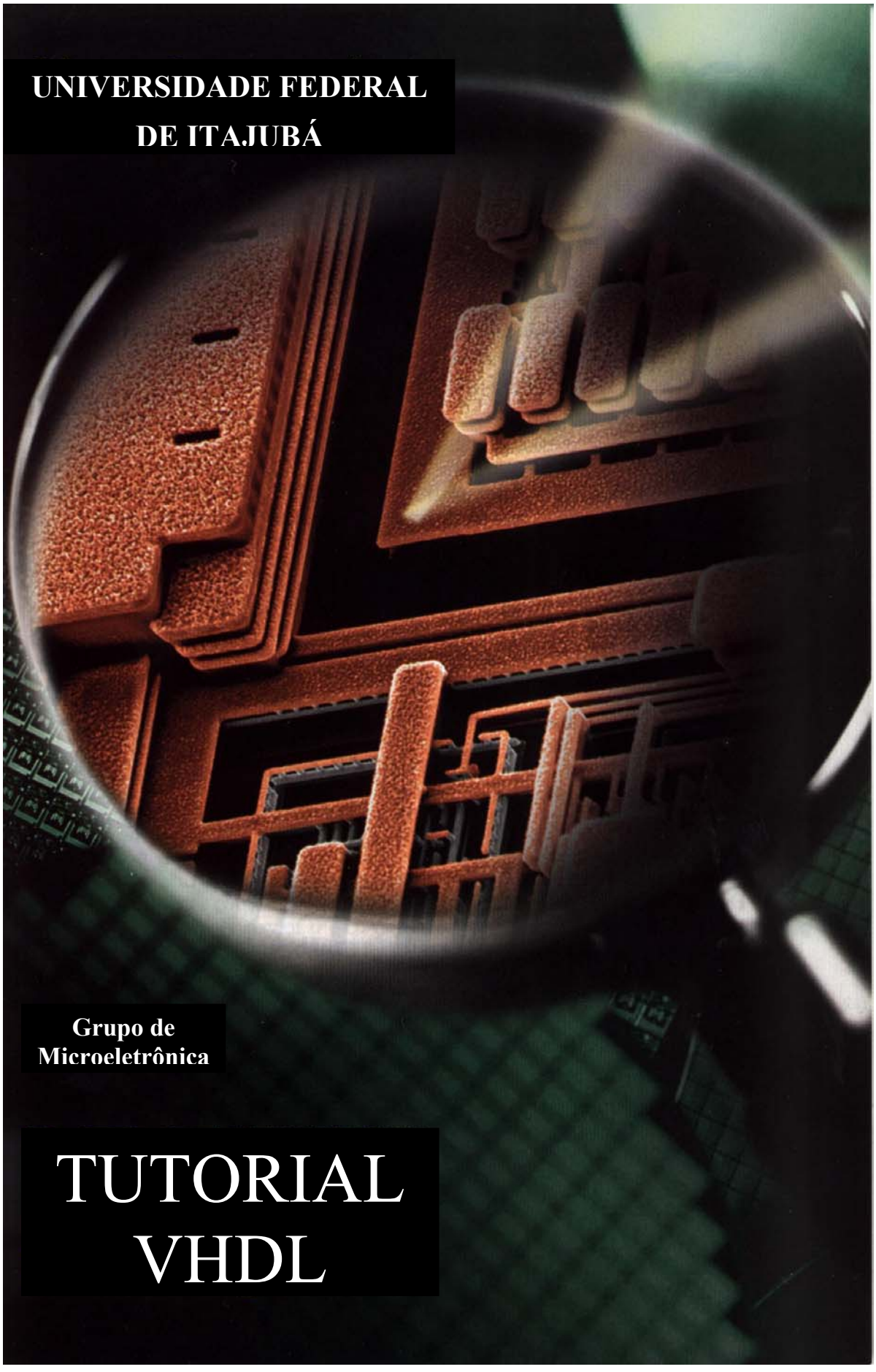

UNIVERSIDADE FEDERAL DE ITAJUBÁ

Grupo de Microeletrônica

TUTORIAL VHDL

TUTORIAL VHDL

I INTRODUÇÃO

Não é possível projetar sistemas digitais sem entender alguns blocos básicos, tais como portas lógicas e flip-flops. A maioria dos circuitos digitais baseados em portas lógicas e flip-flops é normalmente projetada a partir de equações Booleanas. Várias técnicas foram desenvolvidas para otimizar este procedimento, incluindo a minimização de equações para um uso mais racional de portas lógicas e flip-flops.

Pela técnica Booleana deve-se obter uma expressão para cada entrada dos flip-flops e dos blocos lógicos. Isto torna esta técnica impraticável para projetos maiores, que contenham muitos componentes devido ao grande número de equações. Teoricamente qualquer sistema pode ser representado por equações Booleanas, apesar de proibitivo para grandes circuitos. A Figura 1 ilustra a composição de um circuito pela técnica Booleana.

Os projetos desenvolvidos por esquemático ampliaram as capacidades da técnica Booleana por permitir o uso de portas lógicas, flip-flops e sub-circuitos. Estes sub-circuitos por sua vez podem ser compostos por portas lógicas e outros sub-circuitos, e assim sucessivamente podendo formar vários níveis hierárquicos. Muitos projetistas preferem esta técnica, pois ela é visualmente mais atrativa e se tem uma visão mais clara do relacionamento entre os vários blocos.

Nestas técnicas o sistema é normalmente especificado como uma interconexão de elementos, e não na forma de comportamento do sistema. Infelizmente, o projetista recebe as especificações de um sistema na forma de comportamento desejado, isto é, o que o sistema deve fazer sob certas condições. Adicionalmente, mesmo utilizando-se esquemáticos, o projeto começa a ficar proibitivo para circuitos com muitas portas lógicas.

A maior dificuldade nos métodos tradicionais de projeto é a conversão manual da descrição do projeto em um conjunto de equações Booleanas. Esta dificuldade é eliminada com o uso de linguagens de descrição de hardware – *HDL* (*Hardware Description Languages*). Pode-se, por exemplo, a partir de uma tabela verdade, ou da descrição de uma máquina de estado, implementar um circuito usando-se *HDL*. Dentre as várias *HDL*s, as mais populares são *VHDL, Verilog* e *Abel*.

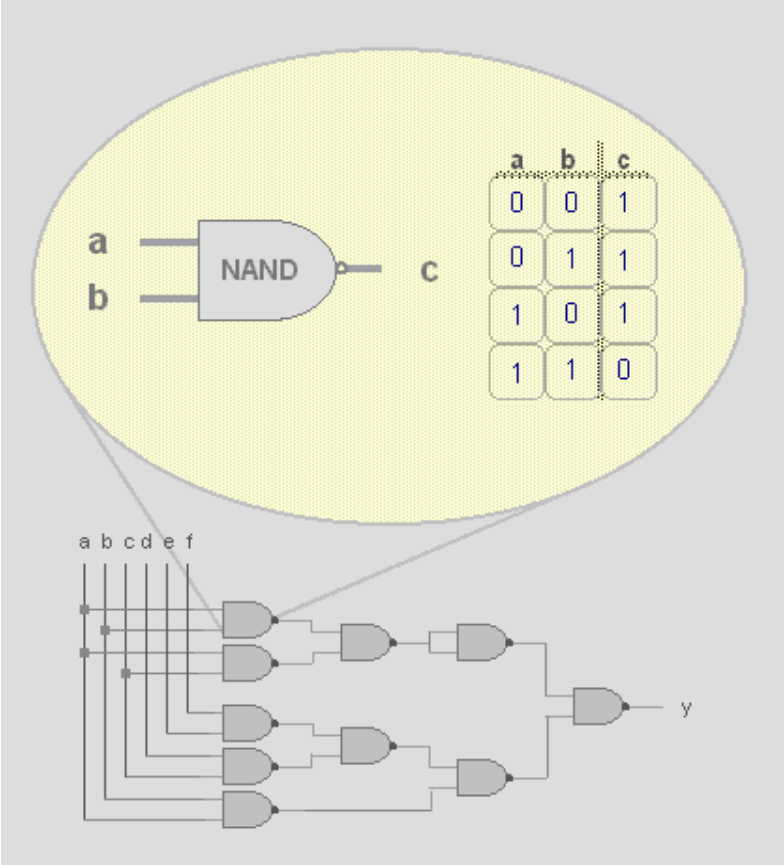

Figura 1 - Composição de um Circuito pela Técnica Booleana.

A letra V da palavra VHDL significa *Very High Speed Integrated Circuit* (Circuito Integrado de Alta Velocidade) e as demais letras, HDL significam *Hardware Description Language*. Esse nome complicado foi criado pelo departamento de defesa dos Estados Unidos, que foi a primeira instituição a reconhecer as suas vantagens. O VHDL foi criado visando simulação, modelagem e documentação, mas acabou recebendo mais tarde a possibilidade de síntese, com o objetivo de se automatizar o projeto de circuitos.

O VHDL ganhou popularidade fora do ambiente militar, graças ao IEEE (*Institute of Electrical and Electronics Engineering)* que estabeleceu padrões para tornar a linguagem universal. Com as ferramentas atuais, pode-se especificar um circuito a partir de seu comportamento ou de sua estrutura, em vários níveis.

II ESTRUTURA

Todo sistema necessita de uma interface com o mundo externo. Em VHDL esta interface é a *entity*, e é fundamental para todo e qualquer sistema.

Para se atingir a funcionalidade desejada, os dados devem sofrer transformações dentro do sistema. Esta parte interna do sistema responsável pelas transformações dos dados é chamada de corpo ou *architecture*. Qualquer que seja o sistema, independente de sua complexidade, necessita de uma interface (*entity*) e de um corpo (*architecture*).

Algumas vezes, alguns sistemas necessitam de funcionalidades adicionais, que são conhecidas como *package*.

II.1 *Entity*

A *entity* é a parte principal de qualquer projeto, pois descreve a interface do sistema. Tudo que é descrito na *entity* fica automaticamente visível a outras unidades associadas com a *entity*. O nome do sistema é o próprio nome da *entity*. Assim, deve-se sempre iniciar um projeto em VHDL pela *entity*. Como exemplo, considere a Figura 2, onde há 8 linhas de transmissão associadas ao sistema A, que é formado por três sub-sistemas; B, C e D. Somente as linhas TL_1, TL_2, TL_3, TL_7 e TL_8 serão descritas na *entity* do sistema A. As demais linhas não são necessárias, pois são internas ao sistema.

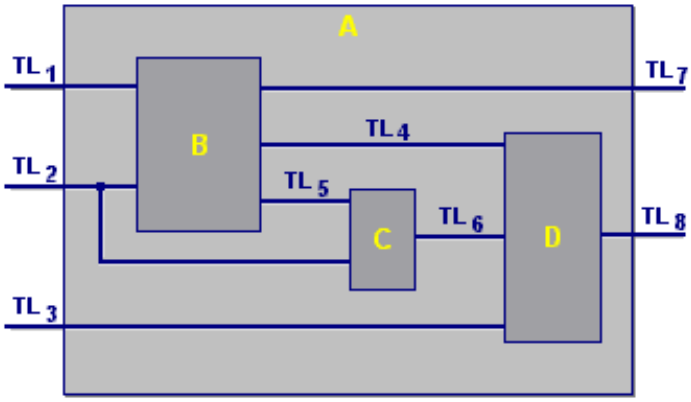

Figura 2 - Sistema Genérico.

