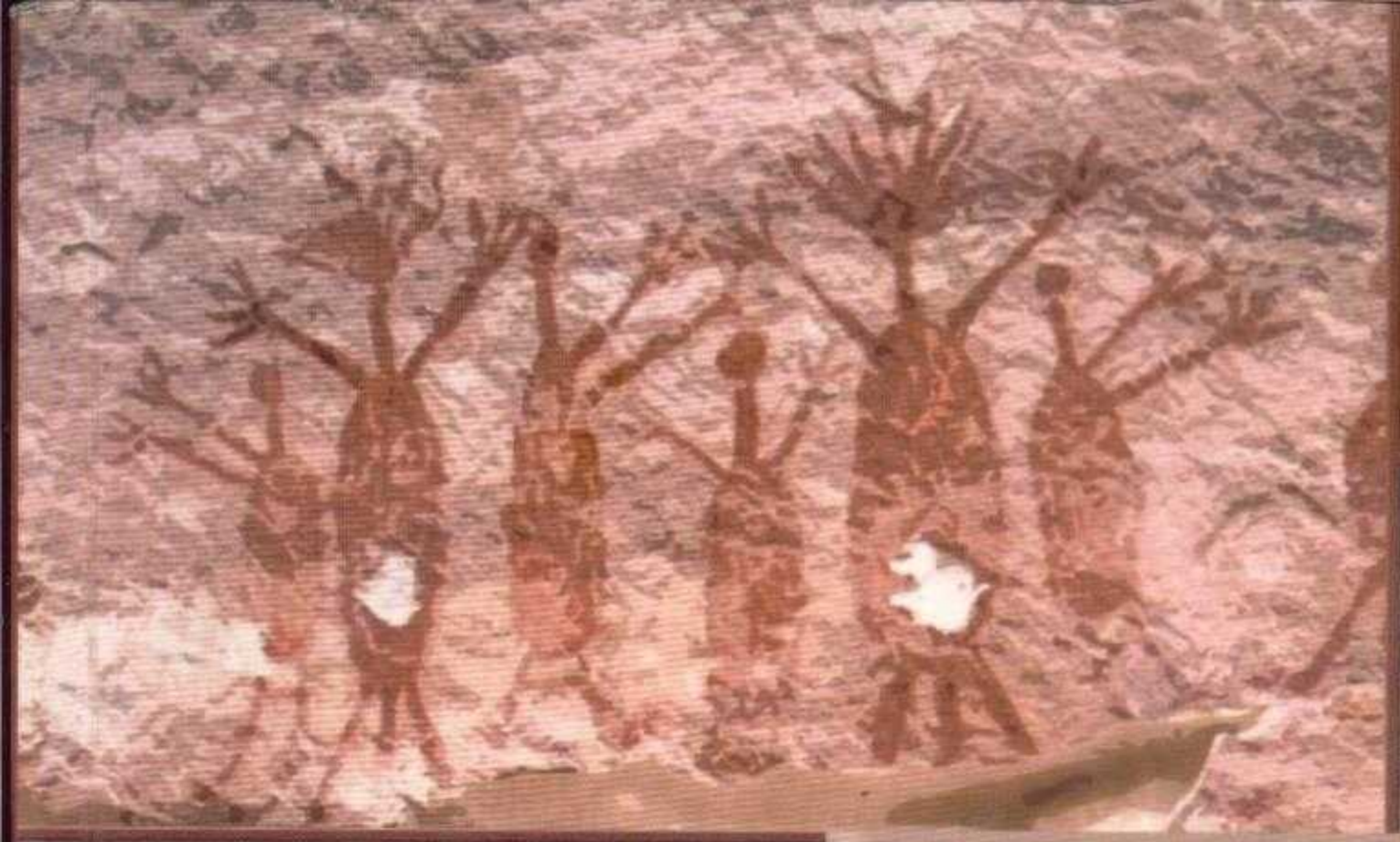
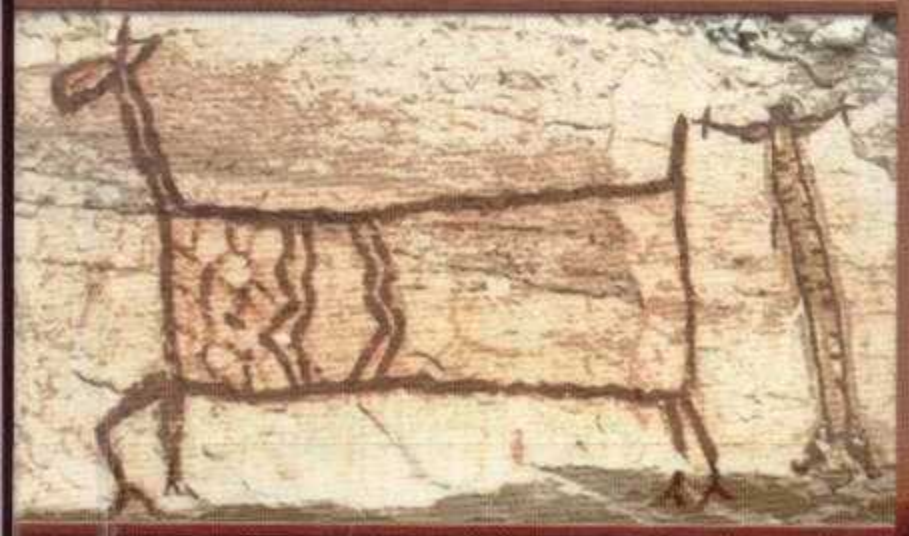




VHDL

»Descrição e Síntese de Circuitos Digitais«

Roberto d'Amore

LTC

VHDL

»Descrição e Síntese de Circuitos Digitais«

Roberto d'Amore

VHDL — Descrição e Síntese de Circuitos Digitais apresenta a linguagem de descrição de circuitos VHDL de maneira gradual, em uma seqüência de tópicos que evita a referência a conceitos ainda não abordados. Por meio da simulação de mais de 170 códigos propostos e com o auxílio de mais de 220 figuras, o leitor experimentará um aprendizado da linguagem rápido e motivador.

Devido à abrangência da obra são propostas várias seqüências de leitura, de forma a atender leitores com diferentes necessidades. Assim, o estudo de tópicos específicos dos seis primeiros capítulos é ideal para o leitor com urgência em obter resultados práticos. Enquanto isso, um aprofundamento maior pode ser atingido com os capítulos subseqüentes — que tratam de estruturas hierárquicas, esquemas de geração, esquemas de iteração, subprogramas, padrões IEEE 1164 e 1076.3, síntese de memórias, operações com arquivos, configurações, funções de resolução e estratégias de teste. As distinções entre as versões VHDL-1987 e VHDL-1993 são mencionadas ao longo do texto.

VHDL — Descrição e Síntese de Circuitos Digitais dá ênfase à síntese de circuitos empregando a linguagem e procurando, sempre que possível, estabelecer um elo entre o tópico abordado e sua aplicação. Nos 15 capítulos são discutidos mais de 60 casos de síntese através da análise dos circuitos gerados por ferramentas de síntese, além dos cuidados na descrição e formas preferenciais de código visando à síntese.

VHDL — Descrição e Síntese de Circuitos Digitais atende às necessidades de:

- Projetistas de circuitos digitais, com experiência no desenvolvimento de circuitos digitais através de captura esquemática, que necessitam migrar para esta nova técnica de projeto;
- Profissionais da área de informática que desejam aplicar sua experiência em programação na síntese de circuitos;
- Estudantes de graduação e pós-graduação que tenham conhecimentos na área de circuitos digitais.

É possível empregar esta obra, também, como livro-texto de cursos da linguagem, pois ela conta com imagens de auxílio didático que podem ser solicitadas ao autor e mais de 140 exercícios propostos.

LTC LIVROS TÉCNICOS E CIENTÍFICOS EDITORA

www.ltceditora.com.br

Prefácio

Introdução

Uma exposição gradual dos conceitos envolvidos em linguagens de programação envolve algumas dificuldades, devido à interligação dos temas. A explicação de um tópico pode necessitar de um conceito ainda não abordado, comprometendo, assim, o seu entendimento. A seqüência proposta procura não mencionar temas ainda não abordados; quando necessário, é dada uma breve explicação sobre o novo tópico, e o capítulo que trata o tema com mais profundidade é indicado.

A ordem dos tópicos abordados visa a uma curva de aprendizado acelerado para projetistas com experiência no desenvolvimento de circuitos digitais e profissionais da área de programação. Logo após o terceiro capítulo, é possível realizar a descrição, simulação e síntese de vários circuitos básicos. Assim, leitores habituados a desenvolver circuitos através de captura esquemática ficam motivados, devido à identificação entre as estruturas da linguagem e circuitos reais. Profissionais da área de programação são, também, beneficiados, pois a complexidade dos circuitos propostos nos exercícios vai sendo elevada gradualmente, conforme novos tópicos são apresentados.

Sempre que possível, procuramos manter uma correlação entre circuitos reais e os exemplos apresentados. Para projetistas de circuitos digitais esse vínculo é importante, pois estabelece uma ligação entre a estrutura desejada e o modo de descrevê-la. No caso de profissionais da área de programação, que muitas vezes desconhecem as reais possibilidades de implementação de um circuito, o elo com exemplos reais evita o emprego de códigos complexos e por demais artificiosos, que nem sempre conduzem a circuitos possíveis de serem sintetizados.

Formato do texto

O texto contém figuras, tabelas e quadros de código. A numeração das figuras, tabelas e quadros identifica a seção a que se referem. A figura que segue ilustra dois tipos de quadros utilizados. No exemplo à esquerda da figura, o quadro contém apenas uma parcela de um código. O quadro à direita, dividido em duas regiões, ilustra um código completo de uma descrição. A área contendo a numeração de linhas é utilizada apenas para referência no texto; ela não deve ser inserida no código. Deve-se observar que é dada preferência a caracteres alfanuméricos pertencentes ao código ASCII nos quadros, e, portanto, a grafia de algumas palavras nessa área pode estar incorreta.

Os termos em inglês foram, em sua maioria, vertidos para o português e podem ser encontrados no índice.

área para numeração de linhas:
não pertence ao código

área de código:
somente caracteres ASCII

área de código:
somente caracteres ASCII

```
ot <= (i0 AND NOT s1 AND NOT s0) OR
      (i1 AND NOT s1 AND     s0) OR
      (i2 AND     s1 AND NOT s0) OR
      (i3 AND     s1 AND     s0);
```

Quadro a.1.2 Segmento de um código

```
1   ENTITY mux_0 IS
2     PORT (i0, i1, i2, i3      :  IN  BIT;
3           s0, s1              :  IN  BIT;
4           ot                  :  OUT BIT);
5   END mux_0;
6
7   ARCHITECTURE teste OF mux_0 IS
8   BEGIN
9     ot <= (i0 AND NOT s1 AND NOT s0) OR
10          (i1 AND NOT s1 AND     s0) OR
11          (i2 AND     s1 AND NOT s0) OR
12          (i3 AND     s1 AND     s0);
13  END teste;
```

Quadro a.1.1 Exemplo de um código completo

Exemplos de quadros contendo códigos de descrições.

Conteúdo dos capítulos

O Capítulo 1 tem como objetivo apresentar um breve histórico da linguagem e os passos envolvidos na síntese de um circuito digital. As etapas de uma síntese são, normalmente, dependentes das ferramentas utilizadas. Procuramos apresentar uma visão que englobe, de uma maneira mais ampla, os possíveis passos envolvidos. Como neste capítulo são abordadas as etapas de um projeto, e devido à falta de elementos necessários para desenvolver uma descrição, as atividades podem ser complementadas com o aprendizado das ferramentas que serão utilizadas. Descrições completas são oferecidas, e todo o processo de compilação, simulação, síntese e programação de uma *FPGA*, pode ser visto.

No Capítulo 2 são introduzidos os primeiros conceitos da linguagem, como tipos e operadores predefinidos. Novamente, os exemplos e exercícios visam sedimentar o conhecimento das ferramentas utilizadas empregando os conceitos vistos. Uma breve explicação sobre a definição de tipo é apresentada. Ela é suficiente para os capítulos seguintes.

Os Capítulos 3 e 4 apresentam alguns comandos concorrentes e seqüenciais que permitem elaborar descrições de complexidade mediana. Como a linguagem VHDL fornece diversas alternativas para descrever o comportamento de um mesmo circuito, muitas das construções abordadas têm a sua operação exemplificada através de um mesmo circuito. Isso evita a exposição de um novo circuito a cada novo comando e enfatiza as múltiplas opções para a descrição. Os exercícios propostos procuram confrontar essas opções, com o objetivo de evitar que uma determinada construção fique associada a um tipo de circuito.

No Capítulo 5 são dados os conceitos de atraso, variável e atributos. Alguns tópicos podem ser omitidos ou resumidos, permitindo que esse capítulo seja inserido em um outro capítulo. A descrição de todos os atributos é um pouco tediosa, e pode ser abordada rapidamente. Atributos como " event " e " stable " não devem ser omitidos, devido a suas aplicações na síntese de circuitos seqüenciais.

O Capítulo 6 contém as principais estratégias para descrição de circuitos seqüenciais visando à síntese. Registradores, máquinas de estado e a inicialização desses elementos são discutidos. No final desse capítulo, é possível realizar a descrição de circuitos com complexidade média para alta, pois muitos dos tópicos importantes da linguagem já foram abordados.

No Capítulo 7 as possibilidades de um projeto são estendidas: estruturas hierárquicas, e esquemas de iteração que permitem gerar circuitos regulares são apresentados. A hierarquia é constituída de descrições que são referenciadas na forma de componentes. Os esquemas de geração e iteração por laço permitem, ainda, a proposta de descrições escaláveis, onde o tamanho do circuito pode ser programado no momento da síntese. Os exercícios propostos abordam estruturas regulares, como somadores e multiplicadores celulares, onde é necessária a interligação de diversos circuitos em um modo preestabelecido.

O Capítulo 8 aborda os subprogramas função e procedimento. Ainda nesse capítulo, é estabelecido um paralelo entre a criação de circuitos empregando componente, vista no capítulo anterior, e a criação de circuitos empregando subprogramas. São abordadas, também, outras operações de subprogramas, como funções de conversão e sobrecarregamento de funções predefinidas.

O Capítulo 9 fornece subsídios para os tópicos a serem abordados no capítulo que segue, discutindo bibliotecas e pacotes. Esse capítulo tem a possibilidade de ser condensado, se necessário. Vários dos conceitos abordados podem ser apresentados nos Capítulos 10 e 11, onde são discutidos dois pacotes padronizados pelo IEEE. Os exercícios propostos são interessantes, sendo viável adaptá-los, caso todos os conceitos do capítulo não tenham sido abordados. O conjunto de exercícios leva à criação de um circuito que realiza uma operação de divisão.

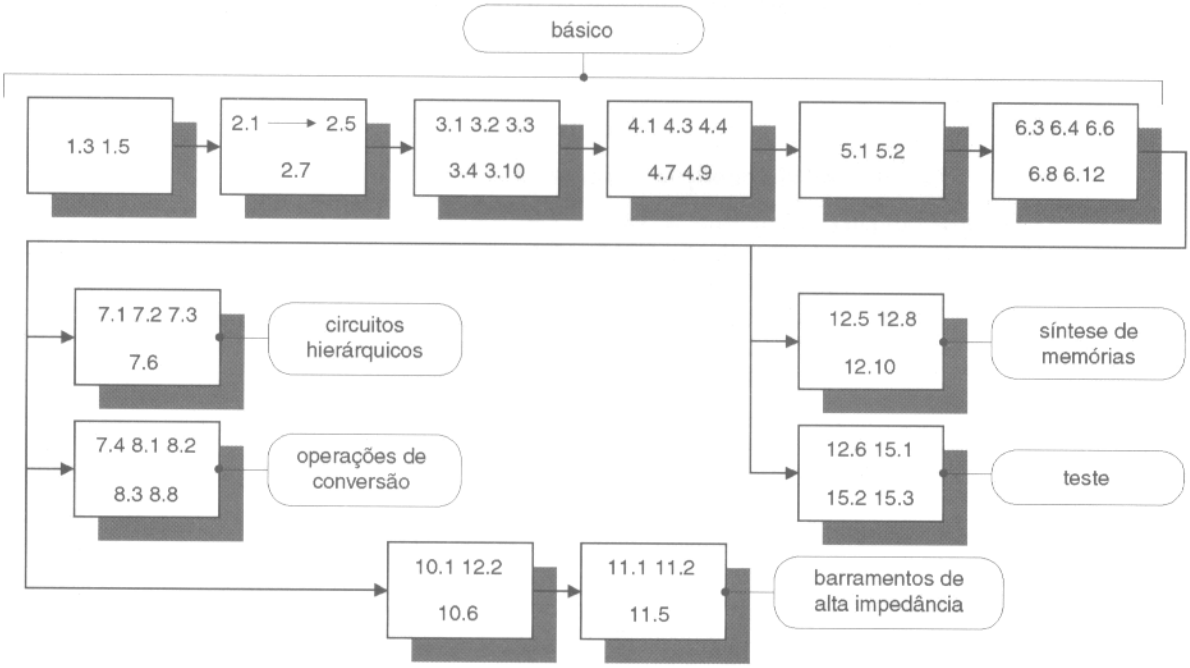
Os Capítulos 10 e 11 são imprescindíveis para a síntese de circuitos que envolvam barramentos e portas com saída em estado de alta impedância. Esses capítulos, podem ser apresentados logo após o Capítulo 6 ou 7, conforme a necessidade. Naturalmente, uma breve menção dos conceitos de pacotes e bibliotecas é necessária, caso o Capítulo 9 venha a ser omitido.

No Capítulo 12 é abordada a criação de novos tipos, ampliando os conceitos vistos no Capítulo 2. Com relação à síntese, é apresentada a descrição de memórias RAM e ROM empregando os novos conceitos introduzidos. O Capítulo 13 continua a abordagem da definição de tipos, com a apresentação do tipo arquivo. Esse tipo permite a operação com arquivos no sistema hospedeiro. A discussão engloba arquivos no formato binário e no formato texto. Também são abordadas as diferenças entre as versões VHDL-1987 e VHDL-1993.

No Capítulo 14 são apresentados comandos que permitem gerar mensagens na fase de simulação, condicionar a execução de um comando em função do término das iterações que ocorrem na simulação de comandos concorrentes, e configurar descrições estruturais. A criação de funções de resolução é, também, discutida. O Capítulo 15 contém uma série de técnicas para o teste de descrições empregando os conceitos abordados nos Capítulos 13 e 14.

Sugestões para leitura e proposta de curso

Uma opção natural para a leitura do texto, bem como uma proposta de um curso sobre a linguagem, é a adoção da seqüência de capítulos apresentada no texto. Entretanto, nem sempre o leitor ou o instrutor dispõem de tempo suficiente, ou necessitam de todos os tópicos apresentados. Na figura que segue, são sugeridas alternativas para o estudo, procurando selecionar itens de interesse específico, de modo a melhor atender uma determinada necessidade e reduzir o tempo necessário.



Opções de leitura do texto.

Material didático de apoio

O estudo de uma linguagem de programação é extremamente monótono se não forem oferecidas atividades para exercitar os conceitos lecionados. Visando motivar o aprendizado, foi inserido ao fim de cada capítulo um conjunto de exercícios. Como complemento aos exercícios, está disponível no endereço eletrônico do autor um material contendo a solução de todos os exercícios propostos, com comentários.

Além desse material, cada capítulo dispõe de uma série de imagens para auxílio didático dos professores. Mais informações sobre esses complementos podem ser obtidas no endereço eletrônico do autor www.ele.ita.br/~damore/vhdl/ indicado no sítio www.ltceditora.com.br.

Agradecimentos

Gostaria de agradecer ao Prof. Jacobus Willibrordus Swart, responsável pelo "Programa Especial para Fabricação de Circuitos Integrados no Exterior", da FAPESP, e ao comitê organizador desse projeto. Graças a esse programa, foi possível o acesso a ferramentas que auxiliaram na elaboração da obra.

Não poderia deixar de mencionar, também, o apoio da equipe do Editorial Técnico da Editora LTC ao longo do desenvolvimento deste trabalho, em especial ao Prof. Bernardo Severo e à Carla.

Direitos autorais e capa

A remuneração do autor foi doada à FUMDHAM, entidade científica sem fins lucrativos, co-gestora do Parque Nacional Serra da Capivara.

A capa é ilustrada com pinturas rupestres do Parque Nacional da Capivara. A idéia é estabelecer um paralelo entre duas formas de linguagem: a utilizada pelos primeiros habitantes no nosso continente para *descrever* o cotidiano e a empregada para *descrever* circuitos eletrônicos.

Comentários finais

A elaboração do livro foi executada com o máximo cuidado, tendo como base o manual de referência de linguagem. Todos os códigos foram compilados, e operaram corretamente para os estímulos de teste aplicados. O autor, entretanto, não fornece garantias com relação às descrições e à documentação contida no texto. Deve-se lembrar, ainda, de que os resultados obtidos, principalmente no caso de síntese, são dependentes do aplicativo e versão da ferramenta empregada. Assim, alterações nos resultados podem ser esperadas. A interpretação dessas diferenças foge do escopo da obra, e devem ser analisadas com base no manual de referência da linguagem e nas informações fornecidas pelo fabricante da ferramenta.

Sumário

Capítulo 1

Introdução 1

1.1 Histórico 1
1.2 Aspectos gerais da linguagem 2
1.3 Síntese de circuitos 3
1.4 Principais pontos abordados 6
1.5 Exercícios 6

Capítulo 2

Primeiro contato com a linguagem 10

2.1 Entidade de projeto 10
Declaração da entidade e corpo da arquitetura 11
2.2 Classe de objetos: constante, variável, sinal e arquivo 12
2.3 Tipos 14
Tipos escalares 15
Tipos compostos 15
Nomes indexados, parte de vetores e agregado 16
Definição de novos tipos 17
2.4 Operadores 17
2.5 Exemplos de utilização 21
2.6 Principais pontos abordados 25
2.7 Exercícios 25

Capítulo 3

Comandos concorrentes básicos 28

3.1 Atraso delta Δ 28
3.2 Atribuição de valor para um sinal 30
3.3 Construção " WHEN ELSE " 31
3.4 Construção " WITH SELECT " 32
3.5 Comando " BLOCK " 34
3.6 Comando " PROCESS " 36
3.7 Palavra reservada " UNAFFECTED " 37
3.8 Cuidados na descrição 37
Comparações entre as construções " WHEN ELSE " e " WITH SELECT " 37
Criação de *latch* com as construções " WHEN ELSE " e " WITH SELECT " 39
3.9 Principais pontos abordados 40
3.10 Exercícios 40

Capítulo 4

Comandos seqüenciais básicos 43

4.1 Lista de sensibilidade em processos 43
4.2 Atribuição de valor para um sinal – região seqüencial 44
4.3 Construção " IF ELSE " 46
4.4 Construção " CASE WHEN " 48
4.5 Comando " WAIT " 49
4.6 Comando " NULL " 51
4.7 Cuidados na descrição 53
Comparações entre " WHEN ELSE " e " IF ELSE " 53
Comparações entre " WITH SELECT " e " CASE WHEN " 53
Construções " IF ELSE " e " CASE WHEN " aninhadas 53
O emprego da construção " IF ELSE " e " CASE WHEN " 55
Expressão de escolha das construções " WITH SELECT " e " CASE WHEN " 57
4.8 Principais pontos abordados 58
4.9 Exercícios 58

Capítulo 5

Atraso, variável e atributos 62

5.1 Atraso 62
5.2 Sinal e variável 63
5.3 Atributos 65
Atributos predefinidos relativos a sinais que resultam em novos sinais 65
Atributos predefinidos relativos a sinais que não resultam em novos sinais 67
Atributos predefinidos relativos a vetores 68
Atributos predefinidos relativos a tipos 70
Atributos definidos pelo usuário 71
5.4 Cuidados na descrição 73
Ordem na atribuição de valores 73
5.5 Principais pontos abordados 75
5.6 Exercícios 75

Capítulo 6

Estratégias de descrição de circuitos síncronos 78

6.1 Conceitos iniciais 78
6.2 Registrador sensível a nível 79
6.3 Registrador sensível a borda – inicialização síncrona 81
6.4 Registrador sensível a borda – inicialização assíncrona 82
6.5 Registrador sensível a borda com sinal para habilitação do sinal de relógio 83
6.6 Máquinas de estados 86
6.7 Inferência de registrador empregando variável 90
6.8 Contadores 91
6.9 Alternativas para descrição de registradores 93
6.10 Cuidados na descrição 94
Sinais na lista de sensibilidade 94
Ciclo de inicialização 95
Inserção desnecessária de elementos de memória 95
6.11 Principais pontos abordados 95
6.12 Exercícios 96

Capítulo 7

Componentes e esquemas de iteração 100

7.1 Componentes 100
Declaração de componente 100
Solicitação de componente 100
Solicitação direta de componente 102
7.2 Genéricos 103
Definição de genéricos 103
Mapa de genéricos na solicitação de componentes 104
7.3 Comando " GENERATE " 106
Esquema de geração " FOR " 106
Esquema de geração " IF " 108
7.4 Comando " LOOP " 108
Esquema de iteração " FOR " 108
Esquema de iteração " WHILE " 111
Comandos " NEXT " e " EXIT " 113
Laços infinitos 113
7.5 Principais pontos abordados 114
7.6 Exercícios 114

Capítulo 8

Subprogramas 119

8.1 Subprogramas 119
8.2 Função 121
8.3 Procedimento 123
8.4 Sobrecarregamento de subprogramas e operadores 125
8.5 Vetores sem especificação de limites em subprogramas 127
8.6 Cuidados na descrição 128
Comportamento de variáveis declaradas em subprogramas 128
Chamada de procedimentos: regiões concorrentes 129
Variáveis compartilhadas 130
8.7 Principais pontos abordados 131
8.8 Exercícios 132

Capítulo 9

Bibliotecas e pacotes 135

9.1 Bibliotecas de projeto 135
9.2 Pacotes 137
Ferramentas de síntese 141
9.3 Ordem de análise 141
9.4 Constantes deferidas 145
9.5 Sinais globais 146
9.6 Principais pontos abordados 147
9.7 Exercícios 148

Capítulo 10

Padrão IEEE 1164 151

10.1 Tipos definidos 151
10.2 Funções e subtipos definidos no pacote " std_logic_1164 " 153
10.3 Descrições empregando o pacote " std_logic_1164 " 156
Funções lógicas 156
Operação da função de resolução em caso de conflito 157
Operações com o valor não importa " - " 158
10.4 Síntese empregando tipos do pacote " std_logic_1164 " 158
Saídas com estado em alta impedância 159
Resultado da codificação de valores no circuito gerado 161
Síntese com o valor "não importa" – comparação 162
Síntese com o valor "não importa" – minimização de tabelas 163
Detecção de bordas de subida ou descida 164
Comando " NULL ": construção " CASE WHEN " e valores " std_logic " 165
10.5 Principais pontos abordados 166
10.6 Exercícios 166

Capítulo 11

Padrão IEEE 1076.3 170

11.1 Tipos definidos 170
11.2 Operações definidas 170
11.3 Exemplos de utilização do padrão IEEE 1076.3 175
Funções aritméticas 176
Funções lógicas 176
Saídas com estado em alta impedância 177
Operações relacionais com tipos " unsigned " e " signed " 177
11.4 Exemplo de síntese com o padrão IEEE 1076.3 178
Valor "não importa" e a função " std_match " 178
Transbordo em operações de soma com tipos " unsigned " e " signed " 179
Circuitos síncronos 181
11.5 Principais pontos abordados 183
11.6 Exercícios 184

Capítulo 12

Declaração de tipos e pseudônimos 188

12.1 Tipos 188
12.2 Definição de tipos enumerados 188
12.3 Definição de tipos físicos 189
12.4 Definição de subtipos 189
12.5 Definição de tipos vetor 190
Declarando um vetor 190
Definição de um vetor com limites em aberto 192
Vetor composto de elementos do tipo vetor 193
Vetor multidimensional 195
12.6 Definição de tipos registro 196
12.7 Pseudônimos 197
12.8 Síntese de memórias 198
Memórias ROM 199
Memórias RAM 201
12.9 Principais pontos abordados 203
12.10 Exercícios 204

Capítulo 13

Arquivos 206

13.1 Emprego de arquivos 206
Declaração de um tipo arquivo 206
Declaração de objetos tipo arquivo – versão VHDL-1987 207
Operações implicitamente declaradas – versão VHDL-1987 207
Exemplos – versão VHDL-1987 208
Declaração de objetos tipo arquivo – versão VHDL-1993 209
Operações implicitamente declaradas – versão VHDL-1993 210
Exemplos – versão VHDL-1993 211
13.2 Pacote " TEXTIO " 212
Versão VHDL-1987 215
Versão VHDL-1993 217
13.3 Principais pontos abordados 218
13.4 Exercícios 219

Capítulo 14

Tópicos avançados 221

14.1 Comando " ASSERT " 221
14.2 Comando " REPORT " 223
14.3 Tipo com função de resolução 224
14.4 Configurações 227
Especificação de configuração 227
Declaração da configuração 229
Síntese contendo configuração 234
14.5 Comandos concorrentes pospostos 236
14.6 Principais pontos abordados 238
14.7 Exercícios 238

Capítulo 15

Teste 241

15.1 Introdução 241
15.2 Geração de estímulos 241
Sinais aleatórios 242
Sinais periódicos 243
15.3 Teste de uma descrição 246
Teste empregando constantes para geração dos estímulos 246
Teste empregando dados em arquivo para geração dos estímulos 250
Teste empregando técnica mista 252
15.4 Principais pontos abordados 253
15.5 Exercícios 253

CAPÍTULO

1 INTRODUÇÃO

Uma linguagem de descrição de circuito padronizada possibilita o intercâmbio de informações referentes ao comportamento de um circuito entre fabricantes, fornecedores de sistemas e empresas de projetos.

Outras aplicações possíveis de uma linguagem para descrição de circuito, ou HDL, *Hardware Description Language*, podem ser encontradas no teste de circuitos e na síntese do circuito descrito. Esta última aplicação permite substituir o processo de desenvolvimento de circuitos com captura esquemática com grande vantagem. Por exemplo, máquinas de estados empregadas no controle de um sistema podem ser sintetizadas através da inferência de construções do tipo " IF THEN ELSE ". Uma outra opção reside na descrição de circuitos parametrizáveis, onde as dimensões das unidades internas são definidas no momento da síntese. A descrição de um circuito somador pode ser condicionada a uma constante genérica " n " que, definida no momento da síntese, estabelece a dimensão desejada para o somador.

Este capítulo contém um breve histórico e os motivos que levaram ao desenvolvimento da linguagem de descrição de circuitos denominada VHDL. São abordados, também, os aspectos principais dessa linguagem e as etapas envolvidas em um processo de síntese.

1.1 Histórico

A linguagem VHDL deve o seu desenvolvimento à necessidade de uma ferramenta de projeto e documentação padrão para o projeto VHSIC — *Very High Speed Integrated Circuit*, do Departamento de Defesa dos Estados Unidos da América (DARPA). No ano de 1981, esse departamento patrocinou um encontro de especialistas para discutir os métodos para descrição de circuitos. Em 1983, o Departamento de Defesa definiu os requisitos de uma linguagem de descrição de circuitos padrão. Ainda nesse mesmo ano, foi concedido às firmas " IBM ", " Texas " e " Intermetrics " um contrato para o desenvolvimento da linguagem e ferramentas. A padronização da linguagem pelo IEEE — *Institute of Electrical and Electronic Engineer*, teve como base a versão 7.2. O processo foi auxiliado pela empresa " CLSI ", contratada por um laboratório da Força Aérea dos Estados Unidos, em 1986. No ano de 1987, após revisões propostas por acadêmicos e representantes de indústrias e do governo dos Estados Unidos, surgiu o padrão IEEE 1076-1987.

Em 1993 uma nova versão da linguagem, denominada IEEE 1076-1993, foi aprovada pelo IEEE. As alterações introduzidas nessa nova versão não acrescentam características significativas para a síntese de circuitos. As diferenças marcantes entre as versões estão relacionadas com o tratamento de arquivos. Neste texto, quase todas as descrições podem ser compiladas em ambas as versões. Os códigos não aceitos por uma determinada versão contêm uma nota de alerta nas primeiras linhas, na forma de comentário.

Com o objetivo de adicionar facilidades à linguagem, foram propostos dois padrões, o IEEE 1164 e IEEE 1076.3. O primeiro define o pacote " Std_logic_1164 ", e o segundo os pacotes " Numeric_std " e " Numeric_bit ". Um pacote em VHDL, ou " package ", é um local para armazenamento de informações de uso comum, como tipos de dados, funções etc. O pacote padrão da linguagem define vários tipos de dados, sendo o tipo " bit " o que possui correlação mais direta com um dado a ser manipulado por um circuito digital. Vetores, ou *arrays*, contendo elementos do tipo " bit " são também definidos. O tipo " bit " pode assumir apenas dois valores, " 0 " ou " 1 ", o que limita, em muitos casos, o modelamento. O padrão 1164 cobre esta lacuna, definindo novos tipos de dados que permitem modelar condições de alta impedância, e casos em que mais de uma porta aciona um mesmo nó. Os novos tipos de dados definidos modelam, também, situações normalmente encontradas na simulação de um circuito digital, como, por exemplo, um nó não-inicializado. O padrão 1076.3 visa auxiliar a síntese de circui-

tos, propondo novos tipos de dados e funções. Operações aritméticas, como soma e multiplicação, somente são definidas em tipos de dados inteiros ou reais. No padrão 1076.3 são definidas funções que realizam essas operações em tipos de dados no formato de vetores, dando, assim, mais opções para o projetista, tendo em vista que qualquer tipo de dado é convertido para um conjunto de bits no circuito implementado.

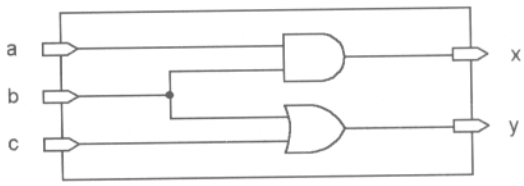
1.2 Aspectos gerais da linguagem

Como a linguagem VHDL suporta projetos com múltiplos níveis de hierarquia, a descrição pode consistir na interligação de outras descrições menores, a um código que representa o comportamento esperado do circuito. Esses estilos são denominados estrutural e comportamental, e podem ser mesclados em uma mesma descrição. A estrutura hierárquica e a opção de combinar diversos estilos de descrição facilitam a condução de projetos complexos que partem de um nível mais elevado para um nível mais baixo de especificação, conhecidos como *top down design*. As ferramentas de síntese suportam descrições de diferentes estilos, desde que o nível de abstração seja moderado. Para a inferência correta das estruturas, as construções empregadas devem seguir modos preferenciais indicados pelos fabricantes das ferramentas.

Com exceção de regiões específicas no código, todos os comandos são executados concorrentemente. Isto significa que a ordem na apresentação dos comandos é irrelevante para o comportamento da descrição. A ocorrência de um evento em um sinal leva à execução de todos os comandos sensíveis àquele sinal, da mesma forma que, em um circuito, a mudança de um valor em um determinado nó afeta todas as entradas ligadas a esse ponto do circuito.

A mudança de valor em " b " na Figura 1.2.1, por exemplo, leva à execução conjunta dos comandos contidos nas linhas 8 e 9. A ordem da avaliação dos comandos executada pelo simulador é irrelevante, e o resultado será sempre o mesmo. As operações internas executadas pela ferramenta de simulação, contudo, são efetuadas seqüencialmente. Assim, o simulador necessita de um mecanismo interno para armazenar o resultado de cada comando, até que a avaliação de todos os comandos envolvidos tenha sido realizada.

A linguagem permite, também, delimitar regiões de código seqüencial, onde a execução dos comandos segue a ordem de sua apresentação no código. Para essas regiões são utilizados comandos específicos, que não podem ser empregados na região de código concorrente. Subprogramas e processos são regiões de código seqüencial.



descrição VHDL

```
1  ENTITY portas IS
2    PORT (a, b, c : IN  BIT;
3          x, y    : OUT BIT);
4  END portas;
5
6  ARCHITECTURE teste OF portas IS
7  BEGIN
8    x <= a AND b;
9    y <= c OR  b;
10 END teste;
```

Figura 1.2.1 Concorrência em uma descrição.

Assim como as linguagens de programação, a linguagem VHDL permite a definição de subprogramas na forma de procedimentos e funções. Os subprogramas podem ser empregados em rotinas de conversão, definição de novos operadores e outras operações não diretamente ligadas a um circuito passível de síntese. Nada impede, entretanto, que um subprograma tenha uma correspondência direta a um elemento do circuito a ser sintetizado. O código de um circuito multiplicador composto por um conjunto de células, por exemplo, pode ter a função de cada célula descrita por um subprograma.

Um outro recurso é a definição de biblioteca e pacote, " library " e " package ". Os pacotes armazenam subprogramas, constantes ou novos tipos definidos, evitando a repetição de uma definição em todas as descrições. Excetuado o pacote padrão, ou " Standard Package ", todos os pacotes devem ser declarados para se tornarem visíveis numa declaração. As bibliotecas armazenam as informações compiladas, sendo a biblioteca corrente de trabalho denominada " work ". A linguagem fornece mecanismos de acesso a bibliotecas, permitindo que o trabalho de equipes envolvidas em um mesmo projeto seja armazenado numa área comum.

Além dos tipos estabelecidos no pacote padrão, é possível definir novos tipos. Conforme visto, o padrão 1164 contém tipos que modelam condições não encontradas nos tipos do pacote padrão, como de alta impedância e não-inicializado. A definição de novos tipos pode ser empregada, por exemplo, na representação dos estados

de uma máquina. Neste caso, pode ser criada uma correspondência mais natural entre o estado da máquina e a maneira como ele é representado no código. Naturalmente, no processo de síntese é atribuída uma codificação binária para cada estado.

O procedimento de teste de uma descrição pode ser comandado por um outro código VHDL. Estímulos de teste podem ser definidos para comparar a correspondência entre o circuito descrito e a especificação do projeto. Além disso, uma vez disponível a estimativa de temporização do circuito sintetizado, o código de teste pode verificar se restrições como *hold time* e *set up time* foram satisfeitas.

Operações com arquivos são possíveis. Essas operações são demasiadamente abstratas para serem empregadas na síntese; elas são empregadas em códigos que objetivam o teste de outras descrições.

Na linguagem VHDL não é feita nenhuma distinção entre comandos empregando caracteres maiúsculos ou minúsculos, como ocorre, por exemplo, na linguagem " C ". Os comentários iniciam após dois hifens seguidos, " -- ", e terminam no final da linha.

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

1.3 Síntese de circuitos

A linguagem VHDL não foi originalmente concebida para a síntese de circuitos digitais; assim, nem todas as construções definidas são suportadas pelas ferramentas de síntese. As limitações devem-se à falta de correspondência da construção com um circuito digital, à impossibilidade da síntese com precisão, ou à falta de detalhamento para uma síntese direta. Essas limitações não devem ser consideradas como um problema da ferramenta de síntese ou da linguagem, mas sim uma falha na própria descrição que está por demais afastada de um circuito real. Um projetista pode propor na descrição, por exemplo, um elemento de memória sensível tanto à borda de descida como à borda de subida de um sinal de relógio. Do ponto de vista de um simulador VHDL, a descrição pode ser simulada com sucesso; entretanto, a ferramenta de síntese poderá não sintetizar a descrição por falta de um elemento desse tipo no mundo físico. Uma multiplicação de dois valores reais é permitida pela linguagem; na prática, seria necessária uma unidade de multiplicação de ponto flutuante, que é, certamente, muito complexa. Um outro exemplo é a inclusão de uma cláusula de atraso em um sinal. Esse recurso é útil para o modelamento de um circuito real; contudo, se o funcionamento do circuito depende de um valor específico de tempo, a descrição não é sintetizada corretamente. A impossibilidade de gerar atrasos em um circuito digital sem o auxílio de uma base de tempo leva as ferramentas de síntese a ignorar esses comandos.

Os três passos em um projeto empregando uma linguagem de descrição de circuitos são apresentados na Figura 1.3.1. A partir da especificação de um projeto, é gerada uma descrição VHDL, que é submetida a um simulador para a verificação da correspondência entre a especificação e o código. A mesma descrição é interpretada por uma ferramenta de síntese que infere as estruturas necessárias para um circuito que corresponda à descrição. O

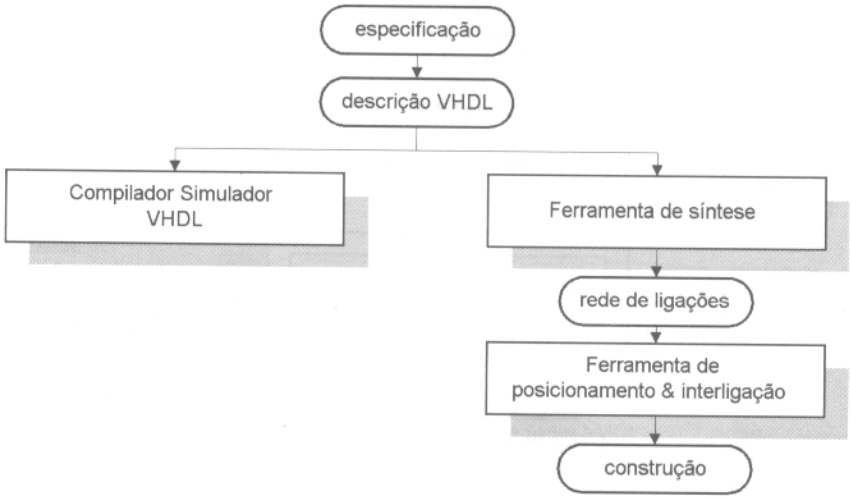


Figura 1.3.1 Etapas de um projeto.

resultado dessa etapa é um arquivo contendo uma rede de ligações de elementos básicos disponíveis na tecnologia do dispositivo empregado. Esse arquivo contendo a rede de ligações é a base de dados para a ferramenta que realiza o posicionamento e a interligação dos componentes, *place and route*. A saída da ferramenta de posicionamento e interligação é um arquivo que contém os dados necessários para confecção do dispositivo.

A etapa de elaboração da descrição é ilustrada na Figura 1.3.2. Como a linguagem VHDL permite descrever um mesmo circuito com diversos graus de abstração, o código gerado inicialmente pode conter estruturas por demais abstratas que não permitem a síntese direta. Um processo iterativo de simulações e detalhamento dos elementos da estrutura é executado até ser atingida uma descrição que permita a síntese, e até que as simulações assegurem a equivalência entre a especificação do projeto e a descrição proposta.

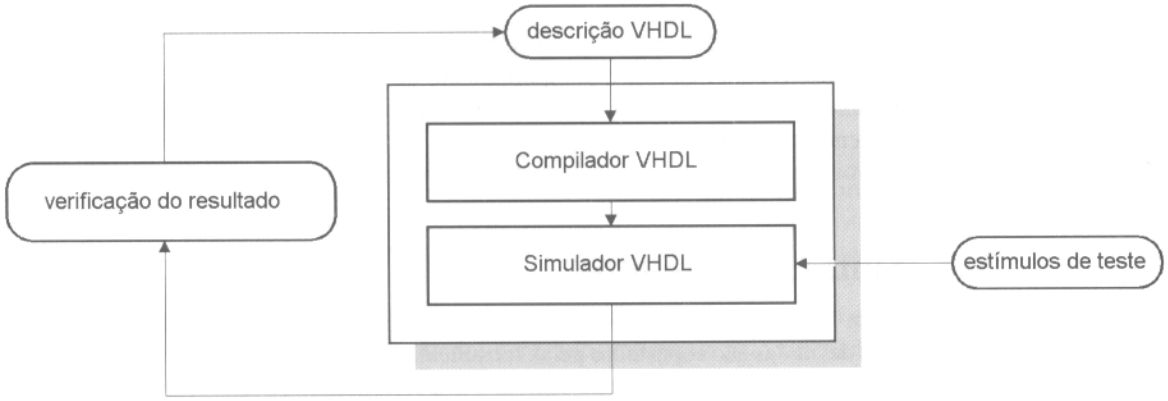


Figura 1.3.2 Elaboração da descrição VHDL.

Uma vez completada a descrição, dá-se início ao processo de síntese (ver Figura 1.3.3). A ferramenta de síntese, após a verificação de erros de sintaxe, executa o processo de inferência e interligação das estruturas necessárias para o circuito a ser gerado a partir da descrição. Nesta etapa, é gerado um circuito no nível RTL — *Register Transfer Level*, empregando primitivas disponíveis na ferramenta como comparadores, somadores, registradores e portas lógicas. O circuito gerado nessa etapa não está associado a nenhuma tecnologia de fabricação em particular, e não está, necessariamente, otimizado.

A próxima etapa do processo de síntese é a geração de um novo circuito a partir da estrutura no nível RTL. A diferença entre os dois circuitos consiste nos elementos utilizados, enquanto no nível RTL são empregadas primitivas genéricas da ferramenta; nessa etapa, o circuito contém unicamente elementos disponíveis na tecnologia empregada para a fabricação. Assim, é necessário especificar o dispositivo empregado para a realização

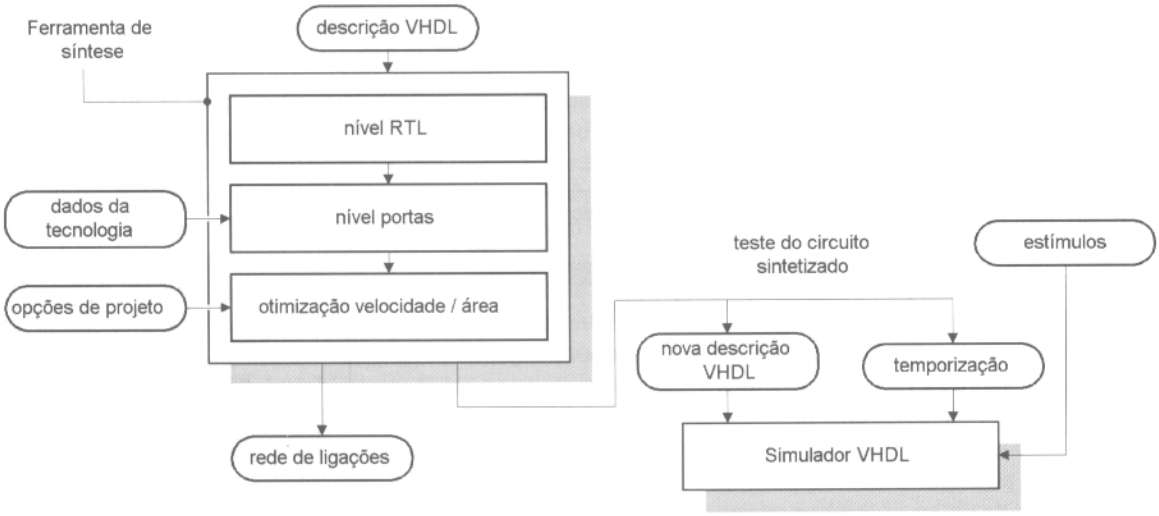


Figura 1.3.3 Síntese da descrição VHDL.

dessa etapa. Muitas vezes a ferramenta pode lançar mão de informações adicionais, fornecidas pelo fabricante do dispositivo, que levam a síntese mais eficiente de módulos específicos para realizar operações aritméticas, contadores etc. Uma minimização é executada nessa etapa e considera dois parâmetros geralmente conflitantes: otimização de custo e otimização para velocidade.

Como um dos resultados dessa etapa, tem-se um arquivo contendo uma rede de ligações entre os elementos disponíveis na tecnologia empregada. O formato do arquivo depende da ferramenta de posicionamento e interligação empregada; um exemplo é o formato EDIF — *Electronic Design Interchange Format*.

As ferramentas de síntese podem fornecer, ainda, um arquivo VHDL descrevendo a rede de ligações sintetizadas, e um outro arquivo contendo informações referentes à temporização considerando a tecnologia de fabricação empregada. Esses arquivos podem ser submetidos novamente a um simulador VHDL excitado pelo mesmo conjunto de estímulos para verificação do comportamento da estrutura sintetizada com relação à temporização.

A temporização estimada nessa etapa pode conter diferenças de até 20% com relação ao circuito real, pois os atrasos referentes às interligações não podem ser ainda determinados com precisão. O caminho correto das ligações somente é determinado na próxima etapa.

A Figura 1.3.4 ilustra as etapas e um processo de síntese partindo de uma descrição VHDL que executa a soma de valores entre zero e sete. Com base na descrição, é inferida a necessidade de um bloco somador com três bits. A ferramenta lança mão da primitiva no nível RTL que atenda aos requisitos — no caso, um bloco somador de três bits. Na segunda etapa, a ferramenta elabora a primitiva no nível RTL com os elementos disponíveis na tecnologia escolhida.

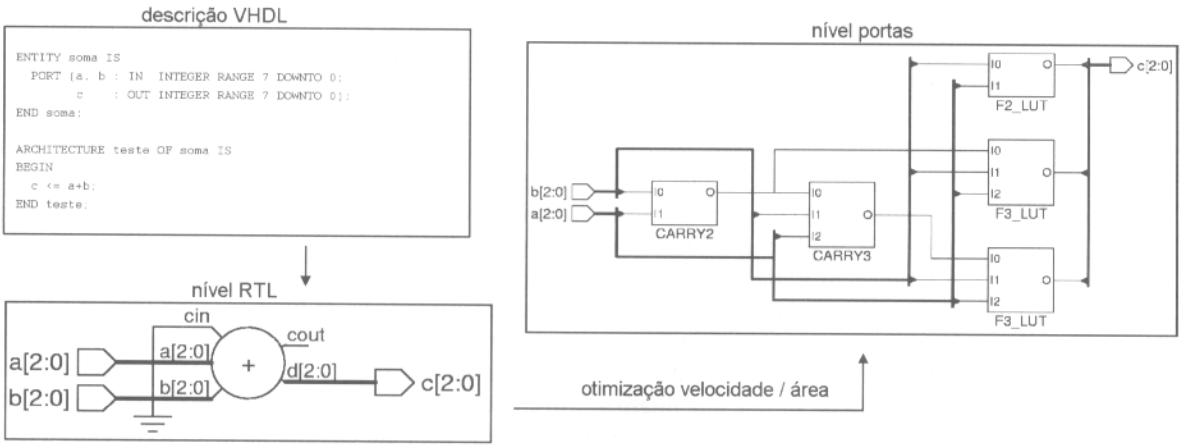


Figura 1.3.4 Exemplo dos passos executados por uma ferramenta de síntese.

Com base na rede de ligações gerada pela ferramenta de síntese, a ferramenta que realiza o posicionamento e interligação dos componentes, *place and route*, assenta cada primitiva em um local do dispositivo empregado e define o caminho para interligação com as demais primitivas (ver Figura 1.3.5). No caso de dispositivos lógicos programáveis, as ferramentas de posicionamento e interligação são, normalmente, fornecidas pelos próprios fabricantes devido às particularidades de cada tecnologia. Circuitos integrados de aplicação específica não pos-

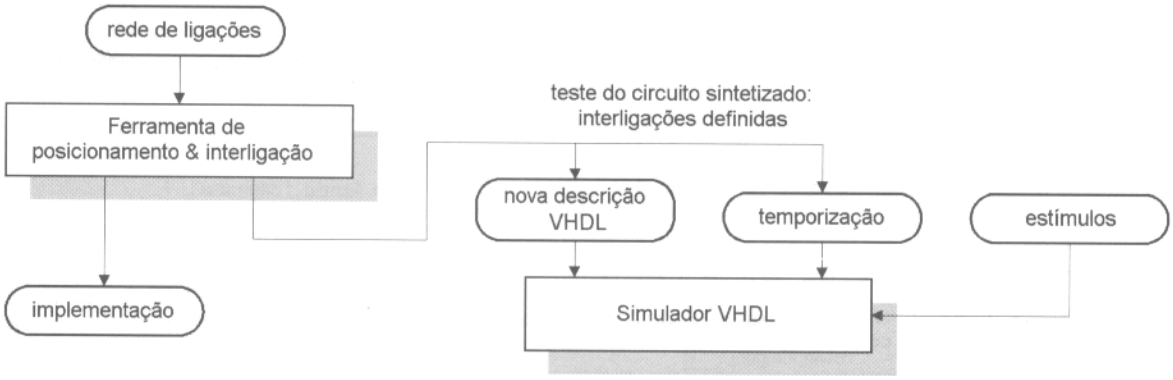


Figura 1.3.5 Etapa final do projeto.

suem grandes variações de arquitetura, permitindo o uso de ferramentas de uso geral para esse fim. Informações referentes a temporização e descrições VHDL da rede de ligações podem, também, ser obtidas. A temporização estimada nessa etapa é mais precisa, pois a ferramenta dispõe de informações sobre o caminho das interligações e, conseqüentemente, o atraso gerado por elas.

As etapas descritas podem variar, ou mesmo ser omitidas, para diferentes ferramentas de síntese. Fabricantes de dispositivos lógicos programáveis fornecem ferramentas que integram várias das etapas, tornando mais simples o processo de síntese. Essas ferramentas, contudo, possuem limitações quanto às estruturas VHDL suportadas, e apenas comandos voltados para síntese são permitidos. Nesses casos, operações de teste e manipulações de arquivos não empregam um código VHDL; elas são executadas com operações específicas dos aplicativos.

Os modelos de temporização fornecidos pelos fabricantes de FPGAs — *Field Programmable Logic Devices*, e ASICs — *Application Specific Integrated Circuits*, podem seguir o padrão IEEE 1076.4 estabelecido no ano de 1995. Esse padrão, conhecido também por VITAL, é um pacote que contém procedimentos e funções com o objetivo de modelar atrasos com precisão, verificar a temporização e notificar violações da temporização.

As ferramentas de síntese e as ferramentas de posicionamento e interligação podem gerar as informações de temporização segundo o formato SDF — *Standard Delay Format*. Esse arquivo, em conjunto com a descrição VHDL contendo a rede de ligações, fornece as informações para um simulador VHDL para verificação da temporização, conforme ilustrado nas Figuras 1.3.3 e 1.3.5.

1.4 Principais pontos abordados

- O desenvolvimento da linguagem VHDL foi motivado pela necessidade de um padrão para o intercâmbio de informações entre diversos fornecedores de equipamentos para o Departamento de Defesa dos Estados Unidos da América.
- VHDL é uma linguagem concorrente. Os comandos envolvidos na ocorrência de um evento são executados simultaneamente, assim como os elementos de um sistema digital executam as suas tarefas conjuntamente.
- Uma descrição pode conter diversos níveis de abstração, variando da interligação de componentes a uma representação do comportamento do sistema. Os diversos níveis de abstração podem ser inseridos numa mesma descrição, sendo permitidos, também, projetos hierárquicos.
- A linguagem VHDL não foi concebida originalmente para a síntese, mas sim para a descrição de circuitos digitais. Desse modo, nem todas as construções possíveis da linguagem podem ser utilizadas para a síntese.
- Um processo de síntese inicia a partir de uma descrição que atenda a especificação de um projeto, e que tenha sido validada através de um conjunto de estímulos de teste. Nessa etapa, a descrição é submetida a uma ferramenta de síntese que infere as estruturas necessárias para atender a especificação do projeto.
- No caso mais geral, uma ferramenta de síntese gera um circuito no nível RTL, onde a tecnologia de fabricação não é definida. Em seguida é feita a transposição da estrutura em nível RTL para a tecnologia-alvo. Nessa etapa, uma estimativa de temporização pode ser fornecida.
- O processo de síntese é finalizado pela ferramenta que realiza o posicionamento e a interligação dos componentes, *place and route,* com base na rede de ligações gerada pela ferramenta de síntese.

1.5 Exercícios

1.5.1 Leia as instruções do compilador e simulador VHDL empregado, para uma familiarização com a operação da ferramenta.

O código do Quadro 1.5.1 descreve o comportamento de uma porta "ou-exclusivo". Os estímulos podem ser aplicados em " a " e " b ", e a saída pode ser observada em " c ". Compile a descrição e aplique os estímulos que julgar necessários para o teste da descrição. O objetivo do exercício é treinar a operação da ferramenta de compilação e simulação.

```
1  ENTITY e_xor IS
2    PORT (a, b : IN  BIT;  -- entradas, valores 0 ou 1
3          c    : OUT BIT); -- saida, valores 0 ou 1
4  END e_xor;
5
6  ARCHITECTURE teste OF e_xor IS
7  BEGIN
8    c <= (a AND (NOT b)) OR ((NOT a) AND b);
9  END teste;
```

Quadro 1.5.1 Descrição do Exercício 1.5.1.

1.5.2 Leia as instruções da ferramenta de síntese empregada, para uma familiarização com sua operação.

Sintetize a descrição proposta no Quadro 1.5.1 e observe os esquemas criados no nível RTL e no nível de portas lógicas. Verifique as diferenças entre os esquemas e explore, também, a diferença entre os resultados obtidos para diversas arquiteturas de implementação.

1.5.3 No Quadro 1.5.2 é apresentada a descrição de um somador. Compile e simule a descrição com alguns estímulos. Observe que " a ", " b " e " c " são do tipo inteiro. Os valores possíveis de serem aplicados nessas interfaces estão restritos ao intervalo de -2^{31} a $2^{31}-1$. Qual o valor inicial apresentado pelo simulador para " a ", " b " e " c "? Existe uma relação entre esse valor e a faixa de valores permitida para o tipo inteiro?

```
1  ENTITY soma IS
2    PORT (a, b : IN  INTEGER;  -- entradas, valores entre -2**31 a 2**31 -1
3          c    : OUT INTEGER); -- saida , valores entre -2**31 a 2**31 -1
4  END soma;
5
6  ARCHITECTURE teste OF soma IS
7  BEGIN
8    c <= a + b;
9  END teste;
```

Quadro 1.5.2 Descrição do Exercício 1.5.2.

1.5.4 Sintetize a descrição proposta no Quadro 1.5.2 e observe os esquemas criados no nível RTL e no nível de portas lógicas. Verifique se o número de bits empregados atende a faixa de valores que as entradas podem assumir. Observe e comente, também, a diferença de complexidade entre esquemas gerados.

1.5.5 A Figura 1.5.1 contém o diagrama de blocos de um circuito composto de contadores, decodificadores e mostradores de sete segmentos. Os contadores são controlados por uma base de tempo, ou relógio, denominada " ck ", e a entrada " reset_b " leva o valor da saída dos contadores a zero na condição " reset_b=0 ". O contador " D " realiza uma divisão no sinal de relógio segundo o valor do genérico " div_por ". Supondo a freqüência de " ck " igual a 25,175MHz e a constante de divisão igual a $25{,}175 \times 10^6$, o contador " L " incrementa a sua contagem a cada 1 segundo. De modo semelhante, o contador " L " habilita a contagem do contador " H " ao final de uma contagem dada pelos genéricos " lmax ". Para " hmax=5 " e " lmax=9 ", o circuito implementa um contador de segundos. No Quadro 1.5.3, é apresentada a descrição VHDL equivalente ao diagrama proposto na figura.

Abra o arquivo contendo a descrição e altere o valor da constante " div_por " na linha 23. Um valor entre 3 e 7 é adequado para simular a descrição. Salve o arquivo, compile e simule o código, verificando a operação da descrição. Retorne o valor da constante para o valor anterior e sintetize a descrição. Observe o esquema do circuito sintetizado, e verifique os blocos empregados no nível RTL. Três contadores devem ser inferidos pela ferramenta de síntese.

A última etapa é o teste e a programação da *FPGA*. Caso disponha de uma placa de teste, programe o dispositivo e verifique a operação.

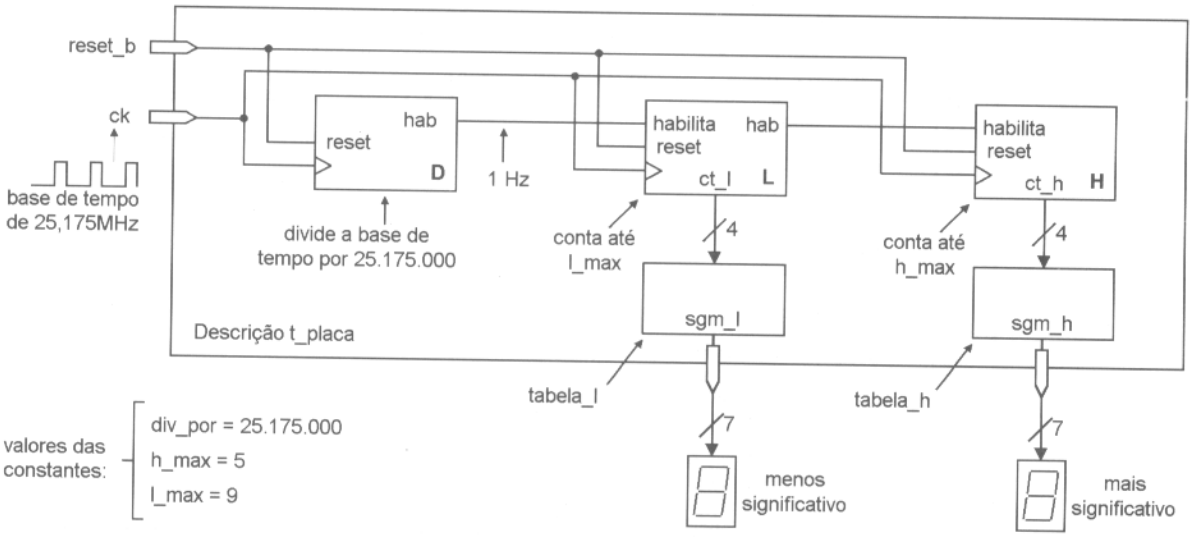


Figura 1.5.1 Esquema de um cronômetro digital.

```
 1  -- Exemplo
 2  -- segmentos do mostrador ativo nivel baixo   (ver Tabela Ap.2.1)
 3  -- segmento  pino_mostrador_h  pino_mostrador_l
 4  --  a = 0          6                17            a
 5  --  b = 1          7                18           ----
 6  --  c = 2          8                19        f|    |b
 7  --  d = 3          9                20         |  g |
 8  --  e = 4         11                21           ----
 9  --  f = 5         12                23        e|    |c
10  --  g = 6         13                24         |  d |
11  --  ponto = 7     14                25           ---- . ponto
12
13  --  oscilador de 25.175MHz  (ver Tabela Ap.2.2)
14  --  pino 91
15
16  -- chaves de contato momentaneo  (ver Tabela Ap.2.3)
17  -- flex_pb1  pino 28
18  -- flex_pb2  pino 29
19
20  -- chaves dip-switch 0-1   (ver Tabela Ap.2.4)
21  -- flex_sw1  1  2  3  4  5  6  7  8
22  -- pino     41 40 39 38 36 35 34 33
23
24  ENTITY t_placa2 IS
25    GENERIC (max_h        : INTEGER  := 5;            -- valor maximo mostrador H
26             max_l        : INTEGER  := 9;            -- valor maximo  mostrador L
27             div_por      : INTEGER  := 16777216);    -- divide 25.175MHz by 2**24
28    PORT   (ck            : IN  BIT;                     -- base de tempo (relogio)
29            reset_b        : IN  BIT;                     -- reset_b=0 inicializa maquina
30            seg_h          : OUT BIT_VECTOR (0 TO 7);     -- mostrador H
31            seg_l          : OUT BIT_VECTOR (0 TO 7));    -- mostrador L
32  END t_placa2;
33
34  ARCHITECTURE bit_type OF t_placa2 IS
35    SIGNAL ct_h        : INTEGER RANGE    max_h  DOWNTO 0;       -- contador mostrador_h
36    SIGNAL ct_l        : INTEGER RANGE    max_l  DOWNTO 0;       -- contador mostrador_l
37    SIGNAL dv          : INTEGER RANGE    div_por DOWNTO 0;      -- base de tempo interna
38  --                          segmento mostrador:  abcdefgh     -- '0' ativa o segmento
39    CONSTANT ds0       : BIT_VECTOR (7 DOWNTO 0) := "00000011";  -- codigo numero 0
40    CONSTANT ds1       : BIT_VECTOR (7 DOWNTO 0) := "10011111";  -- codigo numero 1
41    CONSTANT ds2       : BIT_VECTOR (7 DOWNTO 0) := "00100101";  -- codigo numero 2
42    CONSTANT ds3       : BIT_VECTOR (7 DOWNTO 0) := "00001101";  -- codigo numero 3
43    CONSTANT ds4       : BIT_VECTOR (7 DOWNTO 0) := "10011001";  -- codigo numero 4
44    CONSTANT ds5       : BIT_VECTOR (7 DOWNTO 0) := "01001001";  -- codigo numero 5
45    CONSTANT ds6       : BIT_VECTOR (7 DOWNTO 0) := "01000001";  -- codigo numero 6
46    CONSTANT ds7       : BIT_VECTOR (7 DOWNTO 0) := "00011111";  -- codigo numero 7
47    CONSTANT ds8       : BIT_VECTOR (7 DOWNTO 0) := "00000001";  -- codigo numero 8
48    CONSTANT ds9       : BIT_VECTOR (7 DOWNTO 0) := "00001001";  -- codigo numero 9
49    CONSTANT dsn       : BIT_VECTOR (7 DOWNTO 0) := "11111111";  -- apagado
50
51  BEGIN
52  count: PROCESS (ck, reset_b)
53    BEGIN
54      IF (reset_b ='0') THEN            -- reset assincrono
55          ct_h <= 0;
56          ct_l <= 0;
57          dv <= 0;
58      ELSIF (ck'EVENT) AND (ck = '1') THEN -- base de tempo externa
59        IF (dv = div_por -1) THEN          -- teste fim da divisao do sinal ck
60          dv <= 0;                         -- zera contador incrementa dv
61          IF (ct_l = max_l) THEN           -- teste fim da contagem digit_l
62            ct_l <= 0;                     -- zera contador ct_l caso ct_l = max_l
63            IF (ct_h = max_h) THEN ct_h <= 0;       -- teste fim da contagem digit_h
64            ELSE                   ct_h <= ct_h +1; -- continua contagem digit_h
65            END IF;
66          ELSE   ct_l <= ct_l +1;                   -- continua contagem digit_l
67          END IF;
68        ELSE                                -- continua divisao do sinal ck
69          dv <= dv +1;
70        END IF; -- div_por
71      END IF;
72    END PROCESS count;
73
74  table_h: BLOCK
75    BEGIN
76      seg_h <= dsn WHEN ct_h = 0 ELSE   -- mostrador mais significativo apagado quando '0'
77               ds1 WHEN ct_h = 1 ELSE
78               ds2 WHEN ct_h = 2 ELSE
79               ds3 WHEN ct_h = 3 ELSE
80               ds4 WHEN ct_h = 4 ELSE
81               ds5 WHEN ct_h = 5 ELSE
82               ds6 WHEN ct_h = 6 ELSE
83               ds7 WHEN ct_h = 7 ELSE
84               ds8 WHEN ct_h = 8 ELSE
```

Quadro 1.5.3 Descrição equivalente ao diagrama de blocos proposto na Figura 1.5.1 (continua).

```
 85                ds9;
 86   END BLOCK table_h;
 87
 88 table_l: BLOCK
 89   BEGIN
 90     seg_l <= ds0 WHEN ct_l = 0 ELSE
 91              ds1 WHEN ct_l = 1 ELSE
 92              ds2 WHEN ct_l = 2 ELSE
 93              ds3 WHEN ct_l = 3 ELSE
 94              ds4 WHEN ct_l = 4 ELSE
 95              ds5 WHEN ct_l = 5 ELSE
 96              ds6 WHEN ct_l = 6 ELSE
 97              ds7 WHEN ct_l = 7 ELSE
 98              ds8 WHEN ct_l = 8 ELSE
 99              ds9;
100   END BLOCK table_l;
101 END bit_type;
```

Quadro 1.5.3 Descrição equivalente ao diagrama de blocos proposto na Figura 1.5.1.

CAPÍTULO

2 PRIMEIRO CONTATO COM A LINGUAGEM

O tópico inicial do capítulo é a entidade de projeto por ser, conforme o manual de referência da linguagem, a abstração primária de um circuito VHDL. Em seguida são vistos os conceitos de entidade, arquitetura, portas e modos das portas. Neste capítulo são apresentados, também, as classes de objeto existentes e os tipos e operadores predefinidos.

2.1 Entidade de projeto

Uma entidade de projeto, ou *design entity,* pode representar desde uma simples porta lógica a um sistema completo, e é composta de duas partes: declaração da entidade e arquitetura. A declaração da entidade define a interface entre a entidade e o ambiente exterior, como, por exemplo, entradas e saídas. A arquitetura contém a especificação das relações entre as entradas e saídas de uma entidade (ver Figura 2.1.1).

Uma analogia normalmente feita com um projeto em captura esquemática, é comparar a declaração da entidade ao símbolo de um bloco, e a arquitetura ao esquema contido no bloco.

A simulação de uma entidade de projeto somente pode ser executada após a sua compilação, e o resultado é armazenado numa biblioteca, *library*. Caso não seja especificada uma biblioteca especial, o resultado da compilação é armazenado na biblioteca de trabalho denominada " Work ". Essa biblioteca é, normalmente, um diretório criado pela ferramenta de compilação e simulação empregada, contendo informações específicas do aplicativo.

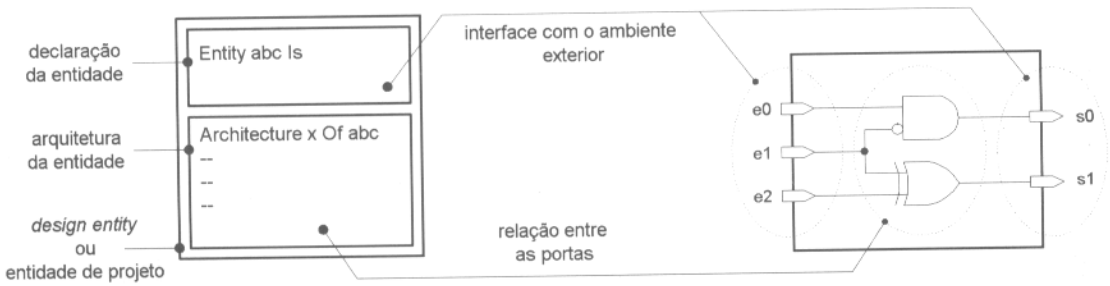


Figura 2.1.1 Exemplo de entidade de projeto.

Uma arquitetura pode ser descrita por meio de uma interligação de componentes, de forma a estabelecer uma hierarquia no projeto (ver Figura 2.1.2). Cada componente deve ser declarado, sendo ele, também, uma entidade de projeto. Descrições que empregam esse estilo são denominadas estruturais, e a complexidade dessas descrições pode variar de um simples registrador, a um processador contendo a interligação de blocos complexos como memórias, unidades lógicas aritméticas etc. Não existe limite para o nível da hierarquia, e a descrição não precisa conter unicamente a interligação de componentes, é possível mesclar diferentes estilos de descrições em um mesmo código.

Figura 2.1.2 Exemplo de entidade de projeto.

Declaração da entidade e corpo da arquitetura

O Quadro 2.1.1 apresenta um exemplo de uma declaração para uma entidade. A declaração inicia com a palavra reservada " ENTITY " seguida do nome que a identifica. A cláusula " PORT " é empregada para definir o modo e tipo das portas de entrada e saída da descrição. A cláusula " GENERIC " permite a passagem de informações estáticas para uma unidade de projeto. Internamente o parâmetro é tratado como uma constante.

O modo de uma porta determina a direção e como os comandos na arquitetura podem operar com as portas. A Figura 2.1.3 ilustra os quatro modos possíveis de uma porta.

São eles:

- " IN ";
- " OUT ";
- " BUFFER ";
- " INOUT ".

Nos modos " IN " e " OUT ", há portas que operam exclusivamente como entrada e saída. No modo " BUFFER ", a porta opera unicamente no modo saída, diferenciando do modo " OUT " porque o valor apresentado pode ser referenciado internamente pela arquitetura. Uma porta do modo " OUT " não pode, por exemplo, controlar um sinal interno da entidade; no caso de uma porta do tipo " BUFFER ", isso é permitido. O modo " INOUT "

```
ENTITY entidade_abc IS

  GENERIC (n        : INTEGER := 5);

  PORT (x0, x1      : IN        tipo_a;     -- entradas
        y0, y1      : OUT       tipo_b;     -- saidas
        y2          : BUFFER    tipo_c;     -- saida
        z0, z1      : INOUT     tipo_d);    -- entrada / saida

END entidade_abc;
```

Quadro 2.1.1 Exemplo de uma declaração para uma entidade.

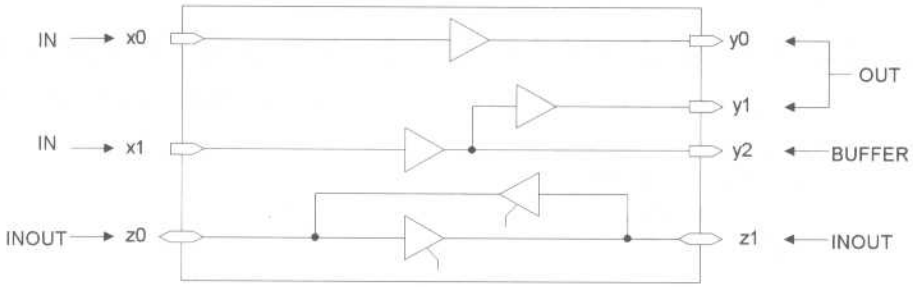


Figura 2.1.3 Quatro modos de portas: " IN ", " OUT ", " BUFFER ", e " INOUT ".

caracteriza uma porta bidirecional onde uma informação pode ser apresentada ou amostrada. Essas portas são adequadas para o modelamento de barramentos de dados, onde a informação pode transitar em ambos os sentidos. O tipo define as características do valor transferido pela porta, e o conjunto de tipos predefinidos pela linguagem é abordado no item 2.3.

O corpo da arquitetura inicia-se com a palavra reservada " ARCHITECTURE " seguida de um nome identificador e o nome da declaração de entidade associada. As linhas que seguem podem conter declarações de sinais e constantes, declarações que identificam os componentes externos utilizados pela arquitetura, bem como a descrição completa de subprogramas que são utilizados localmente pela arquitetura. Os comandos concorrentes que descrevem a entidade, propriamente, ficam entre as palavras reservadas " BEGIN " e " END ".

O nome identificador da arquitetura permite a existência de mais de uma arquitetura para uma mesma declaração de entidade. No exemplo da Figura 2.1.4, o conjunto formado pela arquitetura " prt_e " e a declaração de entidade " comb " representam uma entidade. Do mesmo modo, o conjunto formado pela arquitetura " prt_ou " e a declaração de entidade " comb " representam uma outra entidade. Assim, cada par declaração-arquitetura representa uma entidade distinta com a mesma interface. O emprego de mais de uma arquitetura para uma mesma declaração de entidade é abordado, com mais detalhes, no Capítulo 14.

```
ARCHITECTURE nome_identificador OF entidade_abc IS
  --
  -- regiao de declaracoes:
  --   declaracoes de sinais e constantes
  --   declaracoes de componentes referenciados
  --   declaracao e corpo de sub-programas
  --   definicao de novos tipos de dados locais
  --
BEGIN
  --
  -- comandos concorrentes
  --
END;
```

Quadro 2.1.2 Corpo da arquitetura.

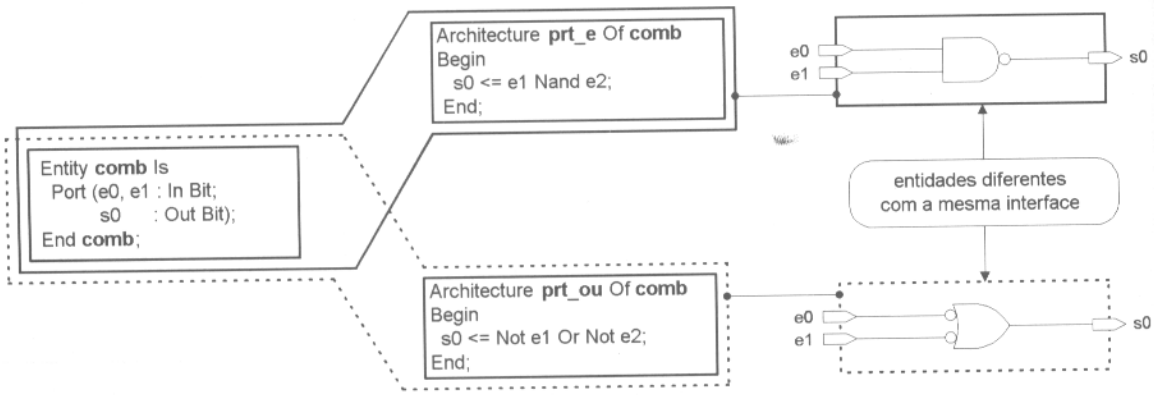


Figura 2.1.4 Duas arquiteturas e uma única declaração de entidade.

2.2 Classe de objetos: constante, variável, sinal e arquivo

Objetos são elementos que contêm um valor armazenado. Quatro classes de objetos são definidas em VHDL:

- constante ou " CONSTANT ";
- variável ou " VARIABLE ";
- sinal ou " SIGNAL ";
- arquivo ou " FILE ".[1]

[1]Na versão VHDL-1987 um objeto arquivo é um membro da classe variável.

A constante é um objeto com um valor estático. No caso da variável, o valor inicial imposto pode ser alterado no decorrer do código, e é empregada em regiões de código seqüencial. Sinais são objetos que podem ter o seu valor alterado, e são empregados em regiões de código concorrente e seqüencial. Os objetos da classe " FILE " estão associados à criação de arquivos, e são abordados no Capítulo 13.

O Quadro 2.2.1 contém exemplos de declaração de objetos das classes " CONSTANT ", " VARIABLE " e " SIGNAL ". A sintaxe da declaração é semelhante para as três classes. Elas iniciam com uma palavra reservada que define a classe do objeto, seguida de uma lista de nomes que identificam os objetos. Após a lista, é declarado o tipo dos objetos. Opcionalmente, pode ser definido um valor inicial para os objetos através do delimitador " := ".

Os objetos são declarados nas regiões de declaração de arquiteturas, blocos, processos, subprogramas e pacotes. Restrições se aplicam, com relação às regiões permitidas, a determinadas classes de objeto, assim como à visibilidade atingida pelo objeto declarado. Conforme ilustrado no Quadro 2.1.2, na região de declarações de uma arquitetura podem ser declarados sinais e constantes. Os objetos ali declarados são visíveis somente naquela arquitetura. Note que a porta de uma entidade realiza uma declaração de um sinal.

```
-- classe   lista de nomes        tipo      valor inicial

   CONSTANT nome_da_constante_a : tipo_x;                      -- constante sem valor inicial
   CONSTANT nome_da_constante_a : tipo_x := valor_inicial;
   CONSTANT fixo_1, fixo_2      : tipo_x := valor_inicial;

   VARIABLE nome_da_variavel_c  : tipo_z;                      -- variavel sem valor inicial
   VARIABLE nome_da_variavel_d  : tipo_z := valor_inicial;     -- variavel com valor inicial
   VARIABLE var_1, var_2        : tipo_z := valor_inicial;     -- variaveis: mesmo tipo e valor

   SIGNAL   nome_do_sinal_a        : tipo_y;                   -- sinal sem valor inicial
   SIGNAL   nome_do_sinal_b        : tipo_y := valor_inicial;  -- sinal com valor inicial
   SIGNAL   nome_x, nome_y, nome_z : tipo_y;                   -- sinais do mesmo tipo
```

Quadro 2.2.1 Declaração de objetos da classe constante, variável e sinal.

No Quadro 2.2.2 são apresentados alguns exemplos de transferência de informação entre objetos das classes " SIGNAL ", " VARIABLE " e " CONSTANT ". Observe que a atribuição de valor para uma variável emprega o delimitador " := " , enquanto a atribuição para um sinal emprega o delimitador " <= ".

```
sinal_2    <= sinal_1;        -- atribuicao valor para sinal
sinal_3    <= variavel_1;     -- atribuicao valor para sinal
sinal_4    <= constante_1;    -- atribuicao valor para sinal

variavel_2 := sinal_1;        -- atribuicao valor para variavel
variavel_3 := variavel_1;     -- atribuicao valor para variavel
variavel_4 := constante_1;    -- atribuicao valor para variavel
```

Quadro 2.2.2 Exemplos de transferência de informação entre objetos.

O Quadro 2.2.3 apresenta uma descrição completa de uma entidade. Na declaração da entidade são definidas três portas, uma de entrada e duas de saída. Todas as portas são do tipo inteiro, " INTEGER ". A arquitetura, denominada " teste ", inicia na linha 6, e são declarados o sinal " s1 " e a constante " c1 ". As operações executadas são simples, o valor presente na entrada " x1 " é transferido para o sinal interno " s1 ", que é transferido, por sua vez, para a saída " y1 ". Em outras palavras, a porta de saída " y1 " segue a porta de entrada " x1 ". Por sua vez, a porta de saída " z1 " recebe o valor da constante " c1 ", mantendo sempre o valor " 7 " (ver Figura 2.1.5).

Note a seqüência dos comandos nas linhas 10 e 11. Em uma primeira análise, tem-se a impressão de que o código apresenta problemas, pois a porta de saída " y1 " recebe o valor de " s1 " na linha 10, e o valor da porta de entrada " x1 " é transferido para o sinal " s1 " numa linha posterior. Entretanto, como esses comandos estão em uma região de código concorrente, a ordem como as linhas são apresentadas é irrelevante. Qualquer alteração de valor em um sinal acarreta a execução de todos os comandos envolvidos com esse sinal. Essa alteração tem como conseqüência a alteração de valores em outros sinais, e a operação perdura até não ocorrerem mais alterações.

```
1  ENTITY atrib_1 IS
2    PORT (x1      : IN  INTEGER;
3          y1,z1   : OUT INTEGER);
4  END;
5
6  ARCHITECTURE teste OF atrib_1 IS
7    SIGNAL   s1 : INTEGER;
8    CONSTANT c1 : INTEGER := 7;
9  BEGIN
10   y1 <= s1;
11   s1 <= x1;
12
13   z1 <= c1;
14 END teste;
```

Quadro 2.2.3 Exemplo de uma descrição completa.

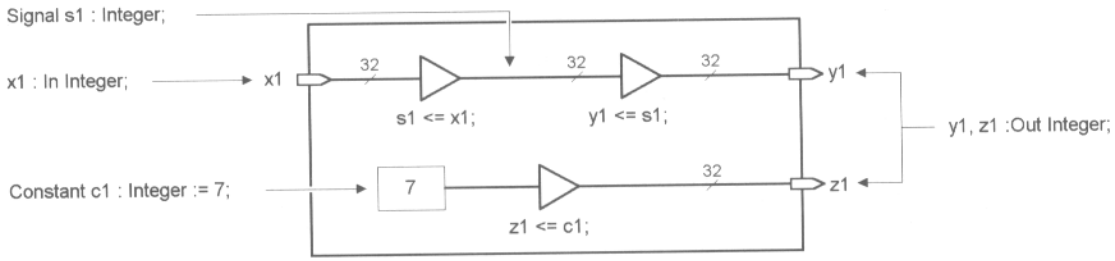


Figura 2.1.5 Representação esquemática do código do Quadro 2.2.3.

2.3 Tipos

Conforme o manual de referência da linguagem, um tipo é caracterizado por um conjunto de valores e um conjunto de operações. Os objetos " CONSTANT ", " VARIABLE " e " SIGNAL " devem ser declarados segundo um tipo definido, para que fiquem estabelecidos os valores que o objeto pode assumir e as operações que podem ser realizadas.

Na Figura 2.3.1 são ilustrados os tipos disponíveis pela linguagem abordados neste capítulo. Na parte inferior da figura são apresentados alguns dos tipos predefinidos empregados na declaração de objetos.

Os tipos escalares consistem nos tipos: enumerado, inteiro, ponto flutuante e físico. Os tipos enumerado e inteiro podem ser designados como tipos discretos, e os tipos inteiro, ponto flutuante e físico podem ser designados como numéricos. Com relação aos tipos compostos, este capítulo se restringe, apenas, ao tipo " Array ".

Não é permitida a transferência de valores entre objetos declarados com tipos diferentes. Caso seja necessário realizar uma troca de valores entre objetos declarados com tipos distintos, é preciso uma operação de conversão.

A linguagem possibilita a definição de subtipos. Um subtipo delimita uma faixa de valores dentro da gama de valores do tipo. Como um subtipo é um subconjunto dos valores de um tipo, é possível a troca de valores entre objetos assim declarados.

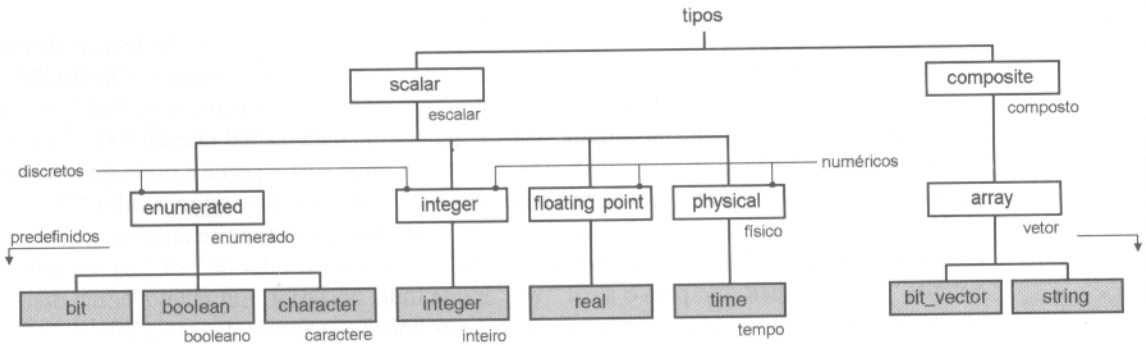


Figura 2.3.1 Tipos VHDL abordados neste capítulo.

Tipos escalares

A Tabela 2.3.1 apresenta tipos escalares predefinidos no pacote padrão. Tipos escalares são ordenados e representam um valor único, permitindo, assim, a aplicação de operadores como maior, menor etc.

O tipo " BIT " é utilizado para representar os níveis lógicos " 0 " e " 1 " de uma linha digital. O tipo " BOOLEAN " pode assumir os valores " FALSE " ou " TRUE ", sendo empregado em comandos que executam um teste ou decisão. O tipo " CHARACTER " corresponde a qualquer caractere do padrão ASCII na versão VHDL-1987, ou do padrão ISO8859-1 na versão VHDL-1993.

O tipo " INTEGER " representa um número inteiro dentre os limites: -2.147.483.647 a 2.147.483.647. Em uma descrição, caso não seja definida na declaração do objeto a faixa de valores que o objeto pode assumir, a ferramenta de síntese pode utilizar linhas de 32 bits para a representação do objeto. Por esse motivo, deve-se sempre estabelecer o limite de valores esperado para um uso adequado de recursos no circuito sintetizado. No pacote padrão são definidos dois subtipos do tipo " INTEGER ": os subtipos " NATURAL " e " POSITIVE ".

Tabela 2.3.1 Tipos escalares predefinidos no pacote padrão.

tipo predefinido	valor	exemplos
BIT	um, zero	1, 0
BOOLEAN	verdadeiro, falso	TRUE, FALSE
CHARACTER	caracteres ASCII	a, b, c, A, B, C, ?, (
INTEGER	$-2^{31}-1 \leq x \leq 2^{31}-1$	123, 8#173#, 16#7B#, 2#11_11_011#
NATURAL	$0 \leq x \leq 2^{31}-1$	123, 8#173#, 16#7B#
POSITIVE	$1 \leq x \leq 2^{31}-1$	2#11_11_011#
REAL	$-3.65\times10^{47} \leq x \leq +3.65\times10^{47}$	1.23, 1.23E+2, 16#7.B#E+1
TIME	ps = 10^3 fs ns = 10^3 ps us = 10^3 ns ms = 10^3 us sec = 10^3 ms min =60 sec hr = 60 min	1 us, 100 ps, 1 fs

Nota 1 " NATURAL " e " POSITIVE " são subtipos " INTEGER ".

Nota 2 Versão VHDL-1993, valores para tipos " CHARACTER " estendido para o conjunto ISO8859-1.

A representação de números no formato ponto flutuante utiliza o tipo predefinido " REAL ". Um valor de um tipo real pode ser distinguido de um valor do tipo inteiro pela presença de um ponto decimal. Deve-se notar que o tipo " REAL " não é, normalmente, suportado pelas ferramentas de síntese devido à complexidade das estruturas envolvidas para realização de operações.

O tipo " TIME " é um tipo físico definido no pacote padrão, empregado para representar quantidades físicas, onde são estabelecidas uma unidade base e sucessivas unidades múltiplas desta. No caso do tipo " TIME ", a unidade base é " fs ", ou 10^{-15}s. Os tipos físicos não são suportados pelas ferramentas de síntese, no caso do tipo " TIME ", os intervalos de tempo definidos são ignorados.

Constantes numéricas podem ser definidas em qualquer base entre 2 e 16, seguindo o formato " base#valor# ". Por exemplo, o valor " 123 " de um tipo inteiro pode ser representado, na base 10, na forma " 123 ", e, na base 16, na forma " 16#7B# ". A inclusão do caractere " _ " entre algarismos é permitida, e tem o objetivo de facilitar a leitura de um número na descrição. Assim, o valor do exemplo anterior na base 2 pode, por exemplo, ser apresentada na forma " 2#111_1011# ".

Tipos compostos

No pacote padrão VHDL são definidos dois tipos " ARRAY ":[2] o tipo " BIT_VECTOR " contendo elementos do tipo " BIT ", e o tipo " STRING " contendo elementos do tipo " CHARACTER " (ver Tabela 2.3.2). Na declaração

[2]Neste texto, a classe " ARRAY " é denominada como vetor.

de um objeto, o número de elementos contidos no vetor é especificado através das palavras reservadas " DOWNTO " e " TO ". A primeira identifica uma ordem descendente nos índices e a segunda o inverso (ver Figura 2.3.2).

Tabela 2.3.2 Tipos " ARRAY " predefinidos no pacote padrão.

tipo predefinido	valor	exemplos
BIT_VECTOR	1, 0	"1010", B"10_10", O"12", X"A"
STRING	tipo character	"texto", ""incluindo_aspas""

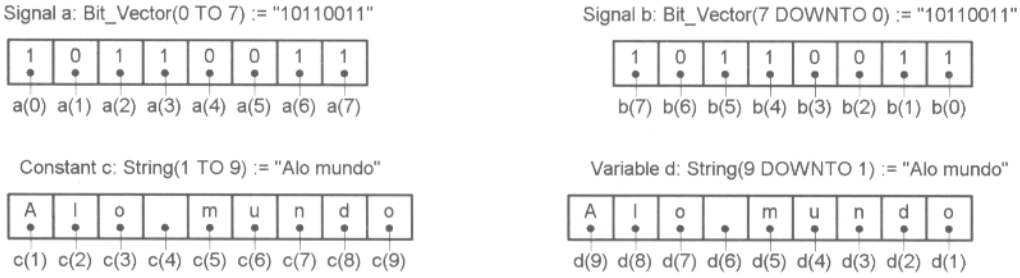


Figura 2.3.2 Declarações de vetores com definição de limites.

Nomes indexados, parte de vetores e agregado

Em um vetor unidimensional os elementos podem ser referenciados através de nomes indexados ou partes de vetores (ver Figura 2.3.3). Na primeira opção, a referência a um elemento é feita identificando-se o objeto e índice do elemento que é colocado entre parêntesis; exemplo: " a(4) ", " b(4) " e " b(3) ". Na outra opção, é especificada uma parcela crescente ou decrescente de um vetor por meio das palavras reservadas " TO " ou " DOWNTO ".

No exemplo proposto, o sinal " b " é declarado como um vetor contendo cinco elementos com índices decrescentes (ver Figura 2.3.3). A parte " b(3 Downto 0) " identifica os elementos " b(3) ", " b(2) ", " b(1) " e " b(0) ". A ordem na identificação dos elementos deve seguir a ordem declarada no objeto. Na versão VHDL-1987, uma parcela na ordem contrária, como, por exemplo, " b(0 TO 3) ", denota uma parte contendo nenhum elemento. Na versão VHDL-1993 isso é um erro.

A atribuição de valor " a(0 To 3) <= b(3 Downto 0); " da Figura 2.3.3 pode parecer incorreta, considerando a ordem dos elementos na identificação da parte de um vetor. Note que as partes " a(0 To 3) " e " b(3 Downto 0) "seguem a mesma ordem declarada nos objetos. Não é um erro a transferência de valores entre vetores com ordenação de índices distinta. A atribuição " a <= b; ", por exemplo, também está correta.

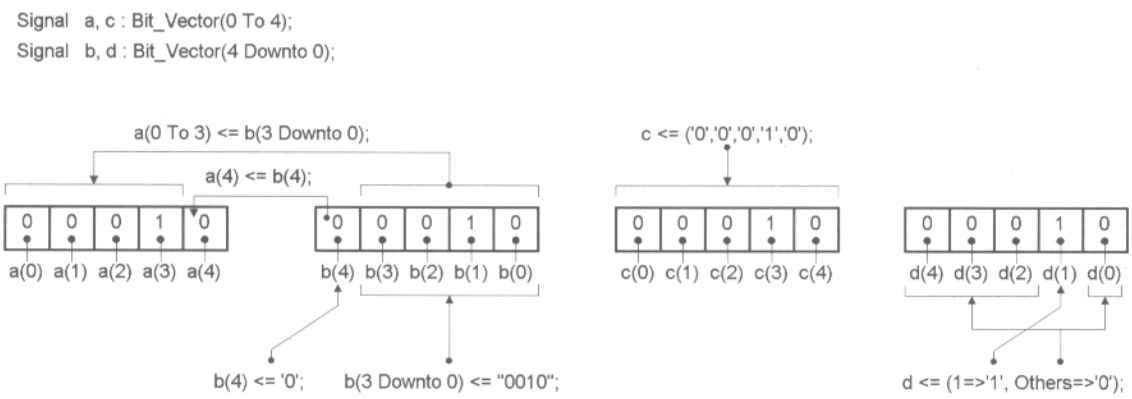


Figura 2.3.3 Ilustração de atribuições de valores com vetores unidimensionais.

Numa atribuição de valor a um objeto do tipo vetor, a expressão que contém o valor pode ser um valor, um nome indexado, uma parte de um vetor ou um agregado (ver Quadro 2.3.1 e Figura 2.3.3).

Conforme o manual de referência da linguagem, um agregado, ou *aggregate*, é uma expressão indicando o valor de um tipo composto. Os valores são especificados dando-se o valor de cada elemento do tipo composto.

O Quadro 2.3.1 contém dois exemplos de agregados. No caso do sinal " c ", o valor de cada elemento é especificado conforme a sua posição, notação posicional. Para o sinal " d ", a associação é feita pelo nome dos elementos. A palavra reservada " OTHERS " pode ser empregada para identificar todos os elementos não-especificados, e deve ser sempre a última associação na lista de associações.

```
a(4)        <= b(4);              -- 1 elemento      <= 1 elemento
a(0 TO 3)   <= b(3 DOWNTO 0);     -- parte do vetor  <= parte do vetor
b(4)        <= '0';               -- 1 elemento      <= valor
b(3 DOWNTO 0) <= "0010";          -- parte do vetor  <= valor

c <= ('0','0','0','1','0');       -- valor 00010 agregado notacao posicional
d <= (1=>'1', OTHERS=>'0');       -- valor 00010 agregado associacao por nomes
```

Quadro 2.3.1 Exemplos de atribuição de valores em vetores: índice, parte e agregado.

Definição de novos tipos

Conforme comentado no Capítulo 1, a linguagem VHDL permite a criação de novos tipos enumerados, físicos e compostos. A criação de um novo tipo pode facilitar a leitura de um código, atribuindo nomes específicos para determinadas condições. A abordagem dada neste capítulo sobre a definição de novos tipos é muito limitada, e visa apenas dar uma introdução sobre o tema. A descrição detalhada desse tópico é apresentada no Capítulo 12.

A declaração de um tipo é feita através da palavra reservada " TYPE ", seguida do nome escolhido para designar novo tipo e a sua especificação. No Quadro 2.3.2 é apresentado um exemplo para criação de um tipo enumerado e a declaração de um sinal que emprega esse tipo. Conforme visto, tipos enumerados permitem a aplicação de operadores de relação como maior, menor, igual. A ordem acompanha a seqüência dada na declaração, o primeiro valor é considerado o menor, e o último valor é considerado o maior.

```
TYPE estado IS (parado, inicio, caso_1, caso_2, caso_3);   -- declaracao do tipo

SIGNAL abc : estado := parado;                             -- objeto empregando o tipo
```

Quadro 2.3.2 Exemplo da criação de um tipo enumerado e declaração de um sinal.

2.4 Operadores

Os operadores definidos são divididos em classes que estabelecem a precedência na execução das operações. Na Tabela 2.4.1 são apresentadas, em ordem crescente de precedência, as classes de operadores. Os operadores de uma mesma classe possuem o mesmo nível de precedência. Parêntesis podem ser empregados para definir a ordem das operações, sendo o seu uso recomendável para evidenciar a ordem desejada em expressões longas. O operador " not " pertence à classe diversos, devido à sua precedência mais elevada; com respeito à sua operação, ele deve ser considerado como um operador lógico.

Devido à rigidez da linguagem VHDL com relação a tipos, os operandos de uma operação são normalmente do mesmo tipo. As exceções podem ser encontradas nas operações de multiplicação e divisão entre tipos físicos, operações de deslocamento e operação de exponenciação.

Tabela 2.4.1 Operadores definidos por classe.

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

Nota a: O operador " not " está inserido na classe diversos devido a sua maior precedência.

Nota b: As operações " xor ", " sll ", " srl ", " sla ", " sra ", " rol " e " ror " não são definidas na versão VHDL-1987.

A Tabela 2.4.2 relaciona os operadores lógicos definidos na linguagem, e a Tabela 2.4.3 descreve as operações executadas para cada condição. No caso de operandos do tipo " boolean ", os valores " 0 " e " 1 " da Tabela 2.4.3 devem ser substituídos pelos valores " FALSE " e " TRUE ", respectivamente. As operações podem ser executadas, também, em vetores compostos por elementos desses dois tipos. Neste caso, os operandos devem ser vetores do mesmo tamanho e compostos por elementos do mesmo tipo. Assim, é possível, por exemplo, realizar operações lógicas entre tipos " bit_vector ".

Tabela 2.4.2 Operadores lógicos.

operadores	operando L	operando R	retorna
not	bit	mesmo tipo de L	mesmo tipo de L
and or xor	boolean	mesmo tipo de L	mesmo tipo de L
nand nor xnor	vetor unidimensional com elementos do tipo bit ou boolean	mesmo tipo de L	mesmo tipo de L

Exemplos: z <= x AND y; sinal_k <= NOT var_a1 ; x1 := a OR NOT b;

Nota a: O operador " not " pertence à classe diversos e possui unicamente o operando L.

Nota b: O operador " xnor " não é definido na versão VHDL-1987.

Tabela 2.4.3 Descrição das operações lógicas.

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

Os operadores relacionais disponíveis testam a igualdade, desigualdade e a ordenação entre dois operandos (ver Tabela 2.4.4). Os operandos devem ser do mesmo tipo e o resultado é um valor do tipo " Boolean ". A operação pode ser efetuada entre vetores com tamanhos diferentes, e é realizada da esquerda para a direita com os vetores alinhados à esquerda. Para exemplos de operações relacionais entre vetores com dimensões distintas (ver Capítulo 11, item 11.3).

Dois valores do tipo escalar são considerados iguais se tiverem o mesmo valor; no caso de tipos compostos, para cada elemento do operando " L " deve haver um elemento igual na mesma posição do operando " R ". A Figura 2.4.1 ilustra exemplos de operadores relacionais entre constantes do tipo " BIT_VECTOR ".

Tabela 2.4.4 Operadores relacionais: igualdade, desigualdade e ordenação.

operadores	operando L	operando R	retorna
= /=	qualquer tipo	mesmo tipo de L	boolean
> < >= <=	qualquer tipo escalar	mesmo tipo de L	boolean
	vetor unidimensional com elementos do tipo inteiro ou enumerado	mesmo tipo de L	boolean
Exemplos: IF x /= y THEN ....; WHILE NOT z = valor_final			

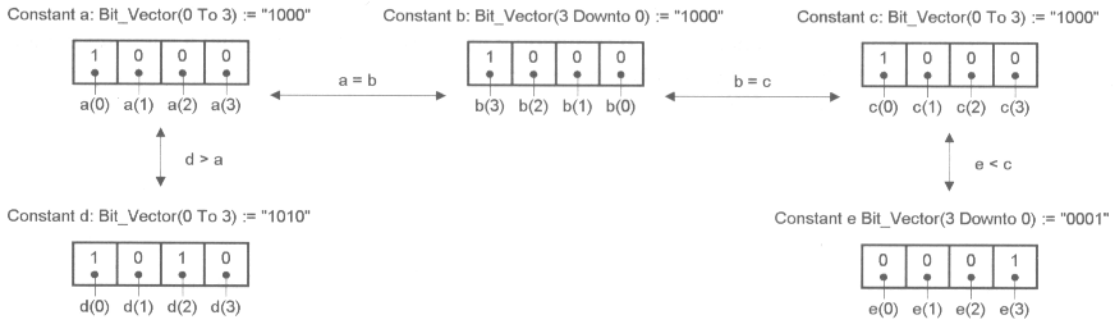


Figura 2.4.1 Operadores relacionais — exemplos.

Os operadores de deslocamento aritmético, deslocamento lógico e rotação não são definidos na versão VHDL-1987 (ver Tabela 2.4.5). O operando da esquerda pode ser qualquer vetor composto por elementos do tipo " bit " ou " boolean ". O operando da direita, que define o valor do deslocamento ou rotação, é um valor inteiro. As Figuras 2.4.2 e 2.4.3 ilustram o comportamento de vetores para diferentes casos. Nas operações de deslocamento, um elemento do vetor é descartado, e um novo elemento é inserido. Nos deslocamentos aritméticos, " sla " ou " sra ", o elemento mais à direta ou mais à esquerda do vetor é repetido nas posições vagas. No caso de um deslocamento lógico, " sll " ou " srl ", um novo valor é inserido: o valor mais à esquerda do conjunto de valores definidos para o tipo. Por exemplo, para um tipo " BIT " o valor " 0 ", para um tipo " BOOLEAN " o valor " FALSE ".

Com relação à síntese de circuitos empregando esses operadores, deve ser considerado o custo de implementação. Para valores de deslocamento constante, o custo de implementação é mínimo, apenas interligações realizam a operação. No caso de valores não-constantes, o número de portas para realização da lógica combinacional pode ser considerável, conforme o tamanho do vetor e as opções de deslocamento.

Tabela 2.4.5 Operadores deslocamento - VHDL-1993 somente.

operadores	operando L	operando R	retorna
sll srl sla sra rol ror	vetor unidimensional com elementos bit ou boolean	integer	mesmo tipo de L
Exemplos: z <= x SLL 1; sinal_a <= x ROR n_shift;			

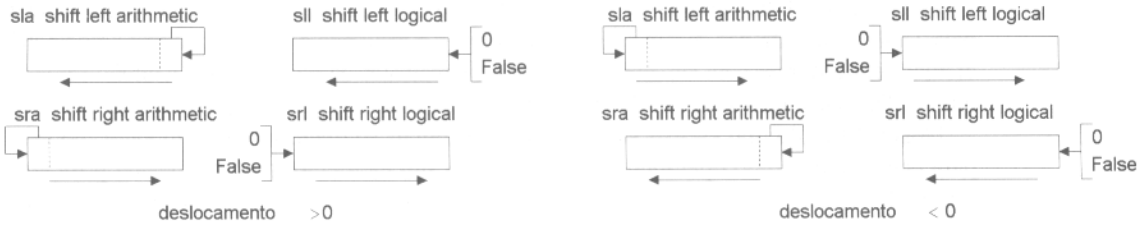


Figura 2.4.2 Operações de deslocamento lógico e aritmético.

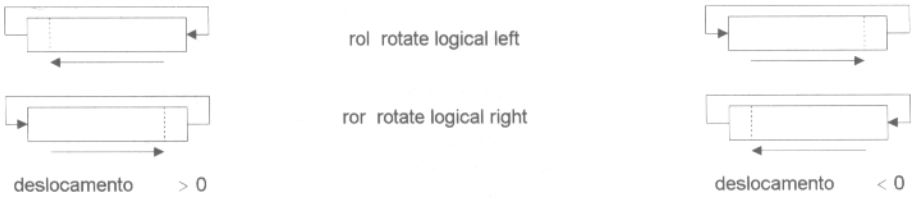


Figura 2.4.3 Operações de rotação.

Os operadores da classe adição " + " e " - " são definidos para qualquer tipo numérico e correspondem aos operadores matemáticos de soma e subtração (ver Tabela 2.4.6). O tipo dos operandos deve ser o mesmo. Na operação de concatenação " & ", o resultado é um vetor de tamanho igual à soma do tamanho dos operandos. A concatenação pode ser executada entre vetores, entre vetores e elementos, e entre elementos. Os operandos e o resultado devem ser do mesmo tipo.

Tabela 2.4.6 Operadores de adição.

operadores	operando L	operando R	retorna
+ -	tipo numérico	o mesmo tipo de L	mesmo tipo
&	vetor unidimensional	mesmo vetor de L	vetor mesmo tipo
	vetor unidimensional	elemento	vetor mesmo tipo
	elemento	vetor unidimensional	vetor mesmo tipo
	elemento	elemento	vetor
Exemplos: z := x + 1; s4 <= '0' & X"B";			
Nota: Na operação " & " os vetores ou elementos devem ser do mesmo tipo.			

Os operadores da classe sinal, identidade e negação possuem o mesmo significado dos operadores matemáticos equivalentes. Eles podem ser aplicados para qualquer tipo numérico, e o valor de retorno é do mesmo tipo (ver Tabela 2.4.7).

Tabela 2.4.7 Operadores aritméticos identidade e negação,

operadores	operando	retorna
+ -	qualquer tipo numérico	mesmo tipo
Exemplos: a <= -b;		

Os operadores de multiplicação e divisão são definidos para operandos inteiros e reais, e o resultado é do mesmo tipo (ver Tabela 2.4.8). Os operadores " mod " e " rem " são definidos para tipos inteiros, e o resultado é do mesmo tipo. Operações com tipos físicos são também possíveis (ver Tabela 2.4.9). No caso de multiplicação, operações entre um tipo físico e um tipo inteiro e entre um tipo físico e real são permitidas. O resultado será sempre um tipo físico. Quanto à divisão, um tipo físico pode ser dividido por um tipo inteiro, real e físico. Nos dois primeiros casos, o resultado é um tipo físico, e no último caso é um tipo inteiro.

Tabela 2.4.8 Operadores aritméticos multiplicação.

operadores	operando L	operando R	retorna
* /	qualquer tipo integer	mesmo tipo	mesmo tipo
	qualquer tipo real	mesmo tipo	mesmo tipo
mod rem	qualquer tipo integer	mesmo tipo	mesmo tipo
Exemplos: l_div_r <= l / r; l_rem_r <= l REM r;			

Tabela 2.4.9 Operadores multiplicação e divisão com tipos físicos.

operadores	operando L	operando R	retorna
*	qualquer tipo physical	integer	mesmo tipo de L
	qualquer tipo physical	real	mesmo tipo de L
	integer	qualquer tipo physical	mesmo tipo de R
	real	qualquer tipo physical	mesmo tipo de R
/	qualquer tipo physical	integer	mesmo tipo de L
	qualquer tipo physical	real	mesmo tipo de L
	qualquer tipo physical	mesmo tipo	integer

A operação " abs ", valor absoluto, pode ser aplicada para qualquer tipo e retorna o mesmo tipo (ver Tabela 2.4.10). Exponenciação, " ** ", pode ser aplicada em tipos inteiros e reais, com expoentes do tipo inteiro. Para expoentes com valor negativo, o operando em que está sendo aplicada a operação deve ser do tipo " real ".

Tabela 2.4.10 Operadores da classe diversos.

operadores	operando L	operando R	retorna
abs	qualquer tipo numérico	-	mesmo tipo
**	qualquer tipo integer	integer	mesmo tipo de L
	qualquer tipo real	integer	mesmo tipo de L
Exemplos: sinal_a <= ABS(-1.7); var_b := a **2;			

2.5 Exemplos de utilização

A descrição do Quadro 2.5.1 permite verificar as possíveis formas de descrever valores para tipos " INTEGER " e " REAL ". No exemplo, o valor inteiro " 11 " é representado em diversas formas e bases numéricas. Quanto às portas de saída, são apresentadas algumas opções para definir os limites que os valores podem assumir (ver

linhas 2 a 6). Esses limites, no caso de descrições voltadas para síntese, são importantes, pois definem para a ferramenta de síntese o menor número de bits necessário para representar todos os valores que a porta pode assumir. A descrição contém constantes do tipo " REAL ", o que inviabiliza a sua aplicação para síntese, pois necessitaria de uma estrutura de ponto flutuante para a representação dos valores.

```
 1 ENTITY int_real IS
 2   PORT (ci1,ci2      : OUT INTEGER RANGE 0    TO 31;
 3         ci3,ci4      : OUT INTEGER RANGE 31   DOWNTO 0;
 4         ci5,ci6,ci7  : OUT INTEGER RANGE -15  TO 15;
 5         cr1,cr2      : OUT REAL    RANGE 0.0  TO 31.0;
 6         cr3,cr4      : OUT REAL    RANGE 31.0 DOWNTO 0.0);
 7 END int_real;
 8
 9 ARCHITECTURE teste OF int_real IS
10   CONSTANT i1 : INTEGER  :=  11;                  -- valor 11, base 10
11   CONSTANT i2 : INTEGER  :=  10#11#;              -- valor 11, base 10
12   CONSTANT i3 : INTEGER  :=  2#01011#;            -- valor 11, base 2
13   CONSTANT i4 : INTEGER  :=  2#01_01_1#;          -- valor 11, base 2
14   CONSTANT i5 : INTEGER  :=  5#21#;               -- valor 11, base 5
15   CONSTANT i6 : NATURAL  :=  8#13#;               -- valor 11, base 8
16   CONSTANT i7 : POSITIVE :=  16#B#;               -- valor 11, base 16
17
18   CONSTANT r1 : REAL     :=  11.0;                -- valor 11, base 10
19   CONSTANT r2 : REAL     :=  1.1E01;              -- valor 11, base 10 formato nn.nExx
20   CONSTANT r3 : REAL     :=  2#01011.0#;          -- valor 11, base 2
21   CONSTANT r4 : REAL     :=  8#1.3#E01;           -- valor 11, base 8  formato nn.nExx
22   CONSTANT r5 : REAL     :=  16#B.0#;             -- valor 11, base 16
23 BEGIN
24   ci1 <= i1; ci2 <= i2; ci3 <= i3; ci4 <= i4; ci5 <= i5; ci6 <= i6; ci7 <= i7;
25   cr1 <= r1; cr2 <= r2; cr3 <= r3; cr4 <= r5;
26 END teste;
```

Quadro 2.5.1 Atribuição de valores em sinais — tipos " INTEGER " e " REAL ".

A entidade do Quadro 2.5.2 representa um conjunto de cinco portas de saída, " s1 " a " s5 ", todas contendo o mesmo valor. Na definição do valor de um objeto do tipo " bit_vector ", é assumido que o valor está expresso na base dois (linhas 11 e 12). No caso de uma base diferente, deve ser inserido um caractere identificando a base antes da abertura de aspas que contém o valor (ver linhas 12 e 13). O valor de um objeto pode, como no caso de um objeto do tipo inteiro, incluir entre os elementos o caractere " _ " para facilitar a leitura da descrição (ver linha 12).

Nas linhas 14 e 15 o valor é atribuído a uma parte do vetor. Note que a ordem que define a parte do vetor, crescente ou decrescente, deve observar a mesma ordem da declaração do vetor. Se a parte de " s5 " na linha 15 fosse alterada para " 0 TO 2 ", ocorreriam alguns problemas. Na versão VHDL-1987 seria gerada uma mensagem de erro na fase de simulação, pois estaria sendo atribuído um valor com três elementos para um vetor com um número de elementos igual a zero. Na versão VHDL-1993 seria gerada uma mensagem na fase de compilação (ver item 2.3).

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

Quadro 2.5.2 Exemplos de atribuição de valores em sinais — tipo vetor.

No Quadro 2.5.3, são apresentados exemplos de atribuição de valor para vetores empregando agregados. Os sinais " s2 " e " s4 " têm o valor de cada um dos seus elementos especificados por expressões empregando a notação posicional. Nos sinais " s3 " e " s5 " a atribuição é estabelecida pela associação de nomes. Note que é possível delimitar um conjunto de elementos que recebem o mesmo valor (declaração " 4 DOWNTO 3 " na linha 12). Observe, também, que o valor de um dado elemento pode ser resultado de uma expressão (operação " um OR '0' " na linha 11).

```
1  ENTITY std_a1 IS
2   PORT(s2, s3, s4, s5   : OUT BIT_VECTOR(4 DOWNTO 0));
3  END std_a1;
4
5  ARCHITECTURE teste OF std_a1 IS
6    CONSTANT zero : BIT := '0';
7    CONSTANT um   : BIT := '1';
8  BEGIN
9    s2 <= ('0','0','0','1','0');                    -- valor 00010, agregado notacao posicional
10   s3 <= (1=>'1', 0=>'1', OTHERS=>'0');            -- valor 00011, agregado associacao por nomes
11   s4 <= (zero, '0', um OR '0', '0', '0');         -- valor 00100, agregado com operacoes
12   s5 <= (4 DOWNTO 3 =>'0', 1=>'0', OTHERS=>'1'); -- valor 00101, agregado faixa discreta
13 END teste;
```

Quadro 2.5.3 Exemplos de atribuição de valores em sinais do tipo vetor — agregados.

Na descrição do Quadro 2.5.4 são apresentados vários exemplos de atribuição de sinal através de expressões com operadores lógicos. A inclusão de parêntesis em muitos casos é necessária para identificar a precedência das operações. Em uma expressão lógica a operação " e " tem precedência sobre uma operação " ou ". Por exemplo, na expressão escrita na forma " x = y.z + w ", está implícito que a operação " y.z " é executada primeiro. Como em VHDL todos os operadores lógicos, excetuando " not ", têm a mesma precedência, deve-se empregar parêntesis para indicar a ordem desejada nas operações. A atribuição da linha 8 está correta, pois o operador " not " tem precedência mais elevada. A linha 9 está também correta, pois não são misturados operadores diferentes. A linha 10 gera uma mensagem de erro na compilação, pois a operação é ambígua, uma opção correta é apresentada na linha seguinte. O mesmo é válido para as linhas 12 e 13. Na linha 14 ocorre um outro problema: apesar de a expressão conter os mesmos operadores, ela não é válida. Deve-se lembrar que a operação " x nand y " equivale a " not (a and b) ", e, portanto, a ordem na aplicação das operações importa.

```
1  ENTITY std_xa1 IS
2    PORT( a, b, c, d          : IN  BIT;
3          x1, x2, x3, x4, x5 : OUT BIT);
4  END std_xa1;
5
6  ARCHITECTURE exemplo OF std_xa1 IS
7  BEGIN
8     x1 <= a OR NOT b;              -- Certo:  operador NOT tem precedencia mais elevada
9     x2 <= a AND b AND c;           -- Certo:  operadores iguais
10 -- x3 <= a AND b  OR c;           -- Errado: expressao ambigua x3=(a.b)+c  ou x3=a.(b+c) ?
11    x3 <=(a AND b) OR c;           -- Certo:  empregando parentesis
12 -- x4 <= a AND b  OR  c AND d;    -- Errado: expressao ambibua, operadores com mesma precedencia
13    x4 <=(a AND b) OR (c AND d);   -- Certo:  x4 = a.b + c.d
14 -- x5 <= a NAND b  NAND c;        -- Errado: operadores com negacao necessitam parentesis
15    x5 <=(a NAND b) NAND c;        -- Certo:  operador com negacao entre parentesis
16 END exemplo;
```

Quadro 2.5.4 Operadores lógicos.

A descrição do Quadro 2.5.5 permite verificar os operadores da classe adição de soma e concatenação. Duas operações de concatenação são ilustradas (linhas 10 e 11). A operação de soma emprega, também, o operador de negação (ver linha 12). Note que, no pacote padrão, operações de soma e subtração são definidas unicamente para tipos numéricos. No pacote numérico 1076.3 são definidas operações aritméticas entre vetores (ver Capítulo 11).

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

Quadro 2.5.5 Operadores da classe adição.

A entidade do Quadro 2.5.6 permite exercitar os operadores da classe multiplicação. O resultado de uma operação " rem " tem o sinal do operando " L ", e o resultado de uma operação " mod " tem o sinal do operando " R ". Para ambos os casos o resultado deve ter um valor absoluto menor do que o valor absoluto do operando " R ". As operações executadas pelos operadores " mod " e " rem " estão nas linhas 11 e 12 do código na forma de comentário. A Figura 2.5.1 ilustra a simulação de alguns casos.

```
1  ENTITY std_xd IS
2    PORT (l, r              : IN  INTEGER RANGE -32 TO 31;
3          l_mul_r, l_div_r  : OUT INTEGER RANGE -32 TO 31;
4          l_rem_r, l_mod_r  : OUT INTEGER RANGE -32 TO 31);
5  END std_xd;
6
7  ARCHITECTURE teste OF std_xd IS
8  BEGIN
9    l_mul_r <= l * r;
10   l_div_r <= l / r;
11   l_rem_r <= l REM r;      -- l = (l/r)*r + (l rem r)
12   l_mod_r <= l MOD r;      -- l =     r*n + (l mod r)   sendo n um valor inteiro
13 END teste;
```

Quadro 2.5.6 Descrição para teste dos operadores da classe multiplicação.

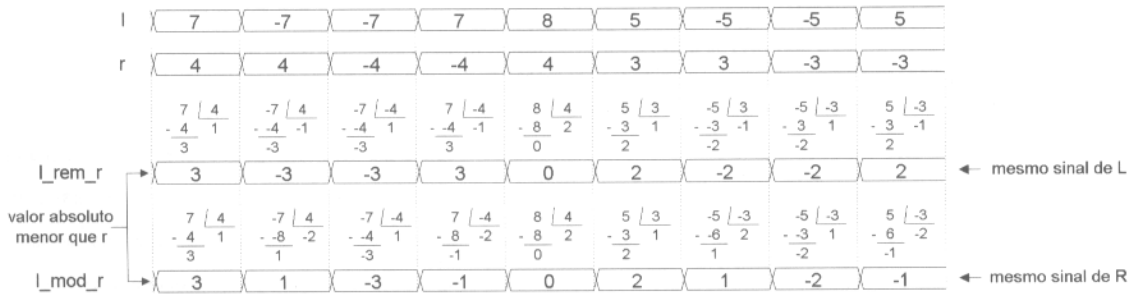


Figura 2.5.1 Exemplos com os operadores " mod " e " rem ".

A descrição do Quadro 2.5.7 ilustra o uso do tipo " STRING " e operações de concatenação de sinais com esses valores. Note que os sinais " x " e " y " poderiam ser declarados como constante, tendo em vista que os seus valores não são alterados no código. Apesar do pouco interesse do código, é possível sintetizar a descrição. Como cada caractere ASCII é representado por oito bits, os dois vetores necessitam de 144 linhas para a representação do valor " Alo mundo ". Do total das linhas, 70 estão ligadas ao nível lógico alto, e 74 estão ligadas ao nível lógico baixo.

```
1  ENTITY lista1 IS
2    PORT (c, d    : OUT STRING(1 TO 9));
3  END lista1;
4
5  ARCHITECTURE teste OF lista1 IS
6     SIGNAL x : STRING(1 TO 3) := "Alo";
7     SIGNAL y : STRING(1 TO 5) := "mundo";
8  BEGIN
9    c <= x & " " & y;
10   d <= x(1 TO 2) & "o " & y;
11 END teste;
```

Quadro 2.5.7 Descrição para teste de operações com tipos " STRING ".

2.6 Principais pontos abordados

- Numa entidade, a declaração da entidade define a interface para com o mundo exterior, e a arquitetura, a operação executada.
- Os modos de uma porta são " IN ", " OUT ", " BUFFER ", e " INOUT ". O primeiro opera exclusivamente como entrada, os modos " OUT " e " BUFFER " como saída, e o modo " INOUT " como entrada e saída.
- Um objeto é um elemento que armazena um valor. Em VHDL são definidas as classes de objeto denominadas " CONSTANT ", " VARIABLE ", " SIGNAL " e " FILE ". Na classe constante o valor não é alterado na execução do código, e nas classes variável e sinal o valor pode ser alterado. A classe arquivo é empregada nas operações com arquivos.
- Todo objeto deve ser declarado segundo um tipo definido. Não é permitida a troca de valores entre objetos de tipos distintos.
- Os tipos escalares predefinidos abordados foram: " BIT ", " BOOLEAN ", " CHARACTER ", " INTEGER ", " REAL " e " PHYSICAL ". Como tipos compostos predefinidos foram apresentados os tipos " BIT_VECTOR " e " STRING ".
- Uma série de operadores é definida na linguagem. Eles são divididos em classes que estabelecem a precedência nas operações. Várias restrições devem ser observadas com respeito ao tipo dos operandos envolvidos em uma operação.

2.7 Exercícios

2.7.1 Apresente o código de uma entidade que descreva as quatro expressões lógicas contidas na Figura 2.7.1. Nessas expressões, considere o operador de negação com maior precedência e o operador lógico " ou " com menor precedência. A entidade deve ter quatro portas de entrada, " a ", " b ", " c " e " d ", e quatro portas de saída, " s1 ", " s2 ", " s3 " e " s4 ", todas do tipo bit.

$s1 = a + \overline{b}$ $\qquad$ $s2 = a + \overline{b} \,.\, c + d$ $\qquad$ $s3 = (a + \overline{b}) \,.\, (c + d)$ $\qquad$ $s4 = (a + \overline{b}) \,.\, \overline{(c + ad)}$

Figura 2.7.1 Expressões lógicas para o Exercício 2.7.1.

2.7.2 Sintetize a descrição proposta para a solução do Exercício 2.7.1. Observe o resultado no nível RTL. Verifique, no esquema criado, se a expressão " $a + \overline{b}$ ", presente nas expressões dos sinais " s1 ", " s2 " e " s3 ", é implementada por uma mesma porta lógica.

2.7.3 Apresente a descrição de uma entidade com duas portas de entrada e uma porta de saída, todas do tipo " bit_vector ". A Figura 2.7.2 ilustra o problema. Ambas as entradas possuem quatro bits, e o valor presente nessas entradas deve ser transferido para a saída, conforme ilustrado na figura.

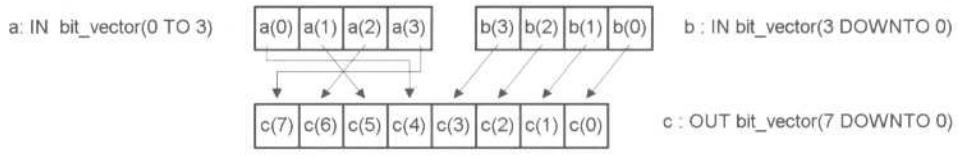


Figura 2.7.2 Exercício 2.7.3.

2.7.4 Considerando a descrição no Quadro 2.7.1, determine, sem o auxílio de um simulador, qual o valor de cada porta de saída. Note que todos os comandos são concorrentes, e, portanto, a ordem nas linhas do código não importa.

```
 1  ENTITY std_yc IS
 2    PORT (a, b   : OUT BIT_VECTOR (2 DOWNTO 0);
 3          c, d   : OUT BIT_VECTOR (0 TO 2));
 4  END std_yc;
 5
 6  ARCHITECTURE teste OF std_yc IS
 7    CONSTANT x : BIT_VECTOR(0 TO 7) := B"1101_1001";
 8    SIGNAL   y : BIT_VECTOR(3 DOWNTO 0);
 9  BEGIN
10    a <= x(1 TO 3);
11    b <= y(3 DOWNTO 1);
12    c <= x(5 TO 7);
13    d <= y(2 DOWNTO 0);
14    y <= x(2 TO 5);
15  END teste;
```

Quadro 2.7.1 Descrição para o Exercício 2.7.4.

2.7.5 No Quadro 2.7.2 é apresentado um código para o teste de operações lógicas. A definição da entidade contém duas entradas e três saídas do tipo " bit_vector ". O valor das saídas é determinado por operações lógicas definidas nas linhas 12, 13 e 14. Desconsiderando inicialmente as linhas 7, 8, 9, 10, 16, 17 e 18, compile a descrição, e simule o resultado para diferentes condições de entrada.

Na arquitetura é definido um vetor composto por elementos do tipo " boolean " (linha 7). Não se preocupe com a definição do tipo, ela será explicada com mais detalhes no Capítulo 12. O intuito dessa definição é verificar o emprego das operações lógicas em vetores contendo elementos " boolean ". Como esse tipo é definido localmente, não é possível definir portas de entrada ou saída empregando esse tipo para entidade. Por esse motivo, foram definidas duas constantes (linhas 9 e 10) para um pequeno teste. Verifique o resultado da simulação. O resultado não é apresentado externamente, mas os simuladores permitem observar sinais internos.

```
 1  ENTITY std_c IS
 2    PORT( a_bit, b_bit              : IN  BIT_VECTOR(2 DOWNTO 0);
 3          not_bit, and_bit, or_bit  : OUT BIT_VECTOR(2 DOWNTO 0));
 4    END std_c;
 5
 6  ARCHITECTURE exemplo OF std_c IS
 7    TYPE     vetor_booleano IS ARRAY (0 TO 2) OF BOOLEAN;
 8    SIGNAL   not_bool, and_bool, or_bool : vetor_booleano;
 9    CONSTANT a_bool  : vetor_booleano := (TRUE,  FALSE, TRUE);
10    CONSTANT b_bool  : vetor_booleano := (FALSE, TRUE,  FALSE);
11  BEGIN
12    not_bit <=   NOT a_bit;
13    and_bit <= a_bit AND b_bit;
14    or_bit  <= a_bit  OR b_bit;
15
16    not_bool <= NOT a_bool;
17    and_bool <= a_bool AND b_bool;
18    or_bool  <= a_bool  OR b_bool;
19  END exemplo;
```

Quadro 2.7.2 Descrição para teste de operações lógicas.

2.7.6 No Quadro 2.7.3 é apresentado um código seguindo a versão VHDL-1993 para o teste da operação lógica " xnor ". Altere a opção de compilação do simulador para a versão VHDL-1993 e simule a entidade.

```
1  -- versao VHDL-1993
2  ENTITY std_c93 IS
3    PORT( a_bit, b_bit       : IN  BIT_VECTOR(2 DOWNTO 0);
4          not_bit, xnor_bit : OUT BIT_VECTOR(2 DOWNTO 0));
5  END std_c93;
6
7  ARCHITECTURE exemplo OF std_c93 IS
8    TYPE vetor_booleano IS ARRAY (0 TO 2) OF BOOLEAN;
9    SIGNAL not_bool, xnor_bool : vetor_booleano;
10   SIGNAL a_bool  : vetor_booleano := (TRUE,  FALSE, TRUE);
11   SIGNAL b_bool  : vetor_booleano := (FALSE, TRUE,  FALSE);
12 BEGIN
13   not_bit  <=   NOT a_bit;
14   xnor_bit <= a_bit  XNOR b_bit;
15
16   not_bool  <= NOT a_bool;
17   xnor_bool <= a_bool  XNOR b_bool;
18 END exemplo;
```

Quadro 2.7.3 Descrição segundo a versão VHDL-1993.

2.7.7 No Quadro 2.7.4 é apresentada uma descrição com problemas. Identifique as linhas com erro e proponha uma correção.

```
1  ENTITY errad_1 IS
2    PORT (a, b, c, d : IN  BIT;
3          s          : OUT BIT_VECTOR (5 DOWNTO 0));
4  END errad_1;
5
6  ARCHITECTURE teste OF errad_1 IS
7  BEGIN
8    s(0) <= a AND  b OR c AND d;
9    s(1) <= a NOR b NOR c;
10   s(2) <= a AND  b OR c;
11   s(3) <= NOT (a AND b) NAND c;
12   s(4) <= a XOR  b XOR c;
13 END teste;
```

Quadro 2.7.4 Descrição contendo erros.

2.7.8 No Quadro 2.7.5 é apresentada uma outra descrição com problemas. Identifique as linhas com erro e proponha uma correção.

```
1  ENTITY errad_2 IS
2    PORT (a, b, c, d : IN  BIT;
3          r, s, t, u, v, x    : OUT BIT_VECTOR (0 TO 5));
4  END errad_2;
5
6  ARCHITECTURE teste OF errad_2 IS
7  BEGIN
8    r(0 TO 2) <= a & b & c;
9    s <= a & b & c & "010";
10   t(0 TO 2) & u(3 TO 5) <= "101101";
11   v <= a & (OTHERS => '0');
12   x <= a & (OTHERS => '0') & '1';
13 END teste;
```

Quadro 2.7.5 Outra descrição contendo erros.

CAPÍTULO

3 COMANDOS CONCORRENTES BÁSICOS

Conforme visto, os comandos concorrentes presentes em um código são executados conjuntamente, e, por isso, a seqüência como são apresentados é irrelevante. O capítulo aborda um conjunto básico de comandos concorrentes: atribuição de valor incondicional, atribuição de valor condicional, blocos e processos. Esse conjunto permite gerar códigos descrevendo circuitos com alguma complexidade em um nível de abstração mais elevado. Além desses tópicos, é apresentado o atraso delta empregado para o simulador avaliar as iterações resultantes dos comandos concorrentes.

3.1 Atraso delta Δ

Em uma região de código concorrente, a ordem dos comandos no código não deve interferir no resultado final da simulação. Para isso, o simulador necessita de um atraso interno, denominado Δ, de modo a coordenar a avaliação de eventos que ocorrem no mesmo instante. Esse intervalo de tempo é nulo, Δ =0. Durante uma simulação, a alteração de uma condição do circuito pode ativar um ou mais comandos do código. Para todos os comandos ativados, o resultado é transferido 1Δ após. A nova condição pode ativar novos comandos, e o processo se repete por sucessivas iterações até atingir uma condição estável.

No Quadro 3.1.1 é apresentado o código de uma possível descrição para o circuito da Figura 3.1.1. Note que a posição dos comandos procura evidenciar o caráter concorrente dos comandos. Em uma linguagem seqüencial, o comando da linha 9 deveria ser apresentado antes do comando da linha 8.

```
1  ENTITY sig_tes0 IS
2    PORT (sa        :  IN  BIT;
3          sb, sc    :  BUFFER BIT);
4  END sig_tes0;
5
6  ARCHITECTURE teste OF sig_tes0 IS
7  BEGIN
8    sc <= sb;
9    sb <= sa;
10 END teste;
```

Quadro 3.1.1 Descrição para estudo das iterações do simulador.

O resultado da simulação e as iterações decorrentes do processo de simulação são apresentados na Figura 3.1.1. A condição estável do circuito é alterada com a mudança do valor do sinal " sa " em t =10 ns. O novo valor em " sa " ocasiona o agendamento de uma modificação no sinal " sb " em t =10 ns +Δ. Após a ocorrência da primeira iteração, o valor de " sb " é alterado para " 1 ", acarretando o agendamento da troca do valor no sinal " sc " em t =10 ns +2Δ. Após a segunda iteração, em t =10 ns+2Δ, o circuito atinge uma condição estável, e nenhuma nova iteração é agendada. Como Δ =0, a condição final visualizada na saída do simulador corresponde à carta de tempo apresentada no lado direito da figura.

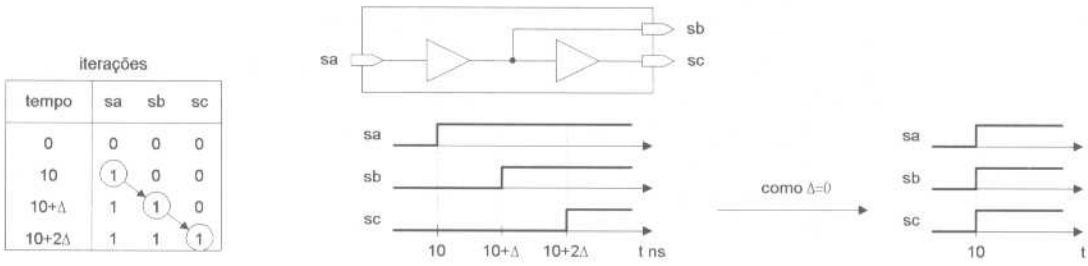


Figura 3.1.1 Exemplo de iterações — caso 1.

Na Figura 3.1.2 é apresentado um outro circuito para análise das iterações, e o seu código é descrito no Quadro 3.1.2. O circuito proposto pode ser redesenhado para um circuito equivalente, considerando que uma porta " ou-exclusivo ", com uma das entradas em nível lógico alto, inverte o nível lógico presente na sua outra entrada. Caso a entrada " z " do circuito passe para o nível lógico 1, a porta " e " transfere o sinal " y " para uma das entradas da porta "ou-exclusivo". Desse modo, temos a porta " ou-exclusivo " operando com um inversor cuja saída é conectada à própria entrada. Como o atraso entre a entrada e a saída é nulo, a simulação desse circuito acarretará um número infinito de iterações, seguindo a seqüência apresentada ao lado direito na Figura 3.1.2.

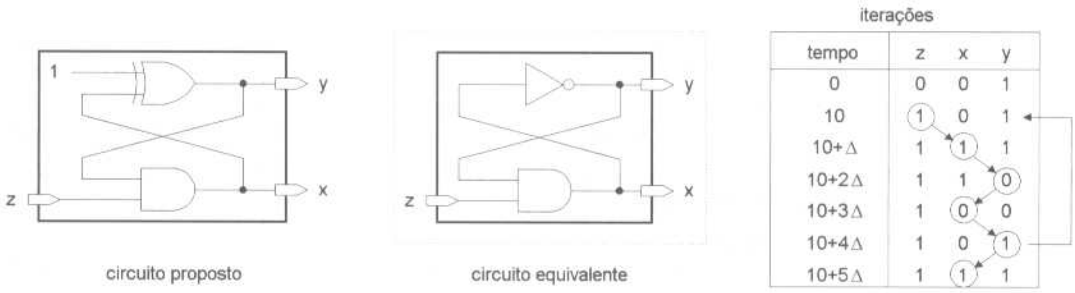


Figura 3.1.2 Exemplo de iterações — caso 2.

```
1  ENTITY c04_delt IS
2    PORT (z        :  IN     BIT;
3          x, y     :  BUFFER BIT);
4  END c04_delt;
5
6  ARCHITECTURE teste OF c04_delt IS
7  BEGIN
8    x <=  z  AND y;
9    y <= '1' XOR x;
10 END teste;
```

Quadro 3.1.2 Código VHDL para teste de iterações.

Os simuladores, normalmente, limitam o número máximo de iterações e abortam a simulação, caso uma condição estável nunca seja atingida. O Quadro 3.1.3 apresenta a mensagem gerada devido à simulação do exemplo da Figura 3.1.1.

```
# ** Error: (vsim-3601) Iteration limit reached at time 10 ns.
```

Quadro 3.1.3 Mensagem devido ao número de iterações ter sido atingido.

3.2 Atribuição de valor para um sinal

A atribuição de um valor a um objeto da classe sinal pode ocorrer tanto em regiões de código concorrente como em regiões de código seqüencial. Uma atribuição ocorrendo numa área de comandos concorrentes é considerada como tal, e, portanto, a ordem dos comandos é irrelevante.

Conforme visto, a atribuição emprega o delimitador " <= ", e a informação pode ser originada de um outro sinal ou expressão (ver Quadro 3.2.1). A linguagem VHDL é rigorosa com relação a operações envolvendo tipos diferentes. Transferência de valores entre objetos de tipos diferentes não é permitida.

```
sinal_destino_a <= sinal_fonte;          -- tipo do sinal_destino_a = tipo do sinal_fonte
sinal_destino_b <= sinal_x OR sinal_y;
```

Quadro 3.2.1 Exemplos de atribuição incondicional de um valor para um sinal.

Na Figura 3.2.1 é apresentado o diagrama esquemático de uma possível implementação de um circuito de seleção de quatro entradas para uma saída. As entradas são denominadas " i0 ", " i1 ", " i2 " e " i3 ", e a saída " ot ". No Quadro 3.2.2 é apresentado o código descrevendo o esquema proposto nessa figura. A descrição emprega apenas uma transferência de sinal, determinada por uma expressão lógica. O uso de parêntesis é necessário devido à igual precedência dos operadores " AND " e " OR ".

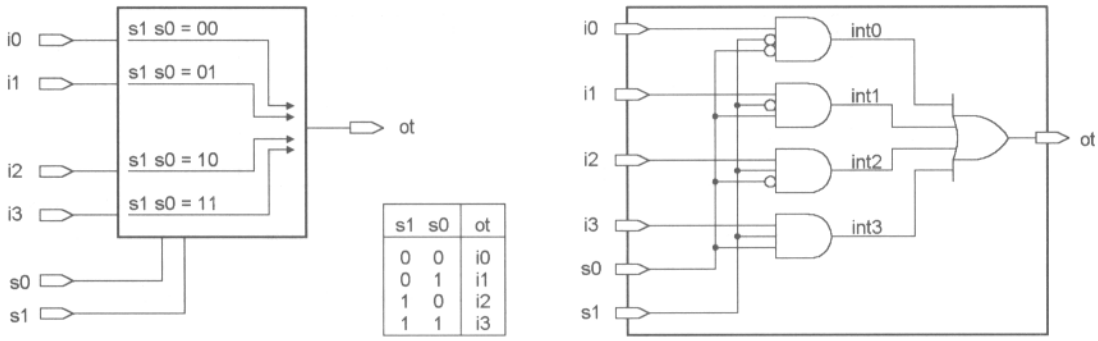


Figura 3.2.1 Diagrama de bloco e esquemático de um circuito de seleção 4 / 1.

```
 1 ENTITY mux_0 IS
 2   PORT (i0, i1, i2, i3     :  IN  BIT;  -- entradas
 3         s0, s1             :  IN  BIT;  -- selecao
 4         ot                 :  OUT BIT); -- saida
 5 END mux_0;
 6
 7 ARCHITECTURE nivel_logico OF mux_0 IS
 8 BEGIN
 9   ot <= (i0 AND NOT s1 AND NOT s0) OR
10         (i1 AND NOT s1 AND     s0) OR
11         (i2 AND     s1 AND NOT s0) OR
12         (i3 AND     s1 AND     s0);
13 END nivel_logico;
```

Quadro 3.2.2 Exemplo do código de um circuito de seleção.

O Quadro 3.2.3 apresenta uma outra descrição para o mesmo circuito de seleção que tem como objetivo explorar as características de uma região de código concorrente. Nesse código são definidos sinais internos para descrever as operações das quatro portas " e " do esquema proposto na Figura 3.2.1 (ver linha 8). A concorrência na execução dos comandos pode ser verificada, por exemplo, na atribuição do sinal " ot " que ocorre na linha 10 do código. No caso de uma linguagem de programação convencional, onde os comandos são executados

seqüencialmente, o sinal " ot " receberia o valor da expressão antes da atualização dos sinais internos " int0 ", " int1 ", " int2 " e " int3 ", o que não seria razoável. Como as atribuições estão numa região de código concorrente, elas são executadas conjuntamente, de modo similar a componentes de um circuito. Assim, a ocorrência de um evento num sinal acarreta a avaliação de todas as expressões que contêm esse sinal. Essas expressões devem ser reavaliadas gerando novos valores. Essa operação gera novas iterações até que seja atingida uma condição estável.

```
1  ENTITY mux_00 IS
2    PORT (i0, i1, i2, i3     :  IN  BIT;  -- entradas
3          s0, s1             :  IN  BIT;  -- selecao
4          ot                 :  OUT BIT); -- saida
5  END mux_00;
6
7  ARCHITECTURE teste OF mux_00 IS
8    SIGNAL int0, int1, int2, int3 : BIT; -- sinais internos
9  BEGIN
10   ot <= int0 OR int1 OR int2 OR int3;
11   int0 <= i0 AND NOT s1 AND NOT s0;
12   int1 <= i1 AND NOT s1 AND     s0;
13   int2 <= i2 AND     s1 AND NOT s0;
14   int3 <= i3 AND     s1 AND     s0;
15 END teste;
```

Quadro 3.2.3 Segundo exemplo do código de um circuito de seleção.

3.3 Construção " WHEN ELSE "

A construção " WHEN ELSE " permite a transferência condicional de um sinal, e segue o formato ilustrado no Quadro 3.3.1. Nessa construção, uma lista de opções é apresentada estabelecendo qual valor de uma expressão deve ser transferido a um sinal de destino. A primeira condição que retorna o valor verdadeiro define o valor que é transferido para o sinal de destino.

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

Quadro 3.3.1 Construção " WHEN ELSE " ou " atribuição condicional de sinal ".

O Quadro 3.3.2 contém algumas linhas de código que ilustram a construção " WHEN ELSE ". O sinal de destino " s0 " pode receber diretamente o valor dos sinais " i0 ", " i1 " ou o resultado de uma operação lógica entre esses sinais. As opções de escolha são definidas por expressões que retornam um valor booleano. Como a construção " WHEN ELSE " define uma prioridade na ordem das opções, o circuito equivalente corresponde a uma cadeia de seletores (ver Figura 3.3.1). Cada seletor é comandado por blocos que detectam as opções; caso a opção seja verdadeira, o seletor transfere o valor de uma expressão, caso contrário, transfere o valor de um outro seletor.

```
s0 <= i0 OR i1   WHEN na= 8 OR  nb= 2 ELSE
      i1         WHEN na= 3 AND nb= 5 ELSE
      i0 AND i1  WHEN na= 7           ELSE
      i0;
```

Quadro 3.3.2 Parte de um código empregando a construção " WHEN ELSE ".

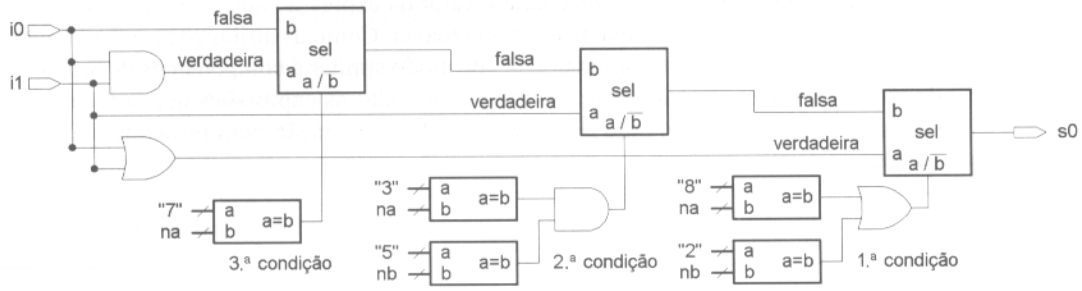


Figura 3.3.1 Circuito equivalente ao código do Quadro 3.3.2.

No Quadro 3.3.3 é apresentado um código completo empregando a construção " WHEN ELSE " para descrever o circuito de seleção proposto na Figura 3.2.1. Neste caso, a descrição possui um estilo mais próximo do comportamento do circuito.

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

Quadro 3.3.3 Circuito de seleção empregando a construção " WHEN ELSE ".

3.4 Construção " WITH SELECT "

A construção " WITH SELECT " transfere um valor a um sinal de destino segundo uma relação de opções. Todas as condições de seleção devem ser consideradas, e elas devem ser mutuamente exclusivas. A lista de opções nessa construção não contém uma prioridade, como se observa na construção " WHEN ELSE ". Essas construções seguem o formato apresentado no Quadro 3.4.1.

As opções podem ser agrupadas através do delimitador " | ", equivalendo, neste caso, a uma condição " ou " entre elas. De modo semelhante, as palavras reservadas " TO " e " DOWNTO " podem ser empregadas para delimitar uma faixa de condições de um tipo escalar. A palavra reservada " OTHERS " é válida, como última alternativa, para englobar as condições restantes.

```
WITH expressao_escolha SELECT                    -- expressao_escolha =
 sinal_destino <= expressao_a WHEN condicao_1,                    -- condicao_1
                  expressao_b WHEN condicao_2,                    -- condicao_2
                  expressao_c WHEN condicao_3 | condicao_4,       -- condicao_3 ou condicao_4
                  expressao_d WHEN condicao_5 TO condicao_7,      -- condicao_5 ate condicao_7
                  expressao_e WHEN OTHERS;                        -- condicoes restantes
```

Quadro 3.4.1 Construção " WITH SELECT " ou " atribuição selecionada de sinal ".

O valor de retorno da expressão de escolha deve ser um tipo discreto ou um vetor unidimensional. Uma expressão de escolha tendo como retorno um valor do tipo não-discreto como " REAL ", por exemplo, não é válida, pois o número de opções de escolha é infinito. Uma boa prática de programação é manter a expressão de escolha simples. Expressões complexas devem ser avaliadas isoladamente, e o resultado transferido para o campo da expressão de escolha.

No Quadro 3.4.2 é apresentada a parte de um código contendo dois exemplos com construção " WITH SELECT ". No primeiro, a expressão de escolha é o sinal " s0 " do tipo " CHARACTER ", e as condições de escolha ilustradas são do tipo: única, uma ou mais condições, faixa de condições e condições restantes. No segundo exemplo a expressão de escolha retorna um valor do tipo " BIT ", e as duas condições de escolha possíveis são apresentadas. Como a construção não identifica nenhuma prioridade nas opções, o circuito equivalente corresponde a um circuito de seleção comandado por um decodificador que detecta as condições contidas no comando (ver Figura 3.4.1).

```
WITH s0 SELECT   -- s0 tipo CHARACTER
  x0 <= i0 AND i1 WHEN 'a',                -- condicao unica
        i0 OR  i1 WHEN 'b'| 'c',           -- uma ou outra condicao
        i0 XOR i1 WHEN 'd' TO 'g',         -- faixa crescente de condicoes
        i0        WHEN 'x' DOWNTO 'k',     -- faixa decrescente de condicoes
        i1        WHEN OTHERS;             -- condicoes restantes

WITH b1 AND b0 SELECT  -- b1 e b0 tipo BIT
  x1 <= i0 WHEN '0',
        i1 WHEN '1';
```

Quadro 3.4.2 Parte de um código contendo construções " WITH SELECT ".

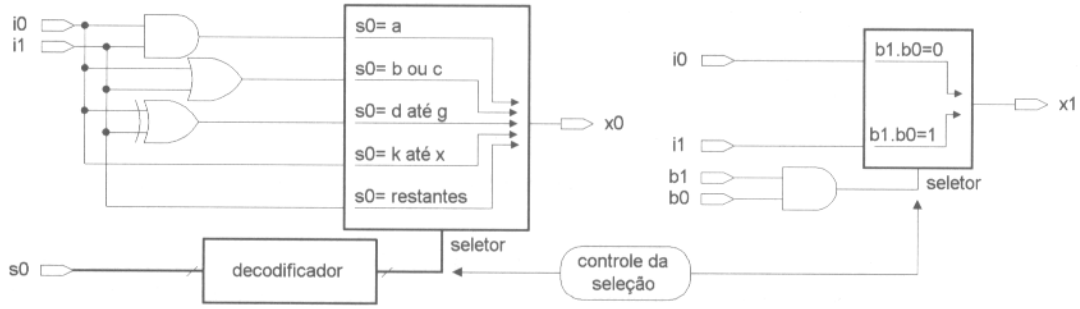


Figura 3.4.1 Circuito equivalente aos códigos do Quadro 3.4.2.

No Quadro 3.4.3 é apresentada uma nova descrição para o circuito da Figura 3.2.1. Na declaração da entidade as portas de entrada responsáveis pela seleção foram concatenadas no sinal interno denominado " sel ". Isso se deve ao fato de que uma operação de concatenação na expressão de escolha não é válida. Observações pertinentes à expressão de escolha são apresentadas no Capítulo 4.

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

Quadro 3.4.3 Circuito de seleção empregando a construção " WITH SELECT ".

3.5 Comando " BLOCK "

O comando " BLOCK " tem a finalidade de dividir o código em regiões para facilitar o entendimento da descrição. Sinais podem ser declarados no interior de um bloco, e a visibilidade destes é restrita ao interior do bloco. A divisão do código não interfere, naturalmente, no circuito sintetizado. Essas declarações seguem o formato apresentado no Quadro 3.5.1.

```
nome_rotulo: BLOCK
   -- definicao de sinais visiveis no bloco
BEGIN
   -- comandos
   -- comandos
END BLOCK nome_rotulo;
```

Quadro 3.5.1 Formato de um comando " BLOCK ".

No Quadro 3.5.2 é apresentada uma nova descrição para o circuito de seleção visto nos itens anteriores. Esse código não visa ilustrar a melhor maneira para descrever o circuito de seleção, mas, sim, um possível emprego do comando " BLOCK ", e evidenciar que os blocos são executados concorrentemente. O bloco " def " identifica qual entrada deve ser transferida para a saída. O resultado é transferido para o sinal visível apenas no interior do bloco denominado " interno_def ", e o valor contido nesse sinal é transferido para o sinal " global ". No bloco " abc " é executada a transferência do valor contido na entrada selecionada para a saída " ot ".

```
1  ENTITY mux_5 IS
2    PORT (i0, i1, i2, i3     :  IN  BIT;                          -- entradas
3          sel                :  IN  INTEGER RANGE 3 DOWNTO 0; -- selecao
4          ot                 :  OUT BIT);                         -- saida
5  END mux_5;
6
7  ARCHITECTURE com_block OF mux_5 IS
8    SIGNAL global      : BIT_VECTOR(0 TO 1);
9  BEGIN
10   abc: BLOCK
11   BEGIN
12     ot <= i0 WHEN global = "00" ELSE
13           i1 WHEN global = "01" ELSE
14           i2 WHEN global = "10" ELSE
15           i3;
16   END BLOCK abc;
17
18   def: BLOCK
19     SIGNAL interno_def  : BIT_VECTOR(0 TO 1);  -- sinal visivel no bloco def
20   BEGIN
21     WITH sel SELECT
22       interno_def <= "00" WHEN 0,
23                      "01" WHEN 1,
24                      "10" WHEN 2,
25                      "11" WHEN OTHERS;
26     global <= interno_def;
27   END BLOCK def;
28 END com_block;
```

Quadro 3.5.2 Circuito de seleção empregando o comando " BLOCK ".

Um outro emprego do comando " BLOCK " é desabilitar sinais no interior do bloco segundo condições definidas por expressões de guarda (ver Quadro 3.5.3). A expressão de guarda retorna um valor booleano. Na condição falsa dessa expressão, um sinal que recebe um valor condicionado a uma expressão de guarda fica desconectado da expressão que o aciona. Com relação à síntese, essa desconexão pode ser interpretada como uma condição de alta impedância ou manutenção do valor anterior. Para ser interpretada como uma condição de alta impedância, o sinal deve ser declarado como tipo " BUS "; caso contrário, é inferido um elemento de memória sensível a nível, ou *latch*. Com relação à inferência de estados de alta impedância, é necessário, ainda, o emprego de tipos com função de resolução, tópico a ser abordado no Capítulo 10, pois um sinal em VHDL não pode ser acionado por mais de um controlador simultaneamente.

Deve-se notar que as expressões de guarda não são aceitas por muitas ferramentas de síntese, e são consideradas, atualmente, obsoletas. Os tipos " sdt_logic ", apresentados no Capítulo 10, são mais adequados para definição de condições de alta impedância, além de serem aceitos pela maioria das ferramentas de síntese.

```
nome_rotulo: BLOCK expressao_guarda
   -- definicao de sinais visiveis no bloco
BEGIN
   sinal_a <= GUARDED expressao_1
   -- comandos
END BLOCK nome_rotulo;
```

Quadro 3.5.3 Formato de um comando " BLOCK " com sinal de guarda.

No Quadro 3.5.4 é apresentada uma pequena descrição contendo uma expressão de guarda. A descrição corresponde a um *latch* controlado pelo sinal " enb ". O resultado da síntese dessa descrição é apresentado na Figura 3.5.1.

```
1  ENTITY block_0 IS
2    PORT (d, enb     :  IN  BIT;
3          s_lat      :  OUT BIT);
4  END block_0;
5
6  ARCHITECTURE teste OF block_0 IS
7  BEGIN
8    lat: BLOCK (enb = '1')
9    BEGIN
10     s_lat <= GUARDED d;
11   END BLOCK lat;
12 END teste;
```

Quadro 3.5.4 Descrição empregando uma expressão com guarda.

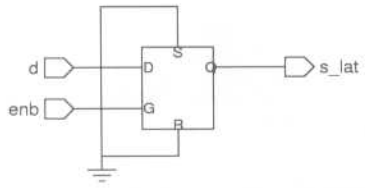


Figura 3.5.1 Circuito sintetizado a partir da descrição do Quadro 3.5.4 — nível RTL.

3.6 Comando " PROCESS "

Um processo permite definir uma área contendo comandos seqüenciais. A palavra reservada " PROCESS " identifica o comando conforme o formato apresentado no Quadro 3.6.1. Um processo é composto de duas regiões:

- parte de declaração;
- parte de comandos seqüenciais.

A primeira região é o local para a declaração de tipos e subtipos, constantes, variáveis, entre outros. A declaração de sinais não é permitida. A segunda região, após a palavra reservada " BEGIN ", inicia a área contendo comandos seqüenciais que representam o comportamento de uma parte ou de toda a descrição.

A palavra reservada " PROCESS " pode ser antecedida de um rótulo qualquer, e ser seguida de uma lista de sensibilidade. Essa lista define a quais sinais o conjunto de comandos contido no processo é sensível. Isto é, ocorrendo uma mudança no valor de um dos sinais contidos na lista, o processo será executado. Exemplos do emprego desse comando são apresentados no Capítulo 4.

```
abc: PROCESS (lista de sensibilidade)
  -- parte_de_declaracao_do_processo
    -- declaracao_de_tipo_e_subtipo;
    -- declaracao_de_constante;
    -- declaracao_de_variavel;
    -- declaracao_de_pseudonimo;
  -- nao_e_permitido: declaracao_de_sinal e declaracao_de_variavel_compartilhada
BEGIN
  -- parte_de_comandos_sequenciais
    -- comando_sequencial;
    -- comando_sequencial;
          ..
    -- comando_sequencial;
END PROCESS abc;
```

Quadro 3.6.1 Formato do comando " PROCESS ".

3.7 Palavra reservada " UNAFFECTED "

A palavra reservada " UNAFFECTED " pode ser empregada na atribuição de valor a um sinal, quando não se deseja afetar o valor do sinal. A palavra " UNAFFECTED " somente é permitida em atribuições concorrentes de valores a sinais. Ver o comando " NULL " no Capítulo 4, para resultados equivalentes em regiões de código seqüencial.

A palavra reservada " UNAFFECTED " não é definida para a versão VHDL-1987, e nem sempre é suportada pelas ferramentas de síntese.

O Quadro 3.7.1 ilustra dois exemplos com o emprego da palavra " UNAFFECTED " para as construções " WHEN ELSE " e " WITH SELECT ". No Quadro 3.7.2 são apresentadas duas descrições completas. Note que os sinais " xa ", " ya " e " xu " e " yu " têm o mesmo comportamento.

```
sinal_destino <= expressao_a  WHEN condicao_1 ELSE
                 UNAFFECTED   WHEN condicao_2 ELSE   -- valor nao alterado na condicao_2
                 expressao_c;
```

```
WITH expressao_escolha SELECT
 sinal_destino <= expressao_a WHEN condicao_1,
                  expressao_b WHEN condicao_2,
                  UNAFFECTED  WHEN OTHERS;            -- valor nao alterado nas condicoes restantes
```

Quadro 3.7.1 Palavra reservada " UNAFFECTED " na atribuição de sinal.

```
1  -- versao VHDL-1993
2  ENTITY unaff_2b IS
3    PORT (sel    : IN     BIT_VECTOR(1 DOWNTO 0);
4          ya, yu : BUFFER BIT);
5  END unaff_2b;
6
7  ARCHITECTURE teste OF unaff_2b IS
8  BEGIN
9    ya <= '0' WHEN sel ="00" ELSE
10         '1' WHEN sel ="10" ELSE
11         ya;
12
13   yu <= '0' WHEN sel ="00" ELSE
14         '1' WHEN sel ="10" ELSE
15         UNAFFECTED;
16  END teste;
```

```
1  -- versao VHDL-1993
2  ENTITY unaff_2a IS
3    PORT (sel    : IN     BIT_VECTOR(1 DOWNTO 0);
4          xa, xu : BUFFER BIT);
5  END unaff_2a;
6
7  ARCHITECTURE teste OF unaff_2a IS
8  BEGIN
9    WITH sel SELECT
10     xa <= '0' WHEN "00",
11           '1' WHEN "10",
12           xa  WHEN OTHERS;
13
14   WITH sel SELECT
15     xu <= '0'         WHEN "00",
16           '1'         WHEN "10",
17           UNAFFECTED WHEN OTHERS;
18 END teste;
```

Quadro 3.7.2 Descrições empregando a palavra reservada " UNAFFECTED ".

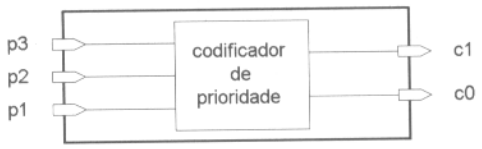
3.8 Cuidados na descrição

Comparações entre as construções " WHEN ELSE " e " WITH SELECT "

Na construção " WHEN ELSE " a ordem em que as condições são apresentadas indica a precedência na execução: a primeira tem prioridade máxima, e a última tem a menor prioridade. Na construção " WITH SELECT " todas as condições possuem a mesma prioridade. A diferença entre as construções pode ser aproveitada pelo projetista para obter circuitos mais eficientes e descrições mais sucintas, conforme o comportamento do circuito a ser descrito.

A Figura 3.8.1 ilustra o caso de um circuito em que a construção " WHEN ELSE " é mais vantajosa. O circuito é um codificador de prioridade entre três entradas denominadas " p3 ", " p2 " e " p1 ", sendo a entrada " p3 " a de maior privilégio. Caso ela esteja ativa, " p3 =1 ", o seu código é representado nas saídas " c1 " e " c0 " independentemente da condição das outras entradas. As entradas restantes, "p2" e "p1", têm uma prioridade decrescente, conforme ilustrado na tabela-verdade da figura.

O Quadro 3.8.1 contém a descrição do circuito empregando a construção " WHEN ELSE ". Essa descrição tem uma maior correspondência com a operação do circuito, onde a condição de cada entrada é verificada individualmente segundo a sua prioridade. No caso da construção " WITH SELECT " (Quadro 3.8.2), como todas as condições devem ser incluídas na construção, o código contém um detalhamento maior em comparação com o código anterior.



p3	p2	p1	c1	c0
1	-	-	1	1
0	1	-	1	0
0	0	1	0	1
0	0	0	0	0

- ⇒ não importa

Figura 3.8.1 Proposta de um codificador de prioridade.

```
ENTITY pr_cod1 IS
PORT (p      : IN  BIT_VECTOR(3 DOWNTO 1);
      c      : OUT BIT_VECTOR(1 DOWNTO 0));
END pr_cod1;

ARCHITECTURE teste OF pr_cod1 IS
BEGIN                                  -- p3 p1 p0  c1 c0
  c <= "11" WHEN p(3)='1' ELSE  --  1  -  -   1  1
       "10" WHEN p(2)='1' ELSE  --  0  1  -   1  0
       "01" WHEN p(1)='1' ELSE  --  0  0  1   0  1
       "00";                    --  0  0  0   0  0
END teste;
```

Quadro 3.8.1 Detector de prioridade — construção " WHEN ELSE ".

```
ENTITY pr_cod2 IS
PORT (p      : IN  BIT_VECTOR(3 DOWNTO 1);
      c      : OUT BIT_VECTOR(1 DOWNTO 0));
END pr_cod2;

ARCHITECTURE teste OF pr_cod2 IS
BEGIN
  WITH p SELECT                                -- p3 p1 p0  c1 c0
  c <= "11" WHEN "111"|"110"|"101"|"100",  --  1  -  -   1  1
       "10" WHEN "011"|"010",              --  0  1  -   1  0
       "01" WHEN "001",                    --  0  0  1   0  1
       "00" WHEN "000";                    --  0  0  0   0  0
END teste;
```

Quadro 3.8.2 Detector de prioridade — construção " WITH SELECT ".

As Figuras 3.8.2 e 3.8.3 ilustram o circuito sintetizado a partir das descrições contidas nos Quadros 3.8.1 e 3.8.2 no nível RTL. No caso desse circuito, a descrição contendo a construção " WHEN ELSE " leva diretamente a um circuito bem mais compacto em comparação com a construção " WITH SELECT ". Como a ferramenta de síntese realiza uma minimização a partir do nível RTL, o esquema final do circuito a ser sintetizado é o mesmo para ambos os casos (ver Figura 3.8.4).

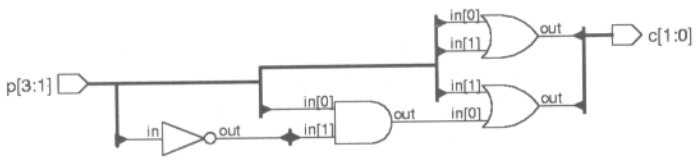


Figura 3.8.2 Entidade sintetizada " pr_enc1 ", nível RTL.

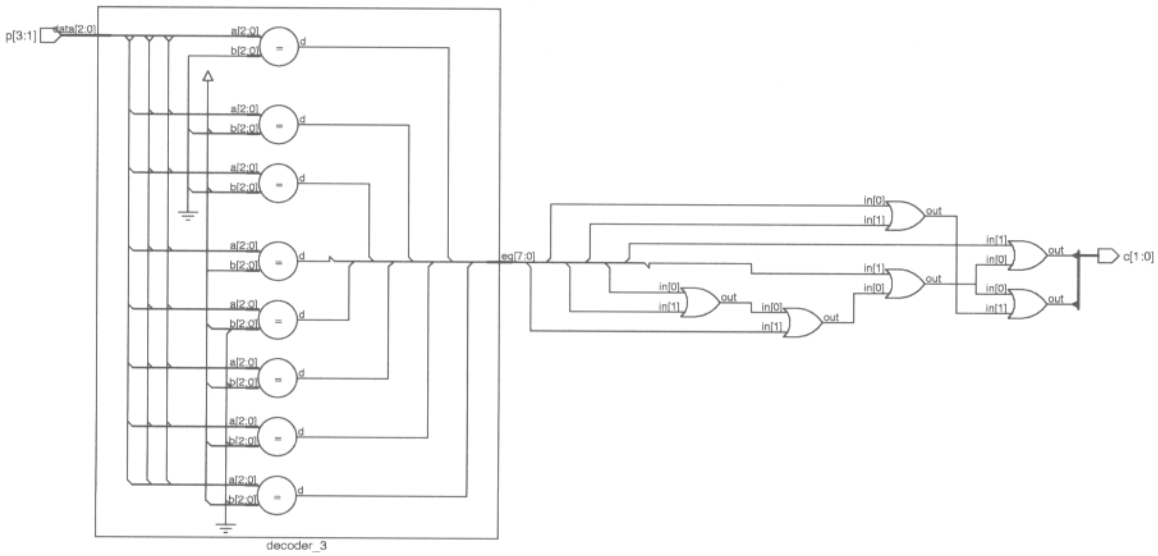


Figura 3.8.3 Entidade sintetizada " pr_enc2 ", nível RTL.

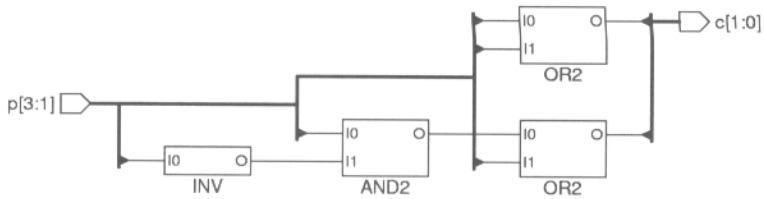


Figura 3.8.4 Circuito final otimizado das descrições " pr_enc1 " e " pr_enc2 ".

Criação de *latch* com as construções " WHEN ELSE " e " WITH SELECT "

Os circuitos sintetizados com as construções " WHEN ELSE " e " WITH SELECT " são do tipo combinacional. Caso o sinal de destino seja um dos possíveis sinais a serem transferidos, ou nem todas as condições de escolha tenham sido consideradas, é necessário um circuito para armazenar a informação.

No Quadro 3.8.3 são apresentados três exemplos que levam à inferência de um elemento de memória sensível a nível. No exemplo da primeira construção " WHEN ELSE " a porta de saída " q1 " recebe o valor de " d1 " na condição " enb ='1' ". Caso contrário, o valor anterior é mantido, caracterizando a operação de um *latch*. O mesmo ocorre com a construção " WHEN ELSE " da linha 11. A condição " enb ='0' " não foi considerada, resultando que o valor de " q2 " deve ser mantido nessa condição. De modo semelhante, a porta de saída " q3 " recebe o seu próprio valor na condição " enb ='0' ", levando à necessidade de manutenção do valor nessa condição. Na Figura 3.8.5 é apresentado o circuito sintetizado, nível RTL, para essa descrição.

```
1  ENTITY latch_0a IS
2    PORT (enb, d1, d2, d3 : IN     BIT;
3          q1, q2, q3      : BUFFER BIT);
4  END latch_0a;
5
6  ARCHITECTURE teste OF latch_0a IS
7  BEGIN
8   q1 <= d1 WHEN enb ='1' ELSE
9         q1;
10
11  q2 <= d2 WHEN enb ='1';
12
13  WITH enb SELECT
14    q3 <= d3 WHEN '1',
15          q3 WHEN '0';
16 END teste;
```

Quadro 3.8.3 Descrição de elementos de memória.

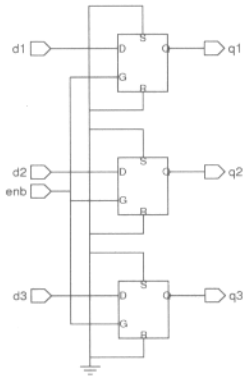


Figura 3.8.5 Circuito sintetizado, descrição " latch_0a ", nível RTL.

3.9 Principais pontos abordados

- A execução dos comandos concorrentes por parte do simulador é possível pela introdução de um atraso interno nulo denominado Δ. Um sinal somente recebe um novo valor após um intervalo de tempo igual a 1Δ.
- Na construção " WHEN ELSE ", a seqüência das condições define a prioridade, pois a primeira condição verdadeira encontrada na lista estabelece a expressão para o sinal de destino.
- Na construção " WITH SELECT " as condições devem ser mutuamente exclusivas; todas as condições devem ser especificadas, e elas possuem a mesma prioridade.
- O comando " BLOCK " permite isolar regiões de código concorrente. Sinais visíveis unicamente nessas regiões podem ser declarados.
- Sinais no interior de um bloco podem ser desabilitados conforme a condição de uma expressão de guarda. Esse tipo de construção pode ser empregado na inferência de um circuito com saída em alta impedância ou um elemento de memória.
- O comando " PROCESS " delimita as regiões contendo comandos seqüenciais. A execução dos comandos contidos em um processo pode ser condicionada a uma lista de sinais denominada lista de sensibilidade.

3.10 Exercícios

3.10.1 Proponha descrições para os circuitos da Figura 3.10.1. Analise a seqüência de iterações que devem ser executadas pelo simulador, com base nas formas de onda apresentadas na figura. Note que o atraso de propagação das portas é nulo.

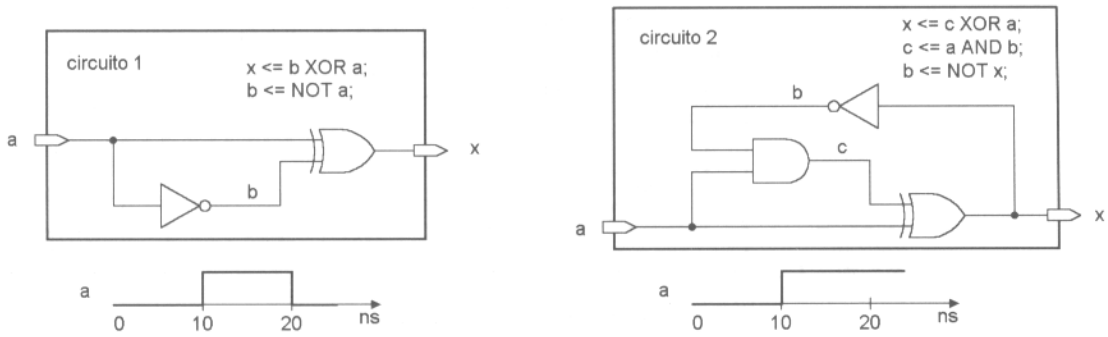


Figura 3.10.1 Exercício 3.10.1.

3.10.2 Apresente o código de um circuito *latch* transparente de 4 bits, conforme ilustrado na Figura 3.10.2, empregando a construção " WHEN ELSE ". As saídas " q " seguem as entradas " d " na condição " en =1 "; na condição " en =0 " a informação é mantida.

3.10.3 Repita o Exercício 3.10.2 empregando a construção " WITH SELECT ".

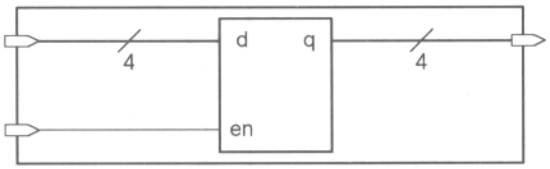
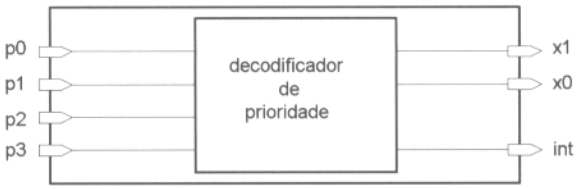


Figura 3.10.2 Ilustração para os Exercícios 3.10.2 e 3.10.3.

3.10.4 Desenvolva o código para um detector de prioridade com quatro entradas empregando a construção " WHEN ELSE ". A entrada de prioridade máxima é " p0 ", e a entrada de prioridade mínima é " p3 ". O detector deve ser empregado num sistema microprocessado, onde quatro periféricos podem solicitar uma interrupção. Uma das entradas em nível alto corresponde a um pedido de interrupção; neste caso, a saída " int " deve ser ativada. Na Figura 3.10.3 é apresentada a tabela-verdade do decodificador.

3.10.5 Repita o Exercício 3.10.4 empregando a construção " WITH SELECT ".



p0	p1	p2	p3	x1	x0	int
1	X	X	X	0	0	1
0	1	X	X	0	1	1
0	0	1	X	1	0	1
0	0	0	1	1	1	1
0	0	0	0	1	1	0

Figura 3.10.3 Ilustração para os Exercícios 3.10.4 e 3.10.5.

3.10.6 Apresente o código de um circuito decodificador para a seleção de memórias. O sistema é composto por três memórias: a primeira respondendo na faixa de endereços de " 00000 h " a " 1FFFF h ", a segunda na faixa de " 20000 h " a " 27FFF h ", e a terceira na faixa de " 28000 h " a " 2FFFF h " (ver Figura 3.10.4). Empregue a construção " WHEN ELSE ".

3.10.7 Repita o Exercício 3.10.6 empregando a construção " WITH SELECT ".

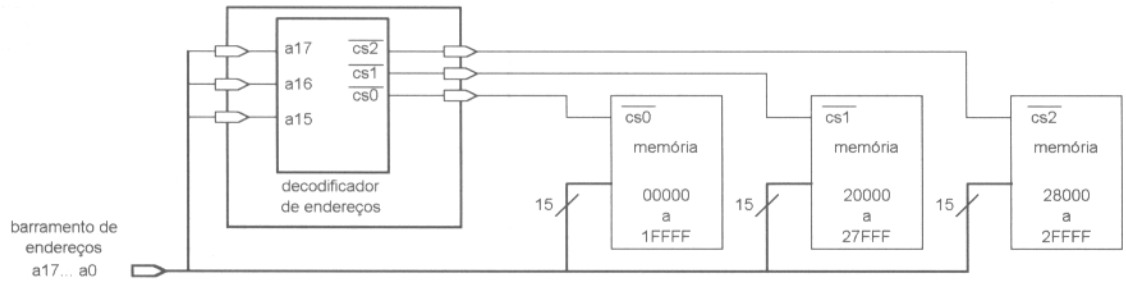


Figura 3.10.4 Ilustração para os Exercícios 3.10.6 e 3.10.7.

3.10.8 Apresente o código de um conversor do código BCD, *binary-coded decimal*, para um mostrador de sete segmentos. Os segmentos são acionados em nível lógico alto.

3.10.9 Com base na Figura 3.10.5, desenvolva o código descrevendo o sistema contendo um decodificador BCD para sete segmentos e um circuito de seleção. A descrição deve ser dividida em dois blocos, um contendo o decodificador, e o outro bloco, o circuito de seleção. Os mostradores são acionados em nível baixo, e os segmentos acionados em nível alto.

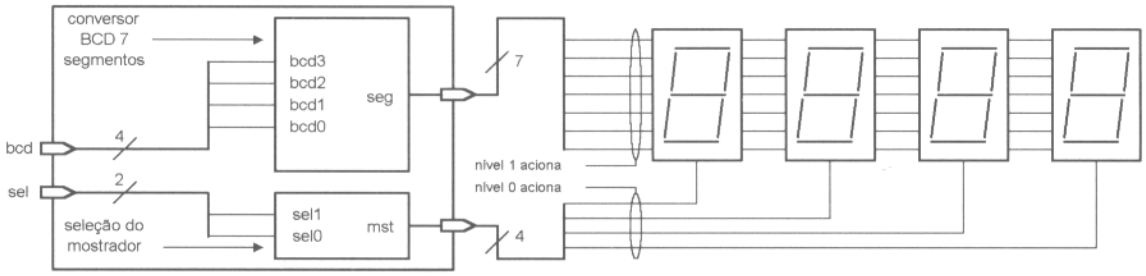


Figura 3.10.5 Ilustração para o Exercício 3.10.9.

3.10.10 Sintetize as descrições criadas para o registrador de quatro bits nos Exercícios 3.10.2 e 3.10.3 e compare os resultados no nível RTL.

3.10.11 Sintetize as descrições criadas para o detector de prioridade nos Exercícios 3.10.4 e 3.10.5 e compare os resultados. Verifique os diagramas gerados no nível RTL e no nível portas lógicas.

3.10.12 Sintetize as descrições criadas para o seletor de memória nos Exercícios 3.10.6 e 3.10.7 e compare os resultados. Verifique os diagramas gerados no nível RTL e no nível portas lógicas.

CAPÍTULO

4 COMANDOS SEQÜENCIAIS BÁSICOS

Comandos seqüenciais ficam contidos em regiões específicas de código como processos ou subprogramas. Uma vez iniciada a execução de comandos em região de código seqüencial, os comandos são avaliados na seqüência em que são apresentados no código, independentemente da troca de valor nos sinais ou variáveis aos quais o comando é sensível.

Este capítulo apresenta os comandos seqüenciais " IF ", " CASE " e " WAIT ", o modo de operação de um processo e a lista de sensibilidade, bem como a atribuição de valores a sinais nas regiões de código seqüencial.

4.1 Lista de sensibilidade em processos

O comando " PROCESS " foi apresentado no Capítulo 3, pois um processo é um comando concorrente. A região delimitada por um processo contém comandos que são executados seqüencialmente, como na maioria das linguagens de programação. Uma descrição pode conter vários processos, e todos esses processos, assim como outros comandos concorrentes, são executados simultaneamente. A Figura 4.1.1 ilustra a execução do código de uma descrição contendo dois processos.

Logo após a palavra reservada " PROCESS ", é possível definir uma lista de sinais que causam a execução do processo. Essa relação de sinais é denominada lista de sensibilidade. A alteração de valor em um dos sinais contidos na lista de sensibilidade ativa o processo, causando a execução dos comandos seqüenciais segundo a ordem de apresentação. Ao término da execução, o processo fica suspenso até a ocorrência de uma nova alteração de valor em um dos sinais relacionados na lista de sensibilidade.

Figura 4.1.1 Regiões de código contendo comandos seqüenciais — PROCESS.

O Quadro 4.1.1 contém uma descrição que ilustra a operação de uma lista de sensibilidade. No processo " abc ", os valores das portas de entrada " a " e " b " são transferidos para as portas de saída " sa " e " sb ", respectivamente. Na lista de sensibilidade do processo foi incluído apenas o sinal " a ". Assim, somente na alteração do valor desse sinal ocorre a execução do processo e, portanto, a atualização dos valores nas portas de saída. Desse modo, o valor da porta de saída " sa " corresponde sempre ao valor da porta " a ", pois qualquer alteração no valor causa a execução do processo (ver Figura 4.1.2). No caso da porta " sb ", o valor é atualizado somente quando ocorre uma alteração no valor de " a ". Assim, o sinal " sb " se comporta como um circuito que memoriza o último valor da porta de entrada " b ", e permanece com o mesmo valor até o instante em que

a porta " a " sofre uma alteração. Conforme se pode observar na figura, a porta de saída " sb " é atualizada nos instantes 100 ns, 200 ns, 500 ns, 600 ns, 800 ns e 900 ns. O efeito de manutenção do valor fica claro nos instantes 500 ns e 800 ns.

A descrição do Quadro 4.1.1 não deve ser considerada como um exemplo para síntese de circuitos que representam um registrador de qualquer espécie. O primeiro motivo deve-se ao fato de que o elemento de memória descrito é ativado tanto na borda de descida como na borda de subida de um sinal, um elemento sem correspondência com circuitos reais. Além disso, as ferramentas de síntese não inferem, necessariamente, a necessidade de um elemento de memória pela ausência de um sinal na lista de sensibilidade; apenas uma mensagem de advertência pode ser gerada. A Figura 4.1.3 e o Quadro 4.1.2 ilustram o problema: o circuito gerado não corresponde ao comportamento da descrição.

```
 1  ENTITY sens_tes IS
 2    PORT (a,  b   :  IN  BIT;
 3          sa, sb  :  OUT BIT);
 4  END sens_tes;
 5
 6  ARCHITECTURE teste OF sens_tes IS
 7  BEGIN
 8    abc: PROCESS (a) -- executado na alteracao do valor de "a"
 9    BEGIN
10      sa <= a;
11      sb <= b;
12    END PROCESS abc;
13  END teste;
```

Quadro 4.1.1 Descrição para o teste de uma lista de sensibilidade em um processo.

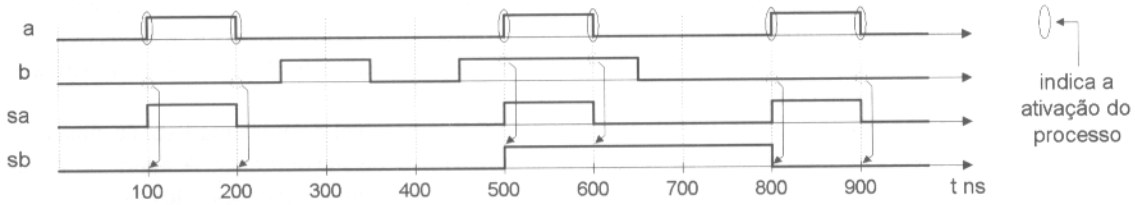


Figura 4.1.2 Simulação da entidade " sens_tes " — destaque para a execução do processo.

Figura 4.1.3 Resultado da síntese da descrição proposta no Quadro 4.1.1 — nível RTL.

```
"/Vhdl/sens_tes.vhd",line 11: Warning, b should be declared on the sensitivity list of the process.
```

Quadro 4.1.2 Mensagem gerada pela ferramenta de síntese — código do Quadro 4.1.1.

4.2 Atribuição de valor para um sinal - região seqüencial

A atribuição incondicional de valor para um sinal pode ocorrer em regiões de código concorrente ou seqüencial. Em uma região de código concorrente, a atribuição é executada como um comando concorrente, isto é, na ocorrência de uma alteração de valor de um sinal da expressão. Numa região de código seqüencial, um processo, por exemplo, uma atribuição é agendada quando o comando é executado.

Uma questão que pode surgir é quanto à influência da ordem de comandos contendo atribuições de sinal em regiões de código seqüencial. Para o entendimento, é preciso recordar do intervalo de tempo Δ, empregado pelo simulador para ordenar a avaliação de eventos que ocorrem no mesmo instante. Alem disso, é necessário saber que na suspensão da execução do processo ocorre a atribuição de valores agendados.

Supondo, por exemplo, que o circuito ilustrado na Figura 4.2.1 seja descrito pelo código apresentado no Quadro 4.2.1. Analisando o código do Quadro 4.2.1, e a Figura 4.2.1, uma alteração do valor no sinal " sa " de " 0 " para " 1 ", em t = 10 ns, causa a execução do processo. Assim, o valor de " sb ", no caso " 0 ", é agendado para ser transferido em 10 ns +Δ, e o valor de " sa ", " 1 ", é agendado para ser transferido em 10 ns +Δ. Após a execução dos comandos, o processo é suspenso e uma iteração ocorre. Como " sb " recebe um novo valor após a iteração, e " sb " está na lista de sensibilidade, o processo é executado novamente, e os dois comandos são reavaliados. O processo é suspenso mais uma vez, e uma nova iteração ocorre em 10 ns +2Δ, resultando em " sb =1 " e " sc =1 ". Como o sinal " sc " não foi incluído na lista de sensibilidade, o processo não é mais executado até uma nova alteração de valor em " sa ". Uma vez que o intervalo de tempo Δ é nulo, o resultado final da simulação corresponde à carta de tempo à direita da figura.

Se o sinal " sb " não fosse incluído na lista de sensibilidade do processo, o comportamento da descrição seria diferente. Neste caso, ocorreria somente uma iteração, e o resultado final da simulação corresponderia à condição da carta de tempo em t = 10 ns +Δ, isto é, " sb =1 " e " sc =0 ". Esse tipo de comportamento não é implementado pelas ferramentas de síntese, e deve ser evitado.

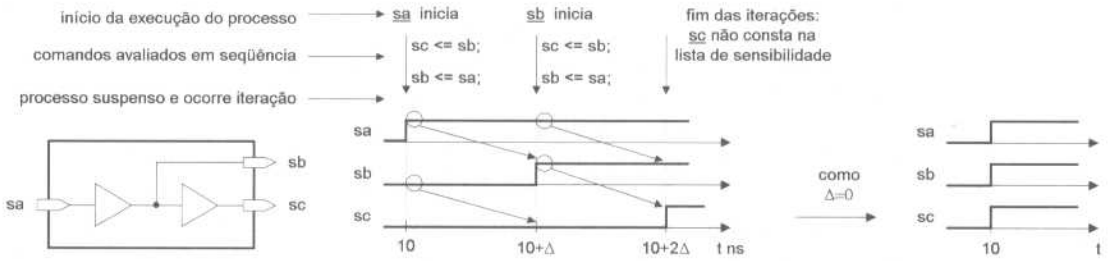


Figura 4.2.1 Eventos na simulação do código do Quadro 4.2.1.

```
1  ENTITY sig_tes1 IS
2    PORT (sa        :  IN  BIT;
3          sb, sc    :  BUFFER BIT);
4  END sig_tes1;
5
6  ARCHITECTURE teste OF sig_tes1 IS
7  BEGIN
8    PROCESS (sa, sb)
9    BEGIN
10    sc <= sb;
11    sb <= sa;
12   END PROCESS;
13 END teste;
```

Quadro 4.2.1 Descrição para teste das iterações em um processo.

A descrição do circuito de seleção do Capítulo 3, empregando a atribuição incondicional de valor para um sinal em região de código seqüencial, pode ser reescrita no formato apresentado no Quadro 4.2.2. Note que os sinais " int0 " a " int3 " devem ser colocados na lista de sensibilidade do processo. A ausência de um desses sinais na lista introduziria um comportamento seqüencial na descrição.

```
1  ENTITY mux_p0 IS
2    PORT (i0, i1, i2, i3     :  IN  BIT;  -- entradas
3          s0, s1             :  IN  BIT;  -- selecao
4          ot                 :  OUT BIT); -- saida
5  END mux_p0;
6
7  ARCHITECTURE teste OF mux_p0 IS
8    SIGNAL int0, int1, int2, int3 : BIT;
9  BEGIN
10   PROCESS (i0, i1, i2, i3, s0, s1, int0, int1, int2, int3)
11   BEGIN
12     ot <= int0 OR int1 OR int2 OR int3;
13     int0 <= i0 AND NOT s1 AND NOT s0;
14     int1 <= i1 AND NOT s1 AND     s0;
15     int2 <= i2 AND     s1 AND NOT s0;
16     int3 <= i3 AND     s1 AND     s0;
17   END PROCESS;
18 END teste;
```

Quadro 4.2.2 Descrição do circuito de seleção — sinais em região de código seqüencial.

4.3 Construção " IF ELSE "

A construção " IF ELSE " permite a execução condicional de um ou mais comandos seqüenciais, segundo uma lista de condições (ver Quadro 4.3.1). As condições são avaliadas na ordem da apresentação no código, e a condição de teste pode ser uma expressão que retorne um valor do tipo " BOOLEAN ". A primeira condição verdadeira encontrada define o conjunto de comandos seqüenciais a ser executado.

O comando " IF " inicia a lista de condições, e pode ser seguido por cláusulas " ELSIF " contendo, também, condições a serem verificadas. Se nenhuma condição verdadeira for encontrada, e existir uma cláusula " ELSE ", o conjunto de comandos que a segue é executado. Se nenhuma condição for verdadeira e não constar uma cláusula " ELSE " na construção, nenhum comando é executado.

As cláusulas " ELSIF " e " ELSE " são opcionais, e o número de cláusulas " ELSIF " não é limitado. Como o comando " IF " é, também, um comando seqüencial, é possível aninhar vários níveis de construções " IF ELSE ", conforme ilustrado no Quadro 4.3.2.

```
IF condicao_1 THEN
  comando_sequencial;
  comando_sequencial;
ELSIF condicao_2 THEN      -- clausula ELSIF opcional
  comando_sequencial;
  comando_sequencial;
ELSIF condicao_3 THEN
  comando_sequencial;
ELSE                       -- clausula ELSE opcional
  comando_sequencial;
END IF;
```

Quadro 4.3.1 Formato da construção " IF ELSIF " " ELSE ".

```
IF condicao_1 THEN
  IF condicao_2 THEN
    comando_sequencial;
  ELSE
    comando_sequencial;
  END IF;
ELSE
  IF condicao_3 THEN
    comando_sequencial;
  ELSE
    comando_sequencial;
  END IF;
END IF;
```

Quadro 4.3.2 Construções " IF ELSE " aninhadas.

O Quadro 4.3.3 contém um exemplo da construção " IF ELSE ". Dois sinais são controlados da seguinte forma: o comando " IF " especifica a primeira condição, a cláusula " ELSIF " especifica as novas condições, e a cláusula " ELSE " engloba as condições restantes. Assim como na construção concorrente " WHEN ELSE ", a construção " IF ELSE " define uma prioridade na ordem das opções, e o circuito equivalente corresponde a uma seqüência de seletores comandados por blocos que detectam as condições (ver Figura 4.3.1). Na construção " IF ELSE ", entretanto, cada circuito que detecta uma condição pode comandar mais de um seletor, pois vários comandos podem ser executados para cada condição.

```
IF na =3 THEN
  s0 <= i0 OR i1;
  s1 <= i3;
ELSIF na =8 OR y ='1' THEN
  s0 <= i1;
  s1 <= i4;
ELSE
  s0 <= i0 AND i1;
  s1 <= i5;
END IF;
```

Quadro 4.3.3 Parte de um código empregando a construção " IF ELSE ".

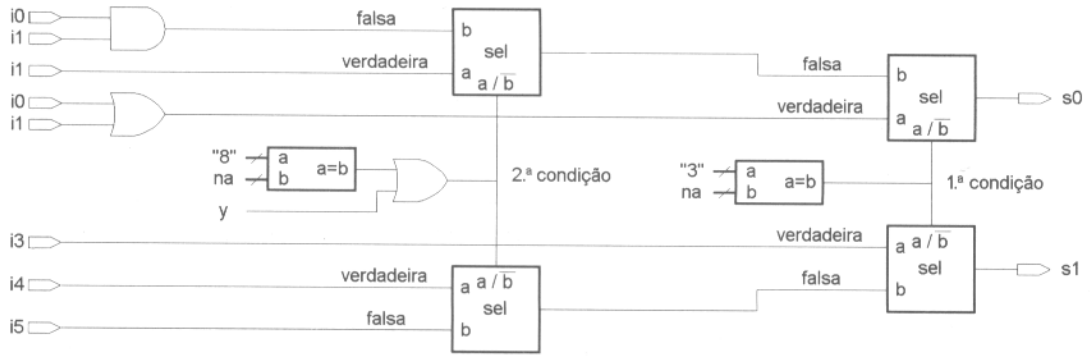


Figura 4.3.1 Circuito equivalente ao código do Quadro 4.3.3.

No Quadro 4.3.4 é apresentada uma nova descrição para o circuito de seleção dos exemplos anteriores, empregando a construção " IF ELSE ". Como o comando " IF " é seqüencial, ele foi inserido em um processo (ver linhas 9 a 18). As portas de entrada " s0 " e " s1 " são concatenadas no sinal " sel ". A lista de sensibilidade do processo deve incluir as portas, " i0 ", " i1 ", " i2 " e " i3 ", e o sinal " sel ". A falta de um desses sinais na lista de sensibilidade tem como conseqüência que o processo não é executado nas alterações de valor do sinal ausente. A descrição, neste caso, teria um comportamento seqüencial, não correspondendo ao circuito de seleção desejado.

```
 1  ENTITY mux_4a IS
 2    PORT (i0, i1, i2, i3  :  IN  BIT;  -- entradas
 3          s0, s1          :  IN  BIT;  -- selecao
 4          ot              :  OUT BIT); -- saida
 5  END mux_4a;
 6
 7  ARCHITECTURE teste OF mux_4a IS
 8    SIGNAL sel : BIT_VECTOR(1 DOWNTO 0);
 9  BEGIN
10    sel <= s1 & s0;
11    abc: PROCESS (i0, i1, i2, i3, sel) --sinal "sel" inserido na lista
12    BEGIN
13      IF    sel = "00" THEN  ot <= i0;
14      ELSIF sel = "01" THEN  ot <= i1;
15      ELSIF sel = "10" THEN  ot <= i2;
16      ELSE                   ot <= i3;
17      END IF;
18    END PROCESS abc;
19  END teste;
```

Quadro 4.3.4 Código VHDL empregando a construção " IF ELSE ".

4.4 Construção " CASE WHEN "

A construção " CASE WHEN " permite a execução condicional de um ou mais comandos seqüenciais, conforme o valor de uma expressão. Cada condição define um conjunto de comandos seqüenciais a ser executado (ver Quadro 4.4.1).

As condições devem ser mutuamente exclusivas, significando que não é permitido mais de uma condição verdadeira na relação de valores que a expressão pode assumir, e todas as condições possuem a mesma prioridade. Caso a execução de um mesmo conjunto de comandos seja condicionada a mais de uma condição, é possível agrupar as condições através do delimitador " | ". Ele equivale a uma operação " ou " entre as condições de escolha. De modo semelhante, as palavras reservadas " TO " e " DOWNTO " podem ser empregadas para delimitar uma faixa de condições. A palavra reservada " OTHERS " pode ser empregada na última condição para agrupar as condições não-relacionadas na lista.

O valor de retorno da expressão de escolha deve ser um tipo discreto ou um vetor unidimensional; tipos " REAL " e " PHYSICAL ", por exemplo, não são permitidos. Outras restrições são aplicáveis com relação à expressão de escolha (ver item 4.7). Sempre que possível, deve-se manter a expressão de escolha simples; expressões complexas devem ser avaliadas isoladamente, e o resultado da operação transferido para o campo da expressão de escolha no comando.

```
CASE expressao_de_escolha IS                    -- expressao_de_escolha =
  WHEN condicao_1                 => comando_a;                    -- condicao_1
  WHEN condicao_2                 => comando_b; comando_c;         -- condicao_2
  WHEN condicao_3 |  condicao_4   => comando_d;                    -- condicao_3 ou condicao_4
  WHEN condicao_5 TO condicao_9   => comando_d;                    -- condicao_5 ate condicao_9
  WHEN OTHERS                     => comando_e; comando_f;         -- condicoes restantes
END CASE;
```

Quadro 4.4.1 Formato da construção " CASE WHEN ".

O Quadro 4.4.2 apresenta um fragmento de código contendo a construção " CASE WHEN ". A expressão de escolha é o sinal " na " do tipo " INTEGER ", e as condições de escolha ilustradas são do tipo: única, faixa de condições e condições restantes. Cada opção de escolha determina a expressão que controla os sinais " s0 " e " s1 ". A construção " CASE WHEN ", assim como a construção concorrente " WITH SELECT ", não identificam nenhuma prioridade nas opções. O circuito equivalente corresponde a um decodificador que detecta as condições contidas no comando e controla dois circuitos de seleção (ver Figura 4.4.1).

```
CASE na IS
  WHEN 3        =>  s0 <= i0 OR i1;   s1 <= i3;
  WHEN 7 TO 12  =>  s0 <= i1;         s1 <= i4;
  WHEN OTHERS   =>  s0 <= i0 AND i1;  s1 <= i5;
END CASE;
```

Quadro 4.4.2 Exemplo de uma construção " CASE WHEN ".

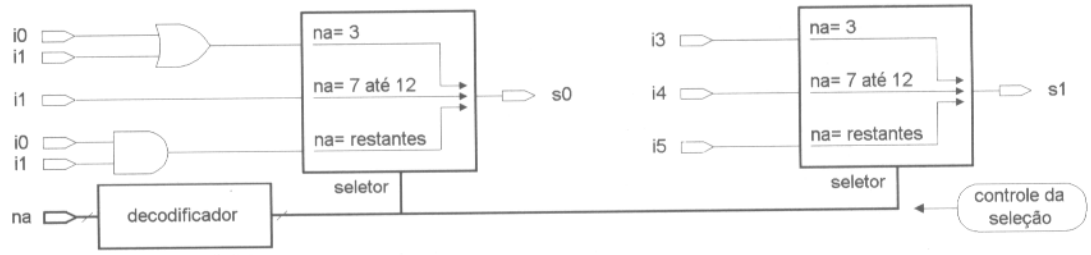


Figura 4.4.1 Circuito equivalente para o código do Quadro 4.4.2.

No Quadro 4.4.3 é apresentada uma nova descrição para o circuito de seleção descrito nos exemplos anteriores. As mesmas observações quanto à lista de sensibilidade, feitas para a descrição do Quadro 4.3.4, são válidas. Observe que a expressão de escolha corresponde ao sinal " sel ", cujo valor é determinado pela concatenação entre " s0 " e " s1 ".

```
1  ENTITY mux_5a IS
2    PORT (i0, i1, i2, i3  :  IN  BIT;
3          s1, s0          :  IN  BIT;
4          ot              :  OUT BIT);
5  END mux_5a;
6
7  ARCHITECTURE teste OF mux_5a IS
8    SIGNAL sel : BIT_VECTOR(1 DOWNTO 0);
9  BEGIN
10   sel <= s1 & s0;
11   abc: PROCESS (i0, i1, i2, i3, sel) --sinal "sel" inserido na lista
12   BEGIN
13     CASE sel IS
14        WHEN "00"   => ot <= i0;
15        WHEN "01"   => ot <= i1;
16        WHEN "10"   => ot <= i2;
17        WHEN OTHERS => ot <= i3;
18     END CASE;
19   END PROCESS abc;
20 END teste;
```

Quadro 4.4.3 Código VHDL — circuito de seleção.

4.5 Comando " WAIT "

O comando " WAIT " suspende a execução de um processo ou de um procedimento. A suspensão pode ser condicionada a três cláusulas, ou a uma combinação delas (ver Quadro 4.5.1). A construção " WAIT ON " fornece um mecanismo equivalente à lista de sensibilidade de um processo. Neste caso, após a execução do comando " WAIT ", o processo fica suspenso até a alteração de valor em um dos sinais relacionados na lista. Na forma " WAIT UNTIL ", o processo é suspenso enquanto a expressão booleana contida no comando não for satisfeita. No último caso, " WAIT FOR ", o processo é suspenso por um período de tempo, reiniciando no comando seguinte, após o período de tempo definido.

```
WAIT ON lista_de_sensibilidade;
WAIT UNTIL expressao_booleana;
WAIT FOR expressao_de_tempo;
WAIT ON lista_sensibilidade UNTIL condicao_booleana FOR expressao_de_tempo;
WAIT;
```

Quadro 4.5.1 Comando " WAIT " e cláusulas associadas.

Não é permitida a ocorrência de um comando " WAIT " em um processo contendo lista de sensibilidade. Entretanto, mais de um comando " WAIT " pode ser inserido num processo. Processos com o comando " WAIT " são úteis em entidades cujo objetivo é o teste de uma outra entidade. Um exemplo desse tipo de aplicação é a geração de sinais de estímulos para verificar o comportamento de uma entidade. Esse tipo de aplicação é abordado no Capítulo 15.

Deve-se observar que nem todas as ferramentas de síntese suportam o comando " WAIT ", e, quando suportado, somente um número restrito de opções são aceitas. As opções normalmente permitidas são: " WAIT ON ", substituindo a lista de sensibilidade de um processo, e " WAIT UNTIL ", empregada na inferência de registradores. Ver o Capítulo 6 para uma amostra do último caso.

O Quadro 4.5.2 ilustra alguns exemplos do comando " WAIT ". No primeiro, o processo é suspenso até a ocorrência de uma alteração de valor no sinal " sig_a " ou no sinal " sig_b ". No segundo exemplo, o processo é suspenso até que o valor de " sig_c " assuma o valor " 1 ". No terceiro exemplo, a execução do comando causa a suspensão do processo por um período de 100 ns. No quarto exemplo, o processo é suspenso aguardando o valor " 0 " no sinal " sig_d " por um intervalo de tempo máximo de 50 ns.

```
WAIT ON sig_a, sig_b;                          -- 1- lista de sensibilidade;
WAIT UNTIL sig_c = '1';                        -- 2- condicao booleana
WAIT FOR 100 ns;                               -- 3- expressao de tempo;
WAIT UNTIL sig_d = '0' FOR 50 ns;              -- 4- condicao boolena por periodo de tempo
```

Quadro 4.5.2 Exemplo de construções " WAIT ".

O código do Quadro 4.5.3 contém dois processos semelhantes empregando o comando " WAIT " com objetivo de ilustrar a execução da instrução. A Figura 4.5.1 apresenta uma carta de tempos da simulação dessa descrição. A cada suspensão de um processo ocorre uma iteração, e os sinais têm os seus valores atualizados. No processo " abc ", o simulador inicia a execução avaliando os comandos das linhas 10, 11 e 12. Após a execução do comando " WAIT ", o processo é suspenso, e ocorre uma iteração: " xa " recebe o valor incrementado, e " ya " o valor " 1 ". Após 30 ns, o processo volta a ser executado, e ocorre a avaliação das linhas 13 e 14. O processo é suspenso, e " ya " recebe o valor " 0 ". Após 20 ns, são executados os comandos das linhas 10, 11 e 12, retornando ao ciclo de operações descrito.

No processo " cde " o primeiro comando causa a suspensão do processo por 30 ns. Após esse período de tempo, o simulador retorna à execução do processo, e os comandos das linhas 19, 20 e 21 são executados. O comando da linha 21 suspende novamente a execução do processo, e os valores de " xb " e " yb " são atualizados. Após 20 ns, são executados os comandos das linhas 22 e 18, e as operações descritas são novamente executadas.

```
 1  ENTITY wait_a IS
 2    PORT (xa, xb  : BUFFER INTEGER RANGE 0 TO 15;
 3          ya, yb  : BUFFER BIT);
 4  END wait_a;
 5
 6  ARCHITECTURE teste OF wait_a IS
 7  BEGIN
 8    abc: PROCESS
 9    BEGIN
10      xa <= xa +1;
11      ya <= '1';
12      WAIT FOR 30 ns;
13      ya <= '0';
14      WAIT FOR 20 ns;
15    END PROCESS abc;
16    cde: PROCESS
17    BEGIN
18      WAIT FOR 30 ns;
19      xb <= xb +1;
20      yb <= '1';
21      WAIT FOR 20 ns;
22      yb <= '0';
23    END PROCESS cde;
24  END teste;
```

Quadro 4.5.3 Descrição para teste do comando " WAIT ".

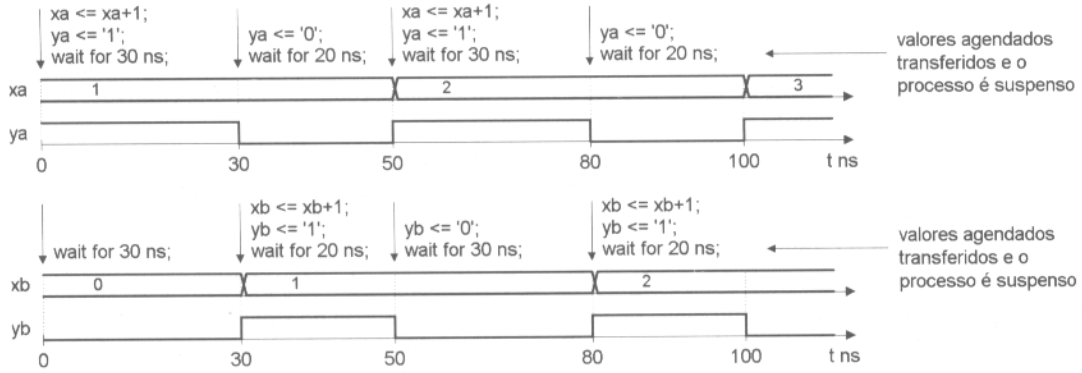


Figura 4.5.1 Simulação da descrição do Quadro 4.5.3.

O Quadro 4.5.4 apresenta uma nova versão para o código proposto no Quadro 4.2.2. A alteração foi a remoção da lista de sensibilidade, e inclusão dos sinais aos quais o processo deve ser sensível no comando " WAIT " (ver linha 16). A execução do processo fica condicionada ao comando " WAIT ON ", isto é, na mudança de valor em um dos sinais contidos na construção " WAIT ON ", a avaliação dos comandos no processo é retomada.

Os códigos dos Quadros 4.5.4 e 4.2.2 são equivalentes do ponto de vista da linguagem. Isto porque um processo contendo uma lista de sensibilidade contém, implicitamente, uma construção " WAIT ON lista_de_sensibilidade " como último comando do processo.

```
 1  ENTITY mux_p2 IS
 2    PORT (i0, i1, i2, i3, s0, s1: IN  BIT;  -- entradas e selecao
 3          ot                    : OUT BIT); -- saida
 4  END mux_p2;
 5
 6  ARCHITECTURE teste OF mux_p2 IS
 7    SIGNAL int0, int1, int2, int3 : BIT;
 8  BEGIN
 9    PROCESS
10    BEGIN
11      ot <= int0 OR int1 OR int2 OR int3;
12      int0 <= i0 AND NOT s1 AND NOT s0;
13      int1 <= i1 AND NOT s1 AND     s0;
14      int2 <= i2 AND     s1 AND NOT s0;
15      int3 <= i3 AND     s1 AND     s0;
16      WAIT ON i0, i1, i2, i3, s0, s1, int0, int1, int2, int3;
17    END PROCESS;
18  END teste;
```

Quadro 4.5.4 Código empregando o comando " WAIT ".

4.6 Comando " NULL "

O comando " NULL " não realiza nenhuma operação; a execução é apenas passada para o próximo comando. Conforme sugestão do manual de referência da linguagem, esse comando é especialmente útil na construção " CASE WHEN ". Como nessa construção é necessário cobrir todos os valores da expressão de escolha, podem ocorrer situações em que nenhuma ação deve ser tomada para uma condição. Assim, para esses casos, pode ser empregado o comando " NULL ".

O Quadro 4.6.1 contém dois exemplos com o comando " NULL "; o código da esquerda emprega a construção " CASE WHEN ", e o da direita emprega a construção " IF ELSE ". O sinal " sel " define qual operação lógica realizada entre os elementos do vetor " z " é transferida para " s ", descrição " null_3a ". O mesmo se aplica à descrição " null_3b ", com relação ao vetor " w ". Nas condições " 00 " e " 10 ", as construções " CASE WHEN " e " IF ELSE " definem uma ação; nas condições restantes, nenhuma operação é executada. Como uma atribuição de sinal para " s " foi estabelecida anteriormente na linha 11 de ambas as descrições, resulta que " s " recebe " z(2) NOR z(0) " nos casos em que o comando " NULL " for executado. Naturalmente, a cláusula " ELSE " na linha 14 da descrição " null_3b " pode ser removida; ela foi inserida apenas para efeito de ilustração do comando.

```
 1  ENTITY null_3a IS
 2    PORT (z   :  IN  BIT_VECTOR(2 DOWNTO 0);
 3          sel :  IN  BIT_VECTOR(1 DOWNTO 0);
 4          s   :  OUT BIT);
 5  END null_3a;
 6
 7  ARCHITECTURE teste OF null_3a IS
 8  BEGIN
 9    abc: PROCESS(z, sel)
10    BEGIN
11      s <= z(2) NOR z(0);
12      CASE sel IS
13        WHEN "00"    => s <= z(0) AND z(1);
14        WHEN "10"    => s <= z(2) XOR z(0);
15        WHEN OTHERS => NULL;
16      END CASE;
17    END PROCESS abc;
18  END teste;
```

```
 1  ENTITY null_3b IS
 2    PORT (w   :  IN  BIT_VECTOR(2 DOWNTO 0);
 3          sel :  IN  BIT_VECTOR(1 DOWNTO 0);
 4          s   :  OUT BIT);
 5  END null_3b;
 6
 7  ARCHITECTURE teste OF null_3b IS
 8  BEGIN
 9    abc: PROCESS(w, sel)
10    BEGIN
11      s <= w(2) NOR w(0);
12      IF    sel ="00" THEN s <= w(0) AND w(1);
13      ELSIF sel ="10" THEN s <= w(2) XOR w(0);
14      ELSE            NULL;
15      END IF;
16    END PROCESS abc;
17  END teste;
```

Quadro 4.6.1 Descrições empregando o comando " NULL ": " null_3a " e " null_3b ".

A Figura 4.6.1 ilustra o resultado da síntese das descrições do Quadro 4.6.1. Nos diagramas esquemáticos, podem-se identificar as portas lógicas que realizam as operações " não ou ", " e " e " ou exclusivo " definidas no código. O valor da saída " s " é estabelecido por um circuito de seleção comandado por " sel " e interligado às portas lógicas.

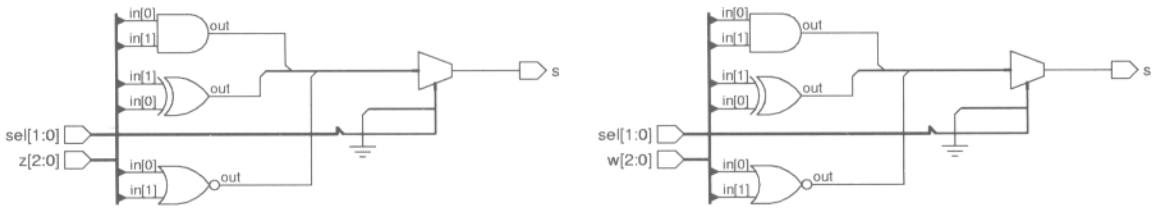


Figura 4.6.1 Resultado da síntese das descrições do Quadro 4.6.1.

A ausência de uma ação pode levar à manutenção de um determinado valor, o que representa a necessidade de um *latch* do ponto de vista de síntese. As descrições apresentadas no Quadro 4.6.2 ilustram o emprego do comando " NULL " nesses casos, através das construções " CASE WHEN " e " IF ELSE ". Em ambas as construções, nenhuma operação é definida nas condições " 01 " e " 11 " devido ao emprego do comando " NULL ". Assim, nessas duas condições, o valor do sinal " s " é mantido. Note, novamente, que a remoção da cláusula " ELSE " na descrição " null_1b " não altera o comportamento da descrição; ela foi introduzida para ilustrar o comando em questão.

```
 1  ENTITY null_1a IS
 2    PORT (x   :  IN  BIT_VECTOR(2 DOWNTO 0);
 3          sel :  IN  BIT_VECTOR(1 DOWNTO 0);
 4          s   :  OUT BIT);
 5  END null_1a;
 6
 7  ARCHITECTURE teste OF null_1a IS
 8  BEGIN
 9    abc: PROCESS(x, sel)
10    BEGIN
11      CASE sel IS
12        WHEN "00"   => s <= x(0) AND x(1);
13        WHEN "10"   => s <= x(2) XOR x(0);
14        WHEN OTHERS => NULL;
15      END CASE;
16    END PROCESS abc;
17  END teste;
```

```
 1  ENTITY null_1b IS
 2    PORT (y   :  IN  BIT_VECTOR(2 DOWNTO 0);
 3          sel :  IN  BIT_VECTOR(1 DOWNTO 0);
 4          s   :  OUT BIT);
 5  END null_1b;
 6
 7  ARCHITECTURE teste OF null_1b IS
 8  BEGIN
 9    abc: PROCESS(y, sel)
10    BEGIN
11      IF    sel ="00" THEN s <= y(0) AND y(1);
12      ELSIF sel ="10" THEN s <= y(2) XOR y(0);
13      ELSE              NULL;
14      END IF;
15    END PROCESS abc;
16  END teste;
```

Quadro 4.6.2 Descrições empregando o comando " NULL ": " null_1a " e " null_1b ".

O resultado da síntese das duas descrições é apresentado na Figura 4.6.2. Como se pode observar, o circuito gerado é o mesmo para ambos os casos, e um *latch* é inserido para manter o valor da saída " s " conforme a condição de " sel ". A condição " sel(0)=0 " leva o *latch* para o estado transparente, isto é " Q=D "; na condição " sel(0)=1 ", o valor presente em " D " é mantido.

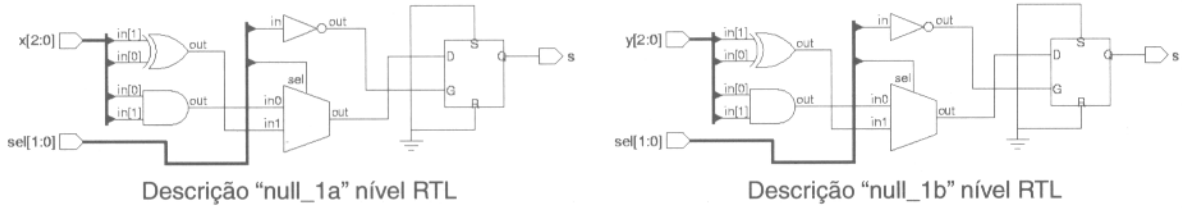


Figura 4.6.2 Resultado da síntese das descrições do Quadro 4.6.2.

4.7 Cuidados na descrição

Comparações entre " WHEN ELSE " e " IF ELSE "

A construção " WHEN ELSE " pode ser empregada unicamente em regiões de código concorrente, e a construção " IF ELSE " em regiões de código seqüencial. Ambas permitem subordinar a atribuição de valor a um sinal segundo um conjunto de condições (ver Quadro 4.7.1). A primeira condição válida detectada no conjunto é a escolhida. No caso da construção " WHEN ELSE ", a operação executada é a simples transferência de um valor para um sinal. A construção " IF ELSE " é mais flexível, pois permite a execução de vários comandos seqüenciais.

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

```
IF    condicao_1 THEN
  sinal_destino <= expressao_a;
ELSIF condicao_2 THEN
  sinal_destino <= expressao_b;
ELSE
  sinal_destino <= expressao_c;
END IF;
```

Quadro 4.7.1 Correspondência entre as construções " WHEN ELSE " e " IF ELSE ".

Comparações entre " WITH SELECT " e " CASE WHEN "

As construções " WITH SELECT " e " CASE WHEN " têm como similaridade o fato de que todas as opções de seleção devem estar presentes na lista de escolha. A construção " WITH SELECT " somente pode ser empregada em regiões de código concorrente, e pode executar apenas a atribuição de um sinal a cada escolha (ver Quadro 4.7.2). A construção " CASE WHEN " possibilita que cada condição de escolha inicie a execução de vários comandos seqüenciais. Essa construção é permitida unicamente em regiões de código seqüencial.

```
WITH expressao_escolha SELECT
 sinal_destino <= expressao_a WHEN condicao_1,
                  expressao_b WHEN condicao_2,
                  expressao_c WHEN OTHERS;
```

```
CASE expressao_escolha IS
   WHEN condicao_1  => sinal_destino <= expressao_a;
   WHEN condicao_2  => sinal_destino <= expressao_b;
   WHEN OTHERS      => sinal_destino <= expressao_c;
END CASE;
```

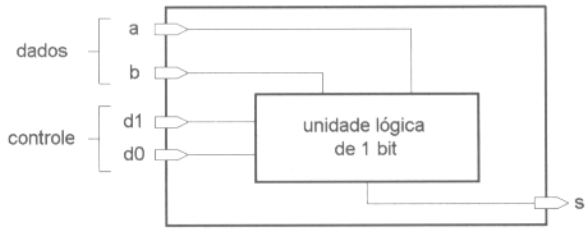
Quadro 4.7.2 Correspondência entre as construções " WITH SELECT " e " CASE WHEN ".

Construções "IF ELSE" e "CASE WHEