
<pre> -- Arquivo componente_inv.vhd -- Modelo do inversor ----- library IEEE; use IEEE.std_logic_1164.all; entity componente_inv is port(x : in bit; y : out bit); end componente_inv; architecture arquitetura_inv of componente_inv is begin y <= not x; end arquitetura_inv; </pre>	<pre> -- Arquivo componente_and.vhd -- Modelo da porta AND ----- library IEEE; use IEEE.std_logic_1164.all; entity componente_and is port(a : in bit; b : in bit; c : out bit); end componente_and; architecture arquitetura_and of componete_and is begin c <= a and b; end arquitetura_and; </pre>

Quadro 3.15 Descrição VHDL do exemplo 3

Programa 3	
<pre> -- Arquivo componente_sistema.vhd -- Modelo da porta AND </pre>	
<pre> library IEEE; use IEEE.std_logic_1164.all; entity componente_sistema is port(in1 : in bit; in2 : in bit; in3 : in bit; in4 : in bit; out1 : out bit); end componente_sistema; architecture arquitetura_sistema of componente_sistema is </pre>	
Declaração Dos Componentes	<pre> { component componente_and port(a : in bit; b : in bit; c : out bit); end component; component componente_inv port(x : in bit; y : out bit); end component; </pre>
Conexão entre Componentes	<pre> signal s1, s2, s3, s4 : bit; begin and1 : componente_and port map (a => in1, b => in2, c => s1); and2 : componente_and port map (a => in3, b => in4, c => s2); and3 : componente_and port map (a => s3, b => s4, c => ut1); inv1 : componente_inv port map (x => s1, y => s3); inv2 : componente_inv port map (x => s2, y => s4); end arquitetura_sistema; </pre>

3.3.2 DESCRIÇÃO POR FLUXO DE DADOS(“*data-flow*”):

Neste tipo de descrição, os valores de saída são atribuídos diretamente, através de expressões lógicas. Todas as expressões são concorrentes no tempo, ou seja, as atribuições ocorrem simultaneamente. Geralmente descrevem o fluxo de dados no sistema.

Um exemplo de descrição de arquitetura por fluxo de dados de um contador completo, mostrado nos circuitos da Figura 3.9 é apresentada no Quadro 3.16.

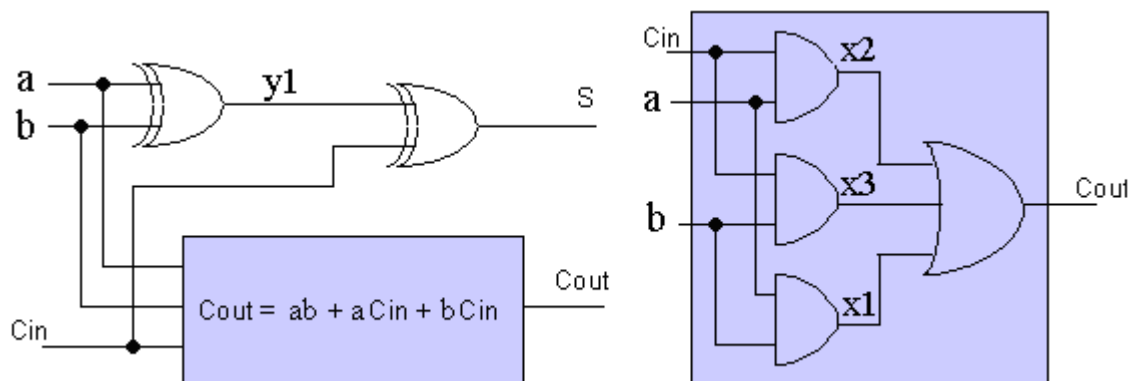


Figura 3.9 Circuito somador completo.

Quadro 3.16

ARCHITECTURE dataflow OF full_adder IS

BEGIN

Cout <= (AND b) OR (a AND Cin) OR (b AND Cin);

S <= a XOR b XOR Cin;

END dataflow;

Pode-se eventualmente utilizar os sinais internos adicionais na descrição, e para isso esses sinais precisam ser declarados na arquitetura, pois não foram declarados na entidade porque não são conexões externas do circuito, como mostra o Quadro 3.17.

Quadro 3.17

ARCHITECTURE dataflow OF full_adder IS

SIGNAL x1, x2, x3, y1 : BIT;

BEGIN

x1 <= a AND b;

x2 <= a AND Cin;

x3 <= b AND Cin;

Cout <= x1 OR x2 OR x3;

y1 <= a XOR b;

s <= y1 XOR Cin;

END dataflow;

Pode-se também descrever uma arquitetura por fluxo de dados usando comandos condicionais, como mostra o Quadro 3.18, onde o circuito da Figura 3.10 é descrito. Portanto, considerando o módulo chamado **Igual**, mostrado na Figura 3.10, sua tabela verdade indica que a sua saída *mesma* = “1” quando as entradas **a** e **b** forem iguais e a saída *mesma* = “0” se **a** e **b** forem diferentes. Um enunciado em, VHDL permite associar um valor a um sinal se algumas condições forem satisfeitas. Usando comandos condicionais pode-se escrever a declaração do módulo **Igual** como mostra o Quadro 3.18 a seguir:

Quadro 3.18 Declaração em VHDL do circuito da Figura 3.10

```
entity Igual is
    port(a,b      :in bit;
          mesma : out bit);
end Igual;
-- Declaração condicional do módulo Igual
architecture fluxo_dados of Igual is
begin
    mesma <= '1' when a = b else
            '0'
end fluxo_dados;
```

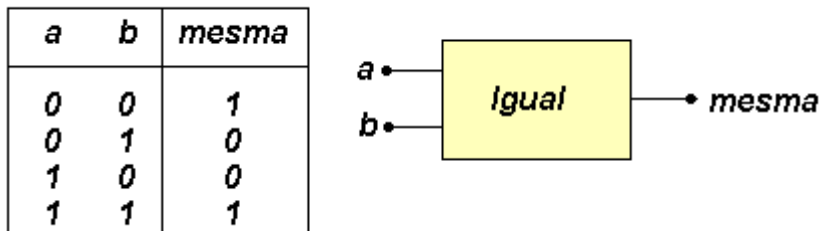


Figura 3.10 Módulo detector de Igualdade

A linha após o comentário -- **Declaração condicional do módulo Igual** indica que esta é a definição de uma nova estrutura chamada *fluxo_dados*, que pertence à entidade chamada *Igual*. Assim esta arquitetura descreve a operação da entidade *Igual*. As linhas entre as diretivas de começo (**begin**) e fim (**end**) descrevem a operação do *Igual*.

Quando se escreve a declaração da arquitetura introduz-se a linha de comando usando a construção geral do Quadro 3.19:

Quadro 3.19

```
Identificador <= '1' when condição else
        '0'
```

Onde essa construção associa ao *identificador* o valor 1 se a condição for verdadeira e associa ao *identificador* o valor Zero ('0') se a condição não for satisfeita.

Este tipo de construção possibilita a descrição em VHDL sem o conhecimento de detalhes internos do módulo, ou sem o conhecimento da expressão lógica do módulo.

3.3.3 MODELOS DE DESCRIÇÃO COMPORTAMENTAL:

A Linguagem de descrição de hardware VHDL permite a utilização de comandos condicionais para construir uma declaração de arquitetura baseada no comportamento do módulo. Esse tipo de descrição, representado no Quadro 3.20, é utilizado na descrição de sistemas seqüenciais cujo comando fundamental é o “**PROCESS**” o qual pode ser, opcionalmente precedido de um rótulo (“*label*”) e seguido de uma *lista de sensibilidade* que indica quais são as variáveis e sinais cuja alteração deve levar à reavaliação da saída. No simulador funcional, quando uma variável da lista é modificada, o processo é simulado novamente. Basicamente, a diferença entre arquitetura com descrição comportamental e por fluxo de dados está no uso da declaração **PROCESS** que define os processos concorrentes.

Obs: Sinais com inicialização assíncrona devem obrigatoriamente estar na lista de sensibilidade. Caso não conste nesta lista, a ferramenta de síntese irá colocar uma mensagem alertando a ausência, mas irá construir o circuito, podendo este não corresponder com o que se deseja implementar. Quando a operação do sinal é síncrona, não há necessidade de incluir o sinal na lista de sensibilidade.

Quadro 3.20

```
process_name: PROCESS( sensitivity_list_signal_1, ... )  
BEGIN  
    -- comandos do processo  
END PROCESS process_name;
```

Esta é a forma mais flexível e poderosa de descrição. O corpo da arquitetura comportamental é montado descrevendo sua função a partir uma ou várias sentenças de processo (**PROCESS**), que são conjuntos de ações a serem executadas. Os tipos de ações que podem ser realizadas incluem expressões de avaliação, atribuição de valores a variáveis, execução condicional, expressões repetitivas e chamadas de subprogramas. Dentro dos processos os comandos são executados seqüencialmente, mas se em uma arquitetura tiver mais de um processo eles serão executados concorrentemente entre si.

Um exemplo de declaração de um comparador de 4 bits é mostrado no Quadro 3.21

Quadro 3.21

```
-- comparador de 4 bits  
entity comp4 is  
    port (    a, b    : in bit_vector (3 downto 0);  
           equals: out bit);  
end comp4;  
architecture comport of comp4 is  
begin  
comp: process (a,b) -- lista de sensibilidade  
    begin  
        if a = b then  
            equals <= '1' ;  
        else  
            equals <= '0' ;  
        end if;  
    end process comp;  
end comport;
```

Assim como em outras linguagens de programação, variáveis internas são definidas também em VHDL, com o detalhe de que estas podem somente ser utilizadas dentro de seus respectivos processos, procedimentos ou funções. Para troca de dados entre processos deve-se utilizar sinais (**signal**) ao invés de variáveis.

Outro exemplo possível de um corpo de arquitetura comportamental é apresentado a seguir no Quadro 3.22, para a entidade *reg4*, registrador de 4 bits:

Exemplo2:

Quadro 3.22

```
architecture comportamento of reg4 is
begin
carga: process (clock)
    variable d0_temp, d1_temp, d2_temp, d3temp : bit;
    begin
        if clk = '1' then
            if en = '1' then
                d0_temp := d0;
                d1_temp := d1;
                d2_temp := d2;
                d3_temp := d3;
            end if;
        end if;
        q0 <= d0_temp after 5ns;
        q1 <= d1_temp after 5ns;
        q2 <= d2_temp after 5ns;
        q3 <= d3_temp after 5ns;
    end process carga;
end architecture comportamento
```

Neste corpo de arquitetura, a parte após a diretiva **begin** inclui a descrição de como o registrador se comporta. Inicia com a definição do nome do processo, chamado *carga*, e termina com a diretiva **end process**.

Na primeira parte da descrição é testada a condição de que ambos os sinais *en* e *clk* sejam iguais a '1'. Se eles são, as sentenças entre as diretivas **then** e **end if** são executadas, atualizando as variáveis do processo com os valores dos sinais de entrada.

Após a estrutura **if** os quatro sinais de saída são atualizados com um atraso de 5ns.

O processo *carga* é sensível ao sinal *clock*, o que é indicado entre parênteses na declaração do mesmo. Quando uma mudança no sinal *clock* ocorre, o processo é novamente executado.

A Figura 3.11 a seguir resume a os campos de um PROCESS.

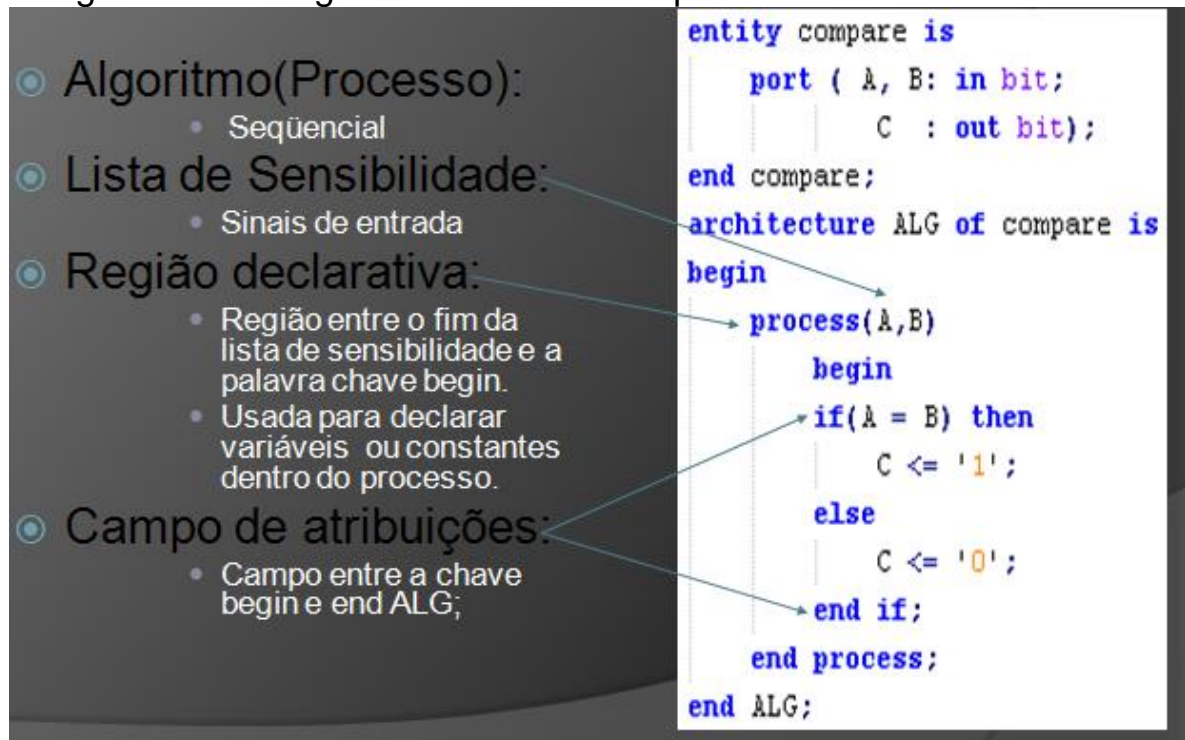


Figura 3.11 Partes de um algoritmo PROCESS.

3.4 CONFIGURATION (Configuração):

A CONFIGURAÇÃO estabelece o elo entre uma declaração de componente e uma entidade de projeto, ela define as arquiteturas que serão utilizadas. Uma mesma entidade pode ter várias arquiteturas. Para ligar a arquitetura à entidade é usada a seguinte declaração:

Especificação do componente	indicação do elo
FOR lista_de_rótulos: nome_componente	USE ENTITY WORK.nome_entidade(arquitetura);

Como por exemplo, o Quadro 3.23 mostra a declaração da entidade “soma” com suas três arquiteturas que são utilizadas na forma de componentes. Na entidade “somadores1” são declarados dois componentes “add” e “adx”. Na declaração do componente “add” o nome das portas coincidem com o das portas da entidade “soma”.

A especificação da configuração é definida nas linhas onde iniciam com declarações “**FOR u1**”, “**FOR u2**” e “**FOR u3**”. Na solicitação “u1” que emprega a declaração do componente “add” é estabelecido um elo com “soma”. Como não foi especificada a arquitetura, é empregada a última arquitetura compilada. A solicitação “u2” emprega também a declaração do componente “add”, para essa solicitação é estabelecido o elo com a arquitetura “lenta” da entidade “soma”. A solicitação “u3” emprega a declaração do componente “adx”, e para essa solicitação é estabelecido o elo com arquitetura “típica” de “soma”. Como nesse último caso os nomes das portas são diferentes, é necessário definir a correspondência entre nomes através de “PORT MAP”.

QUADRO 3.23(exemplo retirado do livro d’Amore

1	-- declaração da entidade “soma” e suas 3 arquiteturas
2	ENTITY soma IS
3	PORT (a, b : IN INTEGER 0 TO 7;
4	s : OUT INTEGER 0 TO 15);
5	END soma;
6	
7	ACHITECTURE lenta OF soma IS
8	BEGIN
9	s<= a + b AFTER 50ns;
10	END lenta;
11	
12	ACHITECTURE típica OF soma IS
13	BEGIN
14	s<= a + b AFTER 25ns;

15	END típica;
16	
17	ACHITECTURE rapida OF soma IS
18	BEGIN
19	s<= a + b AFTER 10ns;
20	END rapida;
21	
22	-- Especificação da configuraçãp
23	ENTITY somadores1 IS
24	PORT (x,y : IN INTEGER 0 TO 7;
25	sl, st, sr : OUT INTEGER 0 TO 15);
26	END somadores1;
27	
28	ACHITECTURE teste OF somadores1 IS
29	--"add": nome_local componente
30	--portas com mesma designação
31	COMPONENT add PORT (a, b : IN INTEGER 0 TO 7;
32	s : OUT INTEGER 0 TO 15);
33	END COMPONENT;
34	
35	--"adx": nome_local componente
36	-- portas com designação diferente
37	COMPONENT adx PORT (k, l : IN INTEGER 0 TO 7;
38	m : OUT INTEGER 0 TO 15);
39	END COMPONENT;
40	
41	-- associação entre "add", "adx" e as entidades de projeto "soma"
42	--arquitetura de u1 é a ultima compilada
43	FOR u1: add USE ENTITY WORK.soma;
44	FOR u2: add USE ENTITY WORK.soma(lenta);
45	FOR u3: adx USE ENTITY WORK.soma(típica) PORT MAP(a=>k, b=>l, s=>m); ;
46	BEGIN
47	u1 : add PORT MAP (x, y, Sr)
48	u2 : add PORT MAP (x, y, Sl)
49	u3 : adx PORT MAP (x, y, St)
50	END teste;

4. Elementos Básicos da Sintaxe de VHDL:

Quadro 4.1 Estrutura de um projeto em VHDL.

Comentários	1	--Exemplos de declarações e comandos de uma descrição VHDL
Declaração de Biblioteca	2	LIBRARY ieee;
	3	USE ieee.std_logic_1164.all;
	4	
Comando Genérico	5	ENTITY contador_generico IS
	6	GENERIC
	7	(Max_Count: natural := 9);
	8	PORT(
	9	(data_input_name : IN INTEGER RANGE 0 TO Max_Count;
Declaração de Entidade	10	clk_input_name : IN STD_LOGIC;
	11	clrn_input_name : IN STD_LOGIC;
	12	ena_input_name : IN STD_LOGIC;
	13	ld_input_name : IN STD_LOGIC;
	14	count_output_name : OUT INTEGER RANGE 0 TO Max_Count);
	15	END contador_generico;
	16	
Declaração de Sinal	17	ARCHITECTURE a OF contador_generico IS
	18	SIGNAL count_signal_name : INTEGER RANGE 0 TO Max_Count;
	19	BEGIN
Declaração de Arquitetura	20	PROCESS (clk_input_name, clrn_input_name)
	21	BEGIN
	22	IF clrn_input_name = '0' THEN
	23	count_signal_name <= 0;
	24	ELSIF (clk_input_name'EVENT AND clk_input_name = '1') THEN
	25	IF ld_input_name = '1' THEN
	26	count_signal_name <= data_input_name;
Declaração de Processo	27	ELSE
	28	IF ena_input_name = '1' THEN
	29	count_signal_name <= count_signal_name + 1;
Comandos Sequenciais	30	ELSE
	31	count_signal_name <= count_signal_name;
	32	END IF;
	33	END IF;
	34	END IF;
	35	END PROCESS;
	36	count_output_name <= count_signal_name;
	37	END a;
	--	

Neste capítulo, serão discutidas as regras básicas de sintaxe de cada comando de uma descrição VHDL. O Quadro 4.1 mostra uma declaração VHDL onde estão indicadas algumas das estruturas e comandos que podem aparecer neste tipo de descrição.

Como em qualquer linguagem de programação, a VHDL utiliza um conjunto bem definido de regras, as quais devem ser seguidas e que definem **palavras-chave** e a **sintaxe** da linguagem que se refere ao uso de uma palavra e a ordem que deve ser obedecida para escrever os comandos. A linguagem não é **case-sensitive**, mas freqüentemente são usadas maiúsculas para as palavras reservadas. Uma **palavra-chave**, mostradas no Quadro 4.2, tem um significado reservado na linguagem e não podem ser usadas para outro propósito. A sintaxe refere-se ao uso de uma palavra e a ordem que deve ser obedecida para escrever os comandos. O Quadro 4.3 mostra uma lista dos símbolos especiais e suas sintaxes.

Quadro 4.2 Palavras reservadas em VHDL.

abs access after alias all and architecture array assert attribute	file for function	nand new next nor not null	then to transport type
begin block body buffer bus	generate generic group guarded	of on open or others out	unaffected units until use
case component configuration constant	if impure in inertial inout is	package port postponed procedure process pure	variable
disconnect downto	label libraries linkage literal loop	range record register reject rem report return rol ror	wait when while with
else	map	select	xor

elseif end entity exit	mod	severity shared signal sla sll sra srl subtype	xnor
---------------------------------	-----	---	------

Quadro 4.3 Símbolos definidos em VHDL.

Símbolo	Significado	Símbolo	Significado
+	Adição ou número positivo	:	Separação entre uma variável e o tipo
-	Subtração ou número negativo	“	Aspas dupla
/	Divisão	‘	Aspas simples ou marca de tick
=	Igualdade	**	Exponenciação
<	Menor do que	=>	Seta indicando “então”
>	Maior do que	=>	Seta indicando “recebe”
&	Concatenador	:=	Associação de valor para variáveis
	Barra vertical	/=	Desigualdade
;	Terminador	>=	Maior do que ou igual a
#	Literal incluído	<=	Menor do que ou igual a
(Parêntese da esquerda	<=	Associação de valor para sinais
)	Parêntese da direita	<>	Caixa
.	Notação de Ponto	--	Comentário

4.1 COMENTÁRIOS:

Os comentários em VHDL são permitidos após dois traços '-' e são válidos até o final da linha corrente.

