Capítulo 9

Uma Breve Introdução à VHDL

The limits of my language means the limits of my world.
Ludwig Wittgenstein.

Minha pátria é minha língua.
Língua, Caetano Veloso.

A linguagem VHDL foi desenvolvida no início da década de 1980 como parte de um programa de pesquisa do Departamento de Defesa dos EUA, chamado de *Very High Speed Integrated Circuits*. Este programa visava o desenvolvimento de circuitos eletrônicos de alta velocidade e um de seus resultados foi a especificação, padronização, e posterior implementação da linguagem *Very High Speed Integrated Circuit Hardware Description Language* ou VHSIC-HDL, ou o mais palatável VHDL. Este capítulo[1] contém uma breve introdução aos elementos da linguagem necessários para a compreensão dos circuitos modelados com VHDL.

Ao contrário de linguagens de programação como C ou *assembly*, nas quais os comandos/instruções são executados na ordem em que aparecem no código fonte, em VHDL o cômputo das expressões e as atribuições de valores ocorrem em paralelo, com um efeito que emula a propagação de sinais num circuito. Neste sentido, o código VHDL é declarativo, ao invés de imperativo, como em C. A diferença entre estes estilos, imperativo e declarativo, ficará mais clara no que se segue.

As ambições do autor, com este capítulo, são particularmente modestas. O que se pertente é apresentar um subconjunto pequeno de VHDL, que é o mínimo necessário para permitir a modelagem dos circuitos estudados neste texto, visando a simulação destes circuitos em atividades de laboratório. Para quem deseja se aprofundar na linguagem, o texto mais completo e o com melhor didática de que o autor tem ciência é o de Ashenden [Ash08].

A leitura deste capítulo deve acompanhar a execução das tarefas de laboratório que correspondem ao material apresentado aqui. Os exemplos no texto podem ser insuficientes para a compreensão dos conceitos e os laboratórios foram projetados para complementar as leituras. Para efetuar as tarefas dos laboratórios é necessário um compilador de VHDL e um visualizador de formas de onda. Com nosas turmas empregamos o compilador `ghdl` e o visualizador `gtkwave`. Existem sistemas comerciais que oferecem a mesma funcionalidade, em versões disponíveis para os sistemas operacionais mais populares.

[1] Versão de 1 de março de 2021.

Ao contrário de C, que não tem um óbvio competidor em seu nicho de aplicação, *Verilog* é outra linguagem de descrição de *hardware* com amplo emprego na indústria. O código fonte Verilog se parece com o da linguagem C ao passo que VHDL se parece mais com Pascal ou Ada. A expressividade e a funcionalidade de Verilog são similares àquelas de VHDL e a escolha dentre as duas me parece ser uma questão de gosto e/ou de fé religiosa.

9.1 A linguagem VHDL

VHDL é uma sigla que contém uma sigla: *VHSIC Hardware Description Language*, sendo VHSIC abreviatura para *Very High Speed Integrated Circuits*. Como o nome diz, VHDL é uma linguagem de **descrição** de *hardware*. Guarde bem essa sigla, *HDL*, que abrevia *Hardware Description Language*.

Diferentemente de C, um programa em VHDL *descreve* uma estrutura, ao invés de *definir* ou *prescrever* uma computação. Neste sentido, um programa em C é uma "receita de bolo" com ingredientes (variáveis) e modo de preparar (algoritmo). Em VHDL um modelo estrutural pode descrever a interligação de componentes, ou pode especificar sua função sem mencionar uma implementação daquela função.

A partir da estrutura, um compilador como `ghdl` gera um simulador para o circuito descrito pelo código fonte. VHDL também permite descrever a função ou o comportamento de um circuito, sem que seja necessário definir sua estrutura.

Um sintetizador é um compilador que gera uma descrição de baixo nível – portas lógicas ou transistores – para que um circuito integrado seja fabricado a partir da descrição.

9.1.1 Descrição gráfica de circuitos

Examine o diagrama da Figura 9.1 e liste, de forma sucinta, as convenções empregadas para que ele possa ser interpretado por alguém que tenha estudado Sistemas Digitais. Relembre DeMorgan.

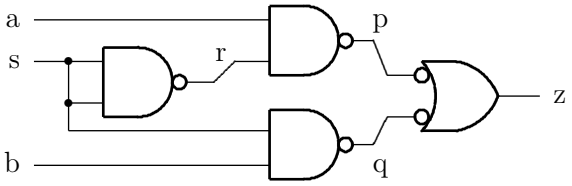


Figura 9.1: Diagrama de um circuito combinacional.

Espaço em branco proposital.

Sua lista deve conter, pelo menos:
(i) uma convenção para representar as portas lógicas;
(ii) uma convenção para representar as ligações;
(iii) uma convenção para representar as entradas;
(iv) uma convenção para representar as saídas; e
(v) uma convenção para nomear os valores lógicos transportados pelas ligações.

Até agora temos usado várias *convenções*:

* portas lógicas tem formatos distintos (*not*, *and*, *or*, *xor*);
* multiplexadores e decodificadores são trapézios;
* somadores são trapézios com um recorte/chanfro;
* registradores são retângulos com o triângulo do relógio;
* circuitos complexos são caixas retangulares;
* ligações (fios, sinais) são linhas;
* sinais fluem da esquerda para a direita;
* entradas ficam no lado esquerdo (ou acima);
* saídas ficam no lado direito (ou abaixo);
* se não é óbvio, setas indicam o fluxo dos sinais; e
* um traço curto com um número indica largura do sinal, se maior do que 1.

O diagrama na Figura 9.2 ilustra estas convenções: são quatro os registradores, chamados de r1, r2, r3 e STAT, os sinais fluem da esquerda para a direita, o sinal clock tem um bit de largura, o sinal oper e a entrada do registrador STAT tem 4 bits de largura, e os demais sinais tem largura de 16 bits. O circuito no centro é uma unidade de lógica e aritmética. O comentário acima do diagrama, é uma indicação da função do circuito, num idioma chamado de *linguagem de transferência entre registradores* (*register transfer language* ou RTL). O comentário indica que, a cada borda do relógio, o resultado da soma dos conteúdos de r1 e r2 é atribuído (transferido) para r3.

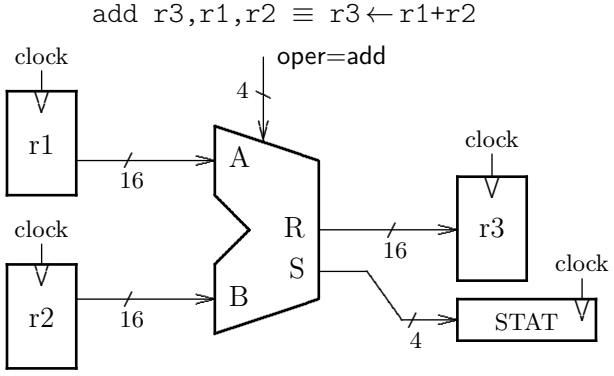


Figura 9.2: Exemplo das convenções usadas em diagramas de circuitos.

9.1.2 E se a descrição for textual?

A interpretação automática de desenhos é uma arte, mesmo que os desenhos sejam diagramas tão simples como nossos circuitos. Para facilitar a interpretação automática de representações para circuitos emprega-se texto.

Circuitos são representados em VHDL por *sinais* (fios) e *componentes* (*design units*). Um componente é descrito por duas construções da linguagem, uma *entidade* – que descreve sua interface com o mundo externo – e uma *arquitetura* – que descreve seu comportamento através das construções da linguagem.

No que segue, empregaremos as seguintes convenções de tipografia para o código VHDL: palavras reservadas são grafadas em negrito (**entity**), nomes de sinais e componentes são grafados sem serifa (meuSinal), e comentários iniciam com '−−' e terminam no final da linha e são grafados com −− *caracteres inclinados*.

Uma entidade (**entity**) descreve as ligações externas do componente:
(i) entradas são **in**
(ii) saídas são **out**
(iii) ligações são representadas por *sinais*, no nosso caso bits (bit) ou vetores de bits (bit_vectors).

Uma arquitetura (**architecture**) descreve as ligações internas do componente. A arquitetura declara quais componentes são usados na implementação, que sinais internos são necessários para interligar os componentes internos, e de que forma os componentes são interligados entre si e com os sinais da entidade.

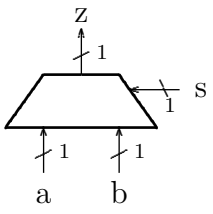
Uma certa entidade pode ser modelada em VHDL de mais de uma forma, como veremos em breve – para uma entidade (interface), podem ser projetadas mais de uma versão da arquitetura (circuito).

MUX2 – entidade e arquitetura

O multiplexador da Figura 9.1 é descrito em VHDL pelas suas entidade e arquitetura. A entidade, mostrada no Programa 9.1, é chamada mux2 e descreve a *interface* deste componente, com três sinais de entrada e um de saída, todos com a largura de um bit.

Programa 9.1: Exemplo de entidade.

```
entity mux2 is
  port(a,b : in  bit;
       s   : in  bit;
       z   : out bit);
end mux2;
```



A entidade tem nome mux2. A associação entre uma entidade (interface) e sua arquitetura (implementação) se dá no comando de definição da arquitetura. No trecho de código abaixo, a entidade tem nome mux2 e a implementação é chamada de estrutural – que é a versão estrutural da entidade mux2.

```
entity mux2 is
  ...
end mux2;

architecture estrutural of mux2 is
  ...
begin
  ...
end architecture estrutural;
```

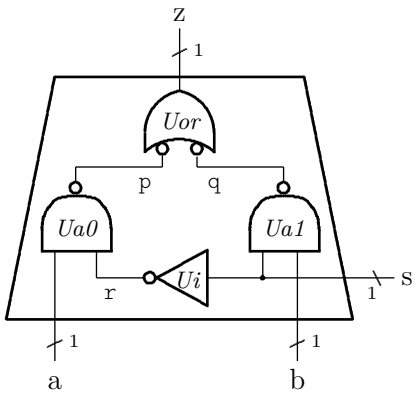
O código com a arquitetura do *mux*-2 é mostrado no Programa 9.2. A arquitetura, chamada `estrutural`, define a *estrutura* do circuito modelado – ela descreve como os componentes são interligados. Entre as palavras chave **architecture** e **begin** devem ser declarados os componentes usados na implementação do multiplexador – neste caso inversor e porta *nand* – e os sinais internos necessários para ligar os componentes e a interface, que são os sinais `r`, `p` e `q`.

Entre o **begin** e o **end architecture** são definidas as ligações entre os componentes da implementação. Como mostra o diagrama ao lado do código, há uma correspondência direta entre o circuito e o código VHDL que o representa. Infelizmente, nas descrições textuais *todos* os sinais devem ser explicitamente nomeados.

A construção **port map** faz a ligação entre os sinais internos à arquitetura e os sinais declarados para cada componente individual usado no modelo. O **port map** é quem liga sinais em componentes ao mapear os sinais internos declarados na arquitetura aos sinais declarados nas entidades dos componentes. A instanciação dos componentes com o mapeamento das portas é similar a um operador aritmético prefixado; ao invés da forma usual, que é infixada ($c = a+b$), emprega-se a forma prefixada ($+(c, a, b)$).

Programa 9.2: Exemplo de arquitetura.

```
architecture estrutural of mux2 is

  -- declaração de componentes
  component inv is
    port(A : in bit; S : out bit);
  end component inv;

  component nand2 is
    port(A,B : in bit; S : out bit);
  end component nand2;

  -- declaração de sinais internos
  signal r, p, q : bit;

begin

  -- instanciação e
  --   interligação dos componentes
  Ui:  inv port map(s, r);

  Ua0: nand2 port map(a, r, p);

  Ua1: nand2 port map(b, s, q);

  Uor: nand2 port map(p, q, z);

end architecture estrutural;
```



Entre **architecture** e **begin**, os componentes e os sinais são *declarados*. Entre o **begin** e o **end architecture**, os sinais e componentes são *instanciados* e conectados através dos sinais internos e do mapeamento dos sinais nas portas dos entidades com o **port map**. O **port map** associa, ou 'gruda', os sinais nos terminais do componente.

9.1.3 Projeto *top–down*

O projeto de sistemas digitais tende a ser complexo por envolver uma quantidade enorme de informações a respeito dos componentes do sistema, sobre as interconexões entre aqueles, bem como sobre os detalhes da implementação de cada componente. Considere um microprocessador comercial de 32 bits que seja relativamente sofisticado – tais circuitos são implementados com cerca de um milhão de transistores [HP12]. Supondo, para simplificar o cálculo, que cada porta lógica e cada *flip-flop* seja implementado com 5 transistores, este projeto empregaria algo como 200 mil componentes. É impossível a um humano lidar com tanta informação e portanto empregam-se diferentes níveis de abstração no projeto de sistemas tão complexos. Por exemplo, no nível de abstração que trata das unidades funcionais do circuito de dados de um processador, tais como somadores, ULA, e bloco de registradores, todos os detalhes sobre a implementação destas unidades no nível de portas lógicas são abstraídos e escondidos do projetista da CPU. O projetista das unidades funcionais preocupa-se com as portas lógicas, mas pode abstrair, e em larga medida ignorar, os detalhes da implementação das portas lógicas no nível dos transistores.

Um dos mecanismos mais eficazes para permitir o uso das abstrações mencionadas no parágrafo anterior é o uso de modelos para descrever os subsistemas e suas interfaces. Neste contexto, um *modelo* consiste de alguma forma de codificação do nosso conhecimento sobre o sistema modelado. Se os modelos descrevem os sistemas formalmente, como as especificações dos circuitos dos Capítulos 4 e 8, então estas podem ser usadas como um contrato entre projetista de um componente e seus usuários. Neste caso, o contrato define as interfaces e o comportamento do componente. Se o modelo for descrito utilizando-se de um formalismo adequado, então este pode ser usado como documentação do projeto. Se o formalismo permitir escrever modelos que sejam executáveis em simuladores, então o modelo executável pode ser usado para testar a validade da especificação, e o mapeamento entre versões nos distintos níveis de abstração, bem como para verificar se as descrições nos diferentes níveis de abstração satisfazem às mesmas especificações. Um modelo executável, ao menos em teoria, pode ser traduzido por um compilador adequado para então permitir a síntese de um circuito integrado. A linguagem VHDL foi projetada para ser empregada em todas estas formas de modelagem.

Uma forma de descrever o "ciclo de vida" do desenvolvimento de um sistema é com a chamada *espiral de projeto*, mostrada na Figura 9.3. Tipicamente, o processo se inicia com uma ideia, e a partir dela escreve-se uma *especificação* (abstrata) para o sistema. Esta especificação é *refinada* para um projeto, que essencialmente é uma versão mais detalhada e menos abstrata da especificação. A *implementação* do projeto resulta numa versão concreta da especificação, e esta pode então ser *verificada* quanto a sua corretude e avaliada quanto ao seu desempenho. Se a implementação não satisfaz à especificação, deve-se efetuar um novo ciclo da espiral, com tantas repetições quantas necessárias até que os requisitos de projeto sejam atendidos.

Da forma como VHDL é empregada nesse texto, (i) a *ideia* é a especificação em Português de um circuito, (ii) o *projeto* é o primeiro refinamento da especificação, que é obtido com trabalho intelectual, lápis e papel; (iii) a implementação é a *codificação* em VHDL, e (iv) a verificação é a confirmação de que a implementação satisfaz ao especificado, através de *simulação*.

Um projeto pode ser refinado de mais de uma forma, na sua fase inicial, e a cada refinamento, o resultado é mais concreto e próximo da implementação. Por exemplo, partindo-se de uma especificação (abstrata) do comportamento desejado, esta sofre um refinamento que resulta num *diagrama de blocos*, que ao ser refinado produz o *projeto dos blocos*, o refinamento dos

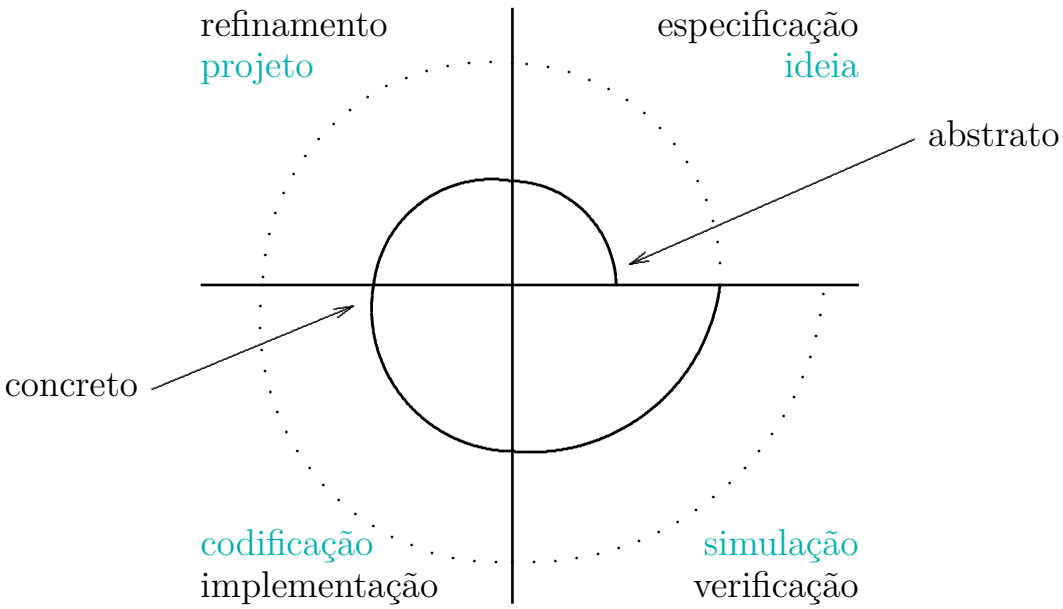


Figura 9.3: Espiral de projeto.

blocos resulta na *formalização da especificação*, que é a modelagem *concreta* dos componentes do projeto em VHDL, gerando-se código VHDL para os *componentes* do projeto.

9.2 Modelagem de circuitos combinacionais em VHDL

VHDL é uma linguagem extremamente versátil e é usada para

- a modelagem e simulação de circuitos;
- especificação, projeto e síntese de circuitos;
- descrever listas de componentes e de ligações; e
- verificação de corretude: se especificação ⇒ implementação.

A linguagem é padronizada pelo IEEE, no padrão IEEE 1076-1987 V[HSIC] H D L, ou *Very High Speed Integrated Circuit Hardware Description Language*. O padrão IEEE Std 1164 define pacote `std_logic`, e o padrão IEEE Std 1076.3 VHDL Numeric Standard define uma biblioteca de funções numéricas. Há mais, muito mais, nos bons textos do ramo tais como [Ash08, PT96].

9.2.1 Tipos de dados

Alguns dos tipos básicos de dados disponíveis em VHDL são mostrados na Tabela 9.1. O tipo `bit` tem dois valores, `'0'` e `'1'` – note as aspas simples. Um vetor de bits, ou um `bit_vector`, é representado por mais de um bit, enfeitado com aspas duplas: `"1001"`. As regras para caracteres (aspas simples) e cadeias (aspas duplas) são similares àquelas para bits e vetores de bits. Constantes booleanas, números inteiros, números reais e constantes de tempo são representadas sem aspas.

A atribuição a um sinal é representada pela *flecha dupla* '<=' e a concatenação de vetores de bits, ou de caracteres, é representada pelo '&', como mostra o exemplo da concatenação das

TIPO	VALOR	EXEMPLO DE USO
bit	'1', '0'	D <= '1';
bit_vector	vetor de bits	Dout <= "0011";
boolean	TRUE, FALSE	start <= TRUE;
integer	-2,-1,0,1,1	Cnt <= Cnt + 17;
real	2.3, 2.3E-4	med <= sum/16.0;
time	100 ps, 3 ns	sig <= '1' **after** 12 ns;
character	'a','1','@'	c <= 'k';
string	vetor de **char**	msg <= "error: " & "badAddr";

Tabela 9.1: Subconjunto pequeno dos tipos de dados em VHDL.

cadeias de caracteres.

Atenção porque isso é importante: VHDL **não** é *case sensitive* – maiúsculas e minúsculas são consideradas iguais.

Sinais e Vetores de Sinais

Os *sinais* em VHDL correspondem aos fios que transportam os bits no circuito e a linguagem define sinais de um bit, e vetores de sinais com muitos bits.

O código abaixo declara um sinal `x` com um bit de largura, e este é inicializado em '1'. O sinal `vb` é um vetor com 4 bits de largura e a posição de cada um dos bits é aquela em que o bit mais significativo está à esquerda, e o menos significativo à direita. Esta é a declaração adequada para vetores que representam números.

```
-- declaração de sinais do tipo bit
signal x: bit := '1';  -- inicializado em '1'
signal vb: bit_vector(3 downto 0) := "1100"; -- inicializa em 12
```

O sinal `bv` é declarado com a direção contrária àquela do sinal `vb`, e o bit mais significativo deste vetor é aquele na direita.

```
signal bv : bit_vector(0 to 3); -- bit 0 é o Mais Significativo
```

As atribuições abaixo produzem resultados que talvez não sejam intuitivos a uma primeira vista.

```
signal bH : bit_vector(7 downto 0); -- bit 7 é Mais Significativo
signal bL : bit_vector(0 to 7);     -- bit 0 é Mais Significativo

bH <= b"11000000"; -- bH se torna 0xc0 = 192
bL <= b"11000000"; -- bL se torna 0x03 = 3
```

É possível selecionar um subconjunto dos bits de um sinal. Os comandos abaixo declaram 4 sinais: um com 1 bit, um com 8 bits, e dois sinais com 4 bits de largura. A primeira atribuição seleciona o bit mais significativo do sinal `v8` e o atribui a `x`. As atribuições a `v4` e `t4` selecionam o quarteto central de `v8`, e armazenam esses quartetos na *ordem numérica* em `v4`, e na *ordem invertida* em `t4`.

```
signal x:  bit; -- um bit
signal v8: bit_vector(7 downto 0); -- vetor de 8 bits
```

```
signal v4: bit_vector(3 downto 0); -- vetor de 4 bits
signal t4: bit_vector(3 downto 0); -- vetor de 4 bits
  ...
x  <= v8(7);          -- atribuição do bit mais significativo
v4 <= v8(5 downto 2); -- atribuição do quarteto central de v8
t4 <= v8(2 to 5);     -- mesmo quarteto, na ordem inversa
```

O operador que seleciona um ou mais bits de um vetor é um par de parenteses (`v8(7)`), ao invés de um par de colchetes como em C ou Pascal. Em VHDL é possível selecionar um subconjunto com vários elementos de um vetor (`v8(2 to 5)`).

