APOSTILA DE

VHDL

Ronaldo Hüsemann

Professor das Disciplinas:
Técnicas Digitais (2000)
Sistemas Digitais (2001)
Microprocessadores II (2002)

Departamento de Engenharia Elétrica
Universidade Federal do Rio Grande do Sul

Histórico

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.

Em 1992 foram propostas várias alterações para a norma, sendo que 1993 foi lançada a versão revisada, mais flexível e com novos recursos. Esta norma revisada, chamada VHDL-93, é a hoje a mais amplamente utilizada e por isso a maioria dos exemplos aqui apresentados são baseados na mesma.

É objetivo desta apostila auxiliar estudantes e pesquisadores que necessitem aprender os preceitos da linguagem VHDL para desenvolvimento e simulação de projetos afins à área de eletrônica e sistemas digitais, mas acima de tudo procurando-se apresentar uma visão geral da linguagem.

Porto Alegre, maio de 2001

Ronaldo Hüsemann

1- Elementos Básicos em VHDL

1.1 - Comentários

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

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

1.2 - Identificadores

Os identificadores são usados para se atribuir nomes a sinais ou processos. Um identificador básico em VHDL:

-pode conter letras do alfabeto ('**A**' a '**Z**' e '**a**' a '**z**'), digitos decimais ('**0**' a '**9**') e o caracter underline ('**_**');

-precisa começar com uma letra do alfabeto;

-não pode terminar com um caracter underline;

-não pode conter dois caracteres underline em sequência.

Alguns exemplos de identificadores:

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 '\'. Por exemplo:

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

1.3 - Números

VHDL define dois tipos básicos de números: inteiros e reais. Um número inteiro consiste de um número sem ponto decimal. Exemplos de números inteiros:

23 0 146

Os números reais, por sua vez, permitem a representação de números fracionários. Por exemplo:

23.1 0.2 34.118

Pode-se trabalhar com notação exponencial, desde que o número seja seguido pela letra '**E**' ou '**e**'. Exemplos:

3E2 18E-6 3.229e9

Além disso pode-se trabalhar em VHDL com bases diferentes da decimal, bastando para isto indicar a base seguido pelo número entre '**#**'. Para bases maiores que 10, deve-se utilizar as letras '**A**' a '**F**' (ou '**a**' a '**f**'), indicando as bases 11 a 15. Não são permitidas bases maiores que 16. Como exemplo, tem-se o número 253 representado em diversas bases distintas:

2#11111101# 16#FD# 16#0fd# 8#0375#

A fim de facilitar a interpretação de números longos é permitido em VHDL utilizar-se o caracter underline ('**_**') entre dígitos. Isto não altera o valor representado pelo número, apenas facilita a sua identificação visual. Por exemplo:

123_456 3.141_592 2#111_1100_0000_0000#

1.4 - Caracteres

A representação de caracteres em VHDL é permitida pelo uso de aspas simples como nos exemplos:

'A' 'b' ';' '0'

1.5 - Strings

As strings, em VHDL, são definidas entre aspas duplas e representam uma sequência de caracteres, mas precisam estar inteiramente definidas em uma linha. Exemplos de strings:

"Teste de operacao"
"2000 iteracoes"

A utilização de aspas dentro de um string é permitida duplicando-se o caracter de aspas ("). Por exemplo:

"O caracter "" faz parte da string"

Se for necessário definir-se uma string cujo comprimento seja maior que o de uma linha deve-se utilizar o operador de concatenação ('**&**') para unir as substrings em uma só. Exemplo:

"Esta string sera' interpretada como uma unica string, "
& "pois foi utilizado o operador de concatenacao"

1.6 - Strings de Bit

Assim como strings de caracteres, é possível a definição de strings de bits ou valores numéricos especiais (como representados por outras bases). O especificador de base pode ser:

- **-B** para binário
- **-O** para octal
- **-X** para hexadecimal

A seguir alguns exemplos de strings de bit em base binária:

B”010001” b”1001_1101_1100”

base octal:

o”372” O"740"

e base hexadecimal:

X”D4” x“0F2”

2 - Tipos de Dados

2.1 - Constantes e Variáveis

Um objeto é um modelo em VHDL que pode conter um valor de um tipo específico. São 4 classes de objetos: constantes, variáveis, sinais e arquivos.

2.1.1 - Constantes e Variáveis

Ambos os tipos constantes e variáveis devem ser definidas antes de serem utilizadas na descrição VHDL. Os elementos do tipo constante não podem ser sobrescritos, enquanto que os do tipo variável podem ser alterados a qualquer momento. A declaração de uma constante é feita através do uso da palavra-chave **constant**, como nos exemplos:

```
constant  valor_de_pi : real := 3.141592653;
constant  ativado : bit := 1;
constant  atraso : time := 5ns;
```

A declaração de variáveis segue a mesma estrutura, utilizando a palavra-chave **variable**. Uma variável pode ser iniciada com algum valor prévio na sua declaração, o qual irá manter até que seja feita a primeira escrita. A seguir, alguns exemplos:

```
variable  numero_de_contagens : integer := 50;
variable  liga_saida1 : bit := '0';
variable  tempo_medida : time := 0ns;
```

As atribuições em VHDL para constantes e variáveis são feitas com o operador "**:=**", como já pode ser visto nos exemplos anteriores.

2.1.2 - Sinais

Um sinal (**signal**) é utilizado para interligação de blocos em VHDL, funcionando de maneira similar a uma variável. O sinal traz porém a capacidade de permitir a ligação (ao ser ligado a uma porta de entrada e saída) ou troca de valores (quando opera com variáveis internas) entre blocos funcionais distintos.

As atribuições de sinais em VHDL é feita normalmente com o operador "<=", por ser tratado analogamente a um pino de entrada e saída. Exemplos de sua utilização são apresentados no capítulo "Programação em VHDL" (item 5.3).

Na atribuição de sinais pode-se adicionalmente gerar de um atraso programado através da palavra-chave **after**. No exemplo:

```
x <= not y after 8ns;
```

o sinal x recebe o valor negado de y após um atraso de 8ns. Isto é especialmente importante para se simular atrasos de portas lógicas.

É possível desta forma até mesmo construir-se um sinal como forma de onda ou pulso variável através do adequado uso dos comandos de atribuição de sinais. Por exemplo, a sentença abaixo gera um pulso positivo com atraso de propagação de 5ns e largura de 10ns no sinal de saída pinout:

pinout <= '1' **after** 5ns, '0' **after** 15ns;

como a representação feita na figura abaixo:

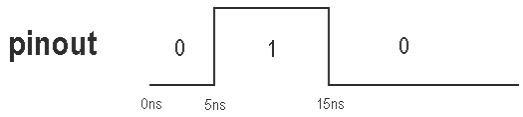

Figura 2.1 : Sinal gerado em *pinout*

A estrutura flexível do VHDL permite a atribuição de valores em processos vinculada a uma ou mais condições, ou seja a atribuição só se efetivará se uma condição (ou conjunto de condições) for verdadeira. Para se realizar esta operação em uma atribuição deve-se utilizar a palavra-chave **when**, seguida da condição que se deseja verificar para a validade da atribuição. Por exemplo, abaixo tem-se listada a descrição de um multiplexador de 4 entradas utilizando-se este recurso:

```
saida   <=   entrada0 when a = '0' and b = '0',
             entrada1 when a = '0' and b = '1',
             entrada2 when a = '1' and b = '0',
             entrada3;
```

O sinal de saída irá receber um dos pinos de entrada de acordo com a condição dos sinais de seleção *a* e *b*. Note que a última atribuição (saida <= entrada3) é realizada se e somente se nenhuma das outras condições anteriores forem satisfeitas. Por isso nenhuma condição precisou ser descrita.

2.1.3 - Arquivos

Um arquivo (**file**) é uma classe de objeto em VHDL usada para armazenamento de dados. A declaração desta classe é feita definindo-se o tipo de arquivo e tipo de dados com que se deseja trabalhar. Por se tratar de um tipo deve-se iniciar a declaração com a palavra-chave **type** seguida pelo nome do arquivo. Em sequência, as palavras-chaves **is file of** e o tipo de dado a ser armazenado. Como exemplo a seguir tem-se uma definição de um tipo de arquivo de bits, chamado *arq_binário*.

type arq_binario **is file of** bits;

Uma vez tendo-se declarado o tipo de arquivo deve-se declarar a arquivo de entrada ou saída que irá ser manipulado. Isto é feito através do uso da palavra-chave **file** seguida pelo identificador do arquivo, dois pontos ':' e do tipo de arquivo declarado. A seguir utiliza-se a palavra-chave **open** com o modo de abertura do arquivo. São três os modos possíveis: modo leitura (**read_mode**), escrita (**write_mode**) e escrita anexada

(**append_mode**). Por fim acrescenta-se **is** seguido pelo nome de referência do arquivo, que pode ser o nome do mesmo em um diretório ou simplesmente um nome usado para interfaceamento com uma memória.