A *entity* é composta de duas partes; *parameters* e *connections*. *Parameters* refere-se aos parâmetros vistos do mundo externo, tais como largura de barramento e freqüência de operação, e que são declarados como *generics*. *Connections* refere-se a onde e como ocorre a transferência de informações para dentro e fora do sistema, e são declarados por *ports*.

II.2 *Architecture*

A *entity* de um sistema é tão importante que a *architecture* é especificada na forma de *architecture of entity*. Um sistema pode ser descrito em termos de funcionalidade, isto é, o que o sistema faz, ou em termos de estrutura, isto é, como o sistema é composto.

A descrição funcional especifica as respostas nas saídas em termos das excitações aplicadas nas entradas. Neste caso não há nenhuma informação de como o sistema deverá ser implementado. A descrição estrutural, por sua vez, especifica quais componentes devem ser usados e como devem ser ligados. Esta descrição é mais facilmente sintetizado, porém exige mais experiência do projetista.

Desta forma, pode-se ter várias *architectures* capazes de implementar um mesmo circuito.

Uma *entity* pode ser formada por mais de uma *architecture*!

II.3 *Package*

Quando for necessário utilizar algo não definido nas bibliotecas do VHDL padrão, faz-se uso do *package*. A única restrição é que o package deve ser previamente definido, antes do inicio da *entity*. O uso do *package* é feito por meio de duas declarações: *library* e *use*.

Dos vários *packages* existentes, o mais conhecido e usado é o *STD_LOGIC_1164* da IEEE que contém a maioria dos comandos adicionais mais usados em VHDL. O uso deste *package* é dado por:

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

III SINAIS

Os sinais são de vital importância em virtualmente todos sistemas eletrônicos, podendo transmitir dados internamente ou externamente ao sistema, assumindo assim um papel muito importante em VHDL. Os sinais externos são apresentados na *entity* e os sinais internos são apresentados na *architecture*.

Os sinais podem ser uma linha (transmissão de um sinal por vez) ou um barramento, também chamado de vetor (transmissão de várias informações simultaneamente). Em VHDL

estes sinais são chamados de *bit* e *bit_vector*, respectivamente. No caso de *bit_vector*, a ordem dos bits é de vital importância. Como exemplo, se o bit7 for o mais significativo e o bit0 o menos significativo, em VHDL isto seria representado por *bit_vector(7downto0)*.

Os sinais externos são apresentados na *entity* pelo comando *port*. Os *ports* funcionam como canais dinâmicos de comunicação entre a *entity* e o ambiente. Cada sinal deve ter nome e tipo únicos. A direção do sinal, que é de vital importância, também deve estar presente e pode ser de entrada (*input*), saída (*output*) ou bidirecional (*inout*). A forma de uso do comando *port* é dada por:

```
port ( nome_port : modo tipo_port )
```

Tem-se o nome do *port*, seguido por dois pontos, o modo ou direção do sinal, o tipo do *port*, seguido opcionalmente por um valor inicial (precedido pelo símbolo ":=") e também opcionalmente por algum comentário. A listagem a seguir ilustra um exemplo do comando *port*. Observe que neste caso há mais de um *port*, e estão separados por ponto-e-vírgula.

```
port (   RESULT  :  inout   bit_vector (0 to 7);
         z       :  in      bit;
         EXTBUS  :  out     bit_vector (4 downto 0)
      );
```

Por outro lado, os sinais internos são apresentados na *architecture,* pelo comando *signal*. A listagem a seguir ilustra um exemplo do comando *signal.*

```
signal x, y : bit
```

Observe que não há a necessidade de se especificar o modo de um *signal* (*in, out* ou *inout*), pois é interno ao sistema.

A "visibilidade" ou disponibilidade dos sinais depende do local onde foi declarado. Um sinal declarado em um *package* é visível em todos os projetos que usam este *package*. Um sinal declarado como *port* de uma *entity* é visível em todas *architectures* desta *entity*. Já um sinal declarado como parte de uma *architecture* só é visível internamente a esta *architecture*. Finalmente, se um sinal for declarado dentro de um bloco em uma *architecture*, só será visível dentro deste bloco.

IV INTERFACES

O nome de uma *entity*, que é o seu identificador serve basicamente para documentação, e assim, deve-se preferencialmente usar nomes associados com a descrição do sistema. O nome pode ser composto por letras e números, sendo que deve iniciar por uma letra.

Toda *entity* deve terminar com o comando *end*. Para se evitar erros, o comando *end* deve ser seguido do nome da *entity*.

IV.1 Comentários

Todo projeto em VHDL deve apresentar uma boa documentação. Isto pode ser feito pela inclusão de comentários ao longo da descrição. Os comentários em VHDL são iniciados por dois traços "--" e terminam ao final da linha. Um boa prática também é a inclusão no início da listagem VHDL da descrição do projeto, indicação de bibliotecas, nomes dos autores e qualquer outra informação relevante. Os comentários são mais relevantes ao nível de sistema. A seguir é apresentado a descrição de uma ULA genérica, conforme ilustrada na Figura 3. Observe os comentários em cada linha.

```
entity ULA is
   port  (
         In1      :  in   bit_vector (3 downto 0); -- operando 1
         In2      :  in   bit_vector (3 downto 0); -- operando
         Sel      :  in   bit_vector (3 downto 0); -- seleção de operação
         Ci       :  in   bit;                     -- carry in
         Modo     :  in   bit;                     -- modo (aritmético/lógico)
         Out      :  out bit_vector (3 downto 0); -- resultado da operação
         Co       :  out bit;                      -- carry out
         Igual    :  out bit;                      -- igualdade de In1 e In2
          );
end entity ULA;
```

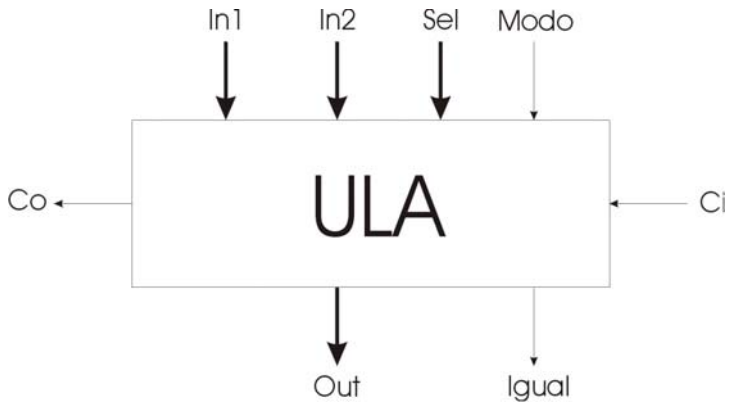

Figura 3 - ULA Genérica.

IV.2 Comando *Generic*

O comando *generic* serve para descrever o valor de uma constante, que por sua vez, serve para estabelecer valores de temporizarão, largura de barramentos, número de bits em somadores e comparadores, etc. O comando *generic* é posicionado dentro da *entity*, normalmente antes dos *ports*. O comando *generic* consiste dos seguintes elementos:

Nome da *generic* seguido por dois pontos,

Tipo da *generic*,

Opcionalmente, o valor da *generic* precedida por **:=**.

A listagem a seguir ilustra o uso de *generic*:

```
generic  ( BusWidth  :  integer := 8;
           MaxDelay  :  time    :=20 us );
```

IV.3 Exercícios

1. Quais as finalidades de *architecture* e *entity*?
2. Obtenha a *entity* do circuito presente nas Figura 1 e Figura 2.
3. Obtenha a listagem do *signal* da Figura 2.
4. Obtenha a *entity* de um contador de 4 bits. Utilize o conceito de *bit_vector* para as saídas.

V ESTRUTURAS PARA DESCRIÇÃO DE COMPORTAMENTO

V.1 *Type*

Um dos tipos de estruturas de dados em VHDL é o *scalar type* (tipo escalar) ou *scalar*. O *scalar type* não tem elemento ou estrutura interna, e todos seus valores são ordenados e estão em uma certa faixa, ou são explicitamente listados. O VHDL já tem previamente definido alguns *scalar type*, que são apresentados na Tabela 1.

Tabela 1 - Tipos Escalares em VHDL.

Tipo	Exemplo	Observação
Boolean	True, False	Diferente de Bit
Character	'0', '+', 'A', '\'	Letras, Números e Caracteres
Integer	-12, 0, 347557	Números inteiros (de -2147483647 a 2147483647)
Real	0.0, 1.0001, -1.0E-5	Números em ponto flutuante (de –1.0E308 a 1.0E308)
Bit	'0', '1'	Valores lógicos '0' e '1'

O usuário também pode definir algum outro tipo escalar, chamado *user defined type*. Considere como exemplo a condição em que se deseja projetar uma máquina de estados, com 4 estados distintos; *esperar, buscar, decodificar e executar*. A declaração deste tipo seria:

```
type MEstado is (Esperar, Buscar, Decodificar, Executar);
```

Outros exemplos de definidos pelo usuário seriam:

```
Type dedos is range 1 to 10
Type dedinho is range 10 downto 1
Type pos_neg is range -1 to 1
```

Outro tipo é o *physical type* que serve para especificar valores físicos. Entretanto, este tipo é previamente definido em VHDL somente para tempo. A listagem a seguir ilustra a descrição do tipo *time*.

```
type time is range –2147483647 to 2147483647
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hour = 60 min;
end units;
```

V.2 *Predefined Arrays*

Estruturas regulares consistindo de elementos do mesmo tipo são chamadas de *Arrays*. Há dois tipos de *arrays* pré-definidos em VHDL; *bit-vector* (elementos do tipo *bit*) e *string* (elementos do tipo *character*). Um único elemento, em ambos tipos de *arrays* é especificado por **‘** (apóstrofe), e dois ou mais elementos são especificados por **“** (aspas). A listagem a seguir ilustra a declaração de um *array* e as duas maneiras de se especificar seus elementos.

```
Signal BusDados : bit_vector (7 downto 0);

BusDados = ‘00110101

BusDados(7) = ‘0’
BusDados(6) = ‘0’
BusDados(5) = ‘1’
BusDados(4) = ‘1’
BusDados(3) = ‘0’
BusDados(2) = ‘1’
BusDados(1) = ‘0’
BusDados(0) = ‘1’
```

V.3 *User-Defined Array*

Os *arrays* predefinidos são unidimensionais (também chamados de vetoriais), entretanto, as vezes se torna necessário definir um *array* de dimensões maiores. Considere

como exemplo de aplicação uma memória. São necessárias duas dimensões para se especificar uma memória, isto é, número de posições e número de bits em cada posição. A descrição de uma memória de 2Kbytes de 8 bits seria:

signal memoria2K8 : array (0 to 2047) of bit_vector (7 downto 0);

VI EXPRESSÕES E OPERADORES

VI.1 Operadores Lógicos

Como os sinais em VHDL são tipicamente lógicos, os operadores lógicos são os mais usados. Os operadores *and, or, nand, nor, xor* e *xnor* exigem dois operandos e o operador *not* necessita de apenas um. Essas operações lógicas são definidas para os tipos *bit, boolean* e *bit_vector*, e exige-se que os operandos sejam do mesmo tipo. No caso de vetores, estas operações são efetuadas bit a bit.

VI.2 Deslocamento

As operações de deslocamento são restritas a *arrays*, cujos elementos devem ser *bit* ou *boolean*. Estas operações exigem dois operandos. Um sendo o *array* e o outro, do tipo *integer,* que determina o numero de posições a serem deslocadas. Se o valor do segundo operando for negativo, o sentido do deslocamento fica invertido.

As operações de deslocamento, conforme indicadas na Figura 4 são:

sll: *shift left logical* (deslocamento lógico a esquerda),

srl: *shift right logical* (deslocamento lógico a direita),

sla: *shift left arithmetic* (deslocamento aritmético a esquerda),

rla: *shift right arithmetic* (deslocamento aritmético a direita),

rol: *rotate left logical* (rotacionamento lógico a esquerda),

ror: *rotate right logical* (rotacionamento lógico a direita).

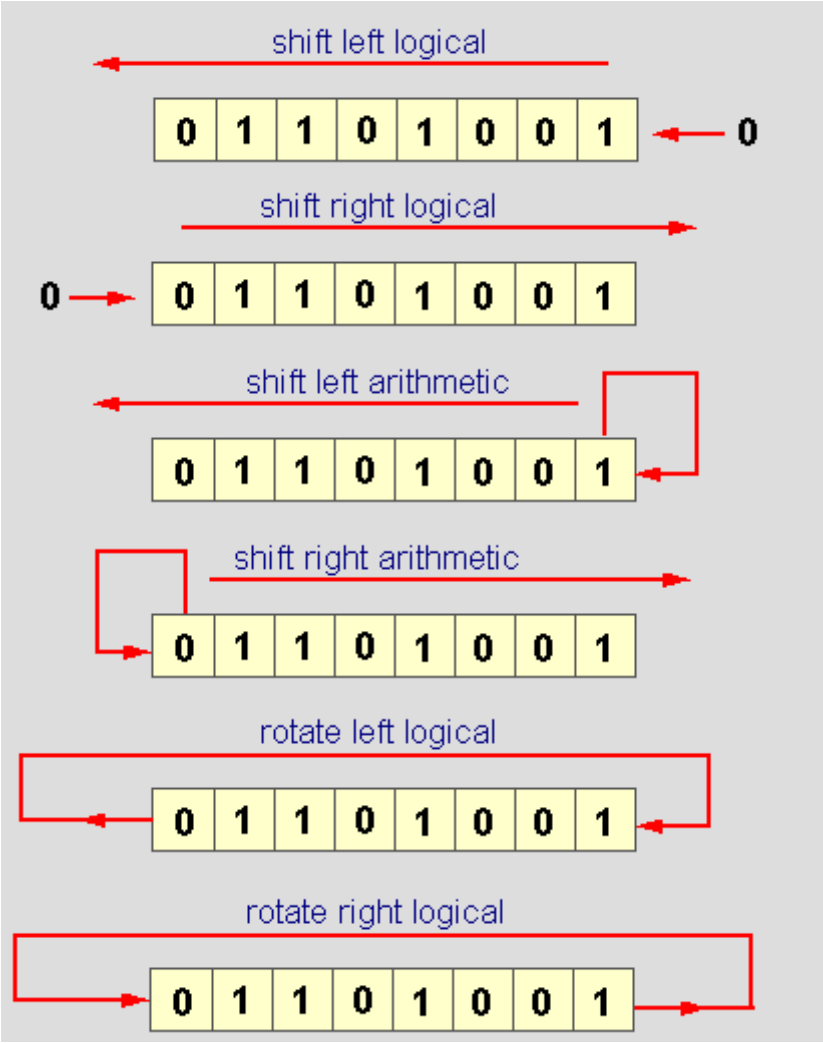

Figura 4 - Operações de Deslocamento.

VI.3 Operadores Numéricos

As operações de adição (+), subtração (-), multiplicação (*), divisão (/), modulo (*mod*), valor absoluto (*abs*), resto (*rem*) e potência (**) são aplicados aos tipos *integer* e *real*. Os operandos devem ser do mesmo tipo. O tipo *time* exige que os operandos sejam iguais nas operações de adição e subtração, mas um operando *time* pode ser multiplicado ou dividido por um operando *integer* ou *real*.

VI.4 Comparações

As comparações entre dois objetos podem ser igual (=), diferente (/=), menor (<), menor ou igual (<=), maior (>), e maior ou igual (>=). Os objetos comparados devem ser do

mesmo tipo e podem ser *boolean, bit, character, integer, real, time, string* ou *bit_vector*, entretanto o resultado é sempre *boolean* (*true* ou *false*). Ao se comparar dois vetores de tamanhos diferentes, eles são primeiramente justificados no tamanho do menor.

VI.5 Concatenação

Concatenação é uma forma conveniente de se unir dois ou mais vetores, criando um novo vetor, cujo tamanho é a soma dos vetores dos operandos. Os vetores devem ser do mesmo tipo e pode-se unir também um bit a um vetor. A listagem a seguir ilustra a situação em que se une parte de dois vetores e um bit para formar um novo vetor.

```
NovoVetor <= ( Vetor1(0 to 3)
             & Vetor2(3 to  5)
             & Bit1);
```

VI.6 Atribuição de Sinais

Eventualmente os resultados das operações em VHDL devem ser atribuídas às saídas. Isto é feito pelo símbolo <= em que o valor da expressão a direita é atribuído à expressão a esquerda. Para auxiliar na memorização desta operação, basta observar que a seta formada indica o fluxo da informação. A listagem a seguir ilustra alguns exemplos de atribuição.

```
x <= y <= z;
a <= b or c;
k <= '1';
m <= "0101";
n <= m & k;
```

Tendo em vista que nada é instantâneo no mundo real, em VHDL também é possível modelar estes atrasos naturais por meio da expressão *after*. A expressão *after* determina o tempo após o qual uma expressão se torna efetiva. A declaração a seguir ilustra o uso da atribuição de sinal com atraso empregando *after*.

```
luz <= chave after 3 s;
```

VI.7 Atraso Inercial

Atraso inercial é tempo mínimo que um pulso deve durar para que seja aceito por um dado circuito. Esta característica de circuitos reais faz parte do modelo de atrasos em VHDL. Assim, para o caso anterior, observe na Figura 5.a o comportamento do circuito para um pulso de 2s, e na Figura 5.b a mesma análise para um pulso de 4s.

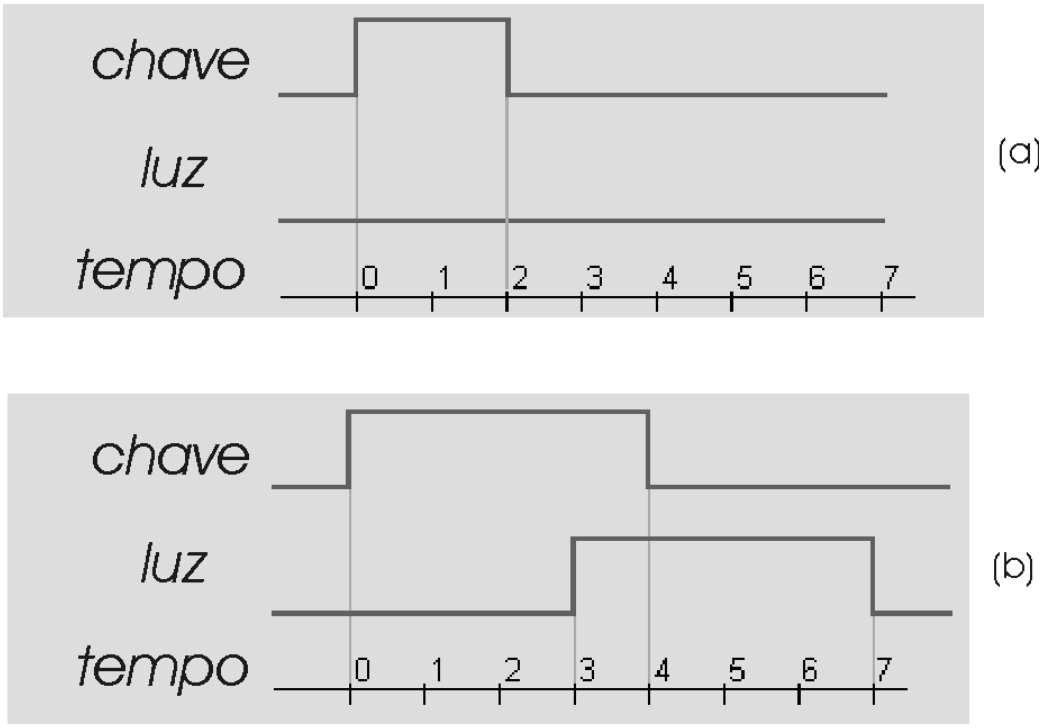

Figura 5 - Efeito de Atraso Inercial para (a) um Pulso de 2s, e (b) um Pulso de 4s.

Como na Figura 5.a o pulso de entrada tem uma duração de apenas 2s, não é o suficiente para vencer o atraso inercial e efetivamente acionar a saída *luz*. Já na Figura 5.b o pulso de entrada tem uma duração de 4s, o que é maior que o atraso inercial e consegue acionar a saída *luz*.

Outro problema do mundo real é o atraso que ocorre em linhas de transmissão. Neste caso não há uma duração mínima do pulso, mas sempre há um atraso no transporte do sinal. Isto é modelado em VHDL pela inclusão do termo *transport* antes da especificação de valor e atraso, conforme ilustrado a seguir.

```
luz <= transport chave after 3 s;
```

Para melhor ilustrar este conceito, considere o exemplo anterior, agora usando este conceito de atraso de transporte, conforme ilustrado na Figura 6.

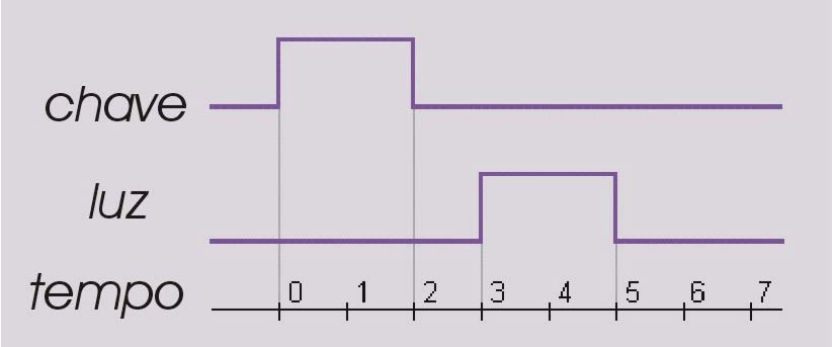

Figura 6 - Comando *Transport* Aplicado em um Pulso de 2s.

VI.8 Constantes

As constantes desempenham o mesmo papel que as *generics*. Ao contrário da *generic* que é declarada na *entity*, a *constant* é declarada na *architecture*. A declaração de uma *constant* consiste do comando *constant* seguido do nome da constante, dois pontos, tipo da constante e o símbolo "**:=**", conforme ilustrado a seguir.

```
constant N : integer := 123
```

A *constant* serve para atribuir valores locais, tornando o projeto mais simples, mais fácil de se ler e de efetuar manutenções/alterações. Como a *constant* é declarada na *architecture*, não pode ser usada para definir dimensões de vetores. Da mesma forma, a *constant* só pode ser usada dentro da *architecture* na qual foi declarada. Por outro lado, a *generic* que é declarada na *entity* é usada por todas as *architectures*.

VI.9 Exercícios

5. Diferencie atraso inercial e atraso de transporte. Cite exemplos.
6. Atribua a seqüência "01010101" a um vetor A, a seqüência "11110000" a um vetor B. Faça um sll no vetor A, um ror em B. A seguir concatene em D os 4 bits mais significativos de A, os três bis menos significativos de B e o valor de uma variável C como sendo o último bit.

VII DESCRIÇÃO DE COMPORTAMENTO (*PROCESS*)

O comportamento, ou funcionalidade, de um sistema corresponde a uma lista de operações a serem executadas para se obter um determinado resultado. *Process* é o modo formal de se fazer uma lista seqüencial dessas operações, e tem um formato estruturado. Existem algumas regras para se fazer uma descrição *process*.

Inicialmente, deve-se especificar que a listagem corresponde a um *process*, e isto é feito pelo comando *process*, que fica posicionado após o nome e os dois pontos. Da mesma forma este nome deve ser colocado após a declaração *end of process*. Para separar as operações seqüenciais de comandos tais como *variable* ou *constant*, a declaração *begin* é usada para marcar o inicio da listagem das operações seqüenciais

Ao contrário de linguagens de programação convencionais, processos descritos em VHDL não terminam. Após a execução do último comando, *process* é executado novamente desde o primeiro comando.

Circuitos eletrônicos em geral operam em modo contínuo, aguardando ocorrência de certas condições de entrada, executando tarefas, suspendendo as atividades quando se completa o serviço, e novamente retomando o serviço na ocorrência de certas condições de entrada.

Em VHDL suspende-se um processo pelo comando *wait*. Este comando pode suspender o processo incondicionalmente, ou apresentar uma lista de condições a serem satisfeitas para se retomar a operação normal. O comando *wait* pode aparecer em três formas distintas.

Na primeira forma, ou *wait for* o processo é suspenso por um certo intervalo de tempo, na segunda, ou *wait until* o processo é suspenso até que uma certa condição seja verdadeira devido a mudança dos sinais envolvidos nesta condição. Observe que se não houver mudança nos sinais, o processo continua suspenso. Na terceira forma, ou *wait on* o processo é suspenso até que um evento da lista de sinais ocorra. Isto é, o processo continua quando ocorrer qualquer uma das condições de sua lista. Eventualmente pode ocorrer uma mistura destas formas. As duas listagens a seguir ilustram exemplos destes casos.

```
wait for 10 ns;
wait for periodoCLK  / 2;
wait until CLK = '1';
```

```
wait until CE and (not RST);
wait until dado > 16;
wait on CLK;
wait on dado;
wait on dado until CLK = '1';
wait until CLK='1' for 10 ns;
```

Como já mencionado, os comandos de um *process* são executados constantemente e ciclicamente. Pode parecer, portanto, que não faz diferença de onde se coloca um *wait*. Na verdade, é de vital importância o seu posicionamento adequado. Se uma instrução *wait* for colocada no início de um *process*, nada ocorre até que as condições do *wait* sejam satisfeitas. Se este mesmo *wait* for colocado no final de um *process*, todas os comandos anteriores ao *wait* serão executados.

A forma *wait on* é usado com mais freqüência que as demais. Ao invés de se utilizar o *wait on* no final de um *process*, pode-se usar o conceito de *sensitivity list*, ou lista de sensitividade, que consiste simplesmente em colocar as condições do *wait on* ao lado do comando *process* (entre parênteses), e se obtém os mesmos resultados. Neste caso, o *process* não pode conter nenhum *wait*. A listagem a seguir ilustra o uso de *process* para se implementar um multiplexador 2x1. Observe que os sinais *A, B* e *Sel* foram colocados na *sensitivity list*.

```
MUX2x1: process (A, B, Sel)
   constant High : Bit := '1';
begin
    y <=A;
    if (Sel = '1') then Y <= B;
    end if;
end process MUX2x1;
```

Deve-se ressaltar que neste caso, os comandos do *process* são executados uma vez, e o processo fica suspenso. Ao haver mudança em qualquer uma das variáveis do *wait on* (listados no final do *process*) ou variáveis listadas ao lado do comando *process* (*sensitivity list*), o *process* é executado mais uma vez.

A descrição de comportamento em VHDL é empregada para descrever a reação da saídas às entradas. Entretanto, qualquer novo valor atribuído a uma variável só passa a valer

efetivamente quando o processo é suspenso. Adicionalmente, somente a última atribuição a uma variável é válida. Como entradas e saídas são sinais, qualquer variação de saída é tido como variação de sinal e novamente o processo é executado. Para entender melhor este conceito, considere o exemplo listado a seguir.

```
process (sinal1, sinal2)
begin
   sinal2 <= sinal1 + 5;
end process;
```

Uma vez iniciado o processo, pode-se ver que o *Sinal2* recebe o valor de *Sinal1* acrescido de 5. Como houve uma variação em *Sinal2*, o processo é executado novamente. Desta vez, como não há variação em nenhuma variável (*Sinal2* continua com o valor de *Sinal1* acrescido de 5), o *process* é suspenso. Neste momento *Sinal2* efetivamente recebe o novo valor.

Infelizmente há sérias limitações no *process*. Não se pode declarar sinais dentro do *process*. Outra grave limitação é que somente o último valor atribuído à um sinal se torna efetivo, e ainda assim, somente quando o processo é suspenso. Considere, como exemplo, a descrição apresentada a seguir. Considere o valor inicial de todas os sinais como sendo "1". Quando D mudar para 2, o processo é iniciado. Quais seriam os valores quando o processo for suspenso?

```
process (C, D)
begin
   A <= 2;
   B <= A + C;
   A <= D + 1;
   E <= A * 2;
end process;
```

Os valores finais das variáveis são:

```
A = 3;   B = 2;   C = 1;   D = 2;   E = 2;
```

Esta limitação do *process* é eliminada com o uso de *variables*, que podem ser declaradas dentro do *process*. O valor atribuído a uma *variable* é válido imediatamente após a sua atribuição, e pode receber várias atribuições no mesmo *process*. A listagem a seguir ilustra como se declara e utiliza as *variables*.

```
process (C, D)
variable At, Bt, Et: integer := 0;
begin
   At := 2;
   Bt := At + C;
   At := D + 1;
   Et <= A * 2;
   A <= At;
   B <= Bt;
   E <= Et;
end process;
```

Considere os mesmos valores iniciais do exemplo anterior. Observe que a atribuição de valores de uma variável é feita utilizando-se "**:**=". Compare os resultados deste exemplo, apresentados a seguir os do exemplo anterior. Os valores finais das variáveis são:

A = 3; B = 3; C = 1; D = 2; E = 6;

Para um melhor entendimento, a Tabela 2 apresenta uma comparação entre *signal* e *variable*, observe a tabela a seguir que ilustra suas semelhanças e diferenças.

Tabela 2 - Comparação entre *Variables* e *Signals*.

	Signal	Variable
Declaração	Em *architecture* ou como *port* na *entity*.	Dentro de *process*.
Atribuição	Recebe valor atribuído na suspensão do *process*. Somente a última atribuição é válida. Ex: A<= B + C	Recebe valor atribuído imediatamente. Toda atribuição é válida. Ex: A:= B + C
Atraso	Inercial e Transporte	Não há

VII.1 Controle da Seqüência

Como os *processes* representam sistemas reais, estão sujeitos a constantes mudanças externas e assim, normalmente não executam todas as instruções. Estas mudanças externas podem ser modeladas em VHDL pelos comandos condicionais.

VII.1.1 Comando *if then*

Algumas operações são executadas somente se certas condições são satisfeitas. Estas operações são chamadas de condicionais e pode-se especifica-las como:

If *condição* then *operação*

Se a condição for verdadeira, então a lista de operações é executada. As operações desta lista devem ser estar separadas por ponto-e-vírgula. Ao termino destas operações deve ser colocado o *end if*. Considere como exemplo o *process* de um flip-flop D ativo por rampa de subida listado a seguir.

```
Dff : process (D, CLK)
   begin
     if rising_edge (CLK)
         then
            Q <= D;
      endif;
end process Dff;
```

VII.1.2 Comando *if then else*

O comando *if then* é bem simples e torna-se necessário incorporar alguma modificação para se obter maior flexibilidade. Este comando pode ser modificado para:

If *condição* then *operação_1* else *operação_2*

Assim se a condição for verdadeira, a operação_1 é executada, caso contrário, a operação_2 é executada. Na verdade, operação_1 e operação_2 podem ser várias operações separadas por ponto-e-vírgula.

Para uma maior flexibilidade ainda, este comando pode incluir novos testes através do *elsif*. A listagem a seguir ilustra como exemplo o *process* de um flip-flop tipo D ativo por rampa de subida, com linha de *reset* ativa por "1".

```
Dff_rst : process (D, CLK, RST)
   begin
     if RST = 1
         then Q <= '0';
             elseif rising_edge (CLK)
                then
                   Q <= D;
      endif;
end process Dff_rst;
```

VII.1.3 Comando *Case*

O uso do comando *if then elsif* serve para selecionar uma ramificação dentre as várias possíveis, e que pode ficar complicado se o número de opções se tornar maior que três. Para esta situação utiliza-se o comando *case.*

Ao invés de se avaliar uma expressão booleana, o comando *case* verifica as condições de uma expressão discreta ou um *array*. Cada alternativa é composta por:

```
when alternativa => operação
```

Considere como exemplo a listagem a seguir. O circuito apresenta três entradas de seleção (*Modo*), duas entradas de dados (*In1* e *In2*) e uma saída *Out*. Dependendo dos valores presentes nas entradas de seleção, diferentes operações são executadas nos dados. Pode-se observar o recurso de se utilizar a opção *when others* para situações não previstas pelas demais opções.

```
PortaProgramavel : process (Modo, In1, In2)
   begin
      case Modo is
          when “000” => Out <= In1 and In2;
          when “001” => Out <= In1 or In2;
          when “010” => Out <= In1 nand In2;
          when “011” => Out <= In1 nor In2;
          when “100” => Out <= not In1;
          when “101” => Out <= not In2;
          when others => Out <= ‘0’;
      end case;
end process PortaProgramavel;
```

VII.1.4 Comando *While Loop*

O comando *while loop* condicional funciona de forma similar ao comando *if then.* Ele inicia com uma condição lógica, porém tem na última linha um *jump* para o início do loop.

O loop é executado enquanto a condição presente no seu inicio for válida. A condição é verificada e ser for satisfeita, os comandos presentes no loop são executados, caso contrário, o loop é considerado completo e o comando passa para a instrução seguinte ao loop.

Na listagem seguinte faz-se a contagem das transições positivas do sinal de clock desde que o sinal *Nivel* esteja em 1. Observe que como não há *sensitivity list,* o *process* continua sendo executado continuamente.

```
process
   variable conta : integer := 0;
   begin
      wait until Clk = ‘1’;
      while Nivel = ‘1’ loop
          conta := conta + 1;
          wait until Clk = ‘0’;
      end loop;
end process;
```

VII.1.5 Comando *For Loop*

O comando *while loop* repete-se enquanto a condição apresentada no seu início for satisfeita. Algumas vezes pode ser necessário repetir o *loop* por um número específico de vezes. Isto pode ser feito de forma mais conveniente com o comando *for loop*. Este comando não usa nenhuma expressão booleana, mas sim um contador, e desde que o valor do contador esteja em uma certa faixa, o *loop* é executado. Ao término de cada execução do *loop*, o contador recebe o novo valor. O contador não precisa ser declarado e é tratado como uma constante e só existe dentro do *loop*.

A estrutura do comando *for loop* é dada por:

```
[nome :] for variável in faixa
          loop
          comandos do loop
          end loop [nome];
```

A listagem a seguir ilustra um exemplo de uso deste comando.

```
signal Dados : bit_vector (3 downto 0);
signal Uns : integer;
Contar1 : process (Dados)
     variable Numero1s : integer := 0);
     begin
          for temp in 3 downto 0 loop
             next when Dados(temp) = '0';
             Numero1s := Numero1s +1;
        end loop;
          Uns <= Numero1s;
end process Contar1;
```

VII.1.6 Comandos *Next* e *Exit*

Algumas vezes se torna necessário pular alguns comandos do loop e ir diretamente para a próxima interação. Esta situação pode ser obtida pelo comando *next*. As formas deste comando são apresentadas a seguir:

```
next ;
next nome_loop ;
next when expressão ;
next nome_loop when expressão ;
```

Considere como exemplo deste comando, a descrição dada a seguir.

```
other : process (flag)
   variable a, b : integer := 0 ;
        begin
          a := 0 ; b := 3 ;
          alfa : for i in 0 to 7 loop
          b := b + 1 ;
          if i = 5 then next alfa ;
          end if ;
          a := a + b ;
          end loop alfa ;
end process other
```

Dependendo da situação pode ser necessário terminar o loop, que pode ser conseguido pelo comando *exit*. Este comando pode ter um dos seguintes formatos:

```
exit ;
exit nome_loop ;
exit when expressão ;
exit nome_loop when expressão ;
```

A Tabela 3 apresenta dois exemplos equivalentes, usando este comando.

Tabela 3 - Exemplo de Usos Equivalentes do Comando Exit.

```
show1: process (flag)
          variable sun, cnt : integer := 0 ;
          begin
            sum := 0; cnt := 0;
            um : loop
                    cnt := cnt +1 ;
                    sum := sum + cnt ;
                    exit when sum > 100 ;
            end loop um ;
end process ;
```

```
show2: process (flag)
          variable sun, cnt : integer := 0 ;
          begin
            sum := 0; cnt := 0;
            dois : loop
                    cnt := cnt +1 ;
                    sum := sum + cnt ;
                    if cnt > 100 then exit ;
                    end if ;
            end loop dois ;
end process ;
```

VII.2 Exercícios

7. Implemente um multiplexador 4x1 usando dois *processes* distintos.
8. Obtenha o *process* de um flip-flop JK ativo por rampa de descida.
9. Obtenha o *process* de um decodificador 2x4, e de um codificador 4x2.

VIII ARQUITETURAS MULTI-PROCESSO

Apesar de ser fácil e conveniente descrever eventos de forma seqüencial, a realidade nem sempre é assim. Na verdade, por inúmeras vezes, vários eventos ocorrem ao mesmo tempo, ou de forma "concorrente". Da mesma forma, muitas vezes alguns sistemas são concorrentes internamente.

Como o VHDL é uma linguagem hierárquica, é possível especificar sistemas como sendo um conjunto de subsistemas concorrentes, onde cada subsistema é um *process* individual. O nível de detalhes de cada *process* depende das necessidades. Como exemplo, um processador ou como uma porta lógica podem ser especificados como um *process*. Assim, a descrição comportamental de um sistema é dada como sendo um conjunto de *processes* seqüenciais concorrentes.

Uma arquitetura comportamental é composta por um cabeçalho e por um corpo. O cabeçalho especifica que se trata de uma arquitetura e estabelece um nome. No cabeçalho também se indica a qual *entity* pertence. No corpo estão listados os *processes* e seus conteúdos, vale lembrar que mesmo sendo listado seqüencialmente, os *processes* de uma *architecture* são executados de forma concorrente. Visualmente, isto equivale a escrever todos os *processes* em paralelo, o que não é feito por questões de espaço.

Vale ressaltar que quando ocorre a mudança de um sinal, **todos** os *processes* que têm este sinal em sua lista de sensitividade são ativados. Como os *processes* não tem como distinguir entre sinais modificados externamente e sinais modificados internamente, sinais que ativam alguns *processes* podem ser gerados por outros *processes*. Assim, toda vez que um sinal em uma lista de sensitividade é modificado, independente da origem da mudança, o *process* é ativado.

Note que somente sinais podem ser usados para transferir informações entre *processes.* As variáveis são internas aos *processes* e não podem transferir informação entre *processes*.

VIII.1 *Process* Simplificado

Para se especificar uma simples porta lógica (ex: OUT <= A + B), pode-se usar um *process*. Entretanto, para isso seriam necessários três comandos adicionais (cabeçalho do *process*, comando *begin* e comando *end*), o que na verdade seria um desperdício. O VHDL permite que se use *processes* de uma única linha, chamados *concurrent signal assigment*. Este comando de uma única linha pode ser posicionado dentro de uma *architecture*, em paralelo com alguns *processes* e são executados de forma concorrente com os demais comandos. Os comandos comuns são executados na seqüência dada, entretanto os *concurrent signal assignment* são executados de forma concorrente.

Os *processes* têm lista de sensitividade e comandos *wait*, mas o mesmo não ocorre com o *concurrent signal assignment*. Na verdade o VHDL interpreta como lista de sensitividade tudo que esta à direita do símbolo de atribuição (<=) do *concurrent signal assignment*. Da mesma forma, um comando pode ser atrasado pelo uso do comando *after*. A Figura 7 ilustra um circuito multiplexador e a seguir a sua respectiva descrição usando os recursos apresentados.

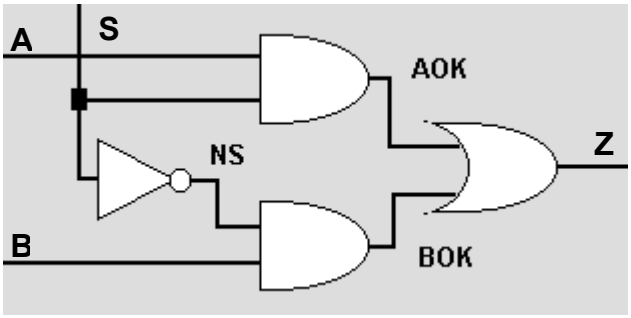

Figura 7 - Estrutura Interna de um Mux 2x1.

```
architecture Portas of Mux2x1 is
signal AOK, BOK, NS : bit;
begin
   AOK <= A and S after 1 ns;
   BOK <= B and NS after 1 ns;
   NS <= not S after 1 ns;
   Z <= AOK or BOK after 1 ns;
end architecture Portas;
```

Algumas vezes se torna necessário utilizar um comando condicional. Como o comando *if then* é um comando seqüencial e restrito a *process*, deve-se usar *conditional signal assignment* que pode ser usado diretamente na *architecture*. Este comando tem sintaxe direta, isto é, um sinal recebe um certo valor quando uma certa condição for atendida. É possível usar *else* "ninhados". As duas listagens seguintes ilustram o uso da *conditional signal assignment* e *process* para executar a mesma função, correspondente ao circuito da Figura 7.

```
architecture Processo of Mux2x1 is
begin
   Mux : process (A, B, S);
   begin
     if S = '1'
          then Z <= A;
          else Z <= B;
      endif;
   end process Mux;
end architecture Process;
architecture Condicional of Mux2x1 is
begin
   Z <= A when S = '1' else
           B;
end architecture Condicional;
```

Quando for necessário selecionar um valor, dentre várias opções, para um sinal, usa-se a estrutura *selected signal assignment*. Esta estrutura não pode ser usada dentro dos *processes*, onde se usa o comando *case*, e da mesma forma, o contrário também não é válido.

As duas listagens a seguir são equivalentes, sendo que a primeira faz uso de *selected signal assignment*, e é equivalente a segunda, que faz uso de *Process*.

```
architecture Selecao of PortaProgramavel is
begin
   with Modo select
      Out <=
          In1 and In2 when “000”,
          In1 or In2 when “001”,
          In1 nand In2 when “010”,
          In1 nor In2 when “011”,
          not In1 when “100”,
          not In2 when “101”,
          ‘0’ when others;
end architecture Selecao;

Architecture Case of PortaProgramavel is
PortaProg : process (Modo, In1, In2)
  begin
      case Modo is
          when “000” => Out <= In1 and In2;
          when “001” => Out <= In1 or In2;
          when “010” => Out <= In1 nand In2;
          when “011” => Out <= In1 nor In2;
          when “100” => Out <= not In1;
          when “101” => Out <= not In2;
          when others => Out <= ‘0’;
      end case;
end process PortaProgr;
end architecture PortaProgramavel;
```

VIII.2 *Drivers* e Atribuição de Sinais

Em um *process*, os sinais recebem os seus novos valores apenas quando o *process* é suspenso. Enquanto o *process* está ativo, os valores que devem ser atribuídos aos sinais ficam armazenados em *drivers*. Quando o *process* é suspenso, o valor dos *drivers* são transferidos aos respectivos sinais. Se durante o *process* houver mais de uma atribuição ao mesmo sinal, somente a última atribuição é válida.

Além do valor presente, os *drivers* também permitem que os sinais tenham valores passados e futuros. Os drivers podem especificar os valores de uma forma de onda de saída esperada para um sinal. Esta forma de onda consiste de transições, que por sua vez consistem

de valores e tempos. O tempo especifica quando um novo valor deverá ser atribuído pela transição. Assim, uma forma de onda pode ser explicitamente especificada como sendo uma seqüência de valores e seus correspondentes atrasos com relação ao mesmo ponto no tempo. Assim, as formas de onda podem ser consideradas como valores projetados dos sinais. Tendo em vista que os simuladores armazenam as transições para cada sinal, tem-se na verdade uma história dos sinais.

A história de um sinal é importante para que se possa determinar se houve mudança no sinal, ou para verificar se ocorreu uma transição positiva ou negativa. A história de um sinal é representada pelos seus atributos, que são informações associadas a um sinal e são atualizadas automaticamente. A Figura 8 ilustra uma forma de onda e a seguir é apresentada a sua respectiva descrição.

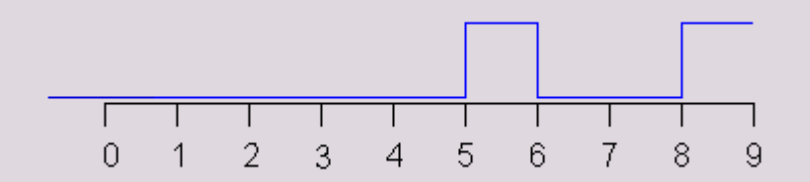

Figura 8 - Forma de Onda Genérica.

```
Exemplo  <=  '0' after 2s,
             '1' after 5 s,
             '0' after 6s,
             '1' after 8 s;
```

A expressão *after* determina o tempo após o qual uma expressão se torna efetiva, tomando como base uma referência de tempo. Se nada for especificado, a referência é a origem (t=0). Já o comando *wait* suspende processo por um tempo determinado, o que permite especificar uma nova referência.

Assim, esses comandos podem ser usados individualmente, ou de forma associada para a geração de formas de onda. O *process* a seguir é usado para a geração da forma de onda da Figura 9. Observe que foi usada uma composição de *after* e *wait*.

```
forma_de_onda : process
    begin
        s <= '0';
        wait for 5ns;
        s <= '1' after 3ns;
        wait for 10ns;
        s <= '0', '1' after 10ns, '0' after 20ns;
        wait;
end process;
```

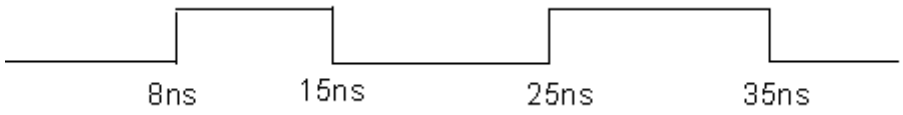

Figura 9 - Forma de Onda Gerada pelo *Process* "forma_de_onda".

A descrição de sinais em VHDL é muito importante quando se pretende montar uma descrição de teste, chamada *test bench*, onde se pode testar o circuito descrito em VHDL.

Existem inúmeros atributos em VHDL, e estes têm inúmeras aplicações. Além dos atributos predefinidos, o usuário pode definir seus próprios atributos. Como exemplo, considere a descrição seguinte que ilustra a detecção de transição de sinal.

```
if CLK'event
  and CLK = '1'
  and CLK'last_value = '0' then ...
```

VIII.3 Sinais com mais de um *Driver*

Quando um sinal tem apenas um *driver* é fácil determinar o seu valor. Entretanto existem situações onde se torna necessário mais de um *driver* para um mesmo sinal. Considere o caso do barramento de um microprocessador onde há várias fontes de dados, tais como o processador, a memória, os dispositivos de entrada/saída e outros. Existem situações onde se torna difícil determinar se somente uma das fontes de sinal estará excitando o sinal ao mesmo tempo. Em alguns casos isto ocorre sempre, entretanto, em outros pode ser necessário misturar os sinais de diferentes fontes, emulando nestes casos, "E" ou "OU" por fios. Nestas situações, é necessário estabelecer um método para se determinar o valor resultante. Em

VHDL, a determinação do valor resultante quando se misturam sinais é conhecida como *resolving*.

Em algumas situações, dependendo da escolha do tipo de sinais, estes não podem ser misturados. Considere que se deseja descrever um sistema de microprocessador, para este caso não é possível utilizar *bit* e/ou *bit_vector*, pois não se tem como misturar "0" e "1". Estes tipos são chamados de tipos *unresolved* e não podem ser usados com sinais de múltiplos *drivers*.

Somente "0" e "1" não são suficientes para sinais de um ou de múltiplos *drivers*, inclusive para o caso dos barramentos. Os circuitos reais exigem alguns outros tipos de dados:
- Algumas vezes não importa o valor de um sinal. Isto é representado por "não-interessa" ou *"don't care*".
- Buffres tri-state apresentam linhas de saída de alta impedância (não são "0" ou "1").
- Ocasionalmente um sistema pode não ter um valor especificado ("*unassigned*") ou não ter um valor conhecido ("*unknown*"), que são diferentes de "não interessa".

Estes e alguns outros tipos de dados comumente usados são especificados pelo tipo *std_ulogic*, definido no *package Std_Logic_1164*. Este *package* também tem os tipos *std_ulogic_vector* que são os seus equivalentes vetoriais. Ambos têm um conjunto de operações lógicas definidas para eles. A letra "u" indica tipos *unresolved*, isto é, não podem ser usados com sinais de múltiplos *drivers*. As listagens a seguir apresentam os valores do tipo *std_ulogic* do *package Std_Logic_1164* para um bit e para barramento.

```
TYPE std_ulogic is
   (     'U', -- Não inicializado
         'X', -- Força a 0 ou 1
         '0', -- Força a 0
         '1', -- Força a 1
         'Z', -- Alta impedância
         'W', -- 0 ou 1 fraco
         'L', -- 0 fraco (para ECL com emissor aberto)
         'H', -- 1 fraco (para Coletor ou Dreno aberto)
         '-', -- Não interessa
   );
TYPE std_ulogic vector IS ARRAY
   ( NATURASL RANGE <> ) of std_ulogic;
```

Convém mencionar que o *std_ulogic* e o *std_ulogic-vector* não são definidos no VHDL padrão e deve-se usar um *package*, como por exemplo o IEEE, tal como ilustrado a seguir. Invoca-se o *package* antes do cabeçalho da *entity*.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

As regras para a mistura de sinais são especificadas na forma de tabela, conhecida como *resolution function*. Esta tabela lista todas as possibilidades de mistura de sinais e o seu respectivo valor final.

Todas operações lógicas disponíveis para *bit* e *bit_vector* são disponíveis para *std_ulogic* e *std_ulogic_vector*, respectivamente. A Tabela 4 mostra os valores esperados quando se aplica o operador *and* aos valores *std_ulogic*.

Tabela 4 - Resultado das Operações "E " para Valores Std_ulogic.

	U	X	0	1	Z	W	L	H	-
U	U	U	0	U	U	U	0	U	U
X	U	X	0	X	X	X	0	X	X
0	0	0	0	0	0	0	0	0	0
1	U	X	0	1	X	X	0	1	X
Z	U	X	0	X	X	X	0	X	X
W	U	X	0	X	X	X	0	X	X
L	0	0	0	0	0	0	0	0	0
H	U	X	0	1	X	X	0	1	X
-	U	X	0	X	X	X	0	X	X

Duas funções adicionais definidas para os sinais *std_ulogic* são *falling_edge* e *raising_edge*, que são usadas para detectar transição de descida e transição de subida nos sinais especificados. Considere como exemplo as descrições de um flip-flop tipo D ativo por transição de subida usando *raising_edge* e uma forma anterior, apresentados nas duas listagens a seguir.

```
if rising_edge(CLK)
   then Q <=  'D';
end if;
```

```
if (CLK'event and
    CLK = '1' and
    CLK 'last_value= '0')
  then Q <= 'D';
end if
```

O *std_ulogic* aceita todos os valores necessários para se especificar um sistema digital típico. Entretanto, é *unresolved*, o que o torna inútil para sinais de múltiplos *drivers*. Assim, mais um tipo, o *std_logic* é definido no *Std_logic_1164*.

Este novo tipo apresenta as vantagens dos nove valores do *std_ulogic* e permite a *resolution*, dando então aos usuários de VHDL um tipo lógico universal. A diferença é que o *std_logic* tem *resolution* e o *std_ulogic* não tem. O *std_logic* tornou-se de fato um padrão industrial. Há também a versão *resolved do std_ulogic-vector*, chamada *std_logic_vector*.

A Tabela 5 apresenta a *resolution table* do tipo *std_logic*.

Tabela 5 - *Resolution Table* do *Std-logic*.

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

VIII.4 Exercícios

9 Obtenha o sinal da Figura 9 usando somente o comando *after*.

10 Obtenha o sinal da Figura 9 usando somente o comando *wait*.

11 Implemente um multiplexador 4x1 usando *conditional signal assignment*.

12 Implemente um decodificador 2x4 e de um codificador 4x2 usando o conceito de *conditional signal assignment*.

13 Usando o conceito de *conditional signal assignment* implemente um flip-flop D ativo por rampa de descida.

IX ESPECIFICACAO DA ESTRUTURA DO SISTEMA

A descrição comportamental em VHDL especifica o sistema em termos de sua operação. A descrição estrutural especifica como o sistema é composto, como os subsistemas ou componentes são constituídos e como estão interconectados.

A descrição estrutural permite a existência de vários níveis hierárquicos e um componente pode ser especificado por sua descrição comportamental ou estrutural. Esta, por sua vez, pode ser especificada como um sub circuito, e assim sucessivamente até o último nível, em que cada componente é especificado pelo seu comportamento.

A especificação estrutural consiste de componentes conectados entre si e com o ambiente externo através de *signals*. Uma especificação estrutural requer dois grupos de elementos:

O componente pode ser um sistema individual, especificado por *architecture* e *entity* individuais e pode ser definido em uma *architecture* pela declaração de componente.

Em ambos casos a declaração de componente é tratada como uma especificação genérica, tal como um item de uma lista ou de um catalogo. Para serem usados, estes itens devem ser invocados, ou *instantiated* na especificação estrutural. Assim, a *instantiation* de um componente é a declaração básica da arquitetura estrutural. Da mesma forma que outros elementos de uma *architecture*, os componentes *instantiated* são concorrentes.

Geralmente um componente é um modulo qualquer especificado de forma estrutural ou comportamental. Na sua forma mais simples, um componente é uma *entity* e sua respectiva *architecture*. Antes que o componente possa ser usado, ele deve ser *instantiated*. A *instantiation* é a seleção de uma especificação compilada de uma biblioteca, e associando-a com a *architecture* onde será usada. Cada *instantiation* consiste do seguinte:

nome_da_instance: *entity*
biblio.nome_*entity*(nome_*architecture*)

onde “biblio” é o nome da biblioteca onde os itens estão localizados.

A *instantiation* de um componente não é completa sem o *port map*, que é uma referência ao nome real do *port* no nível de descrição estrutural mais alto, e é formalmente especificado pelo comando *port*. Sinais nos *port maps* podem ser especificados como *port* ou como internos ao sistema, sendo que neste último caso, devem ser declarados na *architecture*.

Quando os sinais nos *port maps* são especificados como *port*, diz-se que há uma associação por nome. Uma forma mais simples seria pela associação posicional, onde os sinais no *port map* são listados na mesma seqüência que os *ports* na declaração de componentes da *entity*.

A Figura 10 apresenta um circuito de um latch D com entrada habilitadora (enable). O circuito é composto por dois sub circuitos; uma porta lógica "E" e por um latch D, cujas descrições são apresentadas a seguir. Após estas duas descrições, é apresentada a descrição do *latch* D da Figura 10 usando *instantiation* dos componentes anteriores. Observe que foi usada associação por nome no *port map*, e os componentes estão na biblioteca *work*

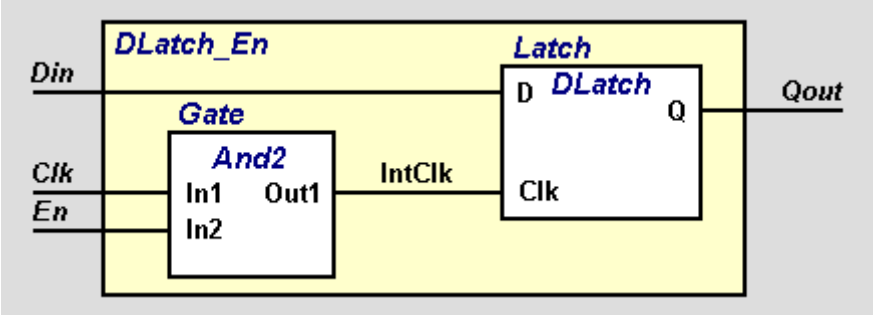

Figura 10 - *Lacth* D Formado por Dois Sub-Circuitos.

```
entity LatchD is
port (D, CLK : in Bit;
      Q : out Bit);
end entity LatchD;
architecture Beta of LatchD is
begin
   process(CLK, D)
   begin
        if CLK = '1' then
            Q <= D after 3 ns;
        end if;
    end process;
end architecture Beta;
```

```
entity And2 is
port (In1, In2 : in Bit;
      Out1 : out Bit);
end entity And2;
architecture Alfa of And2 is
begin
    Out1 <= In1 and In2 after 2 ns;
end architecture Alfa;


entity LatchD_En is
port (Din, CLK, En : in Bit;
       Qout : out Bit);
end entity LatchD_En;
architecture Delta of LatchD_En is
signal ClkInt : Bit;
begin
    process(CLK, D)
    begin
          gate : entity work.And2(Alfa)
               port map (In1 => CLK,
                         In2 => En,
                         Out1 => ClkInt);
          Latch : entity work.LatchD(Beta)
                port map (D => Din,
                          CLK => ClkInt,
                          Q => Qout);
end architecture Delta;
```

IX.1 *Instantiation* Direta

Instantiation direta é a forma mais simples de se especificar um sistema estrutural. São necessários apenas uma especificação do componente compilado e sua respectiva chamada. O comando de *instantiation* direta consiste de:

Um nome para o componente a ser *instantiated*. O nome é obrigatório pois o mesmo componente pode ser *instantiated* mais de uma vez na mesma *architecture*.

O termo ***entity*** seguido pelo nome da biblioteca e da *entity* que contém a especificação do componente.

Opcionalmente o nome da *architecture* dentro de colchetes e especificação de *port map*. O nome da *architecture* só é necessária quando há várias *architectures* dentro de uma mesma *entity*.

Cada *port map* especifica as conexões entre *ports* de uma *entity* (componente) e sinais na *architecture* onde o componente foi *instantiated*. Como já mencionado, há duas forma de se fazer *port map*, associação posicional e associação por nome.

Na associação posicional os sinais são listados na mesma ordem dos *ports* na declaração da *entity* do componente. Desta forma, as conexões entre os sinais e os *ports* são especificados por suas posições. Naturalmente os sinais devem ser do mesmo tipo de seus respectivos *ports*.

A associação posicional parece ser natural, mas eventualmente pode haver dificuldades para se determinar quais sinais são conectados a quais *ports*. Isto pode ocorrer quando há muitos sinais em uma lista. Uma solução é a associação por nome, onde são atribuídos os nomes dos *ports* aos sinais. Desta forma, a ordem dos *ports* é irrelevante e a relação correta é sempre obtida.

A associação entre *ports* e os sinais é representada pelo símbolo "=>". Observe que este símbolo não implica em direção de fluxo de informação. A listagem a seguir ilustra o trecho de uma *architecture* onde se faz uma *instantiation* direta.

```
architecture Gama of Sistema is
begin
...
   GateX : entity work.And2
      Port map (A, B, C);
...
end architecture Gama;
```

Apesar de ter sido mencionado apenas sinais de um bit, pode-se também fazer *instantiation* de sinais complexos, tais como *arrays* e *records*. Pode-se fazer a *instantiation* por elemento ou por segmento. Qualquer combinação de *ports* e sinais é permitida, desde que haja compatibilidade entre *ports* e sinais. Entretanto, todos os elementos do *port* composto devem ser associados a algum sinal.

Os *ports* não conectados podem ser especificados como *open* no *port map*. Adicionalmente, um *port* não usado pode ser deixado omitido no *port map*. Apesar de poder ser empregado em VHDL, não é recomendado pois não há garantia que tenha sido omitido de propósito. Mesmo projetistas experientes em VHDL podem esquecer alguns *ports*. Assim, recomenda-se o uso de *open* para todos os *ports* não usados no *port map*. A Figura 11 ilustra um circuito com uma saída aberta e a seguir a sua respectiva *instantiation* onde se pode visualizar o uso de *open* no *port map*.

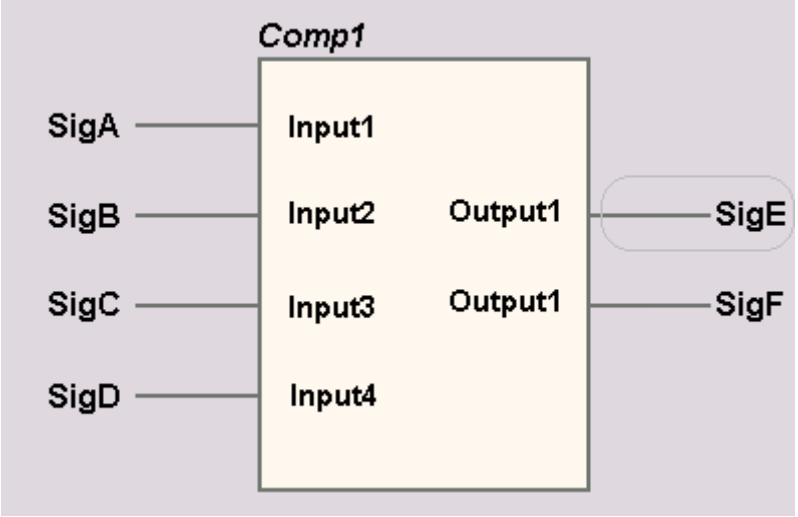

Figura 11 - Circuito Genérico com Saída Aberta.

```
Comp1: entity work.sistema
    port map  (   Input1 => SigA,
                  Input2 => SigB,
                  Input3 => SigC,
                  Input4 => SigD,
                  Output1 => open,
                  Output2 => SigF);
```

IX.2 Componentes e Configurações

No caso de uma simples descrição estrutural onde todos componentes são previamente conhecidos, a *instantiation* direta é adequada. Entretanto, em sistemas maiores, os componentes são criados de forma concorrente, e eventualmente pode ser necessário relaciona-los a implementações estruturais ou comportamentais ainda não definidos.

Na situação de uma especificação estrutural, basta ter a declaração da interface dos *components*, feita no cabeçalho da *architecture*.

A sintaxe de declaração de interface de componentes e a *entity* são similares. Isto não é coincidência, pois *component* e *entity* desempenham papeis semelhantes na definição de interfaces externas dos módulos. Há, entretanto, uma grande diferença entre estas duas construções.

A declaração da *entity* define a interface de um circuito real, o circuito é um sistema separado e pode ser individualmente simulado, analisado e sintetizado.

A declaração de *component* define a interface de um módulo "virtual", especificado dentro de uma *architecture* e funciona mais como o que se espera de uma interface do que a descrição de um circuito.

A listagem a seguir ilustra como se declara um *component*. Pode-se observar que a sua estrutura é muito similar a declaração de uma *entity*.

```
component DFF is
   generic (t_prop : time;
            tset_up : time);
   port (D    :  in bit;
         Clk  :  in bit;
         Rst  :  in bit;
         Q    : out bit);
end component DFF;
```

Da mesma forma, a *instantiation* do *component* é muito similar à da *entity* e é composta de:

Rótulo seguido por dois pontos, sendo que o rótulo é obrigatório, pois serve para identificar a *instantiation*.

O termo ***component*** que é opcional, mas altamente recomendado.

O nome do *component*, como especificado na declaração do *component*.

O comando ***generic map*** que lista valores para as variáveis *generics* do *component*.

O comando ***port map*** que é especificada da mesma forma que na *instantiation* direta.

Deve-se observar que não há ponto-e-vírgula depois do nome do *component* e depois do comando *port map*. Toda a *instantiation* forma um comando, e somente após este comando

aparece o ponto-e-vírgula. A listagem seguinte apresenta a *entity* de um registrador de 4 bits que faz uso de *instantiation* de *component* em sua *architecture*, conforme já apresentado.

```
entity Reg4 is
  port(RegD : in bit_vector (3 downto 0);
       Clk, Rst : in bit;
       RegQ : out bit_vector (3 downto0));
end entity Reg4;

architecture Gama of  Reg4 is
component DFF is
   generic (t_prop : time; t_setup : time);
   port (D, Clk, Rst : in bit;
          Q : out bit);
end component DFF;
begin
   bit0: component DFF
         generic map (t_prop => 2ns, t_setup => 1 ns)
         port map (D => RegD(0), Clk => Clk, Rst => Rst, Q => RegQ(0));
   bit1: component DFF
         generic map (t_prop => 2ns, t_setup => 1 ns)
         port map (D => RegD(1), Clk => Clk, Rst => Rst, Q => RegQ(1));
   bit2: component DFF
         generic map (t_prop => 2ns, t_setup => 1 ns)
         port map (D => RegD(2), Clk => Clk, Rst => Rst, Q => RegQ(2));
   bit3: component DFF
         generic map (t_prop => 2ns, t_setup => 1 ns)
         port map (D => RegD(3), Clk => Clk, Rst => Rst, Q => RegQ(3));
end architecture Gama;
```

A declaração de um *component* e sua *instantiation* não são suficientes para uma completa especificação de uma arquitetura estrutural pois fica faltando ainda a descrição de implementação dos componentes. A informação que une o componente a *architectures* e *entities* é especificada na *configuration*, que é uma unidade de projeto separa e que pode ser simulada e analisada sem outras unidades.

Uma análise mais cuidadosa revela que isso é similar a *instantiation* direta de *entity*, porém mais fácil e mais flexível de manter se uma implementação diferente do *component* deve ser usada. Se não há mudanças, devem apenas ser introduzidas no arquivo de

configuração, que é relativamente fácil. A listagem a seguir ilustra um caso simples de *configuration*.

```
configuration Reg4_conf of Reg4 is
   for Gama
        for Bit0 : DFF
            use entity Dflipflop(fast);
        end for;
        for others : DFF
            use entity Dflipflop(normal);
        end for;
    end for;
end configuration Reg4_conf;
```

IX.3 Exercícios

14. Usando *instantiation* implemente um multiplexador 4x1. Faça a *entity* de cada componente usado.
15. Usando *instantiation* direta implemente um flip-flop D ativo por rampa de descida. Considere o flip-flop implementado a partir de *latches* D.
16. Usando *component* faça a *entity* e a *architecture* de um contador binário crescente assíncrono formado a partir de flip-flops JK.

X *TEST BENCH*

Um projeto fica incompleto se não for verificado. Apesar de haver várias formas de se verificar um projeto VHDL, a forma mais comum é usando *test bench*. *Test bench* é um ambiente onde o projeto, chamado de *design* ou *unit under test* é verificada através da aplicação de sinais ou estímulos, e da monitoração de suas respostas. Em outras palavras, um *test bench* substitui o ambiente de projeto, de forma que o comportamento do projeto possa ser observado e analisado. Um *test bench* consiste de:

Um soquete para a *unit under test* (UUT).

Um gerador de estímulos, que é um subsistema que aplica estímulos, gerados internamente ou provenientes de um arquivo, ao UUT.

Ferramentas para monitorar as respostas do UUT.

O *test bench* na verdade é uma especificação VHDL que é simulada por um simulador VHDL. O *test bench* é composto da *instantiation* da UUT e de processos que dão suporte a aplicação de estímulos ao UUT. Forma-se assim, uma especificação híbrida, que mistura comandos estruturais e comportamentais. Isto é legal em VHDL pois a *instantiation* de *components* e os *processes* são comandos concorrentes.

Os estímulos para a UUT são especificados internamente na arquitetura da *test bench* ou podem ser provenientes de um arquivo externo. Por outro lado, as reações da UUT podem ser observadas através das saídas do simulador (apresentadas em uma tela gráfica) ou gravadas em um arquivo. Há muita flexibilidade para se criar o *test bench*, mas, eventualmente a sua complexidade pode exceder a do próprio UUT.

O *test bench* é como outra especificação VHDL, com sua *entity* e sua *architecture*, porém com algumas características específicas deste tipo de especificação:

A entity da test bench não tem ports.

A relação entre *test bench* e UUT é especificada pela *instantiation* do *component* e especificação do tipo estrutural.

Os estímulos são um conjunto de sinais declarados internamente na arquitetura da *test bench* e passada aos *ports* da UUT pela *instantiation*. Os estímulos são definidos como formas de onda em um ou mais processos comportamentais.

Toda vez que uma nova especificação VHDL é criada deve-se lembrar que o projeto deve ser verificado. Projetos simples podem ser facilmente simulados, mas projetos complexos podem exigir mais tempo para serem simulados.Como o desenvolvimento de um *test bench* é uma tarefa complexa, é altamente recomendável que se escreva algumas orientações de projeto ao longo do desenvolvimento do projeto da UUT.

Assim que a UUT e a sua *test bench* estiverem completas, a verificação pode ser iniciada. Observe que a *test bench* é simulada, mas não a UUT. A UUT é apenas um dos componentes *instantiated* pela *test bench*. A Figura 12 ilustra os passos descritos para a implementação de uma *test bench* e a seguir é apresentado uma descrição simbíloca de uma *test bench*, onde se pode observar no primeiro quando o cabeçalho da *entity*, a declaração dos

estímulos no segundo quadro, a *instantiation* do UUT no terceiro quando e a definição dos estímulos no último quadro.

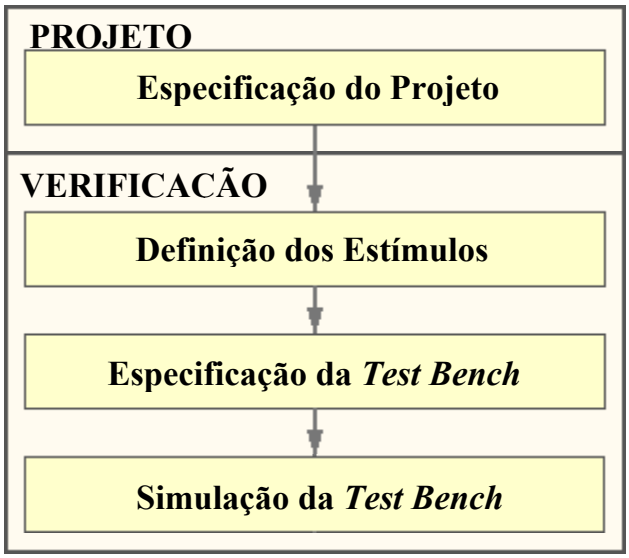

Figura 12 - Fluxograma para Implementação de *Test Bench*.

```
entity TB is
end entity TB;
architecture TB1 of TB is
   signal A, B, … : bit;
   signal …;
begin
UUT: entity work.processador(Beta)
   port map (…);
estimulos: process
   begin
         A <= …;
         B <= …;
         …
         wait for …;
         …
         wait for …;
         …;
         wait;
   end process estimulos;
end architecture TB1;
```

X.1 Estrutura de uma t*est bench*

A *test bench* é como outra especificação VHDL, consistindo de uma *entity* e de uma *architecture*. Uma importante diferença é que a *entity* de uma *test bench* não tem *ports* ou *generics*. Isto se deve ao fato que a *test bench* não é um circuito real que precisa se comunicar com o ambiente e, portanto não tem entradas ou saídas. Todos os valores para os *ports* de entrada da UUT são especificados na t*est bench* como estímulo. As saídas são observadas pelo simulador e são armazenadas em um arquivo. O arquivo a ser testado não precisa de modificações ou comandos adicionais. Assim, qualquer especificação VHDL pode ser testada.

A UUT deve ser *instantiated* na arquitetura da *test bench*. Isto pode ser feito da mesma forma que qualquer especificação estrutural, através de *instantiation* direta ou através de *instantiation* de componente. Aos *ports* de *instantiation* da UUT devem ser atribuídos os estímulos.

Como os processos e a *instantiation* de componentes são concorrentes, não faz diferença se a UUT ou o estímulo for definido primeiro. As descrições a seguir ilustram de forma resumida a descrição de um multiplexador e sua respectiva *test bench*.

```
entity Mux2x1 is
   port (A, B, Sel : in bit;
         Y : out bit);
end entity Mux2x1;
architecture beta of Mux2x1 is
...
end architecture Beta;

entity Testbanch is
end entity Testbench;
architecture TB1 of Testbench is
...
begin
   UUT: entity work.Mux2x1
        port map ( A => ...,
                   B => ...,
                   Sel => ...;
                   Y => ...);
...
end architecture TB1;
```

A parte principal de uma *test bench* é o conjunto de estímulos. Como a *test bench* não se comunica com o ambiente por sinais, todos estímulos devem ser declarados como outros sinais no cabeçalho da *architecture* da *test bench*. No caso de um circuito que se comunica como mundo externo, os sinais seriam declarados na *entity*.

Os estímulos podem ser especificados tanto como declarações concorrentes (mudanças nos sinais especificados como formas de onda), ou como *process* que contenha declarações de sinais separados por comandos *wait* e *for*, introduzindo atrasos entre declarações subseqüentes. Neste ultimo caso, adiciona-se um *wait* (incondicional) como última declaração no *process*. Este comando faz uma parada permanente na simulação (caso contrário, a simulação recomeçaria novamente).

Finalmente, a relação entre estímulo e UUT é obtida através de atribuições pelo *port map* na *instantiation* do UUT.

Como exemplo, a primeira listagem a seguir ilustra a descrição de um multiplexador 2x2 a ser empregado como UUT, e a Figura 13 apresenta as formas de onda desejadas para testa-lo. A seguir é listada a descrição da test bench para a geração dos sinais.

```
entity Mux2x1_2 is
   generic (MuxDel:time := 5ns);
   port(A, B : in std_logic_vector (1 downto 0);
        Sel : in std_logic;
        Y: out std_logic_vector (1 downto0));
end entity Mux2x1_2;
architecture Beta of Mux2x1_2 is
begin
   with sel select
        Y <= A after MuxDel when '0',
        B after MuxDel when '1',
        "XX" when others;
end architecture Beta;
```

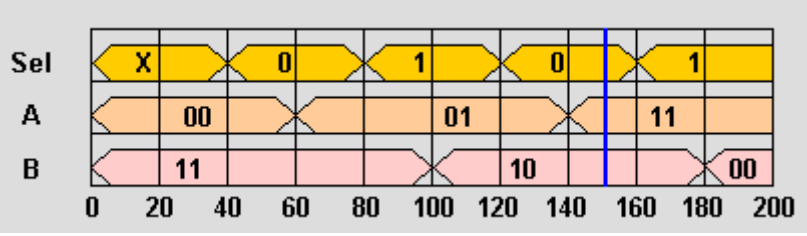

Figura 13 - Formas de Onda para Testar um Multiplexador 2x2.

```
entity Testbench is
end entity Testbench;
architecture TB1 of Testbench is
signal A, B, Y : in std_logic_vector (1 downto 0);
signal Sel : in std_logic;
begin
   UUT: entity work.Mux2x1_2
        Port map (A, B, Sel, Y);
   Sel <= 'X',     '0' after 40 ns,
                   '1' after 80 ns,
                   '0' after 120 ns,
                   '1' after 160 ns;
   A <= "00",
        "01" after 60 ns,
        "11" after 140 ns;
   B <= "11",
        "10" after 100 ns,
        "00" after 180 ns;
end architecture TB1;
```

Ao final de uma verificação tem-se o resultado da simulação e/ou uma listagem de resultados na forma de relatório. Isto é obtido de várias formas; usando aplicativos dos simuladores (listagem das variações dos sinais ao longo do tempo ou telas gráficas), ou usando o comando ***report*** que descreve em um arquivo os resultados de toda a simulação.

Esta ultima forma é fácil de se usar, e é empregada para mostrar uma mensagem quando alguma coisa não está certa. Se esta opção é usada e não há mensagem durante a simulação, então se presume que a UUT tenha funcionado como esperado.

O comando *report* consiste de três elementos:

- Declaração *assert* que verifica uma condição Booleana,
- Declaração *report* que define uma mensagem a ser apresentada quando a condição é falsa.
- Declaração *severity* que informa ao simulador quão severa foi a condição de erro encontrada, variando desde um alerta até uma falha gral no sistema.

A listagem a seguir ilustra o uso do comando *report*.

```
assert (numero1 > referencia)
   report "Numero maior que a referencia"
   severety warning;
```

O comando *assert* é por natureza seqüencial e portando usado em um *process*. Uma regra geral para uso deste comando seria:

Use o par *assert-report* para cada novo valor esperado da UUT.

Especifique o valor esperado como condição na *assertion*.

Tente ser específico no texto a ser apresentado. Uma simples mensagem apresentando "erro" não diz muito. Tente especificar o que ocorreu, quando ocorreu e os valores das entradas.

Lembre-se que os novos valores só são atribuídos quando o processo é suspenso. Não se deve esperar valores nas saídas imediatamente após a sua atribuição.

A listagem seguinte ilustra a *test bench* para verificar o multiplexador 2x2 anterior onde se faz uso de *assert* e *report*.

```
entity Testbench is
end entity Testbench;
architecture TB1 of Testbench is
signal A, B, Y : in std_logic_vector (1 downto 0);
signal Sel : in std_logic;
begin
   UUT: entity work.Mux2x1_2
        port map (A, B, Sel, Y);
   stimuli: process
       begin
        Sel <= 'X'; A <= "00"; B <= "11"; wait for 0 ns;
        assert (Y = "XX") report "Teste falhou em Sel=X";
        Sel <= '0'; wait for 40 ns;
        assert (Y = "00") report "Teste falhou em Sel=0";
        A <= "01"; wait for 20 ns;
        assert (Y = "01") report "Teste falhou em Sel=0  -  A não mudou";
        Sel <= '1'; wait for 20 ns;
        assert (Y = "11") report "Teste falhou em Sel=1";
        B <= "10"; wait for 20 ns;
        assert (Y = "10") report "Teste falhou em Sel=1  -  B não mudou";
   end process;
end architecture TB1;
```

X.2 Exercícios

17. Faça *test benches* para os circuitos dos exercícios 16, 17, 20, 21 e 22.
18. Obtenha a descrição VHDL de um flip-flop SR ativo por rampa de descida e faça a sua correspondente verificação.
19. Obtenha a descrição VHDL de uma máquina de Mealy que detecte todas as ocorrências da seqüência 1101. Faça a verificação de seu projeto.

XI ÍNDICE