Vetores de bits podem ser representados nas bases hexadecimal (com prefixo `x`), octal (prefixo `o`) e binária (prefixo `b`). A representação binária implícita é a normal: se nada for dito em contrário, o vetor é um vetor de bits.

```
hexadecimal  <= x"C0";        -- 1100 0000    OITO bits
octal        <= o"300";       -- 011 000 000  NOVE bits
binaria_expl <= b"11000000"; -- 11000000 base-2 explícita
binaria_impl <= "11000000";  -- 11000000 base-2 implícita
```

9.2.2 Abstração em VHDL: *Packages*

O Programa 9.3 mostra o código VHDL que define uma série de nomes para sinais com mais de um bit. Este conjunto de definições é chamado de *pacote*, ou **package** e pode ser armazenado em um arquivo que é compilado separadamente dos demais arquivos com código fonte. Uma vez que o pacote seja compilado, suas definições podem ser usadas em todas as demais unidades de projeto. Pacotes têm função similar aos arquivos `.h` da linguagem C.

Programa 9.3: Um pacote com abreviaturas.

```
package p_WIRES is -- abreviaturas para
                   -- novos tipos e agrupamentos de sinais

  subtype natur8  is integer range 0 to 255;   -- natural de 8 bits
  subtype natur16 is integer range 0 to 65535; -- natural de 16 bits

  subtype reg2  is bit_vector(1  downto 0); -- par de bits
  subtype reg4  is bit_vector(3  downto 0);
  subtype reg8  is bit_vector(7  downto 0);
  subtype reg15 is bit_vector(14 downto 0);
  subtype reg16 is bit_vector(15 downto 0); -- barramento de 16 bits

package body p_WIRES is

  -- o corpo deste pacote é vazio

end package p_WIRES;
```

Um tipo restringido de dados, tal como um inteiro de 8 bits pode ser declarado como sendo um subtipo de `integer`, com a declaração **subtype** mais a faixa de valores da restrição. Por exemplo, a declaração

```
subtype natur8 is integer range 0 to 255;
```

define um novo tipo, chamado de natur8, que é um número natural com valores no intervalo $[0, 255]$. Sinais de tipo natur8 podem ser usadas nos modelos e o compilador verifica se os valores atribuídos a eles estão dentro do intervalo correto. A atribuição do valor 300 a um sinal de tipo natur8 resulta num erro de compilação ou de simulação.

Outra aplicação dos tipos restringidos é na definição de abreviaturas para tipos de sinais. Por exemplo, os subtipos reg2 a reg16 definem nomes curtos para vetores de bits com larguras 2 a 16. O nome curto reg16 pode ser usado no lugar do verborrágico bit_vector(15 **downto** 0) para declarar sinais com 16 bits de largura.

O corpo do pacote, entre o **package body** e o **end package** é vazio. A Seção 9.4.13 mostra como definir funções no corpo do pacote.

Para que as definições contidas no pacote possam ser usadas, a declaração da entidade deve ser precedida do nome do pacote, como mostra o Programa 9.4. O conteúdo do pacote é compilado e depositado na biblioteca chamada work. A declaração

```
use work.p_wires.all;
```

indica que a entidade fará uso das definições do pacote p_wires, e mais que isso, todas (. **all**) as definições devem estar disponíveis: p_wires. **all**. A alternativa a *tudo* é explicitar *uma* definição para ser usada na unidade de projeto, tal como os naturais de 16 bits p_wires.natur16.

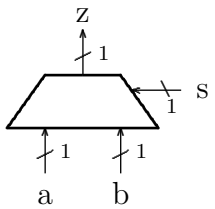
9.2.3 Modelo estrutural do multiplexador

Um *modelo estrutural* descreve a estrutura de um circuito, do ponto de vista das interconexões entre seus componentes. Dependendo das ferramentas disponíveis, um circuito pode ser sintetizado a partir da sua descrição no nível RTL ou do nível estrutural, porque estas descrições são suficientemente concretas e detalhadas para permitir a tradução automática e eficiente da descrição em VHDL para uma lista de componentes e ligações entre aqueles, que é então usada para produzir o circuito integrado. Esta lista de ligações e componentes é chamada de *net list*.

A *entidade* do multiplexador de duas entradas é mostrada abaixo. O circuito tem duas entradas a e b, uma entrada de seleção s e uma saída z, todas de um bit. Um diagrama convencional da interface do *mux*-2 é mostrado ao lado da entidade.

Programa 9.4: Entidade do *mux2*.

```
use work.p_wires.all;
entity mux2 is
  port(a,b : in  bit;
       s   : in  bit;
       z   : out bit);
end mux2;
```



Um *modelo estrutural* do *mux2* é mostrado no Programa 9.5. Um diagrama convencional da interligação dos componentes do *mux*-2 é mostrado ao lado da arquitetura. Numa implementação óbvia do *mux*-2 são necessários três sinais internos, `r` com o complemento de `s`, e `p,q` para interligar as portas *and* e a porta *or*.

No Programa 9.5, os componentes e os sinais internos são *declarados* entre o **architecture** e o **begin**. Os componentes são *instanciados* entre o **begin** e o **end architecture**.

Na instanciação, cada componente tem um *label* (opcional) – Uxx: que significa *design Unit*

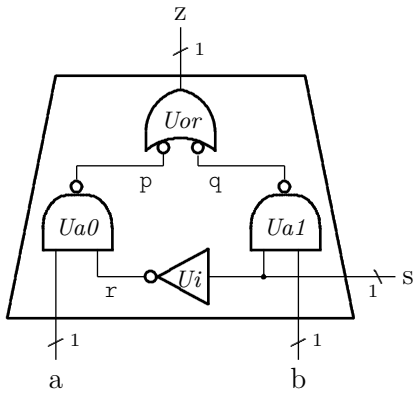
xx – e o mapeamento das portas do componente com os sinais da interface (declarados na entidade) e os sinais internos (declarados na arquitetura). Por exemplo, na instanciação

```
Ua0: nand2 port map(a,r,p)
```

a porta A do componente *nand*2 é ligada ao sinal de interface a, a porta B é ligada ao sinal interno r, e a porta de saída S é ligada ao sinal interno p. Este mapeamento é chamado de *mapeamento posicional* porque os sinais da arquitetura são associados às portas do componente na ordem – na mesma posição – em que as portas são declaradas.

Programa 9.5: Arquitetura do *mux2*.

```
architecture estrutural of mux2 is

  -- declaração dos componentes
  component inv is
    port(A: in bit; S: out bit);
  end component inv;

  component nand2 is
    port(A,B: in bit; S: out bit);
  end component nand2;

  -- declaração sinais internos
  signal r, p, q: bit;

begin
  -- início da área concorrente

  -- instanciação dos componentes
  Ui:  inv   port map (s, r);
  Ua0: nand2 port map (a, r, p);
  Ua1: nand2 port map (b, s, q);
  Uor: nand2 port map (p, q, z);

  -- fim da área concorrente
end architecture estrutural;
```



A região entre o **begin** e o **end architecture** é chamada de *área concorrente* e os comandos nesta área são executados concorrentemente. Isso significa que o código dos componentes instanciados é simulado (executado) em paralelo, imitando o comportamento de um circuito real. Os quatro componentes do *mux*-2 reagem a eventos nas suas entradas e possivelmente provocam eventos nas suas saídas.

Por isso que VHDL é dita *declarativa*: as quatro instâncias dos componentes (Ui, Ua0, Ua1 e Uor) são declaradas numa ordem arbitrária no código fonte enquanto que a ocorrência de eventos nas saídas dos componentes depende somente da sequência de eventos nos sinais que os interligam. Um evento na saída de um componente pode provocar a execução simultânea de vários componentes. Este é o comportamento dos sinais num circuito físico.

Outra diferença entre VHDL e linguagens como C ou Pascal, que também é relacionada com VHDL ser declarativa, é que a instanciação de um componente representa uma *cópia* daquele componente. A instanciação não equivale a uma chamada de função numa linguagem imperativa como C.

VHDL provê outro mecanismo para a conexão de sinais declarados na entidade (nomes locais) a sinais da unidade de projeto externa (nomes externos). Esta forma de ligação é chamada de *associação nominal* e seu uso pode ser mais conveniente do que a associação posicional. Com associação nominal, as ligações entre os sinais internos e externos podem ser codificadas em qualquer ordem, ao invés da ordem estrita em que os sinais estão declarados na entidade do componente externo. O Programa 9.6 mostra um exemplo de associação nominal na instanciação de um componente com uma entrada e uma saída. No exemplo, a ordem dos sinais, de entrada e de saída, está invertida na instanciação da unidExterna. O operador da associação nominal é uma *flecha invertida*: '=>'.

Programa 9.6: Associação nominal de portas.

```
architecture exemplo of assocNominal

  component unidExterna is
    port(INPext : in bit; OUText : out bit);
  end component unidExterna;

  signal ENTRlocal, SAIDAlocal : bit;

begin
  ...
  U: unidExterna port map (SAIDAlocal => OUText, ENTRlocal => INPext);
  ...
end architecture exemplo;
```

9.2.4 Processo de compilação de VHDL

A compilação de código VHDL se dá em três fases: análise, elaboração e simulação. No nosso caso, o resultado da compilação é um programa em C gerado por `ghdl`, que simula o comportamento do modelo escrito em VHDL. Usaremos `gtkwave` para verificar o comportamento dos modelos dos circuitos.

Análise Na fase de análise, o compilador VHDL verifica a corretude da sintaxe e a semântica dos comandos. Na análise sintática, são sinalizados erros de grafia das palavras reservadas, sinais que não foram declarados, etc. Na análise da semântica, são verificados os tipos dos sinais, e se os operadores são aplicados a operandos com os tipos apropriados – somar dois números faz sentido, somar dois **if**s não.

É possível efetuar a análise somente em partes de um modelo. Por exemplo, um par entidade-arquitetura (uma *design unit*) pode ser analisado em separado, e se nenhum erro é encontrado, o resultado da análise é armazenado numa biblioteca, para uso futuro. Com `ghdl`, essa biblioteca é mantida no arquivo `work-obj93.cf`, no diretório em que ocorre a compilação.

Elaboração Na fase de elaboração, o compilador constrói um modelo completo de toda a hierarquia do projeto. Reveja o código da arquitetura do mux2 no Programa 9.5 – naquele trecho de código, os dois níveis da hierarquia são (i) a arquitetura do mux2, e (ii) as declarações dos componentes inv e nand2.

O compilador examina a arquitetura de mais alto nível da hierarquia e expande todas as declarações dos componentes instanciados naquela arquitetura. Para cada componente, o compilador examina sua arquitetura e expande todas as declarações lá instanciadas. A elaboração termina quando todas as declarações forem substituídas por atribuições – para ser mais preciso, a elaboração termina quando só restam processos e os sinais que os interligam. De forma muito simplificada, um *processo* é uma generalização para uma atribuição. Voltaremos a falar de processos na Seção 9.4.3.

Simulação Ao final da elaboração, o compilador `ghdl` traduz a estrutura de processos e os sinais que os interligam para um programa em C que, depois de compilado, permite simular a execução do modelo completo. Veja, na Seção 9.2.5, como ocorre a simulação.

A saída da simulação pode ser mostrada na tela, pode ser gravada em arquivos, ou pode gerar um conjunto de dados que representa um diagrama de tempo para ser exibido por um programa como `gtkwave`.

9.2.5 Mecanismo de simulação de VHDL

A simulação de modelos escritos em VHDL emprega a *Simulação de Eventos Discretos* – o tempo simulado progride em função de eventos que ocorrem em instantes determinados pela própria simulação.

Um *processo* é uma construção básica da linguagem e que estudaremos adiante. No trecho de código abaixo, cada uma das atribuições é um processo, embora o uso do nome 'processo' seja reservado para um comando específico, que veremos em breve.

Uma atribuição pode especificar o instante em que transação ocorrerá:

```
A <= 5;              -- atribuição válida no próximo delta
X <= Y after 10 ns;  -- atribuição válida depois de 10ns
```

A simulação consiste de duas fases: uma fase de inicialização, seguida da fase com os passos de simulação.

Na *fase de inicialização* (i) todos os sinais são inicializados com os valores declarados, ou com os menores valores possíveis para os respectivos tipos dos sinais; (ii) o tempo simulado é inicializado em zero; e (iii) todos os processos (atribuições) são executados exatamente uma vez.

A atribuição a um sinal causa uma *transação*, e a atribuição será efetivada no próximo *delta* (Δt). Se o sinal muda de estado, então esse sinal sofre um *evento*. Um evento no sinal S causa a execução de processos que dependem de S.

A Figura 9.4 mostra uma representação da linha de tempo como uma sequência de intervalos δ_n. No δ_0 ocorre a atribuição de `5` para `A`, e em δ_1 o sinal `A` já carrega o valor `5`. Em δ_0 a atribuição de `X` em `Y` é enfileirada e escalonada para ocorrer no instante $\delta_0 + 10ns$.

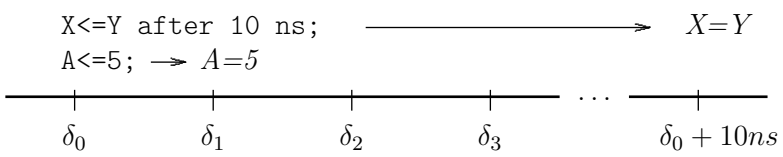


Figura 9.4: Linha de tempo como uma sequência de deltas.

Durante um *passo de simulação* (i) o tempo simulado avança até o próximo instante (δ) em que uma transação está programada; (ii) todas as transações programadas para aquele δ são executadas; (iii) estas transações podem provocar eventos em sinais; (iv) processos que dependem destes eventos são então executados; e (v) depois que todos os processos executam, a simulação avança até o δ em que a próxima transação está programada para ocorrer. Então repete-se o passo de simulação.

O tempo simulado avança em função de eventos nos sinais; a cada mudança no estado de um sinal corresponde um evento. Se ocorre um evento num sinal no lado direito de uma atribuição, a expressão do lado direito é reavaliada, e se houver mudança no estado do sinal representado pela expressão, o novo estado é atribuído ao sinal do lado esquerdo – causando um evento neste sinal. Vejamos um exemplo.

```
A <= 5;
B <= 4;
C <= A + B;
D <= C * 2;
```

As atribuições a A e a B ocorrem em δ_0 e provocam eventos em A e B. Em δ_1, o lado direito de C <= A + B; é avaliado e a alteração de valor em C provoca um evento naquele sinal. Em δ_2, o lado direito de D <= C * 2; é avaliado e a alteração provoca um evento em D. O sinal D valerá 18 em δ_3,

Note que, ao contrário de C, todas as atribuições que podem ocorrer num δ ocorrem 'simultaneamente', e *não na ordem em que aparecem no código fonte*. Como num circuito de verdade, todos os sinais que podem mudar num determinado instante, mudam naquele instante, e as consequências destas mudanças se propagam pelo resto do circuito.

Sinais e expressões são avaliados neste delta, mas eventos têm efeito no próximo delta – enquanto houver eventos no delta corrente, estes são avaliados e eventos resultantes são escalonados para o próximo delta. Eventos podem ser escalonados para instantes futuros com a cláusula **after**.

Considere o modelo de um somador completo, com modelagem da propagação dos sinais através do circuito. Um evento em A em δ_i pode causar um evento em Cout em δ_{i+1}, e em Sum em $\delta_i + 10ns$.

```
Sum  <= A xor B xor Cin after 10 ns;
Cout <= (A and B) or (A and Cin) or (B and Cin);
```

9.2.6 O modelo está pronto, e agora? Testar, testar e testar

Os modelos escritos em VHDL, por si só, não podem ser verificados quanto à sua corretude. O programador dos modelos deve escrever um ou mais *programas de teste* (*testbenches*) para excitar os modelos e observar o seu comportamento. Estes programas são chamados de *testbenches* em Inglês porque são o análogo de uma bancada de testes equipada com geradores de sinais e osciloscópios. Em geral, *testbenches* são programas relativamente complexos porque: (i) se assemelham ao circuito externo àquele que está sob teste; (ii) devem gerar os estímulos apropriados para cobrir todos os casos de teste relevantes; e (iii) devem permitir a observação das respostas para que estas possam ser verificadas. Não é incomum que um programa de teste seja mais complexo do que o modelo que se deseja testar.

Espaço em branco proposital.

Iniciemos o projeto de um *mux*-8 pela tabela verdade ao lado. São três as entradas e portanto as $2^3 = 8$ combinações de entradas devem ser verificadas, para garantir que a saída do circuito é a esperada.

s	a	b	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

O projetista de hardware tem duas obrigações distintas porém inseparáveis: (i) deve projetar um circuito que atenda ao especificado; e (ii) deve prover a garantia de que seu circuito atende à especificação.

Com base na tabela verdade – este não é o único método – o projetista gera um *vetor de testes* para exercitar o circuito, e confirmar que seu comportamento é o correto, segundo a especificação.

A especificação, junto com o vetor de testes, são o *contrato de compra e venda* do circuito, que garante ao *comprador* que o circuito cumpre o que foi prometido pelo *vendedor*.

9.2.7 *Testbenches*

Tipicamente, um arquivo com um *testbench* (TB, ou programa de teste)) contém código para verificar a corretude de modelos. Para simplificar a depuração do seu código VHDL, é uma boa ideia verificar cada novo modelo assim que seu código for completado. Um componente com N entradas tem uma tabela verdade com 2^N linhas; quanto menor for N, mais simples é o conjunto de testes necessário para garantir a corretude do modelo.

A entidade tb_estrut, mostrada no Programa 9.7 é vazia porque o programa de testes é autocontido e não tem interfaces com nenhum outro componente. O TB é o "mundo externo" do componente por testar.

Programa 9.7: Entidade do *testbench* do *mux-2*.

```
use work.p_wires.all;
entity tb_estrut is
  -- entidade vazia, não se comunica com "mundo externo"
end tb_estrut;
```

O código da arquitetura do *testbench* é demasiado longo e está dividido nos Programas 9.8, 9.9 e 9.10. O Programa 9.8 mostra a declaração do componente por testar, que é o *mux*-2.

Programa 9.8: Arquitetura do *testbench*, declaração dos componentes.

```
architecture TB of tb_estrut is

  -- declaração dos componentes por testar
  component mux2 is
    port(A,B : in  bit; S : in  bit; Z : out bit);
  end component mux2;
```

Um *vetor de testes* é um conjunto bem definido de valores, que quando apresentados às entradas do modelo, produzem um conjunto bem definido de saídas. A tabela verdade da página 289 é um vetor de testes que define completamente o comportamento do *mux*-2.

A seguir descrevemos uma das formas de codificar e de aplicar vetores de teste na verificação de corretude do modelo do *mux*-2. A arquitetura do TB, no Programa 9.9, define um **record** que contém os valores necessários para excitar o modelo. Um **record** nada mais é do que uma

tupla com o número adequado de componentes. O registro test_record_mux possui três campos e os valores destes campos devem ser atribuídos de forma a gerar todas as combinações de entradas necessárias para garantir a corretude do modelo. O vetor de testes test_array_mux contém os oito elementos necessários para excitar e verificar o *mux*-2.

No test_record_mux, os quatro campos de tipo bit ('0') correspondem aos sinais da declaração do componente. Um novo tipo, test_array_mux é definido para denotar um vetor de test_record_mux. O tamanho do vetor não é definido *a priori* por causa do positive **range** <> – a faixa de valores é definida como sendo o número de elementos de cada instância de vetores de tipo test_array_mux. O vetor test_vectors_mux é declarado como uma constante e seu tamanho não é fixo mas depende de quantos elementos o projetista julgar necessários durante a execução dos testes.

Programa 9.9: Arquitetura do *testbench*, vetor de testes.

```
type test_record_mux is record
  s    : bit;      -- entrada de seleção
  a,b  : bit;      -- entradas
  z    : bit;      -- saída esperada
end record;

type test_array_mux is array(positive range <>) of test_record_mux;

constant test_vectors_mux : test_array_mux := (
  --s,  a,  b,  z
  ('0','0','0','0'), -- linhas da tabela verdade
  ('0','0','1','0'),
  ('0','1','0','1'),
  ('0','1','1','1'),
  ('1','0','0','0'),
  ('1','0','1','1'),
  ('1','1','0','0'),
  ('1','1','1','1') );
```

O diagrama na Figura 9.5 mostra as ligações entre o modelo e o processo que percorre o vetor de testes e gera as sequências de *entradas de teste* e *saídas de teste*. As entradas de teste (a, b e s) excitam o modelo, que produz saídas de acordo com sua especificação. O comportamento esperado, segundo a especificação, é a coluna z de test_vectors_mux e este é comparado com os valores produzidos pelo modelo.

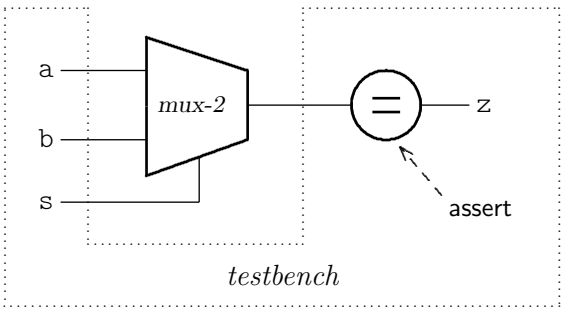


Figura 9.5: Ligações entre o *testbench* e o modelo do *mux-2*.

A sequência de valores de entrada para os testes do modelo é gerada pelo processo U_testValues, com o laço **for** ... **loop**. A variável de indução i itera no espaço definido pelo número de elementos do vetor de testes (test_vectors '**range**) – o atributo '**range** representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações. Processos e o comando **for** são discutidos nas Seções 9.4.3 e 9.4.11, respectivamente.

O i-ésimo elemento do vetor é atribuído à variável v e todos os campos do vetor são então atribuídos aos sinais que excitam o modelo. O processo U_testValues executa concorrentemente com o modelo do *mux*-2 e quando os sinais de teste são atribuídos no laço, estes provocam alterações nos sinais do modelo.

Programa 9.10: Arquitetura do *testbench*, laço de testes.

```
architecture TB of tb_estrut is
  -- declaração do componente por testar
  component mux2 is
    port(A,B : in  bit; S : in  bit; Z : out bit);
  end component mux2;

  -- sinais do testbench
  signal sel, ea, eb, saida, esperada : bit;
begin

  U_mux2: mux2 port map(ea,eb,sel,saida); -- componente por testar

  U_testValues: process        -- este componente efetua os testes
      variable v : test_record_mux;
    begin

      for i in test_vectors_mux'range loop
        v        := test_vectors_mux(i);  -- valores de teste
        sel      <= v.s;      -- bit de seleção
        ea       <= v.a;      -- entrada a
        eb       <= v.b;      -- entrada b
        esperada <= v.z;      -- saída cfe tabela verdade

        wait for 200 ps;      -- duração do passo de simulação

        assert (saida = esperada)
          report LF & "mux2: saida errada sel="& B2STR(sel) &
            " saiu=" & B2STR(saida) & " esperada=" & B2STR(esperada)
          severity error;
      end loop;

    wait; -- encerra a simulação

  end process U_testValues;

end architecture TB;
```

Depois que os valores do i-ésimo vetor de teste são atribuídos aos sinais, o simulador deve esperar para que os sinais estabilizem nas saídas do modelo. Neste exemplo, para simplificar a visualização dos sinais em um diagrama de tempo, o simulador deve esperar 200ps entre cada

mudança nos sinais nas entradas do modelo. Isso é obtido com o comando **wait for** 200 ps; que faz com que o tempo simulado progrida por 200ps antes da próxima atribuição à v. O uso do comando **wait** é discutido na Seção 9.4.4.

O comando **assert** é similar a um `printf()` em C e pode ser usado para exibir o valor de sinais ao longo de uma simulação. Este comando tem três cláusulas, a saber

(i) **assert** condição – a *condição* deve ser **falsa** para que o simulador imprima a *string*;

(ii) **report** string – texto informativo por imprimir; e

(iii) **severity** nível – a severidade pode ser um de quatro *níveis*, note, warning, error , failure , e esta última aborta a simulação.

O **assert** no Programa 9.11 verifica se a saida observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto *com relação aos vetores de teste que o projetista escreveu*. Se o projetista escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no modelo.

Programa 9.11: Mensagem de verificação de comportamento.

```
assert (saida = esperada)
  report "mux2: saida errada sel="& B2STR(sel) &
  " saiu=" & B2STR(saida) & " esperada=" & B2STR(esperada)
  severity error;
```

Se os valores nos sinais saida e esperada diferem, a mensagem no Programa 9.11 é emitida no terminal, indicando o erro:

```
mux2: saída errada   sel=1   saiu=0   esperada=1.
```

A função B2STR converte um bit em uma *string* para que o valor do bit seja emitido na tela. O operador '&' concatena duas *strings*. É obrigação do projetista determinar a saída esperada para cada combinação de entradas no vetor de teste.

Ao final do laço, a simulação termina no comando **wait**; este faz com que a execução do processo U_testValues se encerre silenciosamente.

A Seção 9.2.9 mostra um exemplo completo de modelagem, desde a especificação de um somador de 16 bits, o código VHDL do modelo e do *testbench*, e o resultado da simulação.

9.2.8 Abstração em VHDL: uma entidade, muitas arquiteturas

VHDL provê várias construções que facilitam o emprego de *abstração*. Vimos, na Seção 9.2.2, como definições podem ser agrupadas em **package**s e estas importadas pelas unidades de projeto que delas fazem uso. Outra construção útil é a possibilidade de definir mais de uma arquitetura para uma entidade.

Lembre da espiral de projeto: a cada avanço na espiral, a realização da especificação se torna mais concreta e mais próxima da implementação final.

Uma *entidade* define a interface de um componente com o mundo exterior àquele componente. Uma *arquitetura* é um refinamento, e uma das possíveis implementações, para a especificação daquele componente.

Esta forma de trabalhar é capturada de uma maneira simples por VHDL: uma entidade pode ser implementada por mais de uma arquitetura. Em geral, a primeira arquitetura desenvolvida é a mais abstrata e pode ser uma especificação executável; as intermediárias são mais detalhadas e contêm mais detalhes da implementação; a última é um modelo concreto que pode ser sintetizado como um circuito.

Do ponto de vista do código VHDL, caso haja mais do que uma arquitetura, o compilador emprega aquela última 'vista', que é a que está "mais longe" da definição da entidade no código fonte, e ignora a(s) arquitetura(s) definida(s) 'antes'. O Programa 9.12 mostra um exemplo.

Programa 9.12: Duas arquiteturas para uma entidade.

```
entity meuProjeto is
  port( ... )
end meuProjeto;

architecture especificacao of meuProjeto is
begin
   ...
end architecture especificacao;

architecture concreta of meuProjeto is
begin
   ...
end architecture concreta;
```

A entidade meuProjeto define a interface do componente. A arquitetura especificacao define a função e o comportamento desejado de meuProjeto. A arquitetura concreta é uma versão detalhada e que pode(ria) ser sintetizada diretamente pelo compilador VHDL. Na medida em que o projeto é refinado, cada arquitetura mais concreta é inserida no código abaixo da menos concreta, para que a última seja usada pelo compilador. Dessa forma, é fácil comparar diferentes versões de um mesmo modelo.

9.2.9 Um exemplo completo: somador de 16 bits

Vejamos um exemplo completo de modelagem estrutural em VHDL. Nosso objetivo é desenvolver um somador de 16 bits a partir de 16 cópias de um somador completo. Nosso projeto é uma extensão direta do somador de 4 bits apresentado na Seção 6.2.

Especificação Tipicamente, na fase inicial de um projeto que emprega VHDL como ferramenta de modelagem, um componente é especificado informalmente num elevado nível de abstração, como: *somador de 16 bits*. Esta especificação abstrata é então codificada num *modelo funcional*, similar à Equação 9.1, que define as interfaces e o efeito das operações de uma forma abstrata e que omite os detalhes da implementação.

$$
\begin{array}{l}
vem : \mathbb{B} \\
A, B : \mathbb{B}^{16} \\
S : \mathbb{B}^{16} \\
vai : \mathbb{B} \\
soma16 : (\mathbb{B} \times \mathbb{B}^{16} \times \mathbb{B}^{16}) \mapsto (\mathbb{B}^{16} \times \mathbb{B}) \\
soma16(vem, A, B, S, vai) \ \equiv \ (S, vai) = num(vem + A + B)
\end{array}
\tag{9.1}
$$

A especificação do somador define que os dois operandos A e B e o resultado S são representados em 16 bits e que a entrada de vem-um e a saída de vai-um são bits. O par (S, vai) é o resultado em 17 bits da soma de A, B e do vem-um.

O modelo funcional é então traduzido para um modelo mais concreto em que alguns dos detalhes do projeto são explicitados. Modelos neste nível são chamados de *modelo de fluxo de dados* (*dataflow*) ou *modelos RTL*. RTL é a abreviatura para *Register Transfer Language*, empregada para descrever o fluxo dos dados através de um circuito, no qual valores são transformados por circuitos combinacionais que são delimitados por pares de registradores.

Um *modelo estrutural* descreve a estrutura de um circuito, do ponto de vista das interconexões entre seus componentes. Dependendo das ferramentas disponíveis, um circuito pode ser sintetizado a partir da sua descrição no nível RTL ou do nível estrutural, porque estas descrições são suficientemente concretas e detalhadas para permitir a tradução automática e eficiente da descrição em VHDL para uma lista de componentes e ligações entre aqueles, que é então usada para produzir o circuito integrado. Esta lista de ligações e componentes é chamada de *net list*.

Modelo estrutural do somador completo Nosso projeto faz uso do pacote de definições do Programa 9.3. Definimos o comportamento das portas lógicas necessárias para implementar o somador completo da Seção 6.2 e salvamos o código VHDL num arquivo que chamamos de `aux.vhd`.

VHDL define as operações lógicas **and**, **or**, **xor** para operandos de tipo bit. O Programa 9.13 define o comportamento das portas lógicas *and*2, *or*3 e *xor*3. À primeira vista pode parecer que o código é extremamente verboso – são nove linhas de código para cada porta. Essas definições são coletadas em uma biblioteca para uso posterior, ou como é o caso aqui, mantidas num arquivo separado daquele em que está a parte principal do projeto. Uma vez que os modelos dos componentes estejam corretos, eles podem ser usados de forma mais econômica nos modelos mais complexos, mantidos em bibliotecas ou em arquivos fonte separados.

As três arquiteturas têm o mesmo nome (comport) porque são modelos comportamentais das respectivas portas lógicas. Os modelos são diferenciados pelos nomes de suas entidades, a saber and2, xor3 e or3.

Programa 9.13: Modelo funcional das portas lógicas.

```
entity and2 is
  port(A,B : in bit;
       S   : out bit);
  end and2;

  architecture comport of and2 is
  begin
    S <= A and B;
  end architecture comport;

entity xor3 is
  port(A,B,C : in bit;
       S     : out bit);
  end xor3;

  architecture comport of xor3 is
  begin
    S <= A xor B xor C;
  end architecture comport;

entity or3 is
  port(A,B,C : in bit;
       S     : out bit);
  end or3;

  architecture comport of or3 is
  begin
    S <= A or B or C;
  end architecture comport;
```

Definidos os componentes básicos, podemos escrever o *modelo funcional* do somador completo, como definido pela Equação 6.5. O bit com a soma é o ou-exclusivo das três entradas

$$s = a \oplus b \oplus vem$$

e o bit de vai-um é a soma de produtos das três entradas, tomadas duas a duas

$$vai = (a \wedge b) \ \vee \ (a \wedge vem) \ \vee \ (b \wedge vem)\,.$$

Estas equações são o corpo do modelo funcional do somador completo, como mostrado nas linhas entre o **begin** e o **end architecture** do Programa 9.14.

O *modelo estrutural* do somador completo também é mostrado no Programa 9.14. Como vimos na Seção 9.2.8, caso haja mais de uma, o compilador VHDL examina as várias arquiteturas e produz uma implementação para a última, na ordem do código fonte. Isso permite que o programador use a arquitetura abstrata (anterior) como a especificação para a arquitetura mais concreta (posterior). Note que as duas arquiteturas devem ter nomes distintos, que neste caso são funcional e estrutural .

O modelo estrutural é uma versão textual do diagrama do circuito com portas lógicas. São necessários três sinais internos ao modelo para interligas as saídas dos *and*2 às entradas do

*or*3. Estes sinais devem ser declarados entre o **architecture** e o **begin**. Além dos sinais internos, os componentes empregados no modelo também podem ser declarados explicitamente[2] **architecture** e o **begin**.

Programa 9.14: Modelo estrutural do somador completo.

```
use work.p_wires.all;
entity addBit is
  port(bitA, bitB, vem : in bit;     -- entradas A,B,vem-um
       soma, vai       : out bit);   -- saida C,vai-um
end addBit;

-- modelo funcional ------------------------------------------------
architecture funcional of addBit is
begin
  soma = bitA xor bitB xor vem;
  vai  = (bitA and bitB) or (bitA and vem) or (vem and bitB);
end architecture funcional;

-- modelo estrutural -----------------------------------------------
architecture estrutural of addBit is
  component and2 is
    port (A,B: in bit; S: out bit);
  end component and2;

  component or3 is
    port (A,B,C: in bit; S: out bit);
  end component or3;

  component xor3 is
    port (A,B,C: in bit; S: out bit);
  end component xor3;

  signal a1,a2,a3, : bit;
begin
  U_xor:  xor3 port map (bitA, bitB, vem, soma);
  U_and1: and2 port map (bitA, bitB, a1);
  U_and2: and2 port map (bitA, vem,  a2);
  U_and3: and2 port map (bitB, vem,  a3);
  U_or:   or3  port map (a1, a2, a3, vai);
end architecture estrutural;
```

Espaço em branco proposital.

[2] O autor considera uma boa prática declarar os explicitamente componentes, embora isso seja uma questão de estilo, ou de gosto.

Modelo estrutural do somador de 16 bits A Figura 9.6 mostra o circuito de um somador de 16 bits composto por 16 réplicas do somador completo, e é a extensão óbvia para 16 bits do somador mostrado na Figura 6.4.

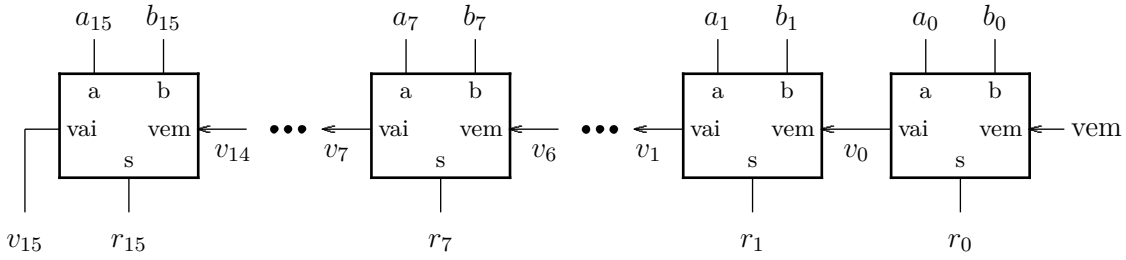


Figura 9.6: Somador composto de 16 cópias de *addBit*.

O Programa 9.15 contém a entidade adder16, com a interface especificada pela Equação 9.1: uma entrada de vem-um, dois operandos de 16 bits, a soma em 16 bits e o bit de vai-um.

Programa 9.15: Modelo estrutural do somador de 16 bits.

```
use work.p_wires.all;
entity adder16 is
  port(vem  : in  bit;   -- vem-um
       A, B : in  reg16; -- entradas de 16 bits
       S    : out reg16; -- saída em 16 bits
       vai  : out bit);  -- vai-um
end adder16;

architecture estrutural of adder16 is
  component addBit port(bitA, bitB, vem : in bit;
                        soma, vai       : out bit);
  end component addBit;

  signal v : reg16;   -- vetor de bits para a cadeia de vai-um

begin
  U_b0: addBit port map ( A(0), B(0), vem,  S(0), v(0) );
  U_b1: addBit port map ( A(1), B(1), v(0), S(1), v(1) );
  U_b2: addBit port map ( A(2), B(2), v(1), S(2), v(2) );
  U_b3: addBit port map ( A(3), B(3), v(2), S(3), v(3) );
  U_b4: addBit port map ( A(4), B(4), v(3), S(4), v(4) );
  U_b5: addBit port map ( A(5), B(5), v(4), S(5), v(5) );
  U_b6: addBit port map ( A(6), B(6), v(5), S(6), v(6) );
  U_b7: addBit port map ( A(7), B(7), v(6), S(7), v(7) );
  U_b8: addBit port map ( A(8), B(8), v(7), S(8), v(8) );
  U_b9: addBit port map ( A(9), B(9), v(8), S(9), v(9) );
  U_ba: addBit port map ( A(10),B(10),v(9), S(10),v(10) );
  U_bb: addBit port map ( A(11),B(11),v(10),S(11),v(11) );
  U_bc: addBit port map ( A(12),B(12),v(11),S(12),v(12) );
  U_bd: addBit port map ( A(13),B(13),v(12),S(13),v(13) );
  U_be: addBit port map ( A(14),B(14),v(13),S(14),v(14) );
  U_bf: addBit port map ( A(15),B(15),v(14),S(15), vai  );
end architecture estrutural;
```

O Programa 9.15 mostra o código da arquitetura estrutural da entidade adder16. A arquitetura faz uso de um único componente, que é o somador completo – as portas lógicas ficam escondidas no modelo addBit – e um vetor de bits para transportar os bits de vai-um entre cada par de somadores – o bit v(i) é gerado pelo somador do i-ésimo bit do resultado.

Entre o **architecture** e o **begin**, o modelo estrutural declara um único componente (addBit) e o vetor da cadeia de vai-um v de tipo reg16. No corpo da arquitetura são instanciadas 16 cópias desse componente, cada uma com seu *label* U_bi. Note que o mapeamento de portas é posicional e que a posição de cada sinal no modelo que usa o componente deve ser exatamente a mesma que na definição do componente. Ao sinal de interface vai é atribuído o valor do bit de vai-um produzido pelo somador dos bits mais significativos.

Este modelo, com o acréscimo de informação sobre temporização, é utilizado nos laboratórios, e uma versão mais sofisticada é empregada no processador do projeto/trabalho semestral.

Testbench O Programa 9.16 mostra o código de teste para o somador de 16 bits, e este é similar ao *testbench* para o modelo do *mux*-2, mostrado no Programa 9.10.

Por razões de espaço, o vetor de testes test_array contém somente três elementos. Teoricamente, seriam necessários $2^1 \times 2^{16} \times 2^{16}$ casos para testar *todas* as combinações para as entradas. A Seção 6.9 indica algumas das possibilidades para reduzir o número impraticável de vetores de testes sem sacrificar a confiança nos resultados.

O vetor de testes não inclui um campo para o vem-um, somente para as entradas a e b do somador, e dois campos para os valores esperados para a soma e o vai-um. Por isso o componente adderEstrut16 é instanciado com a constante '0' no campo correspondente ao vem-um. O acréscimo do vem-um no vetor de testes e no corpo do *testbench* fica como um exercício para a leitora e o leitor dedicados.

Espaço em branco proposital.

Programa 9.16: *Testbench* para o modelo estrutural do somador de 16 bits.

```
use work.p_wires.all;
entity tb_somador is
  -- entidade vazia
end tb_somador;

architecture TB of tb_somador is

  component adderEstrut16 is port(vem  : in bit;
                                  A, B : in bit_vector;
                                  C    : out bit_vector;
                                  vai  : out bit);
  end component adderEstrut16;

  type test_record is record
          a : reg16; b : reg16; -- entradas
          c : reg16;            -- saida esperada
          v : bit;              -- vai-um esperado
          end record;
  type test_array is array(positive range <>) of test_record;
  constant test_vectors : test_array := (
    -- a,      b,      c,    vai-um
    (x"1111",x"1111",x"2222",'0'), -- testes insuficientes
    (x"2222",x"2222",x"4444",'0'),
    (x"4444",x"4444",x"8888",'0'));

  signal inpA,inpB,res,esp_res : reg16; -- entradas de teste
  signal vai, esp_vai : bit;

begin -- TB
  U_addCad: adderEstrut16 port map('0', inpA, inpB, res, vai);

  U_testValues: process                 -- testa o circuito
    variable v : test_record;
  begin
    for i in test_vectors'range loop
      v := test_vectors(i);             -- valores desse teste
      inpA <= v.a;
      inpB <= v.b;
      esp_res <= v.c;
      esp_vai <= v.v;
      assert ((res = esp_res) and (vai = esp_vai))
        report LF & "somador␣estrutural:␣saida␣errada␣" &
        LF & "␣␣␣␣A="& BV2STR(inpA) & LF & "␣␣␣␣B="& BV2STR(inpB) &
        LF & "␣soma="& BV2STR(res) &  LF & "␣␣vai="& B2STR(vai)
        severity error;
        wait for 2 ns;              -- duração do passo de simulação
    end loop;
    wait;                           -- encerra processo
    end process;
end TB;
```

Em caso de erro, se o resultado ou se vai-um produzido(s) pelo circuito é diferente do(s) valore(s) esperado(s), então o simulador emite a mensagem de erro na tela:

```
somador estrutural: saida errada
A=2222   B=2222   soma=4444   vai=1.
```

A Figura 9.7 mostra uma parte da tela do `gtkwave` com o resultado da simulação do *testbench* no Programa 9.16. Os sinais `inpa` (*input A*) e `inpb` (*input B*) são as entradas, o resultado esperado é `esp_res`, e a saída do somador são os sinais `rescad` (*resultado cadeia*) e `vaicad` (*vai-um cadeia*). O diagrama de tempos mostra os resultados para a soma dos pares de operandos definidos pelos elementos do registro `test_array`, com a soma de um par de `x"0000"`s no início e no final da simulação.

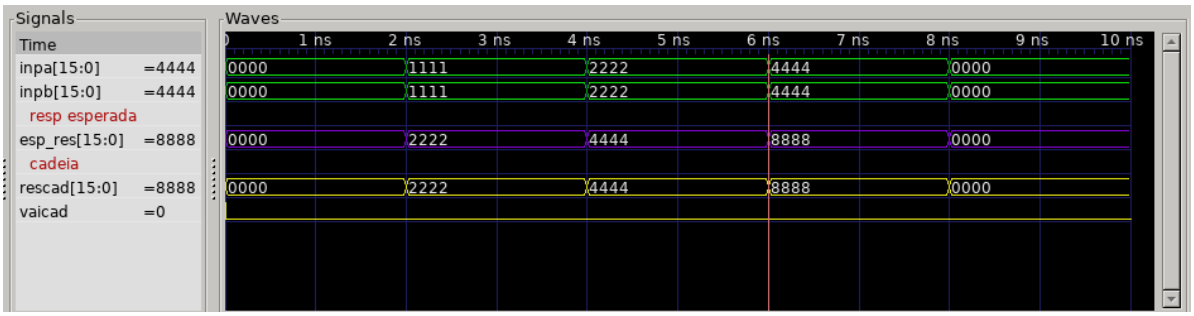


Figura 9.7: Resultado da simulação com o *testbench* do somador de 16 bits.

Os sinais na entrada do somador se alteram a cada 2ns, o que facilita a visualização da sequência de vetores de testes e resultados. Para todas os pares de entradas, a soma produzida pelo modelo é igual ao esperado, logo, para o conjunto *limitadíssimo* de valores empregados neste teste, o modelo do somador está correto. A Seção 6.9 discute a geração de vetores de teste mais completos e eficazes.

9.2.10 Síntese para *Field Programmable Gate Arrays*

O resultado da compilação para *síntese* pode ser a programação de um *Field Programmable Gate Array*, ou ainda pode ser um conjunto de máscaras para fabricar um *Application Specific Integrated Circuit* (ASIC).

Um *Field Programmable Gate Array* (FPGA) é um circuito integrado programável composto por uma matriz com milhares (10^4 a 10^5) de células programáveis. Por célula programável entenda-se algo como o mostrado na Figura 9.8, que é um modelo altamente simplificado.

Cada célula contém um *flip-flop*, duas *look-up tables* (LUT) e multiplexadores para interligar sinais. Uma LUT nada mais é do que um multiplexador com 8 ou 16 entradas, e estas entradas podem ser ligadas para implementar qualquer função lógica de 3 ou 4 variáveis. Tanto as LUTs quando os multiplexadores na entrada do *flip-flop* são acionados pelos bits da *memória de programação.*.

A memória de programação é uma RAM que não pode ser acessada diretamente e que é preenchida quando o FPGA é programado com o 'programa' gerado pelo compilador/sintetizador. Quando o FPGA é inicializado, esta memória é carregada com o padrão de bits necessário para colocar os multiplexadores nas posições apropriadas e implementar as funções das LUTs.

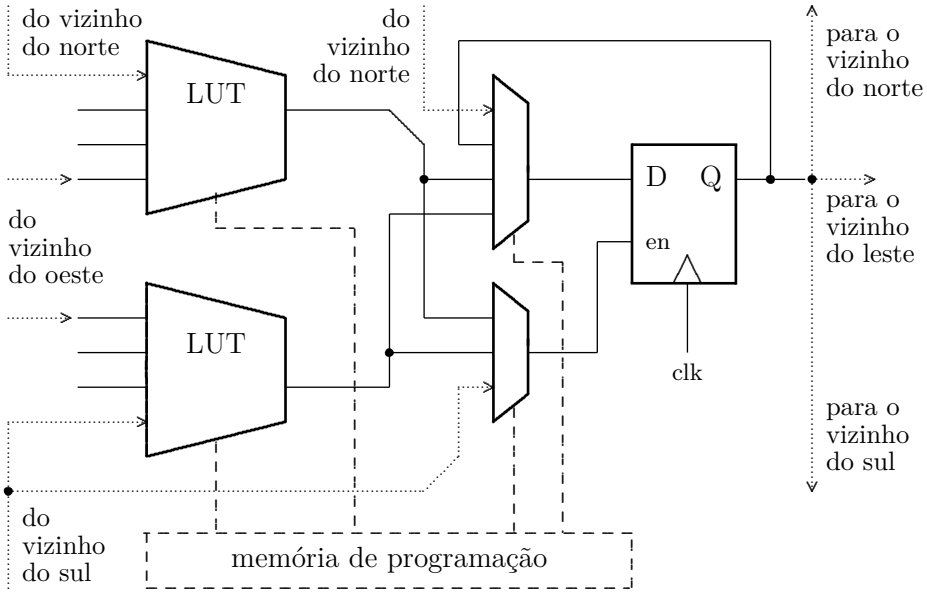


Figura 9.8: Modelo simplificado de uma célula de um FPGA.

Quando se compila VHDL para *sintetizar* um FPGA, o que o compilador produz é o conteúdo para a memória de programação, de forma a configurar o FPGA para que se comporte como definido no código fonte.

O circuito da Figura 9.8 mostra, nas linhas pontilhadas, as ligações para os *flip-flops* e LUTs das células vizinhas, o que permite implementar contadores, registradores, deslocadores ou máquinas de estado.

As células são organizadas em matrizes, separadas por caminhos largos que interligam as matrizes entre si e com células de interface para a periferia. Estas células de interface são circuitos especializados para interligar o FPGA a outros dispositivos ou circuitos integrados.

9.2.11 Comandos concorrentes de seleção: *when-else* e *with-select*

Vejamos o que VHDL nos oferece de mais abstrato do que representações textuais para projetos com portas lógicas. Sendo uma linguagem assaz versátil, VHDL nos permite trabalhar com baixo nível de abstração – interligação de portas lógicas – e também com código um tanto mais abstrato. Vejamos as duas maneiras de modelar seleção.

A *área concorrente* de uma arquitetura é a região de código entre o **begin** e o **end architecture**. O trecho de código no Programa 9.17 mostra a arquitetura chamada exemplo da entidade area_concorrente. Na área concorrente estão dois comandos, ou dois processos, que são executados concorrentemente: a disjunção e a conjunção de dois sinais. Num circuito, esta arquitetura seria implementada com duas portas lógicas que constantemente amostram suas entradas e produzem suas saídas em função daquelas. O modelo VHDL deve ter o mesmo comportamento e portanto os dois processos executam concorrentemente: qualquer alteração em a e/ou em b provocará a avaliação do lado direito das atribuições, e se for o caso, eventos nos sinais r e s.

Programa 9.17: Área concorrente num arquitetura.

```
architecture exemplo of area_concorrente is
begin
  -- inicio da área concorrente

  U_or:  r <= a or  b;
  U_and: s <= a and b;

  -- final da área concorrente
end architecture exemplo;
```

A linguagem nos oferece dois *comandos concorrentes* de seleção que podem ser usados na área concorrente: o comando **when−else** e o comando **with−select**.

when-else A forma mais simples do comando de atribuição condicional **when−else** é

```
a when cond else b;
```

Se a condição cond é verdadeira, então o comando tem o valor de a, do contrário o comando tem o valor de b. Quando usado numa atribuição, este comando se comporta como um comando de seleção de C: `s = (cond ? a : b);`

```
s <= a when cond else b;
```

O comando pode ser encadeado com várias cláusulas **else**:

```
s <= a when cond_1 else
     b when cond_2 else
     c when cond_3 else
     d;
```

As condições deste comando são avaliadas na ordem do texto: primeiro cond_1, se esta é falsa então cond_2 é avaliada, e se esta é falsa então cond_3 é avaliada. Se a última cláusula é falsa, então o valor atribuído é o que segue o último **else**.

O comando **when−else** pode ser usado para implementar um *mux*-4:

```
z <= a when (s = "00") else
     b when (s = "01") else
     c when (s = "10") else
     d;
```

Um modelo completo é mostrado no Programa 9.18, para entradas e saídas que são vetores de 8 bits – o multiplexador tem 4 entradas de 8 bits cada. Este modelo é um tanto mais abstrato do que um modelo estrutural com a mesma função.

Espaço em branco proposital.

Programa 9.18: Modelo do *mux4x8* com when-else.

```
entity mux4x8 is
  port(a,b,c,d: in  bit_vector(7 downto 0);
       s:       in  bit_vector(1 downto 0);
       z:       out bit_vector(7 downto 0));
end mux4x8;

architecture when_else of mux4x8 is
begin
   z <= a when (s = "00") else  -- atribuição condicional
        b when (s = "01") else
        c when (s = "10") else
        d;                      -- todos os demais casos
end architecture when_else;
```

with-select O comando concorrente de atribuição selecionada **with−select** se parece com um `switch` da linguagem C. Para um sinal de seleção `s` que pode assumir um dentre dois valores (`v1`,`v2`), o valor do comando **with−select** corresponde ao valor selecionado. Vejamos um exemplo.

```
  with s select           -- s pode ser v1 OU v2
    r <= a when v1,
         b when v2;
```

Uma cláusula **others** pode ser usada para capturar todos os demais casos:

```
  with s select           -- s pode ser v1 OU v2
    r <= a when v1,
         b when others;   -- todos os demais casos
```

Vejamos como fica o nosso *mux*-4, com um comando de atribuição selecionada.

```
  with s select
    z <= a when "00",
         b when "01",
         c when "10",
         d when others;   -- todos os demais casos
```

Ao contrário do **when−else**, todas as cláusulas do **with−select** tem a mesma precedência e geralmente este comando resulta em circuitos menores e mais rápidos. O Programa 9.19 mostra a arquitetura do *mux*-4x8 com **with−select**.

Programa 9.19: Modelo do *mux4x8* com with-select.

```
architecture with_select of mux4x8 is
begin
  with s select           -- atribuição selecionada
    r <= a when "00",
         b when "01",
         c when "10",
         d when others;   -- todos os demais casos
end architecture with_select;
```

9.3 Modelagem da temporização de circuitos combinacionais

VHDL permite que informação de tempo seja agregada às descrições dos circuitos, possibilitando assim a simulação temporal dos modelos. Tipicamente, estas simulações são usadas em fases iniciais do projeto porque a "anotação 'manual" da temporização dos circuitos permite prever o comportamento dinâmico e a velocidade máxima de operação. As ferramentas de síntese produzem modelos temporais detalhados e com melhor acurácia do que modelos produzidos manualmente e servem para confirmar as previsões das fases iniciais do projeto.

9.3.1 Modelo funcional *versus* modelo temporal

Nas Seções 9.2.3 e 9.2.9 empregamos *modelos funcionais* para portas lógicas, e com aqueles construímos modelos para circuitos combinacionais compostos de várias portas lógicas. Aqueles modelos são chamados de *funcionais* porque representam somente o aspecto *função lógica* dos circuitos, e ignoram informações de tempo.

Nesta seção mostramos como enriquecer os modelos funcionais com informação de tempo, criando modelos mais sofisticados para o comportamento dos circuitos.

Modelos temporais incorporam a informação de tempo de propagação dos componentes e permitem simulações mais detalhadas e realistas dos circuitos. Evidentemente, simulações mais detalhadas são mais custosas em termos de tempo de programação, tempo de simulação e de memória.

Os modelos que usamos nesta seção permitem a visualização do comportamento dos circuitos combinacionais ao longo do tempo, e a medição do seu tempo de propagação. Em breve estudaremos a propagação de sinais em somadores e nestes circuitos os problemas são algo mais severos do que veremos aqui. Cada coisa a seu tempo.

Antes de prosseguirmos, reveja a definição de *circuito combinacional* da Seção 4.1.

9.3.2 Descrição de temporização em VHDL

A atribuição a um sinal pode ser postergada para modelar o comportamento de circuitos reais. Por exemplo, o modelo para o comportamento de um inversor pode incluir o tempo de propagação com o atributo **after**:

```
S <= inertial (not A) after 4 ns;
```

O complemento de A é atribuído a S após o tempo de propagação de 4 nanosegundos. O atributo **after** faz com que o evento no sinal S seja postergado para ocorrer 4ns no futuro, contados desde o tempo corrente. O qualificador **inertial** é optativo e pode ser omitido.

VHDL possibilita a modelagem de dois tipos de atrasos, atraso inercial e atraso de transporte. O *atraso inercial* modela o tempo de propagação de portas lógicas. Com atraso inercial, pulsos com duração menor que o intervalo especificado pelo **after** não são transferidos, por causa da *inércia* natural dos sinais – um pulso na entrada de um circuito é propagado para sua saída caso dure o suficiente para alterar os valores em todos os nós internos ao circuito. A Figura 9.9 mostra um diagrama de tempo de uma simulação na qual ocorrem três pulsos no sinal A e o efeito do comando acima no sinal S. Os dois pulsos iniciais têm duração maior ou igual a 4ns e são transferidos para S; o terceiro pulso tem uma duração de 3ns, menor do que a indicada pelo **after**, não é transferido para S e portanto ignorado.

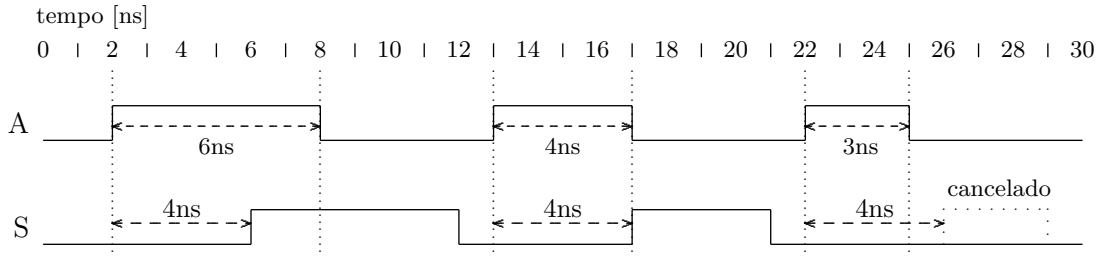


Figura 9.9: Atraso inercial com *after*.

O *atraso de transporte*, indicado pelo atributo **transport**, modela a propagação através de um fio e portanto pulsos de qualquer duração são propagados. O comando abaixo corresponde a um fio longo, com um atraso na propagação de sinais de 2ns.

```
T <= transport A after 2 ns;
```

A Figura 9.10 mostra o diagrama de tempo da simulação da propagação através do fio (longo) que conecta os nós A e T.

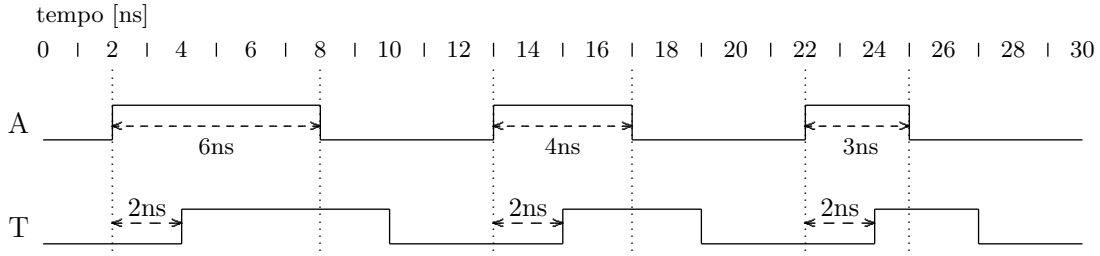


Figura 9.10: Atraso de transporte.

A modelagem do tempo de propagação pode ser refinada para evitar que pulsos com largura menor do que o especificado por **after** sejam rejeitados. A cláusula **reject** faz com que pulsos de largura menor ou igual a seu argumento sejam eliminados; pulsos mais largos que o argumento de **reject** são transferidos. Se o argumento de **after** é t_a, então o argumento de **reject** deve estar no intervalo $[0, t_a]$.

No comando

```
U <= reject 2.5 ns inertial A after 4 ns;
```

a atribuição a U, por conta da cláusula **reject**, rejeita pulsos em A com até 2,5ns de duração, mas propaga pulsos com duração maior que o limite de rejeição. O qualificador **inertial** não pode ser omitido quando se emprega a cláusula **reject**.

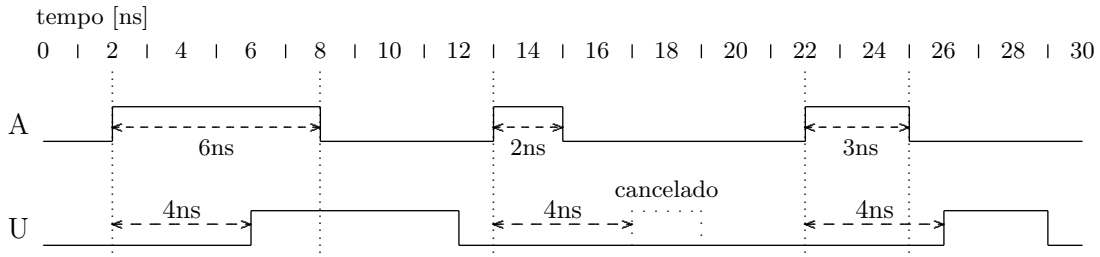


Figura 9.11: Atraso inercial com *reject*.

Não há uma correspondência direta entre o modelo de temporização definido na Seção 5.4 e os três modelos para simular temporização disponibilizados por VHDL. Quanto a contaminação, o modelo para a simulação de tempo com VHDL considera que, (i) não existe contaminação e todos os pulsos são transferidos – o intervalo de rejeição é especificado como zero; ou (ii) a contaminação dura tanto quanto a propagação, e para tanto basta especificar o tempo de propagação com **after**.

A capacitância interna aos nós do circuito é considerada apenas em termos da inércia na transmissão de sinais entre as entradas e as saídas. Se a capacitância no caminho de um sinal é baixa, então o circuito responde a pulsos estreitos – para isso empregamos uma especificação com ambos **reject** e **after**

```
U <= reject t_rej ns inertial A after t_prop;
```

Pulsos com duração menor ou igual a `t_rej` não são transferidos para a saída e portanto a saída do circuito permanece imutável – como se estivesse contaminada – durante o intervalo de rejeição. Pulsos com duração maior do que `t_rej` são transferidos com um atraso de `t_prop`.

A Figura 9.12 mostra um diagrama de tempo para o modelo de temporização para atrasos inerciais com rejeição de pulsos estreitos. As entradas do modelo inicialmente estão na configuração α e sua saída é $f(\alpha)$. Ocorre uma mudança de α para β, possivelmente por causa de alguma corrida de sinais nos circuitos que alimentam o modelo, e esta alteração nas entradas não é refletida para a saída porque sua duração é menor do que o tempo de contaminação T_C do modelo. A nova configuração de entradas γ persiste por mais do que T_C, e decorrido o tempo de propagação T_P, a saída torna-se $f(\gamma)$.

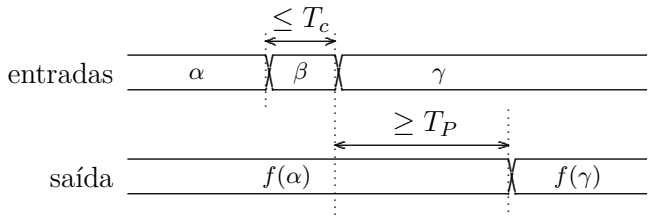


Figura 9.12: Modelagem da temporização em VHDL.

O modelo de temporização de VHDL é similar ao modelo definido na Seção 7.6. O modelo para o tempo de propagação tem a mesma definição: uma vez que as entradas estejam válidas e estáveis, as saídas se tornam válidas e estáveis após T_P. A rejeição de pulsos estreitos tem uma interpretação similar àquela do tempo de contaminação, no sentido de que a saída não é alterada a menos que a alteração nas entradas tenha duração suficiente para provocar uma alteração na saída que perdure até que o nível na saída estabilize.

9.3.3 Definição dos parâmetros de temporização

Há quatro formas de codificar os parâmetros de temporização das portas lógicas, que também são usadas para definir quaisquer constantes empregadas nos modelos. Estas quatro formas satisfazem regras óbvias de escopo, de forma que as constantes fiquem disponíveis: (i) localmente na arquitetura; (ii) em todas as arquiteturas de uma entidade; (iii) nas entidades que importam o conteúdo de um pacote; e (iv) através de parâmetros genéricos.

Na primeira forma, as constantes são definidas na arquitetura, como mostra o Programa 9.20. Esta forma é menos flexível do que a anterior porque as constantes ficam visíveis somente no escopo da arquitetura em que foram definidas.

Programa 9.20: Parâmetros de temporização definidos na arquitetura.

```
entity or2time is
  port(A,B : in bit;
       S   : out bit);
end or2time;

architecture temporiz of or2time is
  constant p_or2 : time := 20.0 ps;
  constant r_or2 : time :=  5.0 ps;
begin
    S <= reject r_or2 inertial (A or B) after p_or2;
end architecture temporiz;
```

A segunda maneira de definir as constantes de temporização é defini-las na entidade, como mostra o Programa 9.21. As constantes ficam assim disponíveis para quaisquer das arquiteturas associadas àquela entidade.

Programa 9.21: Parâmetros de temporização definidos na entidade.

```
entity and2time is
  port(A,B : in bit;
       S   : out bit);
  constant p_and2 : time := 20.0 ps;
  constant r_and2 : time :=  5.0 ps;
end and2time;

architecture temporiz of and2time is
begin
    S <= reject r_and2 inertial (A and B) after p_and2;
end architecture temporiz;
```

Espaço em branco proposital.

A terceira forma é colocar as definições num pacote. Por exemplo, os tempos de propagação das portas lógicas (p_inv, p_and2, p_or2), assim como os intervalos de rejeição (r_inv, r_and2 e r_or2), são constantes declaradas na segunda versão do pacote p_wires, mostrada no Programa 9.22. Assim, a informação quanto aos parâmetros de temporização das portas lógicas fica concentrada num único lugar/arquivo.

Programa 9.22: Abreviaturas, definições e temporização.

```
package p_WIRES is -- versão com temporização

  constant p_inv  : time := 15.0 ps;
  constant r_inv  : time :=  4.0 ps;

  constant p_and2 : time := 20.0 ps;
  constant r_and2 : time :=  5.0 ps;

  constant p_or2  : time := 22.0 ps;
  constant r_or2  : time :=  6.0 ps;

  subtype reg2  is bit_vector(1  downto 0);
  subtype reg4  is bit_vector(3  downto 0);
  subtype reg8  is bit_vector(7  downto 0);
  subtype reg16 is bit_vector(15 downto 0);
  ...
package body p_WIRES is
  ...
end package p_WIRES;
```

A quarta maneira emprega *interfaces genéricas*, que permitem redefinir as constantes a cada vez que um componente é instanciado. A entidade define um valor *default* e os parâmetros definidos como *genéricos* podem ser alterados em cada instanciação. Vejamos como o modelo da porta *and*2 do Programa 9.21 pode ser adaptado para suportar constantes genéricas para temporização. O Programa 9.23 mostra a unidade de projeto para a porta *and*2. As constantes p_and2 e r_and2 são importadas do pacote p_wires. Os parâmetros genéricos são declarados na cláusula **generic**, antes da declaração dos sinais de interface do componente.