Como exemplo a seguir se define um arquivo chamado *arquivo_entrada* do tipo anteriormente declarado *arq_binario*, de modo leitura e com nome "dados.dat":

file arquivo_entrada:arq_binario **open** read_mode **is** "dados.dat";

Para leitura deste arquivo usa-se o procedimento read e para escrita o procedimento write. O conceito de procedimentos é apresentado no capítulo de "Subprogramas" item 7.1. Exemplificando uma leitura de dados para a variável *bit_in* via arquivo:

read(arquivo_entrada, bit_in);

A escrita é feita de modo análogo desde que é claro tenha-se definido um arquivo de escrita ou escrita anexada. Exemplo:

file arquivo_saida:arq_binario **open** write_mode **is** "resultados";
write(arquivo_saida, bit_out);

Para conferência de fim de arquivo usa-se a função **endfile**(nome_arquivo), que retorna 1 se o arquivo chegou ao fim e 0 caso não. Funções são apresentados no item 7.2.

2.2 - Tipos Escalares

Variáveis tipo escalares são aquelas cujos valores são indivisíveis.

2.2.1 - Tipos Inteiros

Variáveis do tipo inteiro podem variar de -2.147.483.647 a +2.147.483.647, por serem representadas por 32 bits. Algumas vezes entretanto é útil definir-se novas faixas de operação para algumas variáveis. VHDL permite a definição destas faixas a partir das estruturas **is range/to** ou **is range/downto**, usadas para variações ascendentes e descendentes respectivamente. Algumas destas estruturas pode ser observadas nos exemplos:

type dia_da_semana **is range** 1 **to** 31;
type valor_contagem **is range** 10 **downto** 0;

Em uma declaração de tipo, o valor inicial de uma variável (caso não atribuído) será igual ao valor mais a esquerda da faixa definida. Por exemplo uma variável do tipo *dia_da_semana* teria valor inicial 1 enquanto que uma do tipo *valor_contagem* teria 10.

2.2.2 - Tipos Ponto Flutuante

Variáveis do tipo ponto flutuante são compostas por duas partes: uma mantissa e uma parte exponencial. Existe um tipo ponto flutuante pré-definido em VHDL que é o tipo real, que varia de -1.0E+38 até +1.0E38, com pelo menos seis dígitos de precisão. É possível a definição de outras variáveis do ponto flutuante conforme a necessidade da aplicação. Alguns exemplos são apresentados a seguir:

```
type  valor_leitura is range -5.0 to 5.0;
type  porcentagem is range 0.0 to 100.0;
```

2.2.3 - Tipos Físicos

Variáveis do tipo físico (*physical*) são utilizadas na representação de quantidades físicas do mundo real, tais como comprimento, massa, tempo e tensão. Na definição destas variáveis é possível a atribuição das unidades respectivas através do uso da palavra-chave **units**. Abaixo apresenta-se um exemplo de uma medida física de resistores em VHDL:

```
type resistencia  is range 0 to 1E9
    units
        ohm;
        kohm = 1000 ohm;
        Mohm = 1000 kohm;
    end units resistencia;
```

Existe um tipo físico muito importante pré-definido em VHDL que é o tipo **time**, usado para operações com tempo. A declaração deste tipo pode ser observada a seguir:

```
type time  is range implementation_defined
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

Note que em VHDL a unidade de tempo é fs, apesar que na maior parte das vezes se adotam outras unidade mais comuns como ms ou ns.

2.2.4 - Tipo Enumeração

O tipo especial de enumeração nada mais é do que uma lista ordenada de possíveis valores que uma determinada variável pode conter. Por exemplo a variável *display* definida por

type display **is** (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ’8’, ‘9’);

pode assumir qualquer um dos 10 possíveis valores ‘0’ a ‘9’. Se por acaso esta variável devesse representar alguns valores hexadecimais, deveria ser definida como:

type display **is** (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ’8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’);

Existe diversos tipos pré-definidos de enumeração em VHDL . Por exemplo:

type severity_level **is** (note, warning, error, failure);

Na prática os tipos pré-definidos de enumeração mais importantes são:

- character
- boolean
- bit

A variável do tipo **character** apresenta 256 possíveis valores, sendo que os 128 primeiros representam a primeira metade da tabela ASCII.

A declaração do tipo **character** em VHDL é feita conforme pode ser visto a seguir na fig. 2.2.

```
type character is (
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em,  sub, esc, fsp, gsp, rsp, usp,

        ' ', '!', '"', '#', '$', '%', '&', ''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',

        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',

        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', del,

        c128, c129, c130, c131, c132, c133, c134, c135,
        c136, c137, c138, c139, c140, c141, c142, c143,
        c144, c145, c146, c147, c148, c149, c150, c151,
        c152, c153, c154, c155, c156, c157, c158, c159,

        -- the character code for 160 is there (NBSP),
        -- but prints as no char

        ' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
        '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
        '°', '±', '²', '³', '´', 'µ', '¶', '·',
        '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',

        'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
        'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
        'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
        'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',

        'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
        'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
        'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
        'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ' );
```

Figura 2.2: Declaração em VHDL do tipo character

Um variável do tipo **boolean** é bastante usada para operações lógicas cujo resultado deve ser verdadeiro ou falso. A definição do tipo **boolean** é:

type boolean **is** (false, true);

As variáveis do tipo **bit** são utilizadas para operações binárias em VHDL. A definição do tipo **bit** é:

type bit **is** (‘0’, ‘1’);

Note que os dois possíveis valores de **bit** são os caracteres '0' e '1' e não os números 0 e 1.

2.3 - Tipos Compostos

Variáveis de tipo composto são montadas com um agrupamento de variáveis de tipos já conhecidos.

2.3.1 - Arrays

De uma forma geral, arrays são conjuntos de valores de um determinado tipo. O agrupamento de determinado tipo em um conjunto é especialmente importante para operações com tratamento de memória ou manipulação de matrizes.

A declaração de um array deve obrigatoriamente conter além do nome do array, o tipo de variável (ou variáveis) que deverá comportar e normalmente a faixa de validade (dimensão) do array. A forma mais simples de compreender a forma de declaração de arrays é através de exemplos práticos. Na declaração abaixo, por exemplo:

type matriz_entrada **is array** (0 **to** 9) **of** integer;

está-se definindo uma matriz de 10 elementos de tipo inteiro (**integer**) chamada *matriz_entrada*. Assim como em outras estruturas VHDL é possível o uso da palavra-chave **downto** para ordenação descendente, o que, em alguns casos, é bem útil. Por ser um arranjo mais genérico a declaração de arrays pode conter também valores não numéricos, como no exemplo a seguir:

type dias_semana **is** (DOM, SEG, TER, QUA, QUI, SEX, SAB);
type dias_uteis **is array** (SEG **to** SEX) **of** dias_semana;

onde se define um array chamado *dias_uteis* que contém somente um conjunto dos cinco dias da semana de SEG a SEX a partir do conjunto *dias_semana*.

Em muitos casos arrays de uma única dimensão não são suficientes para as operações que se deseja. VHDL permite a definição de arrays multi-dimensionais de forma facilitada. Por exemplo, pode-se definir o array que representa uma imagem bidimensional de 10x20 da seguinte forma:

type imagem **is array** (1 **to** 10, 1 **to** 20) **of** integer;

Note que, no exemplo, *imagem* inicia da posição 1. Não é necessário se iniciar sempre do valor 0. Da mesma forma não há a obrigação de que os tipos definidos para cada dimensão de um array sejam iguais. É possível por isso construir-se arrays multi-dimensionais de tipos distintos como no exemplo:

type estado **is** (ativado, desativado, falha);
type conjunto_processos **is array** (0 **to** 10, estado) **of** bit;

que define uma matriz de bits, cujas dimensões são dadas por tipos distintos.

VHDL permite ainda a construção de arrays sem definição de limites através dos símbolos “<>“, como pode ser visto a seguir:

type matriz **is array** (natural **range** <>) **of** integer;

que define um array chamado *matriz* de inteiros, cuja dimensão não foi especificada, mas que deve estar contida dentro do espaço de números naturais (**natural**). Normalmente o tamanho de um array deste tipo é definido quando se atribui no programa uma constante ou variável que irá utilizar este tipo de array. Por exemplo:

variable matriz1 : matriz := (0.0, 0.0, 0.0, 0.0, 0.0);

define uma variável do tipo *matriz* (sem definição de limite) chamada *matriz1* que tem 5 posições. A atribuição de valores a arrays merecem alguns comentários, pois em VHDL são disponibilizados alguns formatos especiais.

A atribuição mais simples já foi mostrada e trata do preenchimento direto dos valores de um array ou de um elemento, como na listagem a seguir:

```
type  medidor is array (1 to 3) of integer := (0.0, 0.0, 0.0);
medidor(1) := entrada1;
medidor(2) := entrada2;
medidor(3) := (entrada1 + entrada2)/2;
```

A primeira linha permite o preenchimento de todo array com valores iniciais. As demais linhas acessam cada uma das posições colocando os valores apropriados, respectivamente. Para arrays muito grandes entretanto alguns recursos são especialmente úteis. Por exemplo para o caso a seguir:

```
type 7_segm is (1 to 7) of  bit;
variable display1: 7_segm := (1 =>1, 2 =>0, 3 =>0, 4 =>0, 5 =>0, 6 =>1,7 =>1);
variable display2: 7_segm := ( 1 =>1, 2 to 5 => 0; 6 to 7 =>1);
variable display3: 7_segm := ( 1 | 6 | 7 => 1, 2 to 5 => 0);
variable display4: 7_segm := ( 1 | 6 | 7 => 1, others => 0);
```

são definidos 4 variáveis de display de 7 segmentos que recebem os mesmos valores iniciais. Note que a palavra-chave **to** agrupa elementos consecutivos enquanto que o símbolo ‘|’ é utilizado para agrupar elementos separados. Além disso, a palavra-chave **others** é válida aqui para se permitir a atribuição de todos os elementos não referenciados anteriormente.

Operações lógicas (ver item 3.3) com arrays são permitidas, desde que os arrays contenham variáveis do tipo **bit** ou **boolean**.

A cópia de arrays é possível desde que sejam arrays de mesmo tipo. Por exemplo:

```
matriz2 := matriz1;
```

realiza a cópia de todos os elementos do array *matriz1* para a *matriz2*.

2.3.2 - Records

Records, assim como arrays, são conjuntos de variáveis, permitindo porém o agrupamento de variáveis de tipos diferentes. Cada elemento de um determinado tipo dentro desta estrutura possui um nome próprio, que é utilizado para referenciá-lo dentro do record. Na sua declaração todos os elementos constituintes de uma estrutura do tipo record devem ser declarados em sequência com seus nomes próprios. Após o final da declaração da estrutura record seu nome deve ser repetido. Para ilustrar o uso deste tipo, no exemplo abaixo:

```
type  horario is record
      segundo : integer  range 0 to 59;
      minuto : integer  range 0 to 59;
      hora : integer  range 0 to 23;
end record  horario;
```

declara-se uma estrutura de horário composta por 3 inteiros *segundo, minuto* e *hora*, com as respectivas faixas de operação.

A atribuição de valores para um elemento específico é feito a partir do nome do record seguido de '.' e o nome da variável que se pretende acessar. Por exemplo:

```
variable horario_atual: horario;
horário_atual.segundo := 23;
horário_atual.minuto := 42;
horário_atual.hora := 10;
```

ajusta o horario atual do sistema para 10:42:23. Outra maneira de se realizar esta atribuição é:

```
horario_atual := (hora => 10, minuto => 42, segundo => 23);
```

Similarmente ao caso de arrays, são permitidos os usos dos recursos '|' e **others** para atribuição de um conjunto de variáveis ao mesmo tempo. Também é possível a cópia de records como em:

```
estrutura2 := estrutura1;
```

que realiza a cópia de todos elementos do record *estrutura1* para a *estrutura2*.

2.4 - Subtipos

Uma vez tendo-se definido um tipo de variável, em VHDL, é possível se definir um subtipo ou seja um tipo de variável que seja um subset do tipo já definido através do uso da palavra-chave **subtype**. Como exemplo são apresentados 3 subtipos pré-definidos em VHDL para números naturais, positivos e atraso de tempo, respectivamente:

subtype natural **is** integer **range** 0 **to** highest_integer;
subtype positive **is** integer **range** 1 **to** highest_integer;
subtype delay_length **is** integer **range** 0 **to** highest_time;

2.5 - Conversão de Tipos

A conversão, ou seja a transformação de um tipo de variável para outro é permitido em VHDL. Para implementar isto deve-se descrever o tipo de variável que se deseja obter seguido pelo valor entre parênteses. Por exemplo a conversão:

integer(39.382)

produz o número inteiro mais próximo (no caso, 39). Enquanto que:

real(123);

gera o número em ponto flutuante 123.0;

2.6 -Diagrama Completo de Tipos de Dados em VHDL

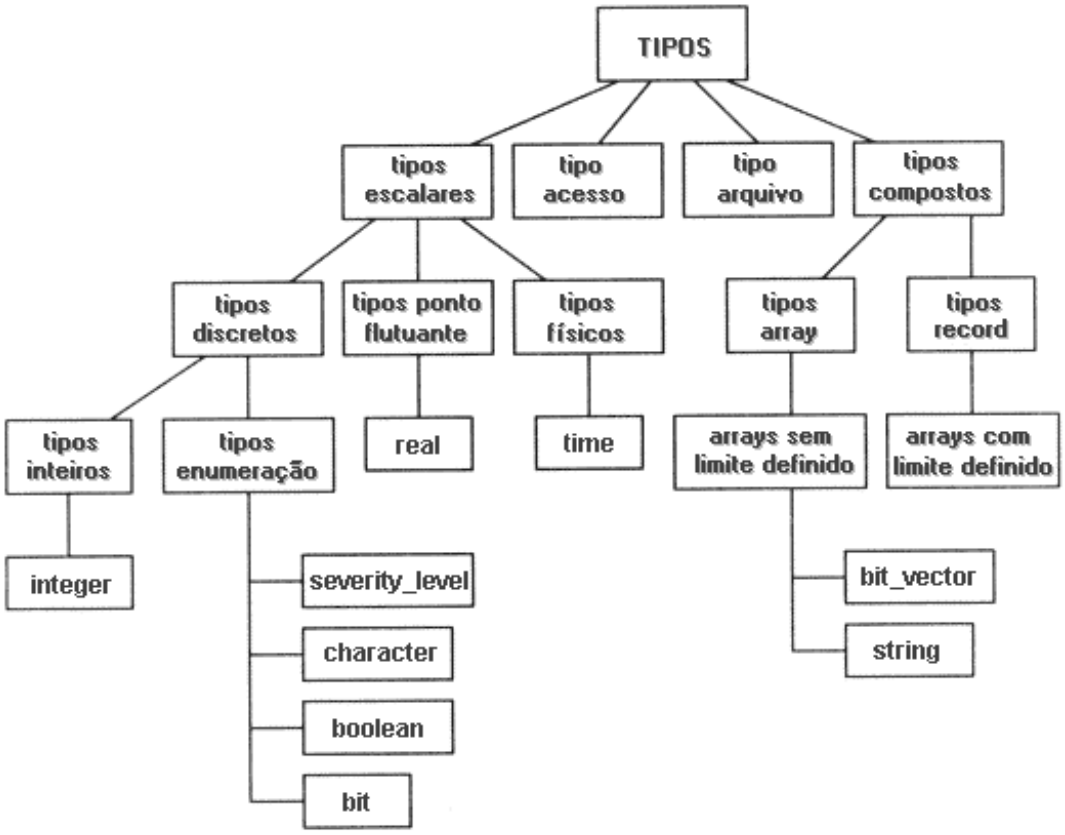

Figura 2.3: Hierarquia dos tipos de dados em VHDL

3 - Expressões e Operadores

3.1 - Operações Aritméticas

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, boolean ou array (bit, boolean)	igual ao operador da direita

3.2 - Operações de Comparação

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

3.3 - Operações Lógicas

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

3.4 - 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	desl. aritmético para esquerda	array de bit ou boolean	inteiro	array de bit ou boolean
sra	desl. 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 - Estruturas e Comandos em VHDL

4.1 - Estrutura IF

Em aplicações onde o estado de alguma variável (ou conjunto de variáveis) determina as operações que devem ser tomadas é muito comum o uso da estrutura IF. A forma mais simples de utilização é através da estrutura **if-then-endif** como apresentado a seguir:

```
if chave = 1 then
        saida1 := valor_saida;
endif
```

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-endif**:

```
if chave = 1 then
        saida1 := valor_saida;
  else
        saida1 := 255;
        registrador := valor_saida
endif
```

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 **elseif**, que funciona como uma nova estrutura de comparação dentro da estrutura if corrente. São possíveis tantos **elseif** quantos forem necessários para refinar as comparações. A seguir tem-se um exemplo:

```
if  painel = OFF then
        led1 := OFF;
        led2 := OFF;
  elseif  leitura = 1 or leitura = 3 then
        led1 := ON;
        led2 := OFF;
  elseif  leitura = 2 or leitura = 4 then
        led1 := OFF;
        led2 := ON
endif
```

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.

4.2 - 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;
endcase;
```

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* (aplicação apresentada no item 4.1):

```
case leitura is
        when 1 | 3 =>
                led1 := ON;
                led2 := OFF;
        when 2 | 4 =>
                led1 := OFF;
                led2 := ON
endcase
```