Exemplo:

```
bit0 := d0; -- Esta linha atribui o valor de d0 a variável bit0
```

No exemplo da Quadro 4.1 : Comentário é:

--Exemplos de declarações e comandos de uma descrição VHDL

4.2 LIBRARY (Bibliotecas):

A explicação da sintaxe de bibliotecas encontra-se no capítulo 3, Quadros 3.2 a e 3.2b. Normalmente, em descrições VHDL, utiliza-se a biblioteca de trabalho *work* (que fica no diretório de trabalho vigente). Para incluir outras bibliotecas que não a *work*, deve-se utilizar a palavra-chave **library** seguida pelo nome da biblioteca desejada, como mostra a descrição a seguir:

Library <nome_da_biblioteca>...
--

Pode-se também incluir todos os componentes de uma biblioteca usando a diretiva **use** em conjunto com a palavra-chave **all** como segue:

Library <nome_da_biblioteca> e/ou
--

Use <nome_da_biblioteca>.all

Por exemplo, a biblioteca IEEE contém um conjunto poderoso de definições como:

- Identificadores de tipo como BIT e BOOLEAN.
- Definições de tempo para ns(nanossegundos), ps(picossegundos) e outras escalas.

A biblioteca é dividida em subgrupos chamados de **pacotes**. Os seguintes itens são encontrados no **pacote 1164** da biblioteca **IEEE** o qual é bastante útil para a linguagem VHDL:

- Identificadores de tipo como '0' e '1'

- Operadores de funções básicas para AND, OR, NOR e outras.
- BIT, VECTORS E BIT_VECTOR.

Para incluir a biblioteca IEEE e o pacote 1164, deve-se iniciar a descrição em VHDL com:

```
Library IEEE
Use IEEE_STD_LOGIC_1164.all
```

Uma vez que a biblioteca IEEE_1164 tenha sido incluída na listagem VHDL, utiliza-se notações diferentes para bit e bit_vector:

- **BIT** é substituído por **STD_LOGIC**
- **BIT_VECTOR** é substituído por **STD_LOGIC_VECTOR**

Essa mudança possibilita mudar de **BIT**, o qual tem apenas dois valores ('0' e '1') para o **STD_LOGIC** mais útil, o qual possui nove valores possíveis:

'0' = 0 forte
 '1' = 1 forte
 'X' = forçando desconhecido
 'Z' = alta impedância (circuito aberto)
 'w' = desconhecido fraco
 'L' = 0 fraco
 'H' = 1 fraco
 'U' = inicialização desconhecida
 '-' = não importa (don't care)

O exemplo a seguir mostra que as regras para descrição continuam iguais, mas os tipos de sinal **STD_LOGIC** são utilizados.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL
ENTITY porta_and IS
    PORT(a, b : IN std_logic;
        f : OUT std_logic);
END porta_and;
ARCHITECTURE lógica OF porta_and IS
BEGIN
```

```
f <= a and b;  
END logica;
```

Praticamente, todos os módulos escritos em VHDL iniciam com:

```
LIBRARY ieee  
USE ieee.std_logic_1164.all  
USE ieee.std_logic_arith.all  
USE ieee.std_logic_unsigned.all
```

Essas quatro linhas significam que deve-se utilizar a biblioteca IEEE, que contém a definição de funções básicas, subtipos, constantes, e todas as definições dos **Packages** incluídos nesta biblioteca.

No quadro 4.1 a declaração de biblioteca é:

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

4.3 IDENTIFICADORES:

Os identificadores são usados para se atribuir nomes à sinais ou processos, devem ser escolhidos de tal maneira que sejam fáceis de serem lembrados e que tenham algum significado no projeto lógico.

Um identificador básico em VHDL:

- pode conter:
 - letras do alfabeto ('A' a 'Z' e 'a' a 'z'),
 - dígitos decimais('0' a '9')
 - caracter underline ('_');
- precisa começar com uma letra do alfabeto;
- não pode terminar com um caracter underline;
- não pode conter dois caracteres underline em sequência.

Exemplos:

contador data0 Novo_valor resultado_final_operacao_FFT

Não há distinção entre letras maiúsculas e minúsculas, logo valor, Valor ou VALOR são interpretados da mesma forma.

VHDL permite ainda a definição de identificadores estendidos que devem ser utilizados somente para interfaceamento com outras ferramentas que usam regras diferentes para definir seus identificadores. A definição de identificadores estendidos é feita entre \'.

Exemplos:

\9data\ \proximo valor\ \bit#123\

Alguns exemplos de identificadores da descrição do Quadro 4.1 são:

contador_generico count_signal_name data_input_name

O Quadro a seguir, mostra o resumo de identificadores válidos e inválidos

<p style="text-align: center;">IDENTIFICADORES VÁLIDOS</p> <ul style="list-style-type: none">palavras seguidas do caracter underline _ Ex.: teste_padrao, input_1_xnão é sensível a LETRA MAIÚSCULA e minúscula Ex.: SOMA = somanome com underline são diferentes de nomes sem underline. Ex.: Teste_Padrao ≠ TestePadrao
<p style="text-align: center;">IDENTIFICADORES INVÁLIDOS</p> <ul style="list-style-type: none">uso de palavras reservadas da linguagem com outras finalidades identificador começando com número => 7AB<ul style="list-style-type: none">uso do caracter @ Ex.: A@Bo caracter underline no fim de um identificador Ex.: soma_<ul style="list-style-type: none">o uso de dois caracteres underlines seguidos Ex.: IN__1

4.4 DECLARAÇÃO GENÉRICA:

A declaração genérica é uma declaração opcional que possibilita passar informações externas para entidades de projeto. O emprego do **GENERIC** possibilita a reconfiguração de um circuito pela simples alteração de seus valores, sem alterar o código do projeto deixando-o generico. A sintaxe da declaração de generico é:

GENERIC (< nome_genérico> : tipo := <valor inicial>);

Uma declaração generic em uma listagem VHDL é:

ENTITY contador IS GENERIC (min_count : NATURAL :=0; max_count : NATURAL :=9); Declaração das portas END contador;
--

No quadro 4.1 a declaração de de genérico é:

GENERIC (Max_count : NATURAL :=9);

4.5 OBJETOS DE DADOS:

Objetos são elementos que contém um valor armazenado. São três as classes de objetos de dados mais usadas : **signals**, **variables**, e **constants**.

- **SIGNAL(Sinal):** representa sinais lógicos sobre um fio no circuito, os quais interligam componentes. Um sinal não tem memória, portanto se a fonte do sinal é removida, o sinal não terá um valor. **PORTS** são exemplos de sinais. Podem ser declarados na entidade ou na arquitetura. Não podem ser declarados em processos, mas podem ser utilizados em seu interior.

Sintaxe:

signal identificador(es) : tipo [restrição] [:=expressão];
--

Exemplo:

```
signal cont : integer range 50 downto 1;  
signal ground : bit := '0';
```

No exemplo do Quadro 4.1 o sinal é:

```
count_signal_name : INTEGER RANGE 0 TO Max_Count;
```

■ **VARIABLE (Variável):** Um objeto VARIABLE lembra seu conteúdo e é usado para cálculos de modelos comportamental.

São utilizadas em processos e devem ser declaradas neles. São atualizadas imediatamente e não correspondem à implementação física, como no caso dos sinais.

Sintaxe se a variável tem valor inicial:

```
variable nome_variavel : tipo [restrição] [:=valor_inicial];
```

Sintaxe se a variável não tem valor inicial:

```
variable nome_variavel : tipo [restrição];
```

■ **CONSTANT(Constante):** Um objeto CONSTANT deve ser inicializado com um valor quando declarado e seu valor não pode ser mudado. Podem ser declaradas nos níveis de:

- package
- entity
- architecture
- process

e valem apenas no contexto em que são declaradas.

Exemplos:

```
SIGNAL x: BIT  
VARIABLE y: INTEGER  
CONSTANT one: STD_LOGIC_VECTOR(3 DOWNT0 0) := "0001"
```

4.6 TIPOS DE DADOS:

Os objetos devem ser declarados segundo uma especificação de **tipo**. Um **tipo** é caracterizado por um conjunto de valores que pode assumir e por um conjunto de operações que podem ser realizadas.

Os **tipos** de objetos são divididos em:

- Escalares
- Compostos

A Figura 4.1 mostra as subdivisões dos tipos escalares e compostos. Porém, neste curso serão estudados apenas os tipos indicados na Figura 4.1.

Os tipos de dados mais utilizados são mostrados no Quadro 4.1 a seguir:

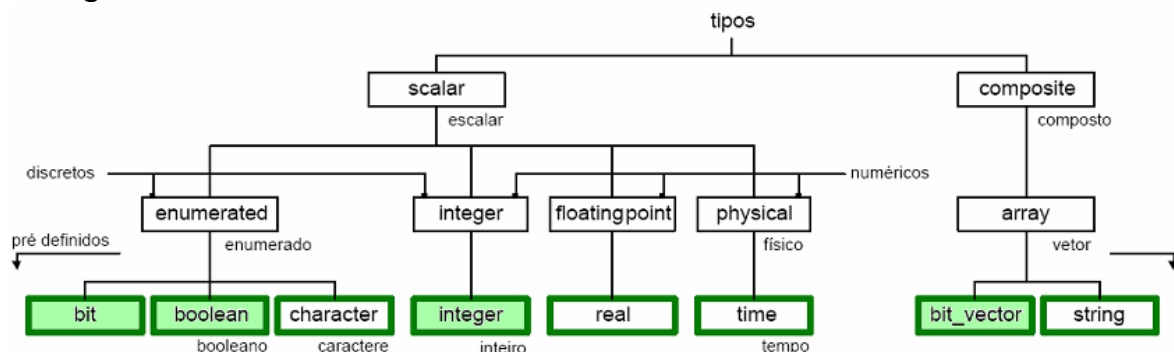


Figura 4.1 Tipos de dados VHDL mais utilizados.

Observação: Não é permitida a transferência de valores entre objetos de tipos diferentes.

4.6.1 Tipos Escalares:

Escalar é um único valor. Os tipos enumerados podem ser ordenados, ou seja, pode-se aplicar operadores relacionais (maior, menor).

O quadro 4.2 apresenta tipos escalares definidos no pacote padrão.

Quadro 4.2

tipo	valor	exemplos
BIT	um, zero	1, 0
BOOLEAN	verdadeiro, falso	TRUE, FALSE
CHARACTER	caracteres ASCII	a, b, c, A, B, C, ?, (
INTEGER	$-2^{31} \leq x \leq 2^{31}-1$	123, 8#173#, 16#7B# 2#11_11_011#
NATURAL	$0 \leq x \leq 2^{31}-1$	
POSITIVE	$1 \leq x \leq 2^{31}-1$	
REAL	$-3.65 \times 10^{47} \leq x \leq +3.65 \times 10^{47}$	1.23, 1.23E+2, 16#7.B#E+1
TIME	ps = 10^3 fs ns = 10^3 ps us = 10^3 ns ms = 10^3 us sec = 10^3 ms min = 60 sec hr = 60 min	1 us, 100 ps, 1 fs

4.6.2 Tipos Compostos ou “Arrays”):

Variáveis de tipo composto são montadas com um agrupamento de variáveis de tipos já conhecidos. No pacote padrão são definidos dois tipos compostos: **BIT_VECTOR** e **STRING**.

■ **BIT_VECTOR**: contém elementos do Tipo “BIT”, ou seja, é um vetor.

Na declaração do objeto, o número de elementos contidos no vetor é especificado através das palavras reservadas “DOWNTO” e “TO”. A primeira indica uma ordem descendente nos índices e a segunda o inverso, como mostra a Figura 4.2

Sintaxe:

TYPE identifier IS ARRAY (range) OF type;

Exemplo:

```
TYPE byte IS ARRAY(7 DOWNT0 0) OF BIT
TYPE memory_type IS ARRAY(1 TO 128) OF byte
SIGNAL memory: memory_type
memory(3) <= "00101101"
```

■ **STRING:** contém elementos do Tipo “CHARACTER”

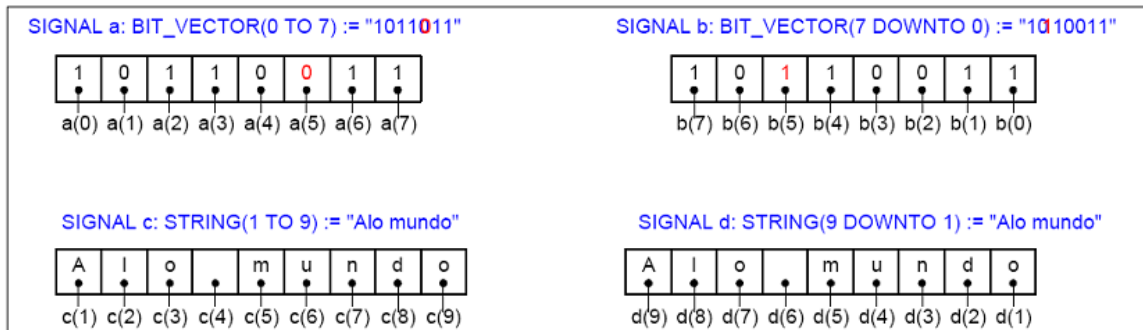


Figura 4.2 exemplos de declarações tipo bit_vector e string

4.6.3 Tipos Utilizados neste curso:

Nesse curso, para fins de simplificação, serão utilizados apenas alguns dos Tipos, mostrados no Quadro 4.3 e descritos a seguir, os quais são os mais utilizados em declaração VHDL.

Quadro 4.3

bit	Assume valores '0' ou '1'. x: in bit;
bit_vector	Vetor de bits. x: in bit_vector(7 downto 0); x: in bit_vector(0 to 7);
std_logic	x: in std_logic;
std_logic_vector	x: in std_logic_vector(7 downto 0); x: in std_logic_vector(0 to 7);
Natural	Assume valores entre 0 e $(2^{31}-1)$
integer	Assume valores $-(2^{31}-1) \leq x \leq (2^{31}-1)$

■ BIT:

Valores: "0" ou "1"

Atribuição de valor: bit_signal <= '0';

Nativo da linguagem VHDL, não precisa de declaração de biblioteca. Exemplo é mostrado no Quadro 4.4

Declaração de BIT:

```
bit_signal : BIT;
```

Quadro 4.4

```
SIGNAL x: BIT;  
x <= '1';
```

■ BIT_VECTOR:

VHDL possibilita que as palavras binárias sejam trabalhadas como vetores de bits, palavras binárias com n-bits, com 2^n valores distintos, simplificando a listagem. Nativo da linguagem VHDL, não precisa de declaração de biblioteca.

Declaração de BIT_VECTOR:

```
bit_vector_signal : BIT_VECTOR( max_index DOWNT0 0 );
```

Atribuição de BIT_VECTOR:

- bit_vector_signal(0) <= '1';
- bit_vector_signal(0) <= bit_signal;
- bit_vector_signal <= "0001";

Um exemplo de aplicação de BIT_VECTOR é mostrado no Quadro 4.5.

Quadro 4.5

```
SIGNAL x: BIT_VECTOR(7 DOWNT0 0)  
x <= "0010"
```

Os exemplos 1 e 2 a seguir mostram a utilização do Tipo BIT_VECTOR:

Exemplo 1:

Considerando a palavra: In_a = a₃ a₂ a₁ a₀

A qual é a entrada do bloco mostrado na Figura 4.2, onde a saída é tem um único bit x . Utilizando o comando **BIT_VECTOR**, os componentes da palavra In_a são especificados pelo número subscrito, como mostrado na listagem de porta:

```
PORT ( In_a : in BIT_VECTOR (3 DOWNT0 0)  
X : out bit)
```

Essa declaração define In_a como sendo quatro componentes nomeados pelo comando **DOWNT0**, em ordem decrescente, a qual não possibilita mais mudar as posições dos bits individuais, os quais são:

In_a(3) = a₃
In_a(2) = a₂
In_a(1) = a₁
In_a(0) = a₀

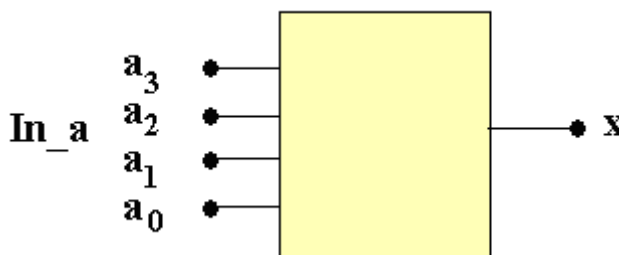


Figura 4.2 Exemplo de uma entrada do tipo BIT_VECTOR

Exemplo2:

Usar o comando **BIT_VECTOR** para descrever o módulo da Figura4.3, onde as entradas são palavras de 4 bits *In_a* e *In_b*, e a saída é também de 4 bits definida por:

$$\text{Saída} = \text{In_a} \text{ OR } \text{In_b}$$

Em que a operação OR é implicitamente indexada, ou seja, a Saída é uma palavra de 4 bits que tem os seguintes valores para cada bit:

$$\text{Saída}(3) = \text{In_a}(3) \text{ OR } \text{In_b}(3)$$

$$\text{Saída}(2) = \text{In_a}(2) \text{ OR } \text{In_b}(2)$$

$$\text{Saída}(1) = \text{In_a}(1) \text{ OR } \text{In_b}(1)$$

$$\text{Saída}(0) = \text{In_a}(0) \text{ OR } \text{In_b}(0)$$

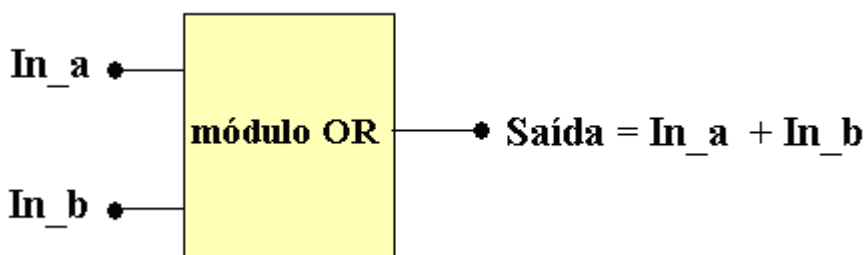


Figura 4.3 Módulo OR usando vetor.

O módulo OR da Figura 4.3 pode ser descrito usando a seguinte listagem em VHDL:

Entity modulo_or **IS**

PORT(*In_a*, *In_b* : in **BIT_VECTOR**(3 downto 0);

Saída : out **BIT_VECTOR**(3 downto 0));

END modulo_or;

ARCHITECTURE listagem **OF** modulo_or **IS**

BEGIN

Saída(3) <= *In_a*(3) OR *In_b*(3);

Saída(2) <= *In_a*(2) OR *In_b*(2);

Saída(1) <= *In_a*(1) OR *In_b*(1);

```
Saída(0) <= In_a(0) OR In_b(0);  
END listagem;
```

Os elementos individuais de um **BIT_VECTOR** podem ser usados em separado. Exemplo:

```
f1 <= In_a(3) AND In_b(3);
```

■ **STD_LOGIC e STD_LOGIC_VECTOR:** Os tipos **STD_LOGIC** e **STD_LOGIC_VECTOR** fornecem mais valores do que o tipo **BIT** para modelagem mais precisa de um circuito real. Objetos desses tipos podem ter os seguintes mostrados no Quadro 4.6 a seguir.

Quadro 4.6

'U': não inicializada	'Z': alta impedância
'X': desconhecida	'W': desconhecida fraca
'0': valor '0'	'L': '0' fraca (Low)
'1': valor '1'	'H': '1' fraca (High)
'-': <i>Don't care.</i>	

Os tipos **STD_LOGIC** e **STD_LOGIC_VECTOR** não são predefinidos e então duas bibliotecas (library) devem ser incluídas para usar esses tipos, a biblioteca em que são definidos deve ser previamente declarada como mostra :

```
LIBRARY ieee  
USE ieee.std_logic_1164.ALL
```

Se objetos do tipo **STD_LOGIC_VECTOR** são usados como números binários em manipulações aritméticas, então uma das duas seguintes instruções de **USE** devem ser incluídas:

Para números aritméticos com sinal:

```
USE ieee.std_logic_signed.ALL;
```

Para números aritméticos sem sinal:

```
USE ieee.std_logic_unsigned.all;
```

Exemplo:

```
LIBRARY ieee  
USE ieee.std_logic_1164.ALL  
SIGNAL x: STD_LOGIC  
SIGNAL y: STD_LOGIC_VECTOR(7 DOWNT0 0)  
x <= 'Z'  
y <= "001-"
```

- **INTEGER:** O tipo predefinido INTEGER define objetos números binários para uso com operadores aritméticos. Um INTEGER usa 32 bits para representar um número com sinal. INTEGER usando poucos bits pode também ser declarado com a chave RANGE.

NÃO é possível realizar operações lógicas sobre inteiros (deve-se realizar a conversão explícita)

Exemplo:

```
SIGNAL x: INTEGER  
SIGNAL y: INTEGER RANGE -64 to 64
```

4.5 .EXPRESSÕES:

Expressões são fórmulas que realizam operações sobre objetos de **mesmo** tipo. As expressões são escritas utilizando sinais e operadores os quais são descritos no item 4.6. As operações descrevem as atribuições dos sinais. A atribuição de valor para um sinal :

- Emprega o delimitador <=
- Ocorrer em regiões de código sequencial ou concorrente.

Exemplo:

```
Sinal_destino <= expressão;
```


Obs: tipo do sinal de destino = tipo do sinal da expressão

O que a seguinte linha de VHDL realiza: $X \leq A$?

Resposta: X assume o valor de A

A informação pode ser originada de:

- Uma expressão
- Um sinal

- Operações lógicas: and, or, nand, nor, xor, not
 - Operações relacionais: =, /=, <, <=, >, >=
 - Operações aritméticas: - (unária), abs
 - Operações aritméticas: +, -
 - Operações aritméticas: *, /
 - Operações aritméticas: mod, rem, **
 - Concatenação
- Menor
- PRIORIDADE
- 

• Observações:

- Operações lógicas são realizadas sobre tipos **bit** e **boolean**.
- Operadores aritméticos trabalham sobre inteiros e reais. Incluindo-se o *package* da Synopsys, por exemplo, pode-se somar vetores de bits.
- Todo tipo físico pode ser multiplicado/dividido por inteiro ou ponto flutuante.
- Concatenação é aplicável sobre caracteres, strings, bits, vetores de bits e arrays.

Exemplos: “ABC” & “xyz” resulta em: “ABCxyz”
 “1001” & “0011” resulta em: “10010011”


4.6 OPERADORES:

Os Operadores são divididos em classe como mostra o Quadro 4.7.

- As classes definem a precedência dos operadores
- Operadores de uma mesma classe possuem mesma precedência.

Quadro 4.7 Operadores separados por classes.

classe	operadores
lógicos	and or nand nor xor xnor
relacionais	= /= < <= > >=
deslocamento	sll srl sla sra rol ror
adição	+ - &
sinal	+ -
multiplicação	* / mod rem
diversos	** abs not

Menor precedência

 Maior precedência

Obs: Operador lógico **NOT** está na classe diversos devido à sua precedência.

4.6.1 Operadores Lógicos:

- Operandos: tipos **bit** e **boolean**
 Vetores unidimensionais compostos de elementos **bit** e **boolean**.
- Os vetores devem ter o mesmo tamanho
- Operação executada entre elementos de mesma posição

operadores	operando L	operando R	retorna
not and or	bit	bit	bit
nand nor xor xnor	boolean	boolean	boolean

Nota: O operador not pertence a classe diversos

- tipo bit: 0,1 tipo boolean: 0 = false 1 = true

L	R	not L	L and R	L nand R	L or R	L nor R	L xor R	L xnor R
1	1	0	1	0	1	0	0	1
1	0	0	0	1	1	0	1	0
0	1	1	0	1	1	0	1	0
0	0	1	0	1	0	1	0	1

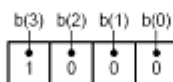
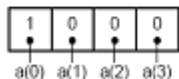
4.6.2 Operadores Relacionais:

- Operadores de igualdade e desigualdade: qualquer tipo
- Operadores de ordenação: tipos escalares(bit, integer, etc)

operadores	operando L	operando R	retorna
= /=	qualquer tipo	mesmo tipo de L	boolean
> < >= <=	qualquer tipo escalar	mesmo tipo de L	boolean

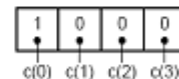
- exemplo de valores tipo bit_vector iguais:

CONSTANT a: BIT_VECTOR(0 TO 3) := "1000"



CONSTANT b: BIT_VECTOR(3 DOWNT0 0) := "1000"

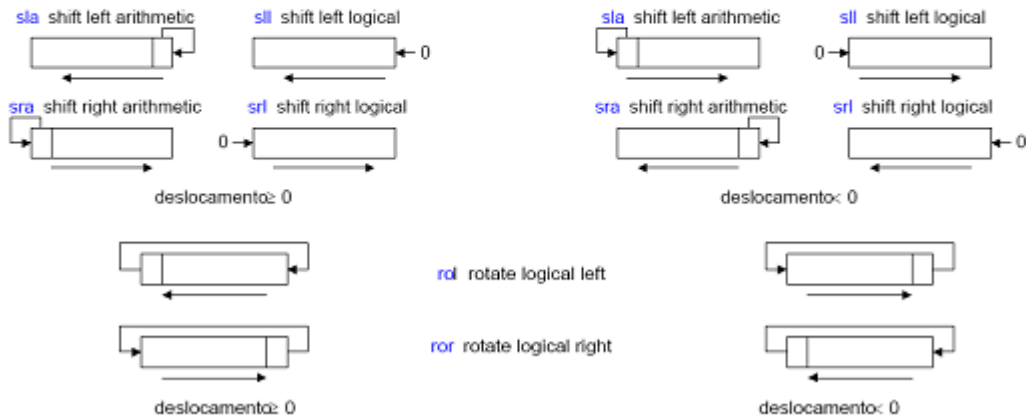
CONSTANT c: BIT_VECTOR(0 TO 3) := "1000"



4.6.3 Operadores de deslocamento e rotação:

- não suportado pela versão VHDL-1987

operadores	operando L	operando R	retorna
sll srl sla sra rol ror	vetor unidimensional com elementos bit ou boolean	integer	o mesmo tipo de L



Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

2.23

4.6.4 Operadores de Adição:

- Adição e subtração: tipo numérico.
- Concatenação: vetor unidiimensional e elementos(mesmo tipo).

operadores	operando L	operando R	retorna
+ -	tipo numérico	o mesmo tipo de L	mesmo tipo
&	vetor	mesmo vetor de L	vetor mesmo tipo
	vetor	elemento	vetor mesmo tipo
	elemento	vetor	vetor mesmo tipo
	elemento	elemento	vetor

• Exemplo:

```
a <= b & c; -- "a" bit_vector 8 elementos, "b", "c" bit_vetor 4 elementos
x <= y & '1'; -- "x" bit_vector 5 elementos, "y" bit_vetor 4 elementos
```

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

2.24

4.6.5 Operadores Aritméticos: identidade, negação

- Operandos: qualquer tipo numérico.
- Possuem o mesmo significado dos operadores matemáticos equivalentes.

operadores	operando R	retorna
+ -	qualquer tipo numérico	mesmo tipo

4.6.6 Operadores Diversos: Absoluto, exponenciação.

- Possuem o mesmo significado dos operadores matemáticos equivalentes.

operadores	operando L	operando R	retorna
abs	qualquer tipo numérico	-	mesmo tipo
**	qualquer tipo integer	integer	mesmo tipo de L
	qualquer tipo real	integer	mesmo tipo de L

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

4.6.7 Operadores Multiplicação:

- Multiplicação e divisão: operandos tipo inteiro e real.
- MOD e REM : operandos tipo inteiro.

operadores	operando L	operando R	retorna
* /	qualquer tipo integer	mesmo tipo	mesmo tipo
	qualquer tipo real	mesmo tipo	mesmo tipo
mod rem	qualquer tipo integer	mesmo tipo	mesmo tipo

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

2.26

Os quatro Quadros seguintes mostram os diversos operadores da VHDL separados por tipos de operação.

Quadro 4.8 – OPERADORES ARITMÉTICOS

Operador	Operação	Tipo do operador da esquerda	Tipo do operador da direita	Tipo do resultado
+	Adição	numérico	mesmo do anterior	igual aos operandos
-	Subtração	numérico	mesmo do anterior	igual aos operandos
&	Concatenação	Array 1 dimensão	mesmo do anterior	igual aos operandos

*	Multiplicação	inteiro ou ponto flutuante	mesmo do anterior	igual aos operandos
		físico	inteiro ou ponto flutuante	físico
		inteiro ou ponto flutuante	físico	físico
/	Divisão	inteiro ou ponto flutuante	mesmo do anterior	igual aos operandos
		físico	inteiro ou ponto flutuante	físico
		físico	físico	inteiro
mod	Módulo	inteiro	mesmo do anterior	igual aos operandos
rem	Resto da Divisão	inteiro	mesmo do anterior	igual aos operandos
**	Potenciação	inteiro ou ponto flutuante	inteiro	Igual ao operador da esquerda
abs	Valor Absoluto		numérico	numérico
not	Negação		Bit, booleana ou array (bit, boolean)	Igual ao operador da direita

Quadro 4.9 OPERAÇÕES DE COMPARAÇÃO:

Operador	Operação	Tipo do operador da esquerda	Tipo do operador da direita	Tipo do resultado
=	Igualdade	qualquer	Mesmo do anterior	boolean
<	Menor do que	Escalar ou array	Mesmo do anterior	boolean
>	Maior do que	Escalar ou array	Mesmo do anterior	boolean
/=	Desigualdade	qualquer	Mesmo do anterior	boolean
>=	Maior do que ou igual a	Escalar ou array	Mesmo do anterior	boolean
<=	Menor do que ou igual a	Escalar ou array	Mesmo do anterior	boolean

QUADRO 4.10 OPERAÇÕES LÓGICAS:

Operador	Operação	Tipo do operador da esquerda	Tipo do operador da direita	Tipo do resultado
and	Lógica E	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean
or	Lógica OR	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean
nand	Lógica E negada	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean
nor	Lógica OR negada	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean
xor	Lógica OR exclusivo	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean
xnor	Lógica OR exclusivo negada	Bit, booleana ou array (bit,boolean)	Mesmo do anterior	boolean

Exemplo:

Operador	exemplo	significado
not	not a	\overline{a}
and	c and b	$c.d$
or	u or v	$u + v$
nand	a nand b	$\overline{a.b}$
nor	c nor d	$\overline{c + d}$
xor	a xor b	$a \oplus b$
xnor	c xnor d	$\overline{c \oplus d}$

Quadro 4.11 OPERAÇÕES DE DESLOCAMENTO E ROTAÇÃO:

Operador	Operação	Tipo do operador da esquerda	Tipo do operador da direita	Tipo do resultado
sll	Desloc. Lógico para esquerda	array de bit ou boolean	inteiro	array de bit ou Boolean
srl	Desloc. Lógico para direita	array de bit ou boolean	Inteiro	array de bit ou Boolean
sla	Desloc. aritmético para esquerda	array de bit ou boolean	Inteiro	array de bit ou Boolean
sra	Desloc. aritmético para direita	array de bit ou boolean	Inteiro	array de bit ou Boolean
rol	Rotação para a esquerda	array de bit ou boolean	Inteiro	array de bit ou Boolean
ror	Rotação para a direita	array de bit ou boolean	inteiro	array de bit ou boolean

4.7 Exemplos de Atribuição de valores de sinais(Retirados do livro de Roberto d'Amore, "Descrição e Síntese de Circuitos Digitais")

Exemplo 1. Tipos INTEGER e REAL

- Operação: valor 11 atribuído a todas as portas de saída

```

9 ARCHITECTURE teste OF int_real IS
10   CONSTANT i1 : INTEGER := 11;           -- valor 11, base 10
11   CONSTANT i2 : INTEGER := 10#11#;       -- valor 11, base 10
12   CONSTANT i3 : INTEGER := 2#01011#;     -- valor 11, base 2
13   CONSTANT i4 : INTEGER := 2#01_01_1#;   -- valor 11, base 2
14   CONSTANT i5 : INTEGER := 5#21#;        -- valor 11, base 5
15   CONSTANT i6 : NATURAL := 8#13#;        -- valor 11, base 8
16   CONSTANT i7 : POSITIVE := 16#B#;       -- valor 11, base 16
17
18   CONSTANT r1 : REAL := 11.0;             -- valor 11, base 10
19   CONSTANT r2 : REAL := 1.1E01;          -- valor 11, base 10 formato nn.nExx
20   CONSTANT r3 : REAL := 2#01011.0#;     -- valor 11, base 2
21   CONSTANT r4 : REAL := 8#1.3#E01;      -- valor 11, base 8 formato nn.nExx
22   CONSTANT r5 : REAL := 16#B.0#;        -- valor 11, base 16
23
24 BEGIN
25   cil <= i1; ci2 <= i2; ci3 <= i3; ci4 <= i4; ci5 <= i5; ci6 <= i6; ci7 <= i7;
26   crl <= r1; cr2 <= r2; cr3 <= r3; cr4 <= r5;
END teste;

```

Exemplo 2: Tipos BIT_VECTOR

- Operação: valor 1011 atribuído a toads as portas de saída
- Diferentes bases de representação
 - Tipos integer, real : formato → 16#B# 16#B.0#
 - Tipos bit_vector: formato → X"B"
- Linha 12: caracter _ em 01_0_11 melhora leitura do valor
- Linhas 14 e 15: valor definido para uma parte real: DOWNT0

```

5 ARCHITECTURE teste OF std_a IS
6   CONSTANT c1 : BIT_VECTOR(4 DOWNT0 0) := "01011"; -- constante
7   CONSTANT zero : BIT := '0';
8   CONSTANT um : BIT := '1';
9 BEGIN
10   s1 <= c1; -- valor atraves de constante
11   s2 <= "01011"; -- valor (01011) direto - base binaria
12   s3 <= B"01_0_11"; -- valor (01011) direto - base binaria com separadores
13   s4 <= '0' & X"B"; -- bit (0) concatenado com valor hexadecimal (1011)
14   s5(4 DOWNT0 3) <= "01"; -- valor (01), parte do vetor
15   s5(2 DOWNT0 0) <= zero & um & um; -- valor (010), parte com concatenacao
16 END teste;

```

Exemplo 3: Atribuição de valores em sinais, tipos BIT_VECTOR agregados

- S2 S4 → atribuição de valor: notação posicional.
- S3 S5 → atribuição de valor: associação de nomes.

```
1 ENTITY std_al IS
2   PORT(s2, s3, s4, s5 : OUT BIT_VECTOR(4 DOWNTO 0));
3 END std_al;
4
5 ARCHITECTURE teste OF std_al IS
6   CONSTANT zero : BIT := '0';
7   CONSTANT um   : BIT := '1';
8 BEGIN
9   s2 <= ('0','0','0','1','0');      -- 00010, agregado notacao posicional
10  s3 <= (1=>'1', 0=>'1', OTHERS=>'0'); -- 00011, agregado associacao por nomes
11  s4 <= (zero, '0', um OR '0', '0', '0'); -- 00100, agregado com operacoes
12  s5 <= (4 DOWNTO 3 =>'0', 1=>'0', OTHERS=>'1'); -- 00101, agregado faixa discreta
13 END teste;
```

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

2.30

Exemplo 4: Atribuição de valores: Expressões com operadores lógicos.

- Linhas 8 e 14: comentário no código
- Linha 15: x and y equivale a not(a and b) → ordem das operações é importante.

```
1 ENTITY std_xal IS
2   PORT( a, b, c, d : IN BIT;
3         x1, x2, x3, x4, x5 : OUT BIT);
4 END std_xal;
5
6 ARCHITECTURE exemplo OF std_xal IS
7 BEGIN
8   x1 <= a OR NOT b;      -- Certo: operador NOT tem precedencia
9                           --      mais elevada
10  x2 <= a AND b AND c;   -- Certo: operadores iguais
11  -- x3 <= a AND b OR c;  -- Errado: expressao ambigua x3=(a.b)+c
12                           --      ou x3=a.(b+c) ?
13  x3 <= (a AND b) OR c;  -- Certo: empregando parentesis
14  -- x4 <= a AND b OR c AND d; -- Errado: expressao ambigua, operadores
15                           --      com mesma precedecia
16  x4 <= (a AND b) OR (c AND d); -- Certo: x4 = a.b + c.d
17  -- x5 <= a NAND b NAND c;  -- Errado: operadores com negacao
18                           --      necessitam parentesis
19  x5 <= (a NAND b) NAND c;  -- Certo: operador com negacao entre
20                           --      parentesis
21 END exemplo;
```

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 2 - revisão 2.4

2.31

Exemplo 5: Operadores classe adição.

- Linhas 10 e 11: Operação de concatenação de dois vetores (tipos bit_vector)
- Linha 12: soma de dois tipos inteiros

```
1 ENTITY std_xc IS
2   PORT (bv_a, bv_b : IN  BIT VECTOR(1 DOWNT0 0);
3         int_a, int_b : IN  INTEGER RANGE -32 TO 31;
4         bv_c, bc_d : OUT BIT VECTOR(3 DOWNT0 0);
5         int_c      : OUT INTEGER RANGE -64 TO 63);
6 END std_xc;
7
8 ARCHITECTURE teste OF std_xc IS
9 BEGIN
10  bv_c <= bv_a & bv_b;
11  bc_d <= bv_a & '1' & '0';
12  int_c <= -int_a +int_b;
13 END teste;
```

5. Comandos em VHDL:

Em VHDL uma descrição estrutural de uma arquitetura apresenta tanto as instruções como comandos concorrentes entre si, ou seja, a ordem de apresentação das instruções ou dos comandos é irrelevante. Todos (comandos ou instruções) são executados paralelamente. Em uma descrição de arquitetura por fluxo de dados ou comportamental, podem ser utilizados alguns comandos com o objetivo de facilitar a descrição de circuitos mais complexos. Esses comandos podem ser concorrentes, e alguns deles se contidos em regiões específicas de códigos, como dentro de processos, são avaliados na sequência em que são apresentados. Neste capítulo serão apresentados alguns comandos básicos de VHDL, assim como alguns conceitos necessários para a utilização desses comandos, os quais são:

- Declaração e Atribuição à sinais
- Atribuição à variáveis
- Conceito de Atributo

■ DECLARAÇÃO E ATRIBUIÇÃO À SINAIS:

A Declaração de sinal é usada como meio de transmissão de informação dentro da arquitetura. As conexões internas entre os componentes são descritas por meio de sinais. A atribuição de um valor a um sinal pode ocorrer tanto em uma região de código concorrente como em regiões de código seqüencial.

Os sinais são declarados após a declaração da arquitetura entre a palavra chave IS e antes de BEGIN da seguinte maneira:

ARCHITECTURE tipo_arquitetura OF nome_entidade IS
SIGNAL nome_sinal1 : tipo_do_sinal;
BEGIN

Uma listagem em VHDL poderia ter a seguinte declaração:

ARCHITECTURE estrutural OF teste IS
SIGNAL X1,X2: STD_LOGIC;
SIGNAL X13: STD_LOGIC_VECTOR(7DOWNT0 0);;
BEGIN

A Declaração de Atribuição do sinal descreve como o dado flui do lado direito do operador (**<=>**) para o lado esquerdo como a seguir

<code><nome_do_sinal > <= <expressão>;</code>
--

Para circuito da Figura 5.1, X deve ser declarado como sinal com o tipo BIT ou STD_LOGIC, como mostra a listagem VHDL seguinte

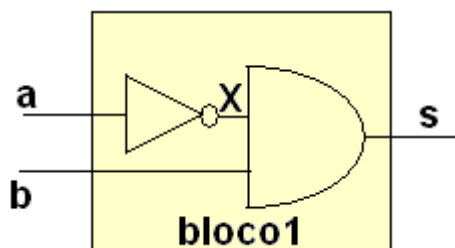


Figura 5.1 Circuito bloco1 com sinal X interno

ENTITY bloco1 IS
PORT(a, b : IN :STD_LOGIC;
s : OUT STD_LOGIC VECTOR);
END bloco1;
ARCHITECTURE estrutural OF bloco1 IS
SIGNAL X: STD_LOGIC;
BEGIN
s <= not a and b;
END estrutural;

A listagem VHDL a seguir mostra uma atribuição de sinal dentro de um processo. (Exemplo retirado de Terroso, A. R. “Dispositivo Lógicos Programável (FPGA) e Linguagem de Descrição de Hardware

(VHDL)"). Na linha 9 o sinal a vale '0', após 20ns passa para '1' e após 40ns volta a ser '0'. A linha 10 e 11 são operações lógicas e a linha 12 o processo é semelhante ao da linha 9 porém, o sinal d recebe valores inteiros.

1	ACHITECTURE comportamental OF exemplo IS
2	SIGNAL a, b, c: BIT;
3	SIGNAL d: INTEGER;
4	BEGIN
5	PROCESS (a, b, c, d)
6	VARIABLE k, m, n q BIT
7	
8	BEGIN
9	a <= '0' , '1' AFTER 20ns, '0' AFTER 40ns;
10	b <= NOT k ;
11	c <= ((k and m) XOR (n nand q));
12	d <= 3,5 AFTER 20ns, 7 AFTER 40ns, 9 AFTER 60ns;
13	END PROCESS;
14	END comportamental;

OBSERVAÇÕES: Os sinais que foram atribuídos valores dentro do processo poderão ser utilizados em outro processo, mas não pode receber atribuição de valores.

■ DECLARAÇÃO E ATRIBUIÇÃO À VARIÁVEIS:

Diferente do sinal, a variável só pode ser utilizada internamente a um processo em que foi criada e o valor é passado à ela através de := sua atualização é imediata enquanto a atualização do sinal só ocorre no final do processo. A função a variável dentro do processo é de intermediar valores.

A variável é declarada depois do comando PROCESS.e antes do comando BEGIN e tem a seguinte sintaxe:

VARIABLE<nome_da_variável > <: <expressão>;

A listagem VHDL a seguir (EXEMPLO 1) apresenta as etapas de simulação de um programa VHDL. Acompanhando horizontalmente cada bloco pode-se observar as atualizações das variáveis. E, analisando a listagem do EXEMPLO2 pode-se verificar a diferença entre sinal e variável(Exemplos retirados de Terroso, A. R. “Dispositivo Lógicos Programável (FPGA) e Linguagem de Descrição de Hardware (VHDL)”).

EXEMPLO 1

D <= 2; process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1	process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1	process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1
process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1	process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1	process (C, D) variable Av, Bv, Ev : integer := 0; begin Av := 2; Bv := Av + C; Av := D + 1; Ev := Av * 2; A <= Av; B <= Bv; E <= Ev; end process; A = 1 B = 1 C = 1 D = 2 E = 1
Av = 2 Bv = 3 Ev = 0	Av = 3 Bv = 3 Ev = 0	Av = 3 Bv = 3 Ev = 6

EXEMPLO 2

D <= 2; process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 B = 1 C = 1 D = 1 E = 1	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 B = 1 C = 1 D = 2 E = 1	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 A <= 2 B = 1 C = 1 D = 2 E = 1	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 A <= 2 B = 1 B <= A + C C = 1 D = 2 E = 1
process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 A <= D + 1 B = 1 B <= A + C C = 1 D = 2 E = 1	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 A <= D + 1 B = 1 B <= A + C C = 1 D = 2 E = 1 E <= A * 2;	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 1 A <= 3 B = 1 B <= 2 C = 1 D = 2 E = 1 E <= 2;	process (C, D) begin A <= 2; B <= A + C; A <= D + 1; E <= A * 2; end process; A = 3 B = 2 e não 3 C = 1 D = 2 E = 2 e não 6

■ ESCOLHA ENTRE SINAL OU VARIÁVEIS:

A escolha entre utilizar sinais (expressões concorrentes) variáveis, as quais são usadas em processos sequencias, vai depender do que se implementar. Por exemplo, nas atribuições a seguir a Atribuição1 faz com que os valores de a e b sejam permutados e a Atribuição2 faz com que a e b assumam o valor anterior de b.

Atribuição1:	Atribuição2:
a <= b; b <= a;	a := b; b := a;

■ CONCEITO DE ATRIBUTO:

Atributos são informações adicionais associadas a objetos, como por exemplo, a sinais. Os atributos possibilitam a verificação de transições de sinal e o modelamento atrasos. A sintaxe de atributo é apresentada a seguir:

' nome_atributo;

A Tabela 5.1 apresenta um resumo de sinais de atributo e SUS funções

Tabela 5.1

ATRIBUTO	FUNÇÃO
nome_do_sinal 'EVENT	Verdadeiro se ocorreu uma troca de valor do sinal no ciclo corrente de simulação; e falso, caso contrário
nome_do_sinal 'ACTIVE	Verdadeiro se foi atribuído um valor durante o ciclo corrente de simulação; e falso, caso contrário
nome_do_sinal 'TRANSACTION	Retorna um sinal tipo BIT complementado em relação ao anterior a cada transição do sinal
nome_do_sinal 'LAST_EVENT	Tempo decorrido desde a ultima troca de valor do sinal
nome_do_sinal 'LAST_VALUE	Retorna o valor do sinal antes do último evento
nome_do_sinal 'LAST_ACTIVE	Retorna o intervalo de tempo desde que a última transição tenha ocorrido no sinal
nome_do_sinal 'DELAYED(T)	Novo sinal equivalente ao sinal atrasado T unidades de tempo. O valor de T é opcional, mas o default é T=0.
nome_do_sinal 'STABLE(T)	Retorna um sinal tipo “booleano” verdadeiro se não ocorreu nenhuma troca de valor durante um período T; caso contrário retorna um valor falso
nome_do_sinal 'QUIET (T)	Retorna um sinal tipo “booleano” verdadeiro se não ocorreu nenhuma transição durante um período T; caso contrário retorna um valor falso

A Figura 5.2 apresenta os termos relativos à “active”, “quiet” e “event”. A condição “active” é empregada acaso seja atribuído um novo valor a um sinal, durante um ciclo de simulação, mesmo que o novo valor seja igual ao anterior. A condição “quiet” é oposta à “active”. E a condição “event” representa uma mudança no valor a ser assumido.

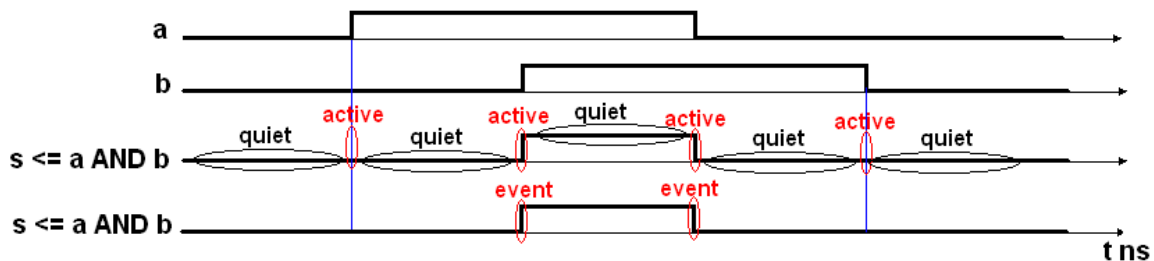


Figura 5.2 condições “active”, “quiet” e “event” de um sinal.

5.1 COMANDOS CONCORRENTES:

5.1.1 Construção “WHEN ELSE”:

Permite associar um valor a um sinal se algumas condições são satisfeitas. Uma lista de opções é apresentada estabelecendo qual valor de uma expressão deve ser transferido a um sinal de destino. A primeira condição verdadeira define a expressão transferida. O Quadro 5.1 mostra o formato utilizado para este comando. E no Quadro 5.2 é apresentado o código completo para o circuito da Figura 5.1. A descrição possui um estilo mais próximo ao comportamento do circuito.

Quadro 5.1

```

sinal_destino <= expressao_a WHEN condicao_1 ELSE
                expressao_b WHEN condicao_2 ELSE
                expressao_c;

```

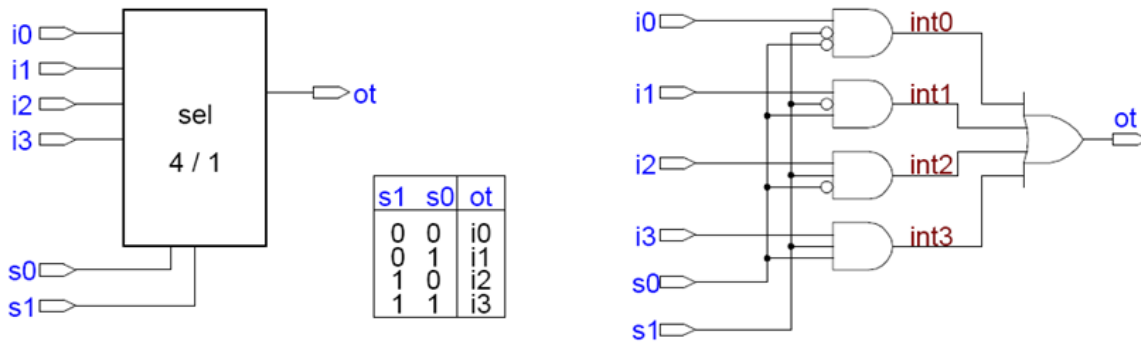



Figura 5.1 Circuito de um Mux 4x1

Quadro 5.2

```

1 ENTITY mux_1 IS
2   PORT (i0, i1, i2, i3      : IN  BIT;
3         s0, s1              : IN  BIT;
4         ot                  : OUT BIT);
5 END mux_1;
6
7 ARCHITECTURE teste OF mux_1 IS
8 BEGIN
9   ot <= i0 WHEN s1= '0' AND s0='0' ELSE
10      i1 WHEN s1= '0' AND s0='1' ELSE
11      i2 WHEN s1= '1' AND s0='0' ELSE
12      i3;
13 END teste;

```

5.1.2 CONSTRUÇÃO “WITH SELECT”:

Transfere um valor a um sinal de destino segundo um relação de opções. Todas as condições de seleção devem ser consideradas e elas devem ser mutuamente exclusivas. Não existe uma prioridade como na construção “WHEN ELSE”. Essa construção segue o formato mostrado no Quadro 5.3. As opções podem ser agrupadas através do delimitador “|”, equivalente a uma condição “ou”. E também pode-se utilizar as palavras reservadas “TO” e “DOWNTO” para delimitar uma faixa de condições de um tipo escalar. A palavra “OTHERS” é válida como última alternativa para englobar as condições restantes.

Quadro 5.3

```
WITH expressao_escolha SELECT      -- expressao_escolha =
  sinal_dest <= expr_a WHEN condicao_1,      -- condicao_1
               expr_b WHEN condicao_2,      -- condicao_2
               expr_c WHEN condicao_3 | condicao_4,      -- condicao_3 ou condicao_4
               expr_d WHEN condicao_5 TO condicao_7,      -- condicao_5 ate condicao_7
               expr_e WHEN OTHERS;          -- condicoes restantes
```

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 3 - revisão 1.7

3.11

O Quadro 5.4 mostra a descrição completa utilizando o comando **“WITH SELECT”** para o circuito da da Figura 5.1 cuja expressão de escolha é:

Sinal **sel** = **s1** e **s0** concatenados

Quadro 5.4

```
1 ENTITY mux_9 IS
2   PORT (i0, i1, i2, i3 : IN BIT;
3         s0, s1         : IN BIT;
4         ot             : OUT BIT);
5 END mux_9;
6
7 ARCHITECTURE teste OF mux_9 IS
8   SIGNAL sel : BIT_VECTOR (1 DOWNTO 0);
9 BEGIN
10  sel <= s1 & s0;
11  WITH sel SELECT
12    ot <= i0 WHEN "00",
13         i1 WHEN "01",
14         i2 WHEN "10",
15         i3 WHEN "11";
16 END teste;
```

Roberto d'Amore - VHDL: Descrição e Síntese de Circuitos Digitais - capítulo 3 - revisão 1.7

3.15

Comparação entre as construções “WHEN ELSE” e “WITH SELECT”:

- Construção **WHEN ELSE**:

- ordem das condições indica a prioridade
- não é necessário apresentar todas condições

```
sinal_destino <= expressao_a WHEN condicao_1 ELSE -- condicao_1 = verdadeira
                    expressao_b WHEN condicao_2 ELSE -- condicao_2 = verdadeira
                    expressao_c;                  -- nenhuma condicao verd.
```

- Construção **WITH SELECT**:

- todas condições têm igual prioridade
- é necessário apresentar todas condições

```
WITH expressao_escolha SELECT -- expressao_escolha =
sinal_destino <= expressao_a WHEN condicao_1, -- condicao_1
                    expressao_b WHEN condicao_2, -- condicao_2
                    expressao_e WHEN OTHERS; -- condicoes restantes
```

5.1.3 COMANDO PROCESS:

O comando **PROCESS** delimita regiões de código seqüencial. É dividido em duas regiões da primeira é a parte da declaração do processo e a outra a descrição do mesmo. Como mostra o Quadro 5.5 a seguir

Quadro 5.5

```
process_name: PROCESS( sensitivity_list_signal_1, ... )
BEGIN
    -- comandos do processo
END PROCESS process_name;
```

Antes da palavra **PROCESS** pode aparecer um rótulo com o nome do processo, o qual é opcional. Logo após a palavra “**PROCESS**” pode-se definir uma lista de sensibilidade que causam a execução do processo, como mostrado na descrição do Quadro 5.5. A alteração de um valor de um dos sinais da lista de sensibilidade ativa o

processo causando a execução dos comandos seqüenciais segundo a ordem de apresentação.

5.2 COMANDOS SEQÜENCIAIS:

Os comandos seqüenciais ficam contidos em regiões onde o código é executado sequencialmente, como dentro de processos, delimitados pelo comando **PROCESS**.

Observação: Esses comandos não podem ser utilizados fora da região delimitada pelo comando **PROCESS**.

- atribuição de variáveis
- if,
- case,
- for,
- while,
- wait

5.2.1 Atribuição de variáveis:

- Variáveis **não** passam valores fora do processo na qual foram declaradas, são locais. Elas sequer existem fora de um processo.
- As atribuições são seqüenciais, ou seja, a ordem delas importa.

O formato da atribuição de uma variável é mostado a seguir:

variable_assignment_statement ::= target := expression ;
target ::= name aggregate

5.2.2 - Estrutura IF:

É utilizada em situações em que a execução de uma sequência de comandos depende de uma ou mais condições. A forma mais simples de utilização é através da estrutura **if-then-end-if** como apresentado a seguir:

```
IF chave = 1 THEN
    saida1 := valor_saida;
END IF;
```

Onde a condição (*chave* = 1) é testada para permitir a operação necessária. Se a condição for verdadeira, *valor_saida* é atribuído à variável *saida1*, caso não nada é feito.

Em alguns casos é desejado realizar-se uma operação (ou sequência de operações) para quando a condição for verdadeira e outra quando a condição for falsa.

Ver exemplo de uma estrutura **if-then-else-end if**:

```
IF chave = 1 THEN
    saida1 := valor_saida;
ELSE
    saida1 := 255;
    registrador := valor_saida;
END IF;
```

Este caso é idêntico ao caso anterior só que quando a condição *chave* = 1 for falsa são executadas as linhas *saida1* := 255 e *registrador* := *valor_saida*.

Uma outra variação desta estrutura é permitida através da utilização da palavra chave **elsif**, que funciona como uma nova estrutura de comparação dentro da estrutura **if** corrente. São possíveis tantos **elsif** quantos forem necessários para refinar as comparações. A seguir tem-se um exemplo:

```
IF painel = OFF THEN
    led1 := OFF;
```

```

    led2 := OFF;
    ELSIF leitura = 1 OR leitura = 3 THEN
        led1 := ON;
        led2 := OFF;
    ELSIF leitura = 2 or leitura = 4 THEN
        led1 := OFF;
        led2 := ON;
    END IF;

```

onde os valores de saída *led1* e *led2* são vinculados aos valores da variável de entrada *leitura*. Se *leitura* for igual a 1 ou 3 (através do uso do operador **or**) a saída *led1* é ativada. Se *leitura* for igual a 2 ou 4 a saída *led2* é ativada. Isto é claro somente se a condição *painel* = OFF for falsa, ou seja a chave de entrada do painel estiver ligada.

IMPORTANTE:

- teste de borda de subida: `if clock'event and clock='1' then ...`
- teste de borda de descida: `if clock'event and clock='0' then ...`
- a seqüência na qual estão definidos os 'ifs' implica na prioridade das ações

5.2.3. - Estrutura CASE:

Uma outra estrutura de comparação de valores para seleção de operações é a estrutura **case-is-when-endcase**. Esta estrutura é mais utilizada para aplicações onde uma determinada variável pode assumir um número limitado de valores, cada qual associado a um conjunto de operações. A seguir tem-se o exemplo de um multiplexador implementado com esta estrutura:

```

CASE seletor IS
    WHEN 0 =>
        saída <= entrada0;
    WHEN 1 =>

```

```
        saída <= entrada1;  
WHEN 2 =>  
        saída <= entrada2;  
WHEN 3 =>  
        saída <= entrada3;  
END CASE;
```

Para se implementar a função **OU** em estruturas **CASE** usa-se o caracter “|”.

Como exemplo apresenta-se uma forma alternativa de implementar a função de acionamento das saídas *led1* e *led2* de acordo com a variável *leitura*

```
CASE leitura IS  
    WHEN 1 | 3 => -- quando 1 ou 3  
        led1 := ON;  
        led2 := OFF;  
    WHEN 2 | 4 => quando 2 ou 4  
        led1 := OFF;  
        led2 := ON  
End CASE;
```

Para a área de sistemas digitais a grande utilidade da estrutura **CASE** é na implementação de máquinas de estado. No exemplo a seguir:

```
CASE estado IS  
    WHEN estado_A =>  
        saída <= '0';  
        IF entrada = '1' THEN estado := estado_B;  
        END IF;  
    WHEN estado_B =>  
        IF entrada = '0' THEN estado := estado_C;  
        END IF;  
    WHEN estado_C =>  
        estado := estado_A;  
        saída <= '1';  
END CASE;
```

No exemplo anterior está implementada máquina de estados da Figura 5.2. Apesar de não indicado, a estrutura apresentada deveria estar dentro de um processo sincronizado com o clock, o que é normalmente subentendido em sistemas de máquinas de estados.

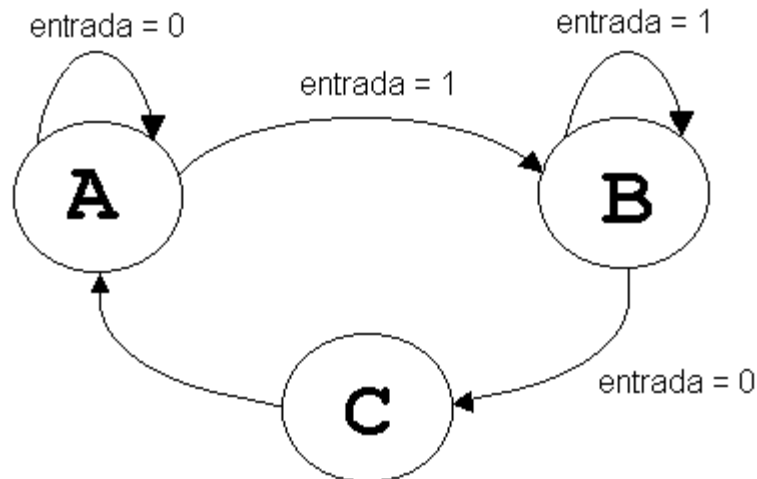


Figura 5.2 : Máquina de estados implementada no exemplo

5.2.4 - Comando WAIT:

A operação ou comando de espera (**wait**) tem como finalidade suspender a execução da sequência corrente até que uma determinada condição ocorra. A sua utilização é feita a partir da palavra-chave **wait**. Este comando pode ser empregado para se esperar simplesmente a mudança de estado de um sinal ou aguardar por uma variação específica.

■ Espera por uma Variação Qualquer:

O modo mais simples de utilização do comando **wait** é para se aguardar uma mudança qualquer de estado em um determinado sinal.

Isto é feito com a palavra-chave **on** seguido pelo sinal que se deseja monitorar. Por exemplo, na execução de uma descrição em VHDL que contenha:

```
...  
wait on a;  
...
```

a sequência de operação irá ser suspensa na linha apresentada até que o sinal *a* mude de valor, quando então irá seguir em execução normal. É possível a combinação de mais de um sinal de entrada como no caso abaixo:

```
...  
wait on a, b;  
...
```

onde a sequência de operação é suspensa até que haja uma variação em qualquer dos sinais *a* ou *b*.

■ Espera por uma Variação Específica:

Para a programação da sensibilidade para um único tipo de variação deve-se utilizar a palavra-chave **until** seguida pelo sinal e a condição que se deseja aguardar. Este formato de comando de espera é especialmente importante para a síntese de componentes sensíveis a um tipo de borda, como latches e flip-flops.

O exemplo abaixo ilustra a utilização deste comando para a implementação de um componente sensível a borda positiva do sinal *clk*:

```
...  
wait until clk = '1';  
...
```

■ Espera por Nível:

Em alguns casos, a função que se deseja sintetizar deve ser sensível não a borda, mas sim a um nível de sinal. Para tanto deve-se especificar além dos parâmetros vistos no item anterior a informação de tempo de persistência da condição. Agrega-se então à estrutura de comando de espera visto anteriormente a palavra-chave **for** seguida pelo tempo que a condição deve ser mantida verdadeira para que o programa saia do modo suspenso. Por exemplo:

```
...  
wait until int_in = '1' for 1ms;  
...
```

implementa a espera de um sinal de nível lógico '1' com duração de 1ms na porta *int_in* para dar continuidade à sequência de operação.

5.2.5 COMPARAÇÃO ENTRE CONSTRUÇÃO IF-ELSE e CASE-WHEN:

Sempre que possível utiliza a construção CASE- WHEN ao invés da IF ELSE, pois as restrições impostas pela construção CASE WHEN evita que circuitos sequenciais sejam criados sem necessidade como no caso da utilização da construção IF- ELSE , onde o esquecimento pelo projetista de incluir alguma condição leva a criação de um elemento de memória, na compilação, para manter a condição anterior.

6. Bibliografia:

6.1 Livros:

D'amore, R., "VHDL Descrição e Síntese de Circuitos Dogitais", Ed. LTC

Hamblen, J. O. & Furman, M. D. "*Rapid Prototyping of Digital Systems- A Tutorial Aproach*". Kluwer Academic Publishers. 2nd. Edition, 2001.

Palnitkar, Samir. "*Verilog Hdl: A Guide to Digital Design and Synthesis*". SunSoft Press, 1996

Perry, D. L. "*VHDL: Programming by Example*", McGraw-Hill 4th Ed., 2002.

Terroso, A. R. "Dispositivo Lógicos Programável (FPGA) e Linguagem de Descrição de Hardware(VHDL)". VII-Semana da Engenharia PUCRS- Minicurso 2, Porto Alegre-RS, 1998.

Uyemura, J. P., " *Sistemas Digitais- Uma Abordagem Integrada*". Ed. Pioneira Thomson Learning, 2002

6.2 Sites Consultados:

<http://pt.wikipedia.org/wiki/VHDL>

<http://ee.ucd.ie/~rreilly/DE%20Website/Lectures/DE-19-VLSI%20Design%20Methodology.ppt>

http://www.ene.unb.br/~juliana/cursos/sistdigital1/vhdl_ronhuse.pdf

<http://www.dimap.ufrn.br/~ivan/orgl/vhdl.PDF>

[http://209.85.165.104/search?q=cache:hYlrOFzhoroJ:www.ee.pucrs.br/~vargas/SisC/poligrafo-vhdl.doc+ANDERSON+ROYES+TERROSO+LINGUAGEM+DE+DESCRIPC3%87%C3%83O+DE+HARDWARE+\(VHDL\)&hl=pt-BR&ct=clnk&cd=5&gl=br](http://209.85.165.104/search?q=cache:hYlrOFzhoroJ:www.ee.pucrs.br/~vargas/SisC/poligrafo-vhdl.doc+ANDERSON+ROYES+TERROSO+LINGUAGEM+DE+DESCRIPC3%87%C3%83O+DE+HARDWARE+(VHDL)&hl=pt-BR&ct=clnk&cd=5&gl=br)

<http://www.angelfire.com/in/rajesh52/verilogvhdl.html>

www.model.com **Comparison of VHDL, Verilog and SystemVerilog**
Stephen Bailey