Programa 9.23: Parâmetros de temporização genéricos da porta *pand2*.

```
use work.p_wires.all;
entity and2gen is
  generic (prop : time := p_and2, reje : time := r_and2);
  port(A,B : in bit;
       S   : out bit);
end and2gen;

architecture generica of and2gen is
begin
    S <= reject reje inertial (A and B) after prop;
end architecture generica;
```

Veremos em seguida, na Seção 9.3.4, o modo de emprego dos parâmetros genéricos.

9.3.4 Modelo temporizado do multiplexador

A entidade e a arquitetura do multiplexador que vimos na Seção 9.2.3 são estendidas para fazer uso dos parâmetros genéricos de temporização. Os modelos do inversor e da porta *or*2 são similares ao mostrado no Programa 9.23. A entidade do *mux*-2 não sofre nenhuma alteração; contudo, os componentes na arquitetura são declarados e instanciados com a informação de tempo.

Note que a entidade mux2 não importa a biblioteca do IEEE porque nenhuma função daquela biblioteca é empregada nesta unidade de projeto. As funções da biblioteca do IEEE ficam escondidas nos modelos dos componentes. O modelo do *mux*-2 faz uso e importa a biblioteca de trabalho.

Entre o **architecture** e o **begin** estão declarados os componentes, com as interfaces genéricas. No corpo da arquitetura, entre o **begin** e o **end architecture**, os componentes são instanciados, cada um com os parâmetros de tempo adequados. O inversor Ui tem o seu tempo de propagação acrescido de 5ps enquanto que o parâmetro da rejeição é inalterado – o **open** indica que o valor *default* declarado[3] na unidade de projeto do inversor é que deve ser usado.

Espaço em branco proposital.

[3] Aparentemente, o compilador `ghdl` *não* aceita o **open** no lugar de um valor explícito, embora [Ash08] assim o preconize. Este autor prefere o padrão à implementação.

Programa 9.24: Modelo do *mux-2* com informação de temporização.

```
use work.p_wires.all;
entity mux2 is
  port(a,b : in  bit;
       s   : in  bit;
       z   : out bit);
end mux2;

architecture temporiz of mux2 is
  -- declaração dos componentes
  component inv is
    generic (prop : time; reje : time); -- propagação, rejeição
    port(A : in bit; S : out bit);
  end component inv;

  component and2 is
    generic (prop : time; reje : time); -- propagação, rejeição
    port(A,B : in bit; S : out bit);
  end component and2;

  component or2 is
    generic (prop : time; reje : time); -- propagação, rejeição
    port(A,B : in bit; S : out bit);
  end component or2;

  -- sinais internos ao mux2
  signal r, p, q : bit;

begin
  -- instanciação dos componentes
  Ui:  inv  generic map (10 ps, 2 ps) port map (s, r);
  Ua0: and2 generic map (16 ps, 4 ps) port map (a, r, p);
  Ua1: and2 generic map (16 ps, 1 ps) port map (b, s, q);
  Uor: or2  generic map ( 9 ps, 3 ps) port map (p, q, z);
end architecture temporiz;
```

As portas *and*2 Ua0 e Ua1 usam o tempo de propagação definido em `packageWires.vhd`, Ua1 altera o tempo de rejeição para 1ps. O *or*2 (Uor) altera seu tempo de propagação *default* em 9ps e redefine a rejeição para 3ps.

Esta forma de modelagem *esconde toda a informação que pode ser escondida* (*information hiding*): ***toda*** *a informação sobre tempo está escondida na arquitetura do mux-2*, e sua entidade não contém nenhuma informação sobre a temporização. Quando usarmos o *mux*-2 em outros modelos, tal como para implementar um *mux*-4, nenhuma informação sobre tempo de propagação do *mux*-2 é transferida explicitamente para o *mux*-4 porque tal informação está escondida na arquitetura do *mux*-2, e fica portanto invisível ao usuário do modelo do *mux*-4. O usuário dos modelos baseados no *mux*-2 não mais se ocupa com definir a temporização dos circuitos.

A Figura 9.13 mostra o digrama de tempo, produzido com `gtkwave`, da simulação do *mux*-2. As entradas estão no vetor de bits entr_2, o sinal de seleção é s0 e a saída simulada é o sinal saidamux2. O sinal esperadamux é a saída esperada para um modelo correto *de acordo com a especificação funcional*, tal como aquela no Programa 9.9.

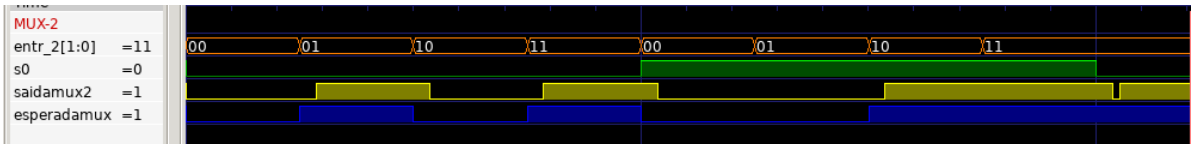


Figura 9.13: Pulsos transitórios na saída do multiplexador.

A menos do atraso de propagação – o sinal saidamux2 está atrasado com relação à esperadamux – o comportamento do modelo é idêntico ao especificado. *Sempre?* No lado direito, após o último valor de teste (entr_2="11"), há um pulso transitório em saidamux2, que seguramente não deveria estar ali. Reveja a Seção 5.4.3 para a causa deste pulso e para um acréscimo ao circuito do multiplexador que elimina o problema.

Note ainda que as entradas permanecem como entr_2="11" *após* o último vetor de teste. Se as entradas, ou esperadamux, tivessem retornado para zero após o último vetor de testes da sequência (entr_2="11" para entr_2="00") então é possível que o pulso espúrio passasse desapercebido e o modelo estaria errado. Ao gerar vetores de testes é importante, mais que isso, é *fundamental* considerar casos limite que podem esconder ou elucidar justamente o comportamento transitório que viola a especificação do circuito.

9.3.5 Modelo temporizado do somador de 16 bits

Nesta seção veremos a codificação em VHDL dos modelos para os somadores apresentados na Seção 6.6, um com adiantamento de vai-um de quatro em quatro bits, e outro com adiantamento de 16 em 16 bits. Iniciemos com o circuito para o adiantamento de 4 em 4 bits.

Primeiramente, vejamos um comando assaz interessante que é o **generate**. Este é um tipo especial de laço que, ao invés de iterar durante a simulação do modelo, ele itera durante a síntese (compilação) e replica as linhas de código em seu corpo ajustando os índices de acordo com o espaço de iteração. Ao invés de a repetição ocorrer durante a execução, ela se dá sobre o código fonte e permite escrever descrições concisas de estruturas cuja forma se repete de uma maneira relativamente simples.

O trecho de código abaixo replica a linha em seu corpo – que equivale a uma porta *and*2 – com o índice i variando de 3 até 0 e produzindo quatro réplicas de uma porta *and*2 com saída c(i) e entradas a(i) e b(i).

```
gen: for i in 3 downto 0 generate
  c(i) <= (a(i) and b(i)) after p_and2;
end generate gen;
```

É possível adicionar informação de temporização aos sinais gerados, como o tempo de propagação indicado acima. No que segue, empregamos o modelo mais completo de temporização, com tempos de propagação (**after**) e contaminação (**reject**):

```
gen: for i in 3 downto 0 generate
  c(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
end generate gen;
```

Adiantamento de 4 em 4 bits

O Programa 9.25 mostra o modelo do circuito de adiantamento de vai-um para 4 bits. O código é uma transcrição literal das Equações 6.20 a 6.23, com informação de tempo de propagação associada aos vários sinais. Para modelar a temporização dos sinais, por exemplo, o atraso do sinal vai(3) é a soma do atraso dominante, que é aquele na porta *or*5 mais o atraso na porta *and*5. Este modelo não é estrutural mas funcional: ao invés da estrutura de interligação das portas lógicas, o código é uma transcrição para VHDL das Equações 6.20 a 6.23.

Programa 9.25: Adiantamento de vai-um para quatro bits.

```
use work.p_wires.all;
entity adianta4 is
  port(a,b : in reg4;           -- entradas A(i),B(i)
       vem : in bit;            -- vem-um
       vai : out reg4);         -- vai(i)
end entity adianta4;

architecture funcional of adianta4 is
  signal p,g : reg4;
begin
  gen_g_p: for i in 3 downto 0 generate
    g(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
    p(i) <= reject (r_or2)  inertial (a(i) or  b(i)) after p_or2;
  end generate gen_g_p;

  vai(0) <= g(0) or (p(0) and vem) after (p_and2+p_or2);
  vai(1) <= g(1) or (p(1) and g(0)) or (p(1) and p(0) and vem)
            after (p_and3+p_or3);
  vai(2) <= g(2) or (p(2) and g(1)) or (p(2) and p(1) and g(0)) or
            (p(2) and p(1) and p(0) and vem) after (p_and4+p_or4);
  vai(3) <= g(3) or (p(3) and g(2)) or (p(3) and p(2) and g(1)) or
            (p(3) and p(2) and p(1) and g(0)) or
            (p(3) and p(2) and p(1) and p(0) and vem)
            after (p_and5+p_or5);
end architecture funcional;
```

Para simplificar o código, considera-se somente o tempo de propagação para a geração dos sinais vai(i). Para modelar a contaminação, o código seria

```
  vai(0) <= reject (r_or2) inertial (g(0) or (p(0) and vem))
            after (p_and2+p_or2);
```

sem esquecer que o intervalo de rejeição deve ser o mínimo dentre todos os caminhos. Neste caso, o caminho mais curto entre as entradas e a saída é g(0) **or** (...) .

O somador de 16 bits, com os circuitos de adiantamento de 4 em 4 bits é mostrado na Figura 6.17 e seu modelo no Programa 9.26, que é um modelo estrutural.

A cadeia de adiantamento não é usada porque os bits de vai-um de cada quarteto são gerados pelo adiantador correspondente. A cada quatro bits, o bit de vai-um (vai(3) no Programa 9.25) produzido pelo circuito adiantador é inserido na entrada de vem-um dos bits nas posições $4, 8$ e 12. O bit de vai-um produzido pelo quarto adiantador (c(15)) é o próprio vai-um do somador de 16 bits.

Os sinais de vai-um gerados em cada somador completo (addBit) são ignorados e por isso estão marcados como **open**.

Programa 9.26: Somador de 16 bits com adiantamento de 4 em 4 bits.

```
architecture adderAdianta4 of adder16 is
  component addBit port(bitA, bitB, vem : in bit;
                        soma, vai       : out bit);
  end component addBit;

  component adianta4 port(a,b : in reg4; vem : in bit;
                          vai: out reg4);
  end component adianta4;

  signal c : reg16;          -- cadeia de adiantamento de vai-um
begin
 U_a0_3: adianta4 port map -- adianta vai(0)..vai(3)
    (A(3 downto 0),B(3 downto 0),vem,c(3 downto 0));

  U_b0: addBit port map ( A(0),B(0),vem, S(0),open );
  U_b1: addBit port map ( A(1),B(1),c(0),S(1),open );
  U_b2: addBit port map ( A(2),B(2),c(1),S(2),open );
  U_b3: addBit port map ( A(3),B(3),c(2),S(3),open );

  U_a4_7: adianta4 port map -- adianta vai(4)..vai(7)
    (A(7 downto 4),B(7 downto 4),c(3),c(7 downto 4));

  U_b4: addBit port map ( A(4),B(4),c(3),S(4),open );
  U_b5: addBit port map ( A(5),B(5),c(4),S(5),open );
  U_b6: addBit port map ( A(6),B(6),c(5),S(6),open );
  U_b7: addBit port map ( A(7),B(7),c(6),S(7),open );

  U_a8_b: adianta4 port map -- adianta vai(8)..vai(11)
    (A(11 downto 8),B(11 downto 8),c(7),c(11 downto 8));

  U_b8: addBit port map ( A(8), B(8), c(7), S(8), open );
  U_b9: addBit port map ( A(9), B(9), c(8), S(9), open );
  U_ba: addBit port map ( A(10),B(10),c(9), S(10),open );
  U_bb: addBit port map ( A(11),B(11),c(10),S(11),open );

  U_a12_15: adianta4 port map -- adianta vai(12)..vai(15)
    (A(15 downto 12),B(15 downto 12),c(11),c(15 downto 12));

  U_bc: addBit port map ( A(12),B(12),c(11),S(12),open );
  U_bd: addBit port map ( A(13),B(13),c(12),S(13),open );
  U_be: addBit port map ( A(14),B(14),c(13),S(14),open );
  U_bf: addBit port map ( A(15),B(15),c(14),S(15),open );

  vai <= c(15);              -- vai-um, sinal da interface
end architecture adderAdianta4;
```

Adiantamento de 16 bits

O modelo do circuito de adiantamento de 16 bits é mostrado no Programa 9.27. O código é uma transcrição literal das Equações 6.24, 6.25 e 6.26. Como VHDL ignora maiúsculas e minúsculas, no modelo os sinais P_i e G_i são chamados de pp(i) e gg(i), respectivamente. Os comandos que geram os sinais pp(i) e gg(i) poderiam ser substituídas por comandos **generate**, com expressões apropriadas para os índices.

Programa 9.27: Adiantamento de 16 em 16 bits.

```
use work.p_wires.all;
entity adianta16 is
  port(a,b : in reg16;          -- entradas A(i),B(i)
       vem : in bit;            -- vem-um
       vai : out reg4);         -- vai(i), de 4 em 4 bits
end adianta16;

architecture funcional of adianta16 is
  signal p,g : reg16;
  signal pp,gg : reg4;
begin
  gen: for i in 15 downto 0 generate
    g(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
    p(i) <= reject (r_or2)  inertial (a(i) or  b(i)) after p_or2;
  end generate gen;

  pp(0) <= p(3) and p(2) and p(1) and p(0) after p_and4;
  pp(1) <= p(7) and p(6) and p(5) and p(4) after p_and4;
  pp(2) <= p(11) and p(10) and p(9) and p(8) after p_and4;
  pp(3) <= p(15) and p(14) and p(13) and p(12) after p_and4;

  gg(0) <= g(3) or (p(3) and g(2)) or (p(3) and p(2) and g(1)) or
           (p(3) and p(2) and p(1) and g(0)) after p_or4+p_and4;
  gg(1) <= g(7) or (p(7) and g(6)) or (p(7) and p(6) and g(5)) or
           (p(7) and p(6) and p(5) and g(4)) after p_or4+p_and4;
  gg(2) <= g(11) or (p(11) and g(10)) or (p(11) and p(10) and g(9)) or
           (p(11) and p(10) and p(9) and g(8)) after p_or4+p_and4;
  gg(3) <= g(15) or (p(15) and g(14)) or (p(15) and p(14) and g(13)) or
           (p(15) and p(14) and p(13) and g(12)) after p_or4+p_and4;

  vai(0) <= gg(0) or (pp(0) and vem) after p_or2+p_and2;
  vai(1) <= gg(1) or (pp(1) and gg(0)) or (pp(1) and pp(0) and vem)
            after p_or3+p_and3;
  vai(2) <= gg(2) or (pp(2) and gg(1)) or (pp(2) and pp(1) and gg(0)) or
            (pp(2) and pp(1) and pp(0) and vem) after p_or4+p_and4;
  vai(3) <= gg(3) or (pp(3) and gg(2)) or (pp(3) and pp(2) and gg(1)) or
            (pp(3) and pp(2) and pp(1) and gg(0)) or
            (pp(3) and pp(2) and pp(1) and pp(0) and vem)
            after p_or5+p_and5;
end architecture funcional;
```

O modelo estrutural do somador de 16 bits com adiantamento de vai-um otimizado é mostrado no Programa 9.28. O modelo é similar àquele do Programa 9.26, exceto pela instanciação do

circuito de adiantamento de 16 bits, e que os sinais de vem-um dos circuitos de adiantamento de 4 em 4 bits são gerados pelo adiantador de 16 bits, nas posições 4, 8 e 12. O sinal de vai-um do somador é o bit mais significativo do adiantador de 16 bits, *cc(3)*.

Programa 9.28: Somador de 16 bits com adiantamento de 16 bits.

```
architecture adderAdianta16 of adderAdianta16 is
  component addBit port( ... ) end component addBit;
  component adianta4 port( ... ) end component adianta4;
  component adianta16 port( ... ) end component adianta16;

  signal v  : reg16;       -- adiantamento de 4 em 4 bits
  signal cc : reg4;        -- adiantamento de 16 bits
begin
  U_a15_0: adianta16 port map (A,B,vem,cc);

  U_a3_0: adianta4 port map
    (A(3 downto 0),B(3 downto 0),vem,v(3 downto 0));

  U_b0: addBit port map ( A(0),B(0),vem, S(0),open );
  U_b1: addBit port map ( A(1),B(1),v(0),S(1),open );
  U_b2: addBit port map ( A(2),B(2),v(1),S(2),open );
  U_b3: addBit port map ( A(3),B(3),v(2),S(3),open );

  U_a4_7: adianta4 port map
    (A(7 downto 4),B(7 downto 4),cc(0),v(7 downto 4));

  U_b4: addBit port map ( A(4),B(4),cc(0),S(4),open );
  U_b5: addBit port map ( A(5),B(5), v(4),S(5),open );
  U_b6: addBit port map ( A(6),B(6), v(5),S(6),open );
  U_b7: addBit port map ( A(7),B(7), v(6),S(7),open );

  U_a8_11: adianta4 port map
    (A(11 downto 8),B(11 downto 8),cc(1),v(11 downto 8));

  U_b8: addBit port map ( A(8), B(8), cc(1), S(8),open );
  U_b9: addBit port map ( A(9), B(9),  v(8), S(9),open );
  U_ba: addBit port map ( A(10),B(10), v(9),S(10),open );
  U_bb: addBit port map ( A(11),B(11),v(10),S(11),open );

  U_a12_15: adianta4 port map
    (A(15 downto 12),B(15 downto 12),cc(2),v(15 downto 12));

  U_bc: addBit port map ( A(12),B(12),cc(2),S(12),open );
  U_bd: addBit port map ( A(13),B(13),v(12),S(13),open );
  U_be: addBit port map ( A(14),B(14),v(13),S(14),open );
  U_bf: addBit port map ( A(15),B(15),v(14),S(15),open );

  vai <= cc(3);
end architecture adderAdianta16;
```

Comparação de desempenho

Finalmente estamos numa posição que nos permite comparar o desempenho dos três modelos de somadores vistos até aqui, aquele sem adiantamento de vai-um (Prog. 9.15), aquele com adiantamento de 4 em 4 bits (Prog. 9.26), e o somador com adiantamento de 16 bits (Prog. 9.28). Os valores empregados para os tempos de propagação e de contaminação das portas lógicas são consistentes para os quatro modelos e servem como medidas quantitativas, ao invés de medidas absolutas de velocidade para cada um dos circuitos. O *testbench* usado para excitar os somadores é similar àquele do Programa 9.16.

O diagrama de tempos na Figura 9.14 mostra a evolução dos resultados ao longo do tempo. As entradas são os sinais inpa e inpb, e são mostrados o resultado (rescad) e o sinal de vai-um (vaicad), para o somador sem adiantamento de vai-um, rescad4 e vaiad4 para o somador com adiantamento de 4 em 4 bits, e rescad16 e vaiad16 para o somador com adiantamento de 16 bits.

Para as entradas, mostrados em hexadecimal como `0001` e `FFFF`, o somador sem adiantamento produz resultado estável após 800ns, quando então o sinal vaicad fica em 1 e o resultado estabiliza em `0000`: `0001+FFFF=1.0000`. O diagrama mostra a sequência de 15 alterações do resultado enquanto os somadores de um bit computam seus resultados bit a bit – a cada alteração é computado um bit definitivo do resultado: `FFFE` $\rightarrow$ `FFFC` $\rightarrow$ `FFF8` $\rightarrow$ `FFF0` $\cdots$ `8000` $\rightarrow$ `0000`. O sinal vaicad estabiliza 20ns depois que o resultado estabiliza em `0000`.

O somador com adiantamento de 4 em 4 bits produz resultado estável após 433ns, e o sinal vaicad4 se define 10ns antes do resultado estabilizar em `0000`. A sequência de valores é a mesma que para o primeiro somador, a menos do intervalo mais curto em cada alteração. Uma vez que o adiantador tenha computado os bits de vai-um, os resultados do quarteto estabilizam rapidamente.

O somador com adiantamento de 16 bits produz resultado estável após 292ns e a sequência de valores é mais curta, com 10 valores intermediários, ao invés de 15, como nos outros dois somadores. O sinal vaicad16 estabiliza após 200ns, e uma vez que todos os bits intermediários de vai-um são computados, os bits do resultado final são obtidos de dois em dois, nas quatro últimas alterações do resultado – a escala horizontal do diagrama esconde estas variações.

Do ponto de vista de desempenho global, e com o modelo de temporização relativamente simples empregado aqui, o somador com adiantamento de 4 em 4 bits é $800ns/433ns = 1,85$ vezes mais rápido que o modelo sem adiantamento, enquanto que o modelo com adiantamento de 16 bits é $800ns/292ns = 2,74$ vezes mais rápido. Os ganhos de desempenho são consideráveis, e são obtidos à custa de acréscimos não triviais no número de portas lógicas, de maior complexidade do projeto, e no tempo adicional para depuração e testes.

Espaço em branco proposital.

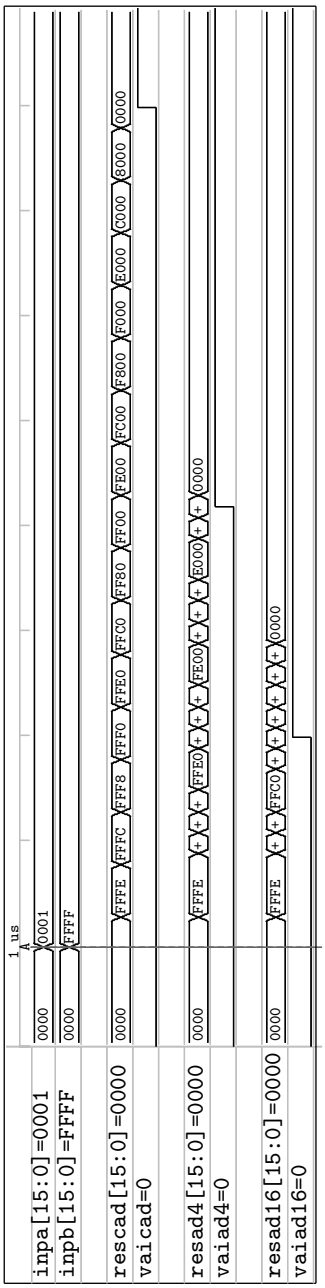


Figura 9.14: Comparação do tempo de propagação dos três somadores para 0001+FFFF.

9.4 Modelagem de circuitos sequenciais em VHDL

Reveja a definição de *circuitos sequenciais síncronos* da Seção 7.6.

9.4.1 Revisão: como funciona o mecanismo de simulação

VHDL emprega *simulação de eventos discretos*: os eventos ocorrem nos instantes determinados pela própria simulação. Por enquanto, consideremos que um *processo* é tão simples quanto uma atribuição; veremos em breve processos 'completos'. Uma simulação consiste de:

fase de inicialização todos os sinais são inicializados com os valores declarados, ou com os menores valores do tipo de cada sinal; tempo simulado $\leftarrow$ 0; todos os processos são executados exatamente uma vez;

passo de simulação – primeira etapa o tempo simulado avança até o próximo instante em que uma transação está programada; todas as transações programadas para aquele instante são executadas; estas transações podem provocar eventos em sinais;

passo de simulação – segunda etapa processos que dependem dos eventos disparados na primeira etapa são então executados; depois que todos os processos executam, a simulação avança até o instante em que a próxima transação está programada.

Uma transação decorre da atribuição a um sinal;
– a atribuição será efetivada no próximo delta (Δt);
– se o sinal muda de estado, então este sinal sofre um evento, na primeira etapa;
– um evento no sinal S causa a execução dos processos que dependem de S, na segunda etapa.

Em nas simulações geradas a partir do compilador `ghdl`, o femtossegundo (1fs = 10^{-15}s) é a menor unidade de tempo simulado, e ocorrem 1.000 Δt em cada femtossegundo.

Em resumo:

(1) O tempo simulado avança em função de eventos nos sinais;

(2) a cada mudança no estado de um sinal corresponde um evento;

(3) em função de um evento num sinal que aparece no lado direito de uma atribuição, a expressão é reavaliada, e se houver mudança no valor da expressão, o novo estado é atribuído ao sinal do lado esquerdo (primeira etapa);

(4) todos os comandos são avaliados e os eventos disparados passarão a ter efeito no próximo delta;

(5) enquanto houver eventos no delta corrente, estes são avaliados e eventos resultantes são programados para o próximo delta;

(6) processos que dependem dos eventos disparados neste delta são executados (segunda etapa);

(7) eventos causados pela execução dos processos são programados para o próximo delta; e

(8) eventos podem ser programados para ocorrer no futuro com **after**.

9.4.2 Comandos Sequenciais

Até agora empregamos somente os *comandos concorrentes* de VHDL. Estes comandos nos permitem escrever e verificar modelos para circuitos combinacionais pela interligação de componentes que são, eles próprios modelados como circuitos combinacionais.

Para que possamos escrever modelos para *flip-flops* e registradores temos que empregar os *comandos sequenciais* da linguagem. Estes comandos descrevem comportamento em termos de sequências de eventos e sua execução depende de ordem em que aparecem no código fonte. Nesse sentido, comandos sequenciais se assemelham àqueles da linguagem C.

Comandos sequenciais são 'executados' em tempo zero numa simulação, e *só podem ser usados em* ***processos***. Comandos sequenciais podem, também, ser usados para descrever lógica combinacional.

9.4.3 Processos

O *processo* é a construção de VHDL que encapsula um ou mais comandos sequenciais, que modelam uma determinada funcionalidade, tais como um *flip-flop*, um registrador ou uma máquina de estados.

Processos são definidos dentro de arquiteturas, e numa arquitetura, todos os processos devem ter nomes distintos.

Exemplo 9.1 Vejamos um exemplo simples de processo, para que o mecanismo de simulação fique mais claro. Nosso primeiro exemplo é um *gerador de sinal de relógio*. Desejamos um período de 50ns e que o sinal seja uma onda quadrada.. Usaremos uma constante para definir o período:

```
constant CLOCK_PER : time := 50 ns;
```

O código do gerador de relógio é mostrado no Programa 9.29. O processo U_clock é executado na fase de inicialização: o sinal clk é inicializado em '0' e a execução do processo é suspensa no primeiro **wait**, e este processo é escalonado para ser executado depois de 25ns de tempo simulado.

Programa 9.29: Processo gerador de relógio.

```
U_clock: process
begin
  clk <= '0';                -- executa
  wait for CLOCK_PER / 2;    -- espera meio ciclo
  clk <= '1';                -- volta a executar
  wait for CLOCK_PER / 2;    -- espera meio ciclo e volta ao topo
end process U_clock;
```

Decorrido o primeiro semiciclo de 25ns, o processo volta a executar a partir do comando que segue o **wait** que causou a suspensão: ao sinal clk é atribuído '1' e o processo se suspende no segundo **wait**, novamente a esperar por 25ns de tempo simulado.

Decorrido este segundo intervalo de 25ns, o processo volta a executar do seu início, e clk torna-se '0' novamente. Assim, o processo executa periodicamente, e a cada execução o nível lógico do sinal de relógio é invertido. ◁

9.4.4 Processos *sem* lista de sensibilidade – *wait*

Processos podem ser usados sem uma *lista de sensibilidade*. Estes processos devem conter um ou mais comandos que suspendem temporariamente a execução do processo, tal como o comando **wait**. Quando um destes comandos é executado, o processo fica suspenso até que a condição de espera seja resolvida. No código abaixo, quando o comando **wait on** é executado e a condição for falsa, a execução do processo é suspensa até que a condição de espera se torne verdadeira, quando então a execução é retomada no comando após o **wait**.

```
nomeDoProcesso: process  -- sem lista de sensibilidade
  declarações;
begin
  ...
  wait on (condição) ;    -- comando que suspende execução
  x <= ...;               -- reinicia neste ponto
end process nomeDoProcesso;
```

Processos podem *ou* conter uma lista de sensibilidade *ou* conter comandos **wait**, mas nunca os dois num mesmo processo.

O Exemplo 9.1 mostra um processo sem lista de sensibilidade que produz um sinal periódico, que pode ser usado como um sinal de relógio.

Exemplo 9.2 O processo U_reset, no Programa 9.30, gera um sinal aperiódico que pode ser usado na inicialização de um circuito sequencial síncrono. Na fase de inicialização, ao sinal reset é atribuído '0' na linha 3 e então o processo se suspende por 3/4 de ciclo na linha 4. Decorrido este intervalo, o processo volta a executar e o sinal reset muda para '1' e o processo se suspende no **wait** da linha 6 e nunca mais é executado. ◁

Programa 9.30: Gerador de *reset*.

```
U_reset: process                -- sem lista de sensibilidade
begin
  reset <= '0';                 -- executa e
  wait for CLOCK_PER * 0.75;    -- espera por 37,5ns
  reset <= '1';                 -- volta a executar e
  wait;                         -- se suspende para sempre
end process U_reset;
```

Existem quatro tipos de condição de parada com **wait**:

(i) **wait for** intervalo ; a execução suspende por um certo tempo (**wait for** 10 ns);

(ii) **wait until** condição; suspende até que a expressão booleana da condição se torne verdadeira (**wait until** (received = true));

(iii) **wait on** (lista de sinais); suspende até a ocorrência de um evento em algum dos sinais na lista de sinais (**wait on** (clk , reset , received));

(iv) **wait**; sem condição, espera para sempre, finalizando a execução do processo.

Um comando **wait** pode combinar mais de uma cláusula de espera, como por exemplo

```
wait on (reset , kill) for 100~ns;
```

O processo fica a esperar por um evento nos sinais reset ou kill ou o decurso de 100ns.

9.4.5 Processos *com* lista de sensibilidade

Processos podem conter uma *lista de sensibilidade*, como mostrado no trecho de código abaixo.

```
nomeDoProcesso: process (listaDeSensibilidade)
  declarações;
begin
  comando_sequencial_1;
  comando_sequencial_2;
  comando_sequencial_3;
end process nomeDoProcesso;
```

Os sinais que são atribuídos em um processo retêm seus valores até a próxima execução daquele processo. A execução de um processo pode ser encarada como ocorrendo em tempo zero, e os processos executam paralelamente com demais comandos concorrentes no corpo da arquitetura. Pode-se dizer que o processo inteiro é um comando concorrente, mas os comandos do corpo do processo são executados sequencialmente. Um evento em qualquer das variáveis na lista de sensibilidade de um processo dispara a sua execução, quando então os comandos no corpo do processo são executados.

Na inicialização de uma simulação, todos os processos são executados exatamente uma vez. Nos passos de simulação, quando ocorre um evento num sinal da *lista de sensibilidade* o processo é escalonado para execução na segunda etapa do delta corrente e os comandos sequenciais são executados na ordem do código fonte.

Se a execução dos comandos provoca eventos em sinais, estes eventos são escalonados para o próximo delta, ou para um instante futuro se a cláusula **after** for usada.

Quando a execução chega ao final do corpo do processo, ela é retomada do primeiro comando sequencial, no topo do processo. O corpo do processo se assemelha a um laço infinito, que só é executado quando ocorrer um evento em sinais da sua lista de sensibilidade. Esta é uma descrição inexata, mas ela será retificada em breve.

O atributo `event`, aplicado a um sinal (`x'event`) tem o valor `TRUE` se ocorreu um evento em `x` no delta corrente.

Exemplo 9.3 O Programa 9.31 mostra um exemplo de processo com lista de sensibilidade. Sempre que ocorrer um evento no sinal `clk`, o corpo do processo é executado *em tempo zero*.

O atributo `clk'event` indica que ocorreu um evento no sinal `clk` e este evento provocou a execução do processo porque `clk` está na lista de sensibilidade. Se o evento colocou `clk` em `'1'` então ocorreu uma borda ascendente em `clk` – o sinal se alterou (evento) e a alteração fez com que o sinal ficasse em `'1'`. Se ocorreu uma borda ascendente em `clk`, então o valor do sinal `D` é atribuído ao sinal `Q`. Se nenhuma outra atribuição é feita ao sinal `Q`, então ele preserva seu valor até a próxima borda ascendente. Uma função que detecta bordas, chamada `rising_edge()`, é definida na Seção 9.4.13 ◁

Programa 9.31: Primeira versão de um *flip-flop* D.

```
FF0: process (clk)  -- somente clk na lista de sensibilidade
begin
  if (clk'event and clk = '1') then -- clk mudou para '1'
    Q <= D;
  end if;
end process FF0;
```

9.4.6 O que é um *flip-flop*?

Consultando as fontes em papel que estão disponíveis na minha estante, e tomado de alguma surpresa, passo a desvelar à dileta leitora que conceitos que emprego há umas tantas décadas são definidos em dicionários de alta qualidade como sendo:

flip-flop circuit *noum: an electronic circuit with two permanently stable conditions (as when one electron tube is conducting while the other is cut off) so that conduction is switched from one to the other by succcessive pulses*, Webster's Third New Internactional Dictionary;

flip-flop *verb: to change your opinion about something, especially when you then hold the opposite opinion*, Oxford Advanced Learner's Dictionary;

latch *noum: to catch or fasten by means of a latch*, Webster's Third New Internactional Dictionary;

basculador *(1)* ELETRÔN *circuito flip-flop*, Dic. Houaiss da Língua Portuguesa[4]; e

básculo *(1) ponte levadiça com mecanismo de contrapeso, (2) peça móvel de metal ou ferro, que gira apoiada num pino, destinada a abrir ou fechar ferrolhos de portas, janelas, etc*, Dic. Houaiss da Língua Portuguesa.

9.4.7 *Flip-flops* em VHDL

Em se tratando de sistemas digitais, um *flip-flop* é um circuito cuja saída, em condições normais de operação, pode estar em um de dois possíveis estados: ou a saída está no estado *flip*, ou está no estado *flop* – um *flip-flop* é um circuito *biestável* cujo comportamento é definido para somente uma de duas condições: ou sua saída é estável em 1, ou é estável em 0. Um *flip-flop* é um modelo para uma ponte levadiça ou um ferrolho: ou estão abertos, ou estão fechados.

Exemplo 9.4 ***Flip-flop*** **com wait** Vejamos o código que modela um *flip-flop*, empregando um processo sem lista de sensibilidade. A entidade FF0 descreve a interface de um *flip-flop* tipo D simples, com entrada D, relógio, e saída Q, e é mostrada no Programa 9.32.

A arquitetura waitSimples modela um *flip-flop* tipo D. O atributo 'event indica ocorrência de evento no sinal clk. O processo executa e fica bloqueado no **wait** a esperar por um evento no sinal clk, e além disso, que o evento faça clk='1'. Estas duas condições equivalem a uma borda ascendente no sinal clk: ocorreu uma mudança, e a mudança foi para 1. Após a detecção da borda, a entrada D é copiada para o sinal Q, que mantém seu estado até a próxima borda em clk.

Quando o processo é desbloqueado no **wait**, a execução prossegue até o fim do processo e é retomada no seu topo. Nesse exemplo, após o desbloqueio acontece a atribuição (Q<=D;) e a execução do processo é logo interrompida novamente no **wait**. ◁

[4]Este me parece um cacófato assaz deselegante. O autor, que é adepto do *uber*-dicionarista, admite que emprega, desavergonhadamente, a forma incorreta 'básculo' ao invés do horrendo 'basculador'. Atravessemos o Rubicão sem mais delongas.

Programa 9.32: *Flip-flop* com wait.

```
1  entity FF0 is
2    port(D:   in  bit;
3         clk: in  bit;
4         Q:   out bit);
5  end FF0;
6
7  architecture waitSimples of FF0 is
8  begin
9    FF: process
10   begin
11      wait for ( clk'event and clk = '1' ); -- borda em clk
12      Q <= D;
13   end process FF;
14 end architecture waitSimples;
```

Exemplo 9.5 ***Flip-flop* com lista de sensibilidade** A arquitetura listaSimples , no Programa 9.33, emprega uma lista de sensibilidade ao invés de um comando **wait** no corpo do processo. A lista de sensibilidade junto à palavra reservada **process** contém somente o sinal clk, e sempre que ocorrer um evento neste sinal, o processo FF é executado. O corpo de um processo com lista de sensibilidade é executado do início ao final sempre que ocorrer um evento num dos sinais da lista.

Programa 9.33: *Flip-flop* com lista de sensibilidade.

```
1  architecture listaSimples of FF0 is
2  begin
3    FF: process (clk)  -- eventos em clk causam execução
4    begin
5       if clk = '1' then      -- valor pós-evento = '1'
6          Q <= D;
7       end if;
8    end process FF;
9  end architecture listaSimples;
```

O corpo do processo, que é o comando **if**, só é executado se ocorrer um evento em clk. Se, após o evento clk='1', então ocorreu uma borda ascendente naquele sinal e a entrada deve ser registrada; se clk tem qualquer outro valor após o evento, não ocorreu uma borda ascendente, e o valor em Q deve permanecer como estava. O corpo do processo é 'executado' *em tempo zero*, somente após eventos no sinal *clk*. ◁

Espaço em branco proposital.

Exemplo 9.6 ***Flip-flop* com *reset* assíncrono** Vejamos como modelar um *flip-flop* com uma entrada de *reset* assíncrona. A entidade FF1, no Programa 9.34, declara duas entradas de controle, clk e rst. O processo na arquitetura rstAssincrono tem dois sinais na sua lista de sensibilidade, clk e rst. O **if** faz Q='0' sempre que ocorrer um evento em rst, e o evento fizer rst=0. O **elsif** testa a ocorrência de uma borda ascendente em clk, e se este for o caso, a entrada D é registrada. Em qualquer outra combinação nas entradas, o sinal Q mantém o seu valor.

A execução do processo pode ocorrer tanto por eventos em rst quanto em clk. É por isso que a cláusula **elsif** verifica se ocorreu um evento em clk, e mais ainda, se clk=1 após o evento. ◁

Programa 9.34: *Flip-flop* com *reset* assíncrono.

```
1  entity FF1 is
2    port(D:         in  bit;
3         clk, rst: in  bit;
4         Q:         out bit);
5  end FF1;
6
7  architecture rstAssincrono of FF1 is
8  begin
9    FF: process (clk, rst)  -- lista de sensibilidade
10   begin
11      if rst = '0' then
12         Q <= '0';
13      elsif clk'event and clk = '1' then  -- borda em clk
14         Q <= D;
15      end if;
16   end process FF;
17 end architecture rstAssincrono;
```

Espaço em branco proposital.

Exemplo 9.7 ***Flip-flop* com *set* e *reset* assíncronos** A entidade FF2, no Programa 9.35, declara três entradas de controle, clk, clr e set.

O processo na arquitetura setclr tem três sinais na sua lista de sensibilidade. O comando **if** coloca a saída Q em 0 caso rst='0'; o primeiro **elsif** a coloca em 1 caso set='0', ou registra a entrada D caso ocorra uma borda ascendente em clk. Em qualquer outra combinação de entradas, Q mantém o seu valor.

Este exemplo evidencia a distinção entre a lista de sensibilidade e o uso de **wait**s: com a lista de sensibilidade, caso ocorram eventos nos três sinais no mesmo delta, o código do processo decide a ordem de avaliação e as consequências dos eventos. Se **wait**s forem usados, a lógica das condições de espera pode ficar mais complexa, e o código confuso e de difícil compreensão. ◁

Programa 9.35: *Flip-flop* com *set* e *reset* assíncronos.

```
1  entity FF2 is
2    port(D:              in  bit;
3         clk, rst, set: in  bit;
4         Q:             out bit);
5  end FF2;
6
7  architecture setclr of FF2 is
8  begin
9    FF: process (clk, rst, set)  -- lista de sensibilidade
10   begin
11     if rst = '0' then          -- reset tem precedência
12       Q <= '0';
13     elsif set = '0' then
14       Q <= '1';
15     elsif clk'event and clk = '1' then -- borda em clk
16       Q <= D;
17     end if;
18   end process FF;
19 end architecture setclr;
```

Num *flip-flop* real, a ocorrência de rst='0' e set='0' é uma condição proibida e que viola a especificação temporal daquele circuito. Caso esta situação ocorra, o valor no sinal Q é indeterminado, e por causa da metaestabilidade, pode permanecer indeterminado por um tempo também indeterminado. O comportamento modelado na arquitetura setClr é uma versão idealizada do circuito e que abstrai a possibilidade de metaestabilidade para simplificar a modelagem e a simulação.

Espaço em branco proposital.

Equivalência entre *lista de sensibilidade* e *wait*

O Programa 9.36 mostra dois processos com comportamento equivalente, o primeiro com lista de sensibilidade e o segundo com um comando **wait**.

Programa 9.36: Processos com lista de sensibilidade e com wait.

```
architecture doisProcessos of SisSeqSin is
begin

  FFlista: process (clk, rst)  -- COM lista de sensibilidade
    begin
      if rst = '0' then
          Q <= '0';
      elsif clk'event and clk = '1' then
          Q <= D;
      end if;
    end process FFlista;

  FFwait: process              -- SEM lista de sensibilidade
    begin
      if rst = '0' then
          Q <= '0';
      elsif clk'event and clk = '1' then
          Q <= D;
      end if;
      wait on (clk, rst);      -- espera eventos em clk, rst
    end process FFwait;

end architecture doisProcessos;
```

9.4.8 Sinais e Variáveis

Enquanto falávamos da modelagem de circuitos combinacionais, sinais eram os equivalentes aos fios de um circuito combinacional – a cada delta, o valor atribuído a um sinal é o resultado da avaliação do lado direito da expressão – RHS ou *Right Hand Side*, ou o *rvalue*, no jargão de compiladores – que lhe atribui valor.

Quando modelamos circuitos sequenciais, alguns sinais "adquirem memória" e mantém seus valores entre as execuções dos processos que os manipulam. Os valores transportados por estes sinais são alterados pela combinação de eventos nas listas de sensibilidade dos processos, e do código sequencial que determina seus valores em função dos eventos apropriados. Sinais somente são atualizados no próximo delta, ou no evento "próxima borda do sinal de relógio" em circuitos sequenciais síncronos.

Frequentemente é necessário empregar 'fios' que transportam informação instantaneamente entre os comandos de um processo e estes 'fios' não podem ser sinais, porque estes somente são atualizados em um delta futuro, e não no delta corrente. Quando é necessária a comunicação instantânea de valores entre os comandos de um processo, deve-se empregar variáveis. Uma *variável* transporta valores entre comandos sequenciais em processos porque um novo valor é atribuído à variável imediatamente, e não num delta futuro.

Variáveis só podem ser declaradas dentro de processos, e a atribuição a uma variável é denotada por ':= ', como `v := '1';` .

As portas de saída de uma entidade não podem ser lidas numa arquitetura ou processo. Para que seja possível ler valores de estado que são atribuídos às saídas, devem ser empregadas variáveis para manter o estado do modelo, e estas variáveis são atribuídas às portas de saída da entidade.

Veremos exemplos em breve.

Atributos de sinais e de vetores

Atributos fornecem informação adicional sobre vários tipos de objetos em VHDL. Dentre muitos, nos interessam os *atributos de sinais* e os *atributos de vetores*.

Para um sinal S, nos interessam os atributos:

S'event	é *true* se ocorreu evento no sinal neste delta;
S'active	é *true* se alguma transação ocorreu no sinal neste delta;
S'last_value	retorna o valor de S, antes do último evento.

Já empregamos a atributo 'event nos modelos dos *flip-flop*s. A condição para a detecção da borda pode ser ainda mais estrita, ao se verificar que o valor no delta anterior era mesmo '0' e não um valor inválido, por exemplo.

```
if( clk'event and clk='1' and clk'last_value='0' ) then
  ...
end if;
```

Para um vetor V, nos interessam os atributos:

V'length	retorna o número de elementos de um vetor;
V'range	retorna a faixa de índices de um vetor, ou se o vetor é um tipo restringido (**subtype**), retorna o conjunto de valores do subtipo.

Por exemplo, para os dois vetores de 12 elementos, as suas faixas são aquelas declaradas, *viz.*:

```
V: bit_vector(11 downto 0);
W: bit_vector(0 to 11);
...
... V'range ...  -- faixa de 11 a 0
...
... W'range ...  -- faixa de 0 a 11
...
```

9.4.9 Processos na modelagem de lógica combinacional

Processos podem ser usados para modelar lógica combinacional. O código de um "processo combinacional" deve evitar que sinais mantenham o mesmo estado entre duas execuções do processo – o processo não pode ter memória. Para tanto, o código do processo deve obedecer à três regras:

Regra 1 a lista de sensibilidade contém todos os sinais de entrada (sinais lidos no RHS, ou *rvalues*) usados pelo processo;

Regra 2 as atribuições às saídas do processo cobrem todas as combinações possíveis das entradas do processo;

Regra 3 a todas as variáveis no processo deve ser atribuído um valor antes que elas sejam usadas como entrada no RHS (*rvalue*).

Exemplo 9.8 Um multiplexador de quatro entradas, que é um circuito combinacional, pode ser modelado com um processo, como mostra o Programa 9.37. O processo mux satisfaz à Regra 1 porque a lista de sensibilidade contém *todos* os sinais que são lidos pelo processo, tanto o sinal Sel que escolhe uma das entradas, quanto as quatro entradas (A,B,C,D). A Regra 2 é obedecida porque, nas quatro combinações da entrada Sel, ocorre uma atribuição ao sinal Y. O processo não emprega variáveis e portanto a Regra 3 é trivialmente satisfeita.

Esta não é a maneira mais eficiente de se codificar um multiplexador porque as entradas são avaliadas em sequência, e na síntese este código resulta num circuito mais complexo do que o necessário. Como está, o circuito resultante da síntese envolve uma cadeia de quatro *and*s. O comando **if** é descrito na Seção 9.4.11.

Em breve veremos um comando VHDL que modela um multiplexador que é sintetizado eficientemente. A *síntese* de um modelo é o processo de compilação que produz uma descrição de baixo nível um circuito – lista de conexões – ao invés de um simulador. ◁

Programa 9.37: Multiplexador modelado com um processo.

```
1  entity simpleMux is
2    port (Sel:        in  bit_vector(0 to 1);
3          A, B, C, D: in  bit;
4          Y:          out bit);
5  end simpleMux;
6
7  architecture ineficiente of simpleMux is
8  begin
9    mux: process (Sel,A,B,C,D)  -- lista de sensibilidade
10   begin
11     if    Sel = "00" then  Y <= A; -- todas as 4
12     elsif Sel = "01" then  Y <= B; --   combinações
13     elsif Sel = "10" then  Y <= C; --   para 2 bits
14     elsif Sel = "11" then  Y <= D;
15     end if;
16   end process mux;
17 end architecture ineficiente;
```

9.4.10 Processos na modelagem de lógica sequencial

Para descrever lógica sequencial, portanto com memória, processos devem obedecer à três regras:

Regra 1 a lista de sensibilidade não inclui todos os sinais no RHS (*rvalue*) das atribuições;

Regra 2 a lógica **if**−**then**−**elsif** é incompletamente especificada, indicando que um ou mais sinais/variáveis devem manter seus valores entre execuções do processo;

Regra 3 uma ou mais variáveis mantêm seus valores entre execuções do processo – variáveis podem ser lidas (RHS) antes que um valor lhes tenha sido atribuído.

Quando um processo atende às Regras 1 e 2, e opcionalmente à 3, durante a síntese o compilador gera um circuito com *flip-flop*s e registradores para manter os valores entre as execuções do processo, que tipicamente ocorrem na borda do sinal de relógio, ou quando sinais de controle assíncronos (reset) são ativados.

Programa 9.38: Modelo para um registrador de deslocamento.

```
1  entity shiftReg is
2    port (clk, rst, load: in  bit;
3          serInp:         in  bit;
4          data:           in  bit_vector(0 to 7);
5          Q:              out bit_vector(0 to 7));
6  end shiftReg;
7
8  architecture correta of shiftReg is
9  begin
10   reg: process (rst, clk)
11     variable Qvar:bit_vector(0 to 7);
12   begin
13     if rst = '1' then          -- inicialização assíncrona
14       Qvar := "00000000";
15     elsif (clk'event and clk = '1') then
16       if load = '1' then                -- carga síncrona
17         Qvar := data;
18       else                             -- rotação síncrona
19         Qvar := Qvar(1 to 7) & serInp;
20       end if;
21     end if;
22     Q <= Qvar;     -- atribui estado ao sinal da interface
23   end process;
24 end architecture correta;
```

Exemplo 9.9 A entidade shiftReg, no Programa 9.38 modela um registrador de deslocamento com 8 bits, que desloca seu conteúdo a cada ciclo do relógio.

Quando o sinal load é ativado, o registrador é carregado com o valor em sua entrada data. O processo reg declara a variável Qvar para manter o conteúdo do registrador entre os eventos no sinal de relógio.

Se o sinal rst é ativado, o registrador é carregado com zero. A cada borda do relógio, se o sinal load está ativo, um novo valor é atribuído à Qvar.

Do contrário, o conteúdo de Qvar é deslocado de uma posição – o bit serInp é inserido na posição do bit Q_0, e os demais deslocam-se de uma posição.

Ao final da execução do processo, o conteúdo da variável Qvar é atribuído ao sinal da interface Q. O valor de Qvar, que mantém o estado do registrador, é lido na linha 18 – é usado no RHS – e portanto o sinal da interface Q não pode ser usado para manter o conteúdo do registrador.

A Regra 1 é atendida porque o sinal load não está na lista de sensibilidade. A Regra 2 é atendida porque os **if**s não cobrem todas as oito combinações possíveis dos três sinais de controle. A Regra 3 é atendida porque, dependendo das combinações das variáveis de controle (rst=1 e load=0), o valor em Qvar pode ser lido antes de que algum valor tenha sido atribuído àquela variável. ◁

Há uma diferença sutil porém importante entre variáveis e sinais: variáveis são atualizadas instantaneamente enquanto que sinais somente são atualizados no próximo delta, ou em momento futuro. No Programa 9.39, a arquitetura ERRADA da entidade shiftReg é praticamente a mesma que a arquitetura correta – a diferença é que, em ERRADA, o valor memorizado em Qsig é mantido num sinal e não numa variável. Suponha que não ocorram eventos no sinal rst, e portanto o processo reg só é executado nas bordas do relógio. O comportamento do modelo é aquele esperado pelo programador?

Programa 9.39: Modelo *errado* para o registrador de deslocamento.

```
1  architecture ERRADA of shiftReg is
2  begin
3    reg: process (rst, clk)
4      signal Qsig:bit_vector(0 to 7);          -- SIGNAL
5    begin
6      if rst = '1' then
7        Qsig <= "00000000";
8      elsif (clk = '1' and clk'event) then
9        if load = '1' then
10         Qsig <= data;
11       else
12         Qsig <= Qsig(1 to 7) & serInp;
13       end if;
14     end if;
15     Q <= Qsig;     -- atribui estado ao sinal da interface
16   end process;
17 end architecture ERRADA;
```

O comportamento do modelo não é o esperado pelo programador porque a atualização do sinal da interface Q só terá efeito visível na próxima execução do processo reg, embora os dois **if**s computem o novo valor de Qsig no delta corrente. Este sinal, Qsig, é atualizado no delta corrente mas a atribuição à Q somente acontecerá na próxima execução do processo, no delta em que ocorrer um evento em clk, portanto com um ciclo de atraso em relação ao que se deseja.

Espaço em branco proposital.

9.4.11 Comandos Sequenciais

Vejamos quais comandos podem ser empregados em processos. Estes são chamados de *comandos sequenciais* porque são executados na ordem do código fonte, de forma similar aos comandos da linguagem C.

Comando Sequencial IF-THEN-ELSE

Já vimos vários usos do comando **if** nos modelos dos *flip-flop*s. O trecho abaixo mostra o comando **if** em toda sua glória. Se condição1 é verdadeira, então os comandos em comandos1 são executados; do contrário, condição2 é avaliada, e se verdadeira, comandos2 são executados; do contrário as duas condições são falsas e comandos3 são executados.

```
if condição1 then      -- condição deve ter tipo Boolean
  comandos1;
elsif condição2 then   -- cláusula opcional
  comandos2;
else                   -- cláusula opcional
  comandos3;
end if;
```

As cláusulas **elsif** e **else** são opcionais, e **if**s aninham da forma óbvia.

```
if cond_externa then  -- if's aninhados
  comandos_externos;
else
   if cond_interna then
      comandos_internos;
   end if;
end if;
```

Comando Sequencial CASE

O comando sequencial **case** é similar ao *switch* da linguagem C. A expressão de escolha é avaliada, e a cláusula que 'casar' com o resultado da avaliação é executada.

No exemplo abaixo a expressão controle é avaliada e se uma das cláusulas teste_i for selecionada, o respectivo comando_i é executado. A palavra reservada **others** 'casa' todos os valores que não estão explicitamente listados e deve ser a última cláusula do comando. Uma cláusula pode conter mais de um valor, sendo estes separados por uma barra vertical.

```
case controle is
  when teste1 =>
    comando1;
  when teste2 | teste3 =>  -- dois casos por testar
    comando2;
  when others =>           -- todos os casos não cobertos acima
    comandoOthers;
end case;
```

Os testes devem ser mutuamente exclusivos, e todas os possíveis valores de controle devem ser cobertos, nem que seja por **others**.

Ao contrario do **if**−**then**−**else**, no **case** não há prioridade na avaliação das cláusulas e portanto este comando é mais adequado para a modelagem de circuitos tais como o multiplexador, como mostra a arquitetura eficiente para o multiplexador de quatro entradas no Programa 9.40. Esse comando é sintetizado como um multiplexador de quatro entradas, e não como uma sequência de **if**s encadeados.

Programa 9.40: Modelo eficiente para processo do multiplexador.

```
1  architecture eficiente of simpleMux is
2  begin
3    mux: process (Sel,A,B,C,D)  -- lista de sensibilidade
4    begin
5      case Sel is
6        when "00" => Y <= A;  -- todas as quatro
7        when "01" => Y <= B;  --   combinações
8        when "10" => Y <= C;  --   estão cobertas
9        when "11" => Y <= D;
10     end case;
11   end process mux;
12 end architecture eficiente;
```

Verifique se este processo obedece às três regras para processos que modelam circuitos combinacionais.

Comando Sequencial FOR

O comando **for** de VHDL é muito mais expressivo do que seu insípido primo da linguagem C. Infelizmente, este comando não é sintetizável e é usado apenas em *testbenches* para gerar sequências de valores de teste.

A variável de indução de um **for** não precisa ser explicitamente declarada, tem escopo somente no corpo do laço, e encobre outra variável com mesmo nome que esteja declarada em escopo externo ao **for**. O tipo da variável de indução é inferido pelo compilador e pode iterar sobre um conjunto de inteiros, sobre os elementos de um tipo, ou sobre o tamanho de um vetor, por exemplo.

```
for i in faixa loop
  comando;
end loop;
```

O valor de i cobre todos os valores em faixa e portanto i pode iterar sobre um subconjunto dos inteiros, ou sobre elementos de um tipo, ou sobre o conjunto de índices de um vetor. O Programa 9.16, na página 299, contém um exemplo de laço que itera sobre os elementos do vetor de tuplas de teste do somador de 16 bits.

Vejamos um exemplo em que a faixa de valores de laços pode ser determinada de três formas diferentes: as duas últimas empregam atributos de vetores de bits para determinar a faixa de valores das variáveis de indução j e k – o tamanho do vetor pode ser alterado pelo programador e o compilador recomputará a nova faixa de valores para indexar o vetor. Os três laços determinam a paridade do vetor de bits D e são mostrados no Programa 9.41. O terceiro formato, com D'**range**, é usado nos *testbenches* para facilitar a indexação dos vetores de testes – a faixa de valores válidos do índice é computada pelo compilador ao determinar o tamanho do vetor D.

Programa 9.41: Exemplos de indexação de laço for.

```
1  signal D: bit_vector(0 to 9);
2
3  paridade3: process(D)
4    variable otmp: Boolean := FALSE;
5  begin
6    ...
7    for i in 0 to 9 loop  -- faixa definida explicitamente
8      if D(i) /= '1' then otmp := not otmp; end if;
9    end loop;
10   ...
11   for j in 0 to (D'length - 1) loop  -- comprimento do vetor
12     if D(j) /= '1' then otmp := not otmp; end if;
13   end loop;
14   ...
15   for k in D'range loop  -- faixa de valores do índice
16     if D(k) /= '1' then otmp := not otmp; end if;
17   end loop;
18   ...
19 end process paridade3;
```

Comando Sequencial WHILE

O comando **while** é similar ao da linguagem C: enquanto a condição for verdadeira, os comandos no corpo do laço são executados. Este comando não é sintetizável e é usado para gerar sequências de valores para testes, ou para ler os caracteres de um arquivo de texto, por exemplo.

```
while condição loop
  comandos;
end loop;
```

Comando Sequencial LOOP

O comando sequencial **loop** é usado para efetuar tarefas repetitivas e tampouco é sintetizável. O laço pode ser terminado com uma cláusula **exit when**; se a condição avaliar como verdadeira, o laço termina.

```
nomeDoLoop: loop
  ...
  exit when condição;
  ...
end loop nomeDoLoop;
```

Este laço tem um *label* que pode ser usado para interromper a execução de laços aninhados; e o *label* do laço que se deseja interromper é usado no **exit**. No exemplo abaixo, quando a condição é verdadeira, a execução dos dois laços é interrompida.

```
Externo: loop
   ...
   Interno: loop
      ...
      if condição then
         exit Externo; -- salta para fora dos dois laços
      end if;
      ...
   end loop Interno;
   ...
end loop Externo;
```

9.4.12 Modelagem de máquinas de estado

Empregaremos o estilo de modelagem de Máquinas de Estado (MEs) descrito em [Ash08]: são necessários dois processos, um para o registrador de estado, e um processo combinacional para as funções de próximo estado e de saída. Dependendo da complexidade da ME, pode ser conveniente separar a função de saída da função de próximo estado.

O Programa 9.42 é um protótipo para as MEs que empregaremos. Um tipo (states) é definido para os estados, com um elemento do tipo para identificar cada um dos estados. Dois sinais do tipo states são declarados para armazenar o estado atual e o próximo estado.

Programa 9.42: Modelo para MEs.

```
type states is (s0, s1, s2, s3);   -- novo tipo para estados
signal curr_st, next_st : states;  -- sinais do novo tipo

U_state_reg: process(reset,clk)    -- registrador de estado
begin                              -- lógica sequencial
   if reset = '0' then
      curr_st <= s0;               -- estado inicial
   elsif rising_edge(clk) then
      curr_st <= next_st;          -- troca de estado na borda
   end if;
end process U_state_reg;

U_st_transitions: process(curr_st) -- função de próx estado
begin                              -- lógica combinacional
   case curr_st is
       when s0 => next_st <= s1;
       when s1 => next_st <= s2;
       when s2 => next_st <= s3;
       when s3 => next_st <= s0;
   end case;
end process U_state_transitions;
```

O processo U_state_reg mantém o estado atual, e nas bordas do relógio, faz com que o próximo estado torne-se o estado atual. Quando reset=0, a ME é colocada em seu estado inicial.

O processo U_st_transitions computa o próximo estado em função do estado atual.

A cada evento no sinal curr_st, disparado por uma borda no relógio no processo U_state_reg, o novo estado é computado em função do estado atual, e na próxima borda o estado atual progredirá para o próximo estado. Este processo modela lógica combinacional.

Exemplo 9.10 Máquina de Moore Vejamos um exemplo completo: uma *Máquina de Moore* que detecta sequências de dois ou mais **1**s em sua entrada. O diagrama de estados da ME é mostrado ao lado do código no Programa 9.43.

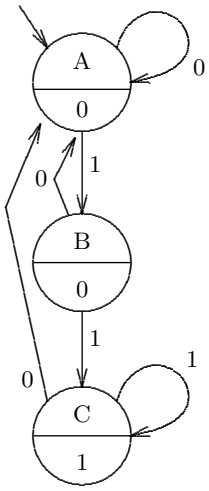
A linha 1 declara o tipo para os estados e a linha 2 declara os sinais para o estado atual e o próximo estado. O sinal found é a saída da ME. O processo U_state_reg mantém o estado atual e o atualiza nas bordas do relógio. O estado inicial é o estado A. O processo U_st_trans computa o próximo estado em função do estado atual e das entradas, e define a saída da ME estado a estado.

Verifique se o código do processo U_st_trans corresponde à função de próximo estado codificada no diagrama de estados. Verifique também se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◁

Espaço em branco proposital.

Programa 9.43: Máquina de Moore para 11^+.

```
1  type states is (A, B, C);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7    if reset = '0' then
8       curr_st <= A;
9    elsif rising_edge(clk) then
10      curr_st <= next_st;
11   end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin                -- Máquina de Moore
16   case curr_st is
17     when A =>
18       if entr = '0' then
19         next_st <= A;
20       else
21         next_st <= B;
22       end if;
23       found <= '0';
24     when B =>
25       if entr = '0' then
26         next_st <= A;
27       else
28         next_st <= C;
29       end if;
30       found <= '0';
31     when C =>
32       if entr = '0' then
33         next_st <= A;
34       else
35         next_st <= C;
36       end if;
37       found <= '1';
38   end case;
39 end process U_st_trans;
```



Espaço em branco proposital.

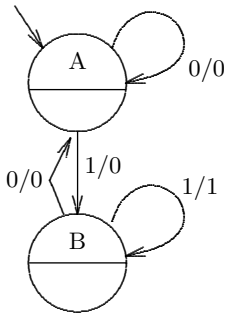
Exemplo 9.11 Máquina de Mealy O Programa 9.44 mostra a implementação para uma *Máquina de Mealy* que detecta sequências de dois ou mais **1**s em sua entrada.

Os modelos Moore e Mealy são semelhantes, exceto que na Máquina de Mealy são necessários apenas dois estados e a saída depende tanto do estado atual quanto da entrada: o sinal found é atualizado nas quatro possíveis combinações de estado e entrada.

Verifique se o código do processo U_st_trans corresponde à função de próximo estado do diagrama de estados, e se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◁

Programa 9.44: Máquina de Mealy para 11^+.

```
1  type states is (A, B);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7    if reset = '0' then
8       curr_st <= A;
9    elsif rising_edge(clk) then
10      curr_st <= next_st;
11   end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin              -- Máquina de Mealy
16   case curr_st is
17     when A =>
18       if entr = '0' then
19         next_st <= A;
20         found   <= '0';
21       else          -- entr='1'
22         next_st <= B;
23         found   <= '0';
24       end if;
25     when B =>
26       if entr = '0' then
27         next_st <= A;
28         found   <= '0';
29       else          -- entr='1'
30         next_st <= B;
31         found   <= '1';
32       end if;
33   end case;
34 end process U_st_trans;
```



Espaço em branco proposital.

9.4.13 Funções

VHDL nos permite definir *funções* que computam valores, e que podem ser usadas no lado direito de expressões. Funções têm argumentos de tipo entrada (**in**) e um valor de retorno, e podem ser usadas, por exemplo, na conversão de tipos, tais como a conversão de inteiros para vetores de bits, e a conversão de vetores de bits para cadeias de caracteres.

Exemplo 9.12 Função para detecção de borda A função rising_edge(), no Programa 9.45, pode ser usada para testar a ocorrência de uma borda de subida num sinal. Seu argumento é um sinal do tipo bit e seu valor é um Booleano. Os registradores de estado nas MEs da Seção 9.4.12 empregam esta função para detectar bordas no sinal de relógio.

Se esta função estiver declarada na entidade de um *flip-flop*, ela pode ser empregada em todas as arquiteturas para aquela entidade. Se a função estiver declarada numa biblioteca, então ela pode ser empregada em todas as entidades que façam uso da biblioteca. ◁

Programa 9.45: Função que detecta bordas ascendentes.

```
function rising_edge(signal S: bit)
  return boolean is
begin
  if (S'event) and          -- ocorreu evento em S
     (S = '1') and          --   e valor atual é '1'
     (S'last_value = '0')   --   e valor anterior era '0'
  then
    return TRUE;
  else
    return FALSE;
  end if;
end rising_edge;
```

Espaço em branco proposital.

Exemplo 9.13 Conversões entre Boolean e Bit O Programa 9.46 mostra duas funções de conversão de tipos: (i) bool2bit() converte um valor do tipo boolean para o tipo bit; (ii) bit2bool() faz a conversão na direção oposta. ◁

Programa 9.46: Funções de conversão entre boolean e bit.

```
function bool2bit(B: in boolean) return bit is
  variable S : bit;
begin
  case B is
    when TRUE   => S := '1';
    when others => S := '0';
  end case;
  return S;
end bool2bit;

function bit2bool(S: in bit) return boolean is
  variable B : boolean;
begin
  case S is
    when '1'    => B := TRUE;
    when others => B := FALSE;
  end case;
  return B;
end bit2bool;
```

Lembre que VHDL é uma linguagem com tipos fortes e estas conversões, aparentemente inúteis, ao um preço módico a se pagar, comparando-se aos benefícios advindos dos tipos fortes. Dentre os benefícios está a detecção, durante a compilação, de erros de programação, tais como a combinação de tipos incompatíveis – soma de bit com inteiro – ou a aplicação de um operador incompatível – conjunção de dois inteiros.

Funções, podem ser definidas em pacotes e usadas da mesma forma que os novos tipos. O tipo da função é declarado no cabeçalho do **package** e o corpo da função é ser definido no corpo (**body**) do pacote. O Programa 9.47 contém a declaração e a definição de uma função que detecta bordas ascendentes em sinais.

Espaço em branco proposital.

Programa 9.47: Um pacote com abreviaturas e funções.

```
package p_WIRES is -- abreviaturas para barramentos e sinais

  subtype int8  is integer range 0 to 255;
  subtype int16 is integer range 0 to 65535;

  subtype reg2  is bit_vector(1  downto 0);
  subtype reg4  is bit_vector(3  downto 0);
  subtype reg8  is bit_vector(7  downto 0);
  subtype reg15 is bit_vector(14 downto 0);
  subtype reg16 is bit_vector(15 downto 0);

  function rising_edge(S: in bit) return boolean;
  function bit2bool(S: in bit) return boolean;
  function bool2bit(B: in boolean) return bit;

package body p_WIRES is

  function rising_edge(signal S: bit) return boolean is
    begin
      if (S'event) and          -- ocorreu evento em S
         (S = '1') and          --   e valor atual é '1'
         (S'last_value = '0') --   e valor anterior era '0'
      then
        return TRUE;
      else
        return FALSE;
      end if;
  end rising_edge;

  function bool2bit(B: in boolean) return bit is
    variable s : bit;
  begin
    ...
  end bool2bit;

  function bit2bool(S: in bit) return boolean is
    variable b : boolean;
  begin
    ...
  end bit2bool;

end package p_WIRES;
```

Índice Remissivo

Símbolos

T_A, 64
T_D, 120
T_I, 109, 116
T_M, 117, 119, 186
T_P, 116
T_h, 188, 192, 199, 204
T_p, 171
$T_{C,F}$, 192
$T_{C,x}$, 118–120, 199, 204, 206
T_{FF}, 192
T_{min}, 199, 206
T_{skew}, 203
T_{su}, 188, 192, 199, 204
%, 18–20
$\Rightarrow$, 37
$\bigvee$, 47
$\bigwedge$, 47
$\equiv$, 36
$\gg$, 155
$\wedge$, 30, 36
$\triangleleft$ $\triangleright$, 36, 42, 66
$\leftarrow$, 193
$\Leftrightarrow$, 36
$\Rightarrow$, 36, 37, 41, 44
$\ll$, 155
$\neg$, 30, 36, 154
$\vee$, 30, 36
$\oplus$, 36, 53, 65
$\mapsto$, 42
$\mathbb{N}$, 146
$\overline{a}$, *veja* $\neg$
π, 25
$\setminus$, 77
decod, 72
demux, 75
e, 24
mod, 158, 218
mux, 67
$\mathbb{B}$, 29–33, 38, 42
$\mathbb{Z}$, 42, 149
$\lceil \mathrm{r} \rceil$, 241, 242
$\mathbb{N}$, 42, 43, 149
num, 44, 55, 72, 77, 261, 262, 264, 274
num^{-1}, 55
$\mathbb{R}$, 42
$|\,\mathrm{N}\,|$, 227
$\langle\rangle$, 29, 42, 43, 180, 201, 225

Números

74148, 75
74163, 218, 272
74374, 232

A

abstração, 27, 280, 283, 292, 293, 301
  bits como sinais, 27–33, 57, 182, 184
  tempo discretizado, 116, 118, 182, 186–188, 191–193, 197, 222
acumulação de produtos parciais, 173–178
adição, 152
adiantamento de vai-um, 164–172, 312–315
alfabeto, 17
Álgebra de Boole, 27
algoritmo, 233
  conversão de base, 18
  conversão de base de frações, 22
amplificador, 125, 136
  diferencial, 138
amplitude, 210
and, *veja* $\wedge$
and array, 128
apontador de pilha, 268
aproximação, 24
árvore, 64, 118
assembly, *veja* ling. de montagem
associatividade, 31
atraso, *veja* tempo de propagação, 57
  de transporte, 305
  inercial, 304
atribuição, 12
autômatos finitos, 236
avaliação de desempenho, 316

B

barramento, 140
barrel shifter, 158
básculo, 138, 185–189
  relógio multi-fase, 226
  SR, 185, 194
binário, 20
bit, 20
  de sinal, 147
bits, 27–37, 65
  definição, 29
  expressões e fórmulas, 30
  expressão, 30
  propriedades da abstração, 31
  tipo VHDL, 281

variável, 30
bloco de registradores, 264
borda, *veja* relógio
branch, *veja* desvio condicional
buffer, 123
buffer three-state, 140
buraco, 87
busca binária, 274
byte, 11

C

C,
deslocamento, 160
overflow, 163
cadeia,
de portas, 64, 118
de somadores, 152
caminho crítico, 117
capacitor, 105, 107, 135, 136
capture FF, *veja flip flop*, destino
CAS, 134
célula de RAM, 81
célula, 100
chave,
analógica, 142
digital, 92
normalmente aberta, 92
normalmente fechada, 78, 92
chip select, 268
ciclo,
combinacional, 57
violação, 81, 184, 186, 188
de trabalho, 225
circuito,
combinacional, 57, 65
de controle, 257
de dados, 257
dual, 99
segmentado, 202
sequencial síncrono, 196, 209
circuito aberto, 57
clear, 194
clk, *veja* relógio
clock, *veja* relógio, 319
clock skew, *veja skew*
clock-to-output, 192
clock-to-Q, 192
CMOS, 59, 65, 85–142
buffer three-state, 140
célula, 100
inversor, 96
nand, 99
nor, 98
porta de transmissão, 141
portas inversoras, 99
sinal restaurado, 125
código,
Gray, 63, 217, 227, 236
Column Address Strobe, *veja* CAS
combinacional,
ciclo, 57
circuito, 57
dispositivo, 57
comparador,
de igualdade, 62, 164
de magnitude, 164, 261, 274
Complementary Metal-Oxide Semiconductor, *veja* CMOS
complemento, *veja* ¬
complemento de dois, 145–151, 154–164
complemento, propriedade, 31
comportamento transitório, *veja* transitório
comutatividade, 31
condicional, *veja* ◁ ▷
condutor, 85
conjunção, *veja* ∧
conjunto mínimo de operadores, 65
contador, 211–223
74163, 218
assíncrono, 220
em anel, 224, 225, 232
inc-dec, 224, 272
Johnson, 227
módulo-16, 215, 218
módulo-2, 212, 223
módulo-32, 223
módulo-4, 212
módulo-4096, 219
módulo-8, 213, 221
ripple, 220, 222
síncrono, 222
contra-positiva, 41
controlador, 242
de memória, 136
conversão de base, 18
conversor,
paralelo-série, 229
série-paralelo, 228
corrente, 85, 105, 106, 112, 113
de fuga, 115
corrida, 123, 125
CSS, 196–197, 209
curto-circuito, 57

D

datapath, *veja* circuito de dados
decimal, 17
decodificador, 72, 78, 81–82, 84, 126
de linha, 126, 135
de prioridades, 74
delay, 57
DeMorgan, 276
demultiplexador, 75, 120
design unit, *veja* VHDL, unidade de projeto, 278
deslocador exponencial, 156
deslocamento, 155–158
aritmético, 155, 158, 164
exponencial, 159, 201

lógico, 155, 162
rotação, 158
detecção de bordas, 123
diagrama de estado, 234
restrições, 236
disjunção, *veja* ∨
dispositivo, 85
combinacional, 57
distributividade, 31, 34, 51
divisão de frequência, 213, 216
divisão inteira, 44, 273
doador, 87
don't care, 70
dopagem por difusão, 86
dopante, 86
DRAM, 134–137
controlador, 136
fase de restauração, 137
linha,
de palavra, 135
linha de bit, 135
linha de palavra, 136
página, 135
refresh, 135
dual, 32, 95, 99
dualidade, 32
duty cycle, 225

E

EEPROM, 133
endereço, 77
energia, 100, 105, 112–115
enviesado, relógio, *veja skew*
EPROM, 133
equação característica do FF, 193
equivalência, *veja* ⇔
erro,
de representação, 23
escopo, 306
especificação, 42, 280, 293
estado, 182
estado atual, 196, 209, 212
exponenciação, 264
expressões, 36

F

fan-in, 109–112, 116, 167, 170
fan-out, 82, 84, 109–112, 116, 120, 170–172
fatorial, 273
fechamento, 31
FET, 91
Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
FIFO, 271
fila, 270–273
circular, 271
filtro digital de ruído, 190
flip-flop, 188–195
adjacentes, 198
comportamento, 193
destino, 198
fonte, 197, 198
mestre-escravo, 189
modelo VHDL, *veja* VHDL, *flip-flop*
temporização, 192
tipo JK, 193
tipo T, 191, 193
um por estado, *veja* um FF por estado
folga de *hold*, 198
folga de *setup*, 198
forma canônica, 48
FPGA, 194, 300
frações, *veja* ponto fixo
frequência, 210
frequência máxima, *veja* relógio
função, 30
tipo, 29, 42
função de próximo estado, 233, 240, 334
função de saída, 233, 240
função, aplicação bit a bit, 32
função, tipo (op. infixo), *veja* ↦

G

ganho de desempenho, 316
gerador de relógio, 192
`ghdl`, 275
glitch, *veja* transitório
GND, 93
gramática, 17
`gtkwave`, 275, 300, 311

H

handshake, 257
hexadecimal, 19
hold time, 188, 197–202, 246
folga, 198, 204, 205

I

idempotência, 31
identidade, 31
igualdade, 30
implementação, 42
implicação, *veja* ⇒
informação, 16
inicialização,
flip-flop, 194
Instrução,
busca, *veja* busca
instrução, 12
busca, *veja* busca
decodificação, *veja* decodificação
execução, *veja* execução
resultado, *veja* resultado
interface,
de rede, 13
de vídeo, 12
inversor, 96
tempo de propagação, 109
involução, 31, 61

isolante, 85

J

Joule, 112
jump, *veja* salto incondicional

L

label, 333
latch, *veja* básculo
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
Lei de Joule, 112
Lei de Kirchoff, 106
Lei de Ohm, 105, 106
LIFO, 268
ligação,
  barramento, 140
  em paralelo, 93, 99
  em série, 93, 99
linguagem,
  assembly, *veja* ling. de montagem
  C, *veja* C
  declarativa, 275, 285
  imperativa, 285
  Pascal, *veja* Pascal
  Verilog, 276
  VHDL, *veja* VHDL
  Z, 27
linha de endereçamento, 78
literal, 38
logaritmo, 43
lógica restauradora, 125
look-up table, 300
LUT, 300

M

MADD, 201
Mapa de Karnaugh, 49, 124
máquina de estados, 233, 236
  Mealy, 238, 245, 256, 273, 337
  modelo VHDL, 334
  Moore, 237, 244, 256, 269, 335
  projeto, 240
Máquina de Mealy, *veja* máq. de estados
Máquina de Moore, *veja* máq. de estados
máscara, 32
máximo e mínimo, 31
maxtermo, 46
Mealy, *veja* máq. de estados
memória, 181
  atualização, 77
  bit, 184
  de programação de FPGA, 300
  de vídeo, 13
  decodificador de linha, 80
  endereço, 77
  FLASH, 133
  matriz, 129, 132, 134
  multiplexador de coluna, 80
  primária, 13
  RAM, 81, 134, 267
  ROM, 78, 126, 245
  secundária, 13
memória dinâmica, *veja* DRAM
memória estática, *veja* SRAM
metaestabilidade, 184, 188, 191, 194
  defesa contra artes das trevas, 191
microcontrolador, 245–253
microrrotina, 253
mintermo, 45, 124, 126
modelagem, 280
modelo, 280
  dataflow, 294
  estrutural, 284, 294, 295
  funcional, 44, 293, 295
  porta lógica, 96
  RTL, 294
  temporal, 304
  temporização, 115
módulo de contagem, 221
módulo, *veja* %, *mod*
Moore, *veja* máq. de estados
MOSFET, 91
multiplexador, 61, 66–70, 80, 101, 117, 119, 123–124, 126, 141, 142, 278–279, 284–286, 309–311
  de coluna, 132, 136
multiplicação, 172–178, 262
  acumulação de produtos parciais, 173–178
multiplicador,
  somas repetidas, 262, 273
multiply-add, *veja* MADD

N

número,
  de Euler, 24
negação, *veja* ¬
net list, 284, 294
níveis de abstração, *veja* abstração
nível lógico,
  0 e 1, 28
  indeterminado, 28, 109, 116, 141
  terceiro estado, 140
nó, 96
non sequitur, 37
not, *veja* ¬
número primo, 53

O

octal, 18
onda quadrada, 188, 210, 319
operação,
  binária, 29
  bit a bit, 32
  infixada, 42
  MADD, 201
  prefixada, 47, 279
  unária, 29
operação apropriada, 197

operações sobre bits, 29–33
operador,
  binário, 29
  lógico, 36
  unário, 29
operation code, *veja opcode*
or, *veja* ∨
or array, 129
ou exclusivo, *veja* ⊕
ou inclusivo, *veja* ∨
output enable, 268
overflow, 148–150, 154, 162–164, 173, 178

P

paridade, 332
  ímpar, 49
  par, 49
período mínimo, *veja* relógio
pilha, 268–270, 273
pipelining, *veja* segmentação, 202
piso, *veja* ⌊v⌋
ponto fixo, 151–152
ponto flutuante, 70
pop, 268
porta,
  de escrita, 265
  de leitura, 265
porta lógica, 65
  and, 59
  carga, *veja fan-out*
  de transmissão, 141, 185
  modelo VHDL, 295
  nand, 60, 99
  nor, 60, 98
  not, 59, 96
  or, 59
  xor, 60, 65, 191
portas complexas, 100
potência, 112–115
  dinâmica, 114
  estática, 115
potenciação, 43
precedência, 30
precisão,
  representação, 23
preset, 194
prioridade,
  decodificador, 74
processador, 12
produtório, 33
produto de somas, 46
programa de testes, *veja* VHDL, *testbench*
PROM, 133
propriedades, operações em $\mathbb{B}$, 31
protocolo,
  de sincronização, 257, 273
prova de equivalência, 40–41
próximo estado, 196, 212
pull-down, 96
pull-up, 96, 126, 141
pulso, 122, 123, 185, 194, 211, 219, 235, 261
  espúrio, *veja* transitório
push, 268

R

raiz quadrada, 274
RAM, 12, 77, 81, 134–139
  célula, 81
  dinâmica, 135
Random Access Memory, *veja* RAM
RAS, 134
Read Only Memory, *veja* ROM
realimentação, 81
receptor, 87
rede, 96
redução, 33
refresh, 137, 139
Register Transfer Language, *veja* RTL
registrador, 195, 232, 250, 264
  carga paralela, 195
  de segmento, 201
  simples, 195
registrador de deslocamento, 228–232
  modelo VHDL, *veja* VHDL, registrador, 329
  paralelo-série, 229
  série-paralelo, 228
  universal, 231
registrador de estado, 196, 334
relógio, 186, 188, 192, 210–225, 319
  bordas, 321
    ascendente, 189
    descendente, 189
  ciclo de trabalho, 225
  enviesado, *veja skew*
  frequência máxima, 199
  multi-fase, 225
  período mínimo, 199
representação,
  abstrata, 28
  binária, 20
  complemento de dois, 147
  concreta, 27
  decimal, 17
  hexadecimal, 19
  octal, 18
  ponto fixo, 151
  posicional, 17
  precisão, 23
reset, 185, 194, 195, 320, 325
resistência, 87, 100, 105
ROM, 12, 77–80, 126–133
rotação, 158, 164
Row Address Strobe, *veja* RAS
RTL, 202, 277, 294
rvalue, 326

S

segmentação, 201

seletor, 72
semântica, 17
semicondutor, 85
  tipo N, 87
  tipo P, 87
set, 185, 194, 325
setup time, 188, 197–202, 246
  folga, 198, 204, 205
silogismo, 37
simplificação de expressões, 38–40
simulação,
  eventos discretos, 287
sinal, 27, 42, 282, 326
  analógico, 27
  digital, 27, 28
  fraco, 125, 126, 138, 141
  intensidade, 90, 139
  restaurado, 125, 139
síntese, *veja* VHDL, síntese, 300
skew, 202–207
Solid State Disk, *veja* SSD
soma, *veja* somador
soma de produtos, 45, 51, 126
  completa, 45
somador, 145, 152–153, 293–300
  adiantamento de vai-um, 165, 232, 311–316
  cadeia, 152
  completo, 104, 152, 295
  modelo VHDL, 295, 312, 314
  overflow, 163
  parcial, 103
  seleção de vai-um, 170–172
  serial, 231
  temporização, 201, 311–316
  teste, 178, 298
somatório, 33
`spice`, 28
SRAM, 138–139
SSD, 14
status, 162
subtração, 154
superfície equipotencial, 96, 110

T

tabela,
  de excitação dos FFs, 193
tabela verdade, 33–35, 45
tamanho, *veja* $|\,\mathrm{N}\,|$
tempo,
  de contaminação, 115, 118–121, 184, 192, 306
  de estabilização, 188, 192
  de manutenção, 188, 192
  de propagação, 57–58, 61, 64–65, 73, 77, 84, 102, 104, 109, 115–117, 192, 207, 222, 223, 304
  discretizado, 186, 188, 192, 197, 222
temporização, 104–126, 304–316
  somador, 311–316
tensão, 105
Teorema,
  Absorção, 49
  DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
  Dualidade, 99
  Simplificação, 49
terceiro estado, 140–141
testbench, *veja* VHDL, *testbench*
teste,
  cobertura, 179
  de corretude, 178
teto, *veja* $\lceil \mathrm{r} \rceil$
three-state, *veja* terceiro estado
tipo,
  de sinal, 42
  função, 29
Tipo I, *veja* formato
Tipo J, *veja* formato
Tipo R, *veja* formato
transferência entre registradores, *veja* RTL
transistor, 87–91, 95–96
  CMOS, 95
  corte, 113
  gate, 88
  saturação, 113
  sinal fraco, 90
  tipo N, 88
  tipo P, 89
Transistor-Transistor Logic, *veja* TTL
transitório, 121–123, 186, 239, 311
transmissão,
  serial, 230
transmission gate, *veja* porta de transmissão
TTL,
  74148, 75
  74163, 218
  74374, 232
tupla, *veja* $\langle\,\rangle$
  elemento, 32
  largura, 43

U

ULA, 160–164, 180, 264
  status, 162
um FF por estado, 253–256
Unidade de Lógica e Aritmética, *veja* ULA
unidade de projeto, 283

V

valor da função, 30
VCC, 93
Verilog, 276
vetor,
  de testes, 289
vetor de bits, *veja* $\langle\,\rangle$, 32
  largura, 43
VHDL, 194, 275–340
  `&`, 281
  `(i)`, 282
  `:=`, 282
  `<=`, 281

=>, 286
active, 327
after, 304, 311
all, 284
architecture, 278
área concorrente, 285
arquitetura, 278
array, 290
assert, 292
associação,
  nominal, 286
  posicional, 285
atraso,
  de transporte, 305
  inercial, 304
atributo, 321, 327
bit, 278, 282, 294
bit_vector, 278
boolean, 282
borda ascendente, 321, 338
case, 331
concatenação, 282
delta, 287–288, 318
design unit, *veja* VHDL, unidade de projeto
entidade, 278
entity, 278
event, 321, 327
evento, 287
exit, 333
flip-flop, 321–326
for, 292, 332
função, 338–340
generate, 311
generic, 308
in, 278, 333
inertial, 304
inicialização de sinal, 282
instanciação, 285
interface genérica, 308
label, 333
last_value, 327
length, 327
linguagem, 275
loop, 332, 333
mapeamento de portas, 285
máquina de estados, 334
modelo executável, 280
open, 309
others, 331
out, 278
package, 283
port map, 279, 285, 286
processo,
  combinacional, 328
  sequencial, 329
processo, 287
range, 327, 333
record, 289
registrador, 329
reject, 305, 311
report, 292
rising_edge, 338, 339
rvalue, 326
síntese, 280
seleção de elemento de vetor, 282
severity, 292
simulação, 287
sinal, 282
síntese, 207, 300, 304
std_logic, 140
subtype, 283
testbench, 288–292, 298–300
time, 282
tipos, 42, 281, 286
to, 333
transação, 287
transport, 305
type, 282, 334
unidade de projeto, 283
use, 284
variável, 326–330
wait, 319, 320, 322
wait for, 320
wait on, 320
wait until, 320
when, 331
while, 333
work, 284

W

Watt, 112
write back, *veja* resultado

X

xor, *veja* ⊕

Z

Z, linguagem, 27