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 =>
                saida <='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;
                saida <= '1';
end case;
```

implementa-se a máquina de estados da figura 4.1. Note que, 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.

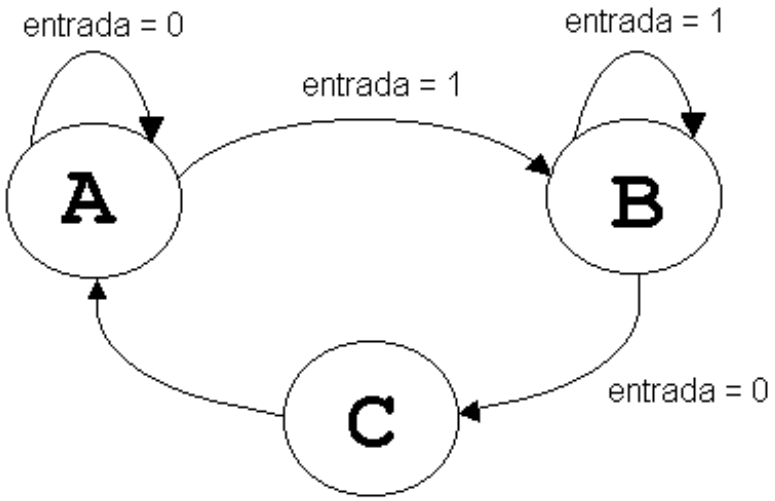

Figura 4.1: Máquina de estados implementada no exemplo

4.3 - Estrutura LOOP

As estruturas de loop em VHDL são bastante usadas principalmente para a execução repetitiva de uma determinada sequência de operações. Existe um série de variações de estruturas de loop possíveis em VHDL, de acordo com a finalidade do software. Estas estruturas serão a seguir apresentadas em sequência para melhor entendimento.

4.3.1 - Estrutura Loop Infinito

A forma mais simples de uma estrutura **for** é chamada desta forma por não apresentar condição de saída. Esta estrutura é particularmente útil para aplicações de sistemas simples que só executam uma tarefa, como por exemplo um controle de nível a seguir:

```
loop
        if nivel_atual < nivel_desejado -2 then
                valvula_entrada := ON;
          elseif  nivel_atual > nivel_desejado +2 then
                valvula_entrada := OFF;
        endif
endloop
```

que tenta manter o nível em um valor desejado em um sistema ON/OFF com uma histerese de largura 4.

4.3.2 - Estrutura Loop com Saída Forçada por Exit

Um estrutura LOOP pode entretanto precisar de uma condição de saída. Isto é previsto em VHDL pela palavra-chave **exit**, que força a saída do loop quando uma determinada condição for atendida. Por exemplo:

```
loop
        loop
                exit when reset_contador = '1';
                contador := contador +1;
                saida := contador;
        endloop
        contador := 0;
endloop
```

A estrutura acima implementa um contador que reseta sua contagem se a condição reset_contador = '1' for verdadeira. Note que é possível o encadeamento de loops, ou seja a montagem de um loop dentro de outro. A linguagem VHDL permite a atribuição de nomes a loops e a seleção de qual saída será forçada a partir destes nomes.

Um exemplo de uso de estrutura for com diversos loops encadeados é descrito a seguir:

```
loop_externo : loop
   contador := 0;
        loop_intermediario : loop
                loop_interno : loop
                     saida := contador;
                    exit  when contagem_habilitada = '0';
                    exit loop_intermediario when reset_contador = '1';
                endloop loop_interno
                contador := contador +1;
        endloop loop_intermediario
endloop loop_externo
```

onde existem duas condições de saída. A primeira testa se a contagem está desabilitada (contagem_habilitada = '0'), quando então sai do *loop_interno* para proceder uma nova contagem. A segunda condição de saída testa se *reset_contador* = '1' e caso for, salta para fora do *loop_intermediario*, resetando o contador.

4.3.3 - Estrutura Loop com Reinício Forçado por Next

Esta estrutura é similar ao caso de saída anterior, só que a palavra-chave **next** serve para reiniciar a sequência de operações do loop. Na descrição a seguir tem-se um monitor de temperatura, que só executa as medidas de emergência (aquecedor:= OFF e

alarme:=ON) se a temperatura medida exceder o limite. Note que com o uso de **next** a execução não sai fora do loop

```
loop
        medida_temperatura =  entrada_medidor;
        next when medida_temperatura < temperatura_maxima;
        aquecedor := OFF;
        alarme := ON;
endloop
```

Assim como no caso do **exit**, também é possível com **next**, a definição de qual loop deve ser iniciado quando se estiver trabalhando com loops encadeados. O formato de utilização é o seguinte:

```
...
next nome_loop when condicao = true
...
```

4.3.4 - Estrutura WHILE-LOOP

A estrutura **while-loop** é um caso especial de estrutura de loop, onde uma condição é testada inicialmente antes de qualquer operação do loop e caso seja verdadeira, o loop é executado. Se a condição não for satisfeita, o loop pára de ser executado, seguindo-se além deste.

Um exemplo desta estrutura é apresentada a seguir:

```
numero := valor_entrada;
raiz_quadrada := 0;
while  raiz_quadrada*raiz_quadrada < numero  loop
        raiz_quadrada := raiz_quadrada + 0.1;
end loop;
```

onde o cálculo simples de uma raiz quadrada é feito até que se encontre o valor mais próximo. Note que se o valor de entrada for zero, a condição do loop não é satisfeita e o mesmo não é executado, resultando no valor raiz_quadrada := 0 que é o valor correto.

4.3.5 - Estrutura Loop com Contagem Tipo For

O uso de estrutura de **for-loop** é utilizado principalmente para execução de operações a serem repetidas por um número determinado de vezes. Por exemplo a sequência abaixo descreve o cálculo do fatorial do número 10.

```
resultado := 1;
for n in 1 to 10  loop
        resultado := resultado *n;
end loop;
```

A estrutura **for-loop** permite também a realização de iterações em ordem decrescente. Neste caso basta utilizar-se a palavra-chave **downto** no lugar de **to**, como em:

```
contador := 5;
resultado := 0;
for contador in 10 downto 1 loop
    resultado := contador;
end loop;
```

É interessante notar que a variável utilizada para as contagens da estrutura **for-loop** mascara dentro do loop uma outra variável externa com o mesmo nome, mas quando for finalizado a variável externa terá seu valor intacto. No caso acima, por exemplo, ao final desta execução a variável contador terá valor 5 e a variável resultado terá o valor 1.

4.4 - 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 esperar-se simplesmente a mudança de estado de um sinal ou aguardar-se por uma variação específica.

4.4.1 - 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*.

4.4.2 - 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';
...
```

4.4.3 - Espera por Nível

Em alguns casos, a função que se deseja sintetizada 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.

4.4 - Comando GENERATE

O comando de geração (generate) é usado para gerar réplicas de uma dada parte da descrição VHDL. De forma simples este comando permite a realização de uma cópia dos comandos ou componentes inseridos entre as palavras-chaves **generate** e **end generate**. A forma clássica de uso deste comando está na réplica de componentes, quando se deseja montar um projeto que apresenta blocos que devem ser repetidos várias vezes.

Por exemplo, a descrição a seguir:

```
architecture mascaramento of processa_registrador
begin
        for n in 0 to 7  generate
                resultado(n) <= registrador(n) and mascara_config(n);
        end generate;
end architecture mascaramento;
```

realiza uma operação de mascaramento, ou seja de acordo com uma palavra de configuração (*mascara_conf*) se ativa ou não os bits de um dado registrador para a saída.

A operação deve ser feita bit a bit e para tanto foi replicada 8 vezes pelo comando **generate**.

4.5 - Comandos de Compilação: Checagens e Mensagens

Existe uma forma de se interagir com a simulação de um programa em VHDL a fim de permitir ao programador checar algumas condições que julgue necessárias ao seu projeto.
São basicamente dois comandos: **assert** para checagem e **report** para exibição de mensagens.

A conferência de alguma anormalidade é feita pela palavra-chave **assert**, seguida pela condição a ser checada. Normalmente esta checagem é seguida pela operação de envio de mensagem de aviso. O envio de mensagens durante o período de simulação é feito pelo uso da palavra-chave **report**, seguida pela mensagem desejada. Por exemplo:

```
assert valor_entrada  < valor_maximo
        report “Valor de entrada excede valor maximo!”;
```

confere a validade da variável *valor_entrada* e quando muito grande sinaliza com erro escrevendo a mensagem "Valor de entrada excede valor maximo!".

Além disso é possível trata-se com até quatro tipos de violação de regras de projeto, a partir do uso de **severity**: **note**, **warning**, **error** e **failure**. O valor **note** é utilizado simplesmente para a passagem de mensagens informativas.

```
assert valor_inicial = 0
        report “Valor inicial indefinido!”
        severity note;
```

Observe que a linha que contém o comando **report** não é finalizada com ';'. Isto só é permitido para uma sentença de comando report seguida de um comando severity. Se não houver esta segunda linha a finalização com ';' é obrigatória.

O valor **warning** é normalmente utilizado para casos em que o modelo pode continuar rodando, mas em que se deve fazer uma advertência ao programador.

```
assert ponteiro_buffer /= 0
        report “Buffer de mensagens ocupado!”
        severity warning;
```

O valor **error** indica que uma condição de falha ocorreu e que deve-se corrigi-la antes de se prosseguir com os trabalhos.

```
assert pulso_entrada >=  largura_clock_minima
        report “Sinal pulso_entrada  excede largura maxima!”
        severity error;
```

Por fim o valor **failure** que é o erro mais grave que poderia ocorrer. Indica um erro extremo.

```
assert numero_contagens <= 0
        report “Parâmetro de contagens impróprio!”
        severity failure;
```

Se não for definida nenhuma mensagem por **report**, o padrão será a mensagem “Assertion violation.” e se não for utilizado a sinalização por **severity**, o nível de erro padrão será **error**.

5 - Programação em VHDL

A fim de se compreender a estrutura de programação VHDL deve-se inicialmente estudar a forma com a qual esta linguagem define os diferentes processos e componentes que compõem um sistema.

Todo componente VHDL tem que ser definido como uma entidade (entity), o que nada mais é do que uma representação formal de um componente ou de um processo. Em descrições mais simples uma entidade pode ser o próprio projeto, mas em implementações de grandes sistemas, o projeto é composto por diversas entidades distintas.

O modelo de definição de uma entidade em VHDL segue uma estrutura bem específica, composta por duas partes, que serão apresentadas a seguir:

-Declaração de entidade;
-Corpo de arquitetura

5.1 - Declaração de Entidade

Declaração de uma entidade (entity declaration) é a definição de uma estrutura funcional (que pode ser um componente completo ou só parte de um sistema), com a identificação apropriada, de modo a permitir que a mesma seja posteriormente utilizada. Normalmente o nome da entidade é declarado após a palavra-chave **entity** e repetido após a palavra-chave **end entity**. Dentro da declaração da entidade é definida a interface do componente de forma similar a definição de pinos de um componente. Isto é feito através da declaração das suas portas (ports) de interface, que além de um nome próprio devem conter um modo, que indica se é um sinal de entrada (**in**), saída (**out**) ou bidirecional (**inout**), e um tipo, que determina o tipo de informação que irá trafegar pela porta. Inicialmente são apresentados os nomes dos sinais e a seguir, separados por dois pontos '**:**', o modo e tipo dos mesmos. Na definição das portas de uma entidade, pode-se definir sinais que comportem não somente bits, mas também tipos complexos como bytes (8 bits), words (16 bits) e até tipos compostos como arrays de variáveis, conforme necessidade do projeto.

A seguir tem-se o exemplo de uma declaração de entidade bem simples:

```
entity  modelo is
port (  a, b : in bit;
        c : out bit);
end entity modelo;
```

chamada *modelo* que possui duas portas de entrada (**in**) do tipo bit, declaradas como *a* e *b* e uma porta de saída (**out**) também deste tipo, declarada como *c*. Normalmente as portas de entrada de uma entidade são declaradas antes das de saída. Cada declaração de interface é seguido por ponto e vírgula '**;**', exceto a última. Também é necessário colocar-se ponto e vírgula no final da definição de porta.

Conforme apresentado no item 2.2.4, bit é um tipo pré-definido em VHDL, que pode ter dois valores de representação '0' e '1'. Este tipo é usado para representar dois níveis lógicos de sinal.

Um tipo mais complexo poderia manter '0', '1' e 'Z' e talvez até 'X'. Na prática para representar valores digitais reais algumas vezes são utilizadas bibliotecas definidas pela organização IEEE ou por um vendedor de componentes. É comum se encontrar os tipos STD_LOGIC ou VL_BIT, que são representações de circuito bem mais próximas da realidade que os simples valores '0' e '1'.

É possível, em VHDL, a definição de valores iniciais para portas, durante a definição da entidade, como no exemplo abaixo:

```
entity  modelo is
port (  a, b : in bit := '1';
        c : out bit);
end entity modelo;
```

onde as entradas de *modelo* são iniciadas com nível lógico '1'.

5.2 - Corpo de Arquitetura

O corpo de arquitetura (*architecture body*) de uma entidade define o seu funcionamento interno, a partir de uma série de comandos de operação em um ou mais processos (**process**). Um processo é como uma unidade básica descritiva de um comportamento. Um processo é executado normalmente em resposta a mudança de valores de sinais e usa os valores correntes de sinais e variáveis para determinar os novos valores de saída.

Em VHDL existem dois conceitos de modelamento da descrição funcional de uma entidade. São eles:

- Modelo comportamental
- Modelo estrutural

5.2.1 - Modelo Comportamental

Pode-se montar um corpo de arquitetura comportamental descrevendo sua função a partir sentenças de processo (*process statements*), 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.

A declaração do comportamento de uma entidade é feito através da programação de seus algoritmos com base nos diversos comandos e estruturas da linguagem. A seguir tem-se um exemplo de uma declaração de arquitetura para a entidade *latch*.

```
architecture opera of latch is      -- Declaração de arquitetura dataflow
begin
  q <= r nor nq;                    -- Saídas q e nq recebem resultado de
  nq <= s nor q;                    -- operacao NOR com as entradas r e s
end dataflow;
```

A primeira linha da declaração indica que esta é a definição de uma nova estrutura chamada *opera*, que pertence à entidade chamada *latch*. Assim esta arquitetura descreve a operação da entidade *latch*. As linhas entre as diretivas de começo (**begin**) e fim (**end**) descrevem o operação do *latch*.

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 para a entidade *reg4*, registrador de 4 bits:

```
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.

5.2.1 - Modelo Estrutural

Existem várias formas pelas quais um modelo estrutural pode ser expresso. Uma forma comum é o esquemático de circuito. Símbolos gráficos são usados para representar subsistemas que são conectados usando linhas representando fios. Esta forma gráfica é geralmente uma das preferidas pelos projetistas. Entretanto a mesma forma estrutural pode ser representada textualmente na forma de uma lista de conexões.

Uma descrição estrutural de um sistema é expressa portanto em termos de subsistemas interconectados por sinais.

O mapeamento de portas (**port map**) especifica que portas da entidade são conectadas para que sinais do sistema que se está montando.

Por exemplo, tomando-se como base a entidade um flip-flop tipo D simples:

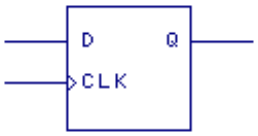

Figura 5.1: Representação da entidade *d_latch*

cuja declaração encontra-se a seguir:

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

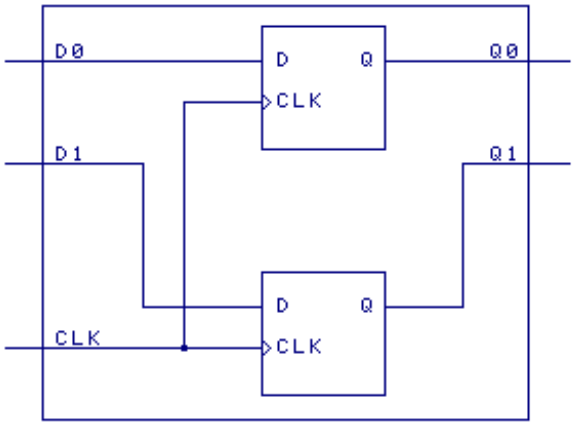

Figura 5.2: Representação de *reg_2bits* a partir de entidades *d_latch*

A descrição VHDL desta estrutura é apresentada abaixo:

```
entity  reg_2bits is
        port (  d0, d1, clk : in bit;
                q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
        begin:
                bit0:   entity work.d_latch(behavioral)
                        port map (d0, clk, q0)
                bit1:   entity work.d_latch(behavioral)
                        port map (d1, clk, q1)
end architecture estrutura;
```

Note que, neste caso, 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.

Entretanto, apesar da forma apresentada ser funcional, muitas vezes na prática se costuma descrever formalmente o mapeamento de portas desejado. Isso torna o esquema de ligações implementado mais visual e intuitivo. O mapeamento então se caracteriza pela indicação dos sinais de origem e destino para cada uma das conexões (ou também chamadas de instâncias). Como exemplo, é apresentada a seguir a mesma entidade *reg_2bits*, descrita anteriormente, porém neste formato mais claro:

```
entity  reg_2bits is
        port (  d0, d1, clk : in bit;
                q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
        begin:
                bit0:   entity work.d_latch(behavioral)
                        port map (d => d0, clk => clk, q => q0)
                bit1:   entity work.d_latch(behavioral)
                        port map (d => d1, clk => clk, q => q1)
end architecture estrutura;
```

Este formato traz por si só uma maior flexibilidade do projeto na manipulação de sinais. Pode-se assim trabalhar com arranjos mais flexíveis como quando se utilizam variáveis de tipo composto. Na descrição a seguir por exemplo:

```
entity  registrador_4bits is
        port (  d : in bit_vector (0 to 3);
                clk : in bit;
                q : out bit_vector (0 to 3);
end entity registrador_4bits;
...
entity  registrador_8bits is
        port (  din : in bit_vector (0 to 7);
                clk : in bit;
                dout : out bit_vector (0 to 7));
end entity registrador_8bits;
architecture estrutura of registrador_8bits is
        begin:
                estrutura: entity work.registrador_4bits
                        port map  ( d (0 to 3) => din (0 to 3),
                                    clk => clk,
                                    q (0 to 3) => dout (0 to 3))
                        port map  ( d (0 to 3) => din (4 to 7),
                                    clk => clk,
                                    q (0 to 3) => dout (4 to 7))
end architecture estrutura;
```

se monta um registrador de 8 bits (*registrador_8bits*) a partir e registradores de 4 bits (*registrador_4bits*) operando com sinais tipo **bit_vector**, o que simplifica a indicação das ligações.

Muitas vezes, como em alguns esquemáticos de circuitos feitos componentes discretos, deseja-se conectar sinais a valores fixos ou até mesmo desligá-los do circuito final. Isto é também possível em VHDL pela adequada atribuição de valores no mapeamento de conexões.

Veja-se por exemplo a o diagrama a seguir:

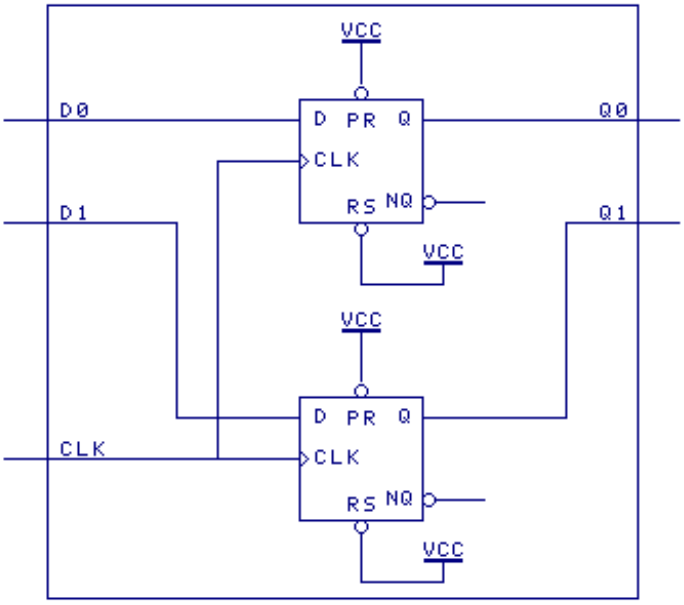

Figura 5.3: Representação de *reg_2bits* a partir de entidades *FF_D*

que pode ser descrito em VHDL como:

```
entity  FF_D is
        port (  d, clk, rs, pr : in bit;
                q, nq : out bit);
end entity FF_D;
...
entity  reg_2bits is
        port (  d0, d1, clk : in bit;
                q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
        begin:
                bit0:   entity work.FF_D(behavioral)
                        port map (d => d0, clk => clk,
                                  rs => '1', pr  => '1',
                                  q => q0, nq => open)
                bit1:   entity work.FF_D(behavioral)
                        port map (d => d1, clk => clk,
                                  rs => '1', pr  => '1',
                                  q => q1, nq => open)
end architecture estrutura;
```

onde monta-se um registrador de 2 bits (*reg_2bits*) a partir de um registrador tipo D completo (*FF_D*). Os pinos de reset (*rs*) e preset (*pr*) dos flip-flops não são úteis para o registrador de 2 bits e então são fixados para lógica '1'. As saídas *nq* (saída negada) dos flip-flops também não tem aplicação por isso são deixadas abertas (**open**).

5.3 - Interligação de Modelos com Sinais

Modelos não precisam ser puramente estruturais ou puramente comportamentais. Frequentemente é útil especificar um modelo com algumas partes compostas de instâncias de componentes interconectados e outras partes descritas usando processos. Utilizam-se sinais (palavra-chave **signal**) no sentido de interligar instâncias de componentes e processos. Um sinal pode ser associado com uma porta de uma instância de componente e pode também ser assinalado para ser lido ou escrito em um processo.

Pode-se escrever um projeto com um modelo híbrido que inclua ambos as abordagens através de instâncias de componentes e sentenças de processo em um mesmo corpo de arquitetura. Estas sentenças são coletivamente chamadas de sentenças concorrentes, uma vez que todos seus correspondentes processos são executados concorrentemente quando o modelo é simulado.

Uma demonstração disto pode ser observada no próximo exemplo. Este modelo descreve um multiplicador que consiste de uma parte de tratamento de dados e uma seção de controle.

A parte de tratamento de dados é descrita estruturalmente, usando um número de instâncias de componentes.

A seção de controle pode ser descrita comportamentalmente usando um processo que liga os sinais de controle com a parte de tratamento de dados.

```
entity multiplicador is
        port (  clk, reset, multiplicando, multiplicador: in integer;
                produto: out integer);
end entity multiplicador;

architecture prj_completo of multiplicador is
        signal produto_parcial, produto_final: integer;
        signal controle_aritmet, result_en, mult_bit, mult_load: bit;
        begin
                unid_aritmet: entity work.shift_adder(behavior)
                  port map (somador => multiplicador, augend => produto_final,
                                sum => produto_parcial,
                                add_control => controle_ aritmet);
                resultado: entity work.reg(behavior)
                  port map ( d  => produto_parcial, q => produto_final,
                                en => result_en, reset => reset);
                multiplicador_rs: entity work.shift_reg(behavior)
                  port map  (d => multiplicador, q => mult_bit,
                                load => mult_load, clk => clk);
                produto <= produto_final;
        secao_controle : process is
                ...
                wait on clk, reset;
        end process secao_controle;
end architecture prj_completo;
```

6 - Bibliotecas

Bibliotecas (*libraries*) são arquivos especiais que contém um conjunto de descrições e definições em VHDL disponibilizadas para uso por outros programas. 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. Para se incluir um componente desta biblioteca deve-se acrescer após o nome da biblioteca o caracter ponto (‘.’) e o nome do componente. Como ilustração, a seguir tem-se apresentada uma descrição que faz uso de um registrador de 32 bits (entidade *reg32*) que se encontra na biblioteca *macro_cell*:

```
library   macro_cel;
...
architecture registradores of somador32 is
begin:
        entrada1: entity macro_cel.reg32
                port map ( en => habilita, clk => clock,
                        d => entrada1, q => dado1);
...
```

Se o uso de um dado componente da biblioteca se tornar muito frequente deve-se utilizar a palavra-chave **use** seguida do nome da biblioteca e do componente (separados pelo caracter ‘.’). Isto torna este componente globalmente visível pelo projeto. Por exemplo a descrição abaixo:

```
library   macro_cel;
use  macro_cel.reg32;
...
architecture registradores of somador32 is
begin:
        entrada1: entity reg32
                port map ( en => habilita, clk => clock,
                        d => entrada1, q => dado1);
...
```

torna o componente *reg32* conhecido pelo projeto. Assim não é mais necessário referenciar a biblioteca *macro_cells*, quando se precisar utilizá-lo.

Pode-se também incluir todos os componentes de uma biblioteca usando a diretiva **use** em conjunto com a palavra-chave **all** (esta deve ser colocada no lugar do nome do componente). A linha abaixo então:

```
use new_lib.all;
```

inclui todos os elementos que compõem a biblioteca *new_lib*. Note que no caso de duas ou mais bibliotecas possuírem componentes com o mesmo nome, quando se necessitar utilizar um específico é necessário referenciar o nome da biblioteca correta.

7 - Subprogramas

Subprogramas em VHDL são conjuntos de comandos ou algoritmos que desempenhem uma determinada operação que é necessária muitas vezes no projeto. Assim esta sequência de comandos é montada e disponibilizada globalmente de forma que o projeto possa acessá-la quando for necessário.

Basicamente são dois os tipos de sub-programas em VHDL:

-Procedimentos;
-Funções.

7.1 -Procedimentos

Um procedimento (**procedure**) em VHDL é um conjunto de operações que executa uma determinada finalidade. Este procedimento pode ser chamado de qualquer parte do programa VHDL, de forma similar à chamada de funções feita em várias outras linguagens.

Para a chamada de um procedimento em uma descrição VHDL basta escrever-se o nome do procedimento seguido de ponto-e-vírgula (';'). Por exemplo:

```
...
procedure delay is
        begin
                for n in 1 to n_delay loop
                wait for 10ns;
                end loop;
end procedure delay;
...
delay;          -- chamada do procedimento de atraso
...
```

implementa um procedimento de atraso de (*n_delay**10ns). Note que *n_delay* deve ser uma constante global.

7.1.1 - Procedimento com Parâmetros

Para ser mais completa, a utilização de procedimentos em VHDL deveria permitir a passagem de parâmetros quando de sua chamada. Isto é também previsto pela linguagem. Para tanto deve-se inicialmente prever na declaração do procedimento o parâmetro (ou parâmetros) que poderá ser passado, descrevendo-se o mesmo entre parênteses. Na sua descrição, o parâmetro é composto por um identificador, um modo (**in**, **out** ou **inout**) e o tipo. Algumas vezes utiliza-se a palavra-chave **func_code** na especificação de tipo, quando deseja-se que o tipo do parâmetro seja dado pelo programa que chama o procedimento.

No exemplo a seguir tem-se uma descrição do uso de um procedimento de envio, por um pino serial chamado dout, de um caracter (*dado*) passado como parâmetro:

```
procedure envia_serial (dado : in func_code) is
        variable indice : dado_serial  := 0;
        variable teste : positive := 0;
        variable temp : bit := start_bit ;
        begin
                if (clock='1') then
                        if  teste >1 and teste < (tamanho_dado_serial + 1) then
                                teste := teste +1;
                                if  indice <= tamanho_dado_serial  then
                                        temp := dado(indice);
                                        indice := indice +1;
                                end if;
                        end if;
                end if;
                if  teste = (tamanho_dado_serial +1) then
                        temp := stop_bit;
                        dout <= temp;
                        return;
                end if
                dout <= temp;
end procedure envia_serial;
...
dado := 34;
envia_serial(dado);
...
```

Para ilustrar que procedimentos também poder retornar valores a seguir apresenta-se um exemplo de um algoritmo para conversão de valor decimal para BCD:

```
...
procedure converte_para_BCD (numero: in integer range 0 to 39;
                        mais_sig, menos_sig : out integer range 0 to 9) is

        begin
                if numero > 39    then   return;
                elsif     numero >= 30    then
                        mais_sig := 3;
                        menos_sig := numero - 30;
                elsif  numero >= 20   then
                        mais_sig := 2;
                        menos_sig := numero - 20;
                elsif  numero >= 10   then
                        mais_sig := 1;
                        menos_sig := numero - 10;
                else      mais_sig := 0;
                        menos_sig := numero;
        end if;
end procedure converte_para_BCD;
...
converte_para_BCD (dado_lido, dezena, unidade);
```

Observe que é possível ter mais de um ponto de saída de um procedimento. No modelo apresentado, por exemplo, se o valor da variável for maior que 39 o procedimento é abortado. Finalizações de procedimentos são possíveis através da palavra-chave **return**.

Além da passagem de valores como parâmetros, um procedimento permite a passagem de sinais, o que na verdade se constitui pela ligação de um pino de entrada (ou saída) de um módulo com a interface do procedimento que está sendo chamado. Assim, se fosse desejado, por exemplo, para o procedimento apresentado anteriormente deixar a atribuição do pino de saída serial a critério do programa que fosse utilizá-lo, poderia-se alterar a declaração do procedimento para:

```
procedure envia_serial (dado : in func_code
                              signal dout: out bit) is
```

7.2 - Funções

Uma função (**function**) em VHDL é similar a um procedimento (descrito no item 7.1) exceto que permite o retorno de um resultado na própria linha de execução. Este tipo de recurso é bastante útil para uso em expressões.

No exemplo a seguir, tem-se implementada uma função que realiza a conversão de um número em formato BCD para decimal.:

```
function bcd_para_decimal (valor_bcd : integer 0 to 255) return integer is
        variable resultado : integer range 0 to 99;
        variable valor_temp : integer range 0 to 255;
        variable nibble: integer range 0 to 15;
        begin
                nibble := valor_bcd;
                valor_temp := (dado - nibble)/2;
                if   valor_bcd > 9   then
                        resultado := valor_temp + valor_temp/4 + nibble;
                else
                        resultado := valor_bcd;
                end if;
        return resultado;
end function bcd_para_decimal;
...
if  bcd_para_decimal(valor_entrada) > valor_limite  then
        atuador <= OFF;
  else
        atuador <= ON;
end if;
...
```

A função é usada para ajustar o formato de uma variável chamada *valor_entrada*, retornando na própria linha de chamada o valor da conversão. Note que a variável (ou variáveis) de entrada de uma função não precisa conter o modo (como **in** ou **inout**), uma vez que isto é subentendido. A variável de saída é declarada fora dos parênteses e não precisa identificador, apenas o tipo.

8 - Blocos Externos

Além da opção de criação de subprogramas em VHDL é possível também a criação e uso de blocos operacionais globais. A finalidade neste caso é de se montar um projeto que fique disponível para reuso no formato de um elemento ou componente fechado. Outros programas podem utilizar estes blocos como "caixas pretas" que são interligadas a sinais ou portas conforme necessário.

Normalmente os tipos de blocos disponíveis em VHDL, são dois:

-Módulo (package);
-Componentes.

Antes porém de se apresentar estes blocos irá-se introduzir o tema parâmetros genéricos, que trata da forma como se pode estar ou definir informações próprias para o bloco que se pretende utilizar. Isto é especialmente importante para blocos muito completos que devem ser configurados para as características de uma dada aplicação antes de ser utilizado.

8.1 - Parâmetros Genéricos

Algumas vezes quando se definem entidades, módulos ou componentes em VHDL tem-se a intenção de reutilizá-las para outras aplicações e até mesmo outras tecnologias. Assim, em alguns casos, para aplicações distintas, as entidades deveriam ter características diferentes, como por exemplo nos atrasos de propagação. Isto sem alterar o funcionamento da entidade. Deve-se lembrar que descrições em VHDL são descrições de hardware genéricas não dependendo das tecnologias atuais no campo de eletrônica digital ou lógicas programáveis.

A fim de se permitir a mudança de parâmetros de entidades deve-se utilizar constantes genéricas. Este tipo de elemento apesar de se comportar como uma constante dentro do corpo da entidade pode ser alterado externamente pelo programa que pretende utilizá-la.

Para a criação deste elemento deve-se fazer uso da palavra-chave **generic** seguido da declaração da(s) constante(s) que se pretende definir entre parênteses. Como exemplo apresenta-se um modelo de um flip-flop simples, chamado *FF*, que permite mudança de seu tempo de propagação, *tpd*:

```
entity FF is
        generic (tpd : time := 3ns)
        port  (  d, clk : in  bit;
                 q : out bit);
end entity FF;
architecture carga_FF of FF is
        begin
                 wait until clk ='1';
                 q <= d after tpd;
end architecture carga_FF;
```

Note que no exemplo se define um valor inicial de 3ns, o qual será utilizado caso o parâmetro não seja alterado externamente.

Para a alteração dos valores destas constantes genéricas deve-se utilizar uma estrutura de mapeamento onde se atribuem os valores desejados a cada constante genérica. Este mapeamento de constantes genéricas deve ser feito através das palavras-chaves **generic map**.

A seguir apresenta-se um exemplo de uso do flip-flop *FF* apresentado anteriormente (supondo-o dentro da biblioteca de trabalho *work*):

```
flipflop1:  entity work.FF
                generic map (tpd => 5ns);
                port map ( d=> d1, clk => clock,  q => q1);
flipflop2:  entity work.FF
                generic map (tpd => 10ns);
                port map ( d=> d1, clk => clock,  q => q1);
```

onde o primeiro flip-flop definido no projeto, *flipflop1*, tem tempo de propagação de 5ns enquanto que o segundo, *flipflop2*, tem tempo de propagação de 10ns.

8.2 - Módulos (Packages)

Módulos *packages*[1] em VHDL 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.

Um módulo *package* é composto por duas descrições isoladas. A primeira é uma visão externa do módulo, chamada declaração do módulo. A segunda é a implementação propriamente dita das funções que o módulo exerce, chamada de corpo do módulo.

8.2.1 - Declaração do Módulo

A declaração do módulo (*package declaration*) deve conter todas as informações necessárias para que um usuário possa importá-lo e trabalhar com ele. Normalmente o uso de módulos *packages* é útil para a composição de bibliotecas de funções e componentes que podem ser incorporadas a programas distintos.

Na declaração deve-se dar o nome do módulo *package*, o qual é usado para referenciá-lo no projeto, bem como da declaração das variáveis utilizadas pelo módulo. Para ilustrar isso a seguir descreve-se um módulo de UART, que apresenta como parâmetros sua frequência base de operação (*frequencia_base*), um registrador de caracter recebido (*caracter_rx*), um de caracter a ser transmitido (*caracter_tx*) e um bit para informar se um novo dado foi recebido (*caracter_recebido*):

[1] A tradução literal para *package* é empacotamento, que entretanto não é muito difundida. Irá-se utilizar neste texto *package* associada ao termo módulo, mais acessível e que mantem a idéia original.

```
package UART is
        constant frequencia_base : positive :=19200;
        variable caracter_rx : byte;
        variable caracter_tx : byte;
        variable caracter_recebido : bit :='0';
end package UART;
```

Uma vez que uma declaração de módulo seja compartilhada por um projeto, todas as variáveis declaradas podem ser livremente acessadas.

A seguir irá-se apresentar um exemplo de uso de um módulo *package* tomando-se como base o módulo UART definido anteriormente:

```
...
entity interface _serial is
        port (byte_ recebido : in serial_lib.caracter_recebido;
                novo_byte : in serial_lib.caracter_rx);
end entity interface_serial;

architecture recepcao_serial of interface_serial is
        type buffer_serial is array (0 to15) of byte;
        variable buffer_cheio : bit := '0';
        variable buffer_vazio : bit := '1';
        variable apontador_buffer : byte := 0;
        begin
                while apontador_buffer < 16 loop
                        if  byte_recebido = '1'
                                buffer_serial (apontador_buffer) := novo_byte;
                                apontador_buffer := apontador_buffer + 1;
                                byte_recebido := '0';
                                buffer_vazio := '0';
                        end if;
                end loop;
                buffer_cheio := 1;
end architecture recepcao_serial;
```

Para este exemplo supõe-se que o módulo UART esteja incluso na biblioteca de trabalho *serial_lib*. No início da descrição da entidade *interface_serial* realiza-se um mapeamento das variáveis do módulo UART para que sejam utilizados posteriormente. Note as variáveis do módulo *package* podem ser associadas nesta etapa a outras variáveis definidas no projeto com identificadores distintos. A variável *caracter_recebido* do módulo *package*, por exemplo, foi associada a outra local chamada *byte_recebido*. Este recurso possibilita, independentemente da declaração do módulo *package*, trabalhar-se no projeto com identificadores definidos a critério do usuário, tornando o código mais compreensível.

8.2.2 - Corpo do Módulo

O corpo do módulo (*package body*) deve conter o comportamento do módulo *package*, a partir das variáveis e funções definidas na declaração do módulo.

Existe uma situação onde o corpo do módulo *package* não precisa ser definido, que é no caso de módulos que são utilizados apenas para definição de tipos, variáveis e constantes globalmente.

As variáveis e constantes definidas na declaração de módulo *package* não precisam ser reescritas, uma vez que o corpo do módulo já as reconhece automaticamente.

Todos os itens tipo funções contidos na declaração do módulo *package* devem ser repetidos no corpo do módulo. É importante principalmente notar que a declaração destas funções devem estar coincidindo integralmente, ou seja contendo suas variáveis de entrada e saída, na mesma ordem, com os mesmos identificadores e tipos.

A seguir um exemplo completo de módulo *package*:

```
package conversor is
    function converte_7seg (byte_entrada : byte) return bit_vector (0 to 7);
end package conversor;

package body conversor is
  begin
    function converte_7seg (byte_entrada : byte) return bit_vector (0 to 7) is
            begin
                case byte_entrada is
                    when 0  =>  convertido := B"11111100";
                    when 1  =>  convertido := B"01100000";
                    when 2  =>  convertido := B"11011010";
                    when 3  =>  convertido := B"11111000";
                    when 4  =>  convertido := B"01100110";
                    when 5  =>  convertido := B"10100110";
                    when 6  =>  convertido := B"10111110";
                    when 7  =>  convertido := B"11100000";
                    when 8  =>  convertido := B"11111110";
                    when 9  =>  convertido := B"11110110";
                end case;
                return convertido;
end function converte_7seg;

end package body conversor;
```

Este exemplo monta um módulo *conversor*, que contém uma função chamada *converte_7seg*, a qual realiza a conversão de um número do tipo **byte** para um conjunto de bits segundo padrão de acionamento usado para displays de 7 segmentos;

Dentro de um corpo de módulo *package* pode-se definir novos tipos, variáveis e constantes que serão utilizados localmente (como no exemplo acima a variável *valor_inteiro*). Além disso pode-se definir funções locais que serão úteis para o corpo do módulo.

A palavra-chave **use** é recomendada para as descrições de projetos que devam incluir módulos *packages*. Assim pode-se importar bibliotecas que contenham módulos *packages*, selecionando quais módulos dentro desta biblioteca e até mesmo quais funções dentro de cada módulo serão utilizadas.

Abaixo apresenta-se um exemplo utilizando uma função *byte_para_string*, localizada na biblioteca *string_lib*, para exibição de mensagens em um display de cristal líquido (LCD):

```
...
architecture  gera_leitura of  msg_lcd is
begin
        conversao: process is
                use string_lib.conversor.all;
                variable byte_lido : byte;
                variable string_lcd : byte_string;
        begin
                wait until flag_entrada = '1';
                byte_lido := valor_entrada;
                string_lcd := byte_para_string(byte_lido);
        end process conversao;
end architecture gera_leitura;
```

8.3 - Componentes

Componentes (*component*) em VHDL, como o próprio nome diz, são conjuntos de funções que ditam o comportamento de um dado bloco funcional. Este componente, apropriado para uma dada aplicação, é disponibilizado para uso por outros módulos.

Em itens anteriores foi apresentado um subsistema em VHDL como descrito por duas partes: a primeira é a declaração da entidade e a segunda, o corpo de arquitetura. Para a descrição de um componente procede-se de uma forma similar, exceto que em vez de declaração de entidade se faz uma declaração de componente. O corpo de arquitetura deste deve descrever as instâncias do componente.

8.3.1 - Declaração do Componente

A declaração de componente (*component declaration*) especifica a visão externa do componente em termos de constantes e portas de interface. Uma declaração de componente e bastante similar a uma declaração de entidade, somente desta vez utilizando a palavra-chave **component**. Normalmente em componentes mais genéricos é comum o uso de constantes genéricas a fim de possibilitar a melhor adequação do componente a diferentes tecnologias.

Para exemplificar, a seguir descreve-se a declaração de um componente chamado *flipflop*:

```
component flipflop is
        generic (Tprop, Tsetup, Thold : delay_length);
        port   (clk : in bit;  d : in bit;
                q , nq : out bit);
end  component flipflop;
```

8.3.2 - Instância do Componente

As instâncias do componente (*component instances*) definem a forma de utilização do componente em um projeto. Nesta descrição pode-se atribuir valores às constantes genéricas bem como conectar sinais às portas do componentes.

Sua utilizada é feita de modo análogo a componentes discretos através de ligações de
sinais com suas portas.

Como exemplo irá-se tomar um sistema divisor por 16, baseado em 4 flipflops. Para tanto tem-se a seguinte descrição em VHDL abaixo apresentada. Note a necessidade de se utilizar vários sinais para esta implementação.

```
architecture divisao of contador is
        component flipflop is
                port   (clk : in bit;  clr : in bit;  d : in bit;
                        q  : out bit);
                end  component flipflop;
        signal dout , ndout: bit_vector (0 to 3);
        begin
                conta1 :  component flipflop
                          port map (clk => clk,  d => ndout(1),
                                    q => dout(1), nq => ndout(1));
                conta2 :  component flipflop
                          port map (clk => dout (1), d => ndout(2),
                                    q => dout(2), nq => ndout(2));
                conta3 :  component flipflop
                          port map (clk => dout (2), d => ndout(3),
                                    q => dout(3), nq => ndout(3));
                conta4 :  component flipflop
                          port map (clk => dout (3), d => ndout(4),
                                    q => saída, nq => ndout(4));
        ...
        end architecture modelo;
```

A representação em diagrama de componentes deste sistema divisor por 16 está ilustrado na figura 8.1.

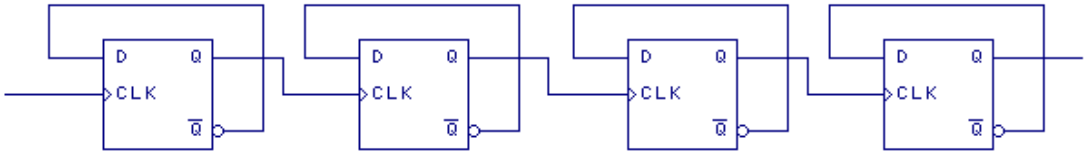

Figura 8.1: Representação do divisor por 16

8.3.3 - Configuração de Componentes

A fim de especificar a tecnologia real que será utilizada no projeto pode-se fazer uso de uma declaração de configuração (*configuration declaration*). Para tanto monta-se uma estrutura de configuração em que pode-se definir para cada instância, quais entidades e modelos de corpo de arquitetura serão utilizados.

Deve-se utilizar a palavra-chave **for** seguida pela instância (ou instâncias separadas por vírgula), o caracter dois pontos (‘**:**’) e o nome do componente. A seguir é descrita a entidade que será utilizada através da palavra-chave **use**. Caso a entidade possua mais de um modelo comportamental, o modelo selecionado deve ser indicado entre parênteses. Por exemplo, a estrutura a seguir:

```
configuration  reg2_gate_level of reg2
        for estrutura
                for bit0: flipflop
                        use entity edge_triggered_Dff (hi_fanout);
                end for;
                for others : flipflop
                        use entity edge_triggered_Dff(basic);
                end for;
        end for;
end configuration reg2_gate_level;
```

define duas configurações para o projeto reg2. A primeira que descreve o elemento *bit0* utiliza um flip-flop definido pela entidade *edge_triggered_Dff* com modelo *hi_fanout*. A segunda que descreve os demais (**others**) elementos utiliza flip-flops definidos pela entidade *edge_triggered_Dff* com modelo *basic*. Para a definição de uma configuração global para o projeto pode-se utilizar a palavra-chave **all**.

Uma configuração feita pode ser compartilhada através do emprego do comando **use configuration** seguida pelo nome da configuração desejada. Isto permite que outras instâncias possam seguir uma mesma configuração definida anteriormente. Como ilustração, observe a estrutura a seguir:

```
for flag_reg : reg2
        use configuration reg2_gate_level;
end for;
```

que utiliza a mesma configuração definida anteriormente *reg2_gate_level*, agora para o elemento *flag_reg*.

É possível ainda efetuar-se a configuração de um componente no seu próprio corpo de arquitetura, usando as estruturas apresentadas logo no início das suas declarações. Isto pode ser melhor visto no exemplo abaixo:

```
architecture modelo of counter is
        use work. counter_types.digit
        signal segundos_clk , reset_meia_noite: bit;
        signal unid_segundos, dez_segundos : digit;
        begin
            segundos :
                    configuration work.counter_down_to_gate_level
                        port map (clk => segundos_clk, clr => reset_meia_noite,
                                    q0 => unid_segundos, q1 => dez_segundos);
        ...
        end architecture modelo;
```

ÍNDICE REMISSIVO

N

O

P

R

S

T

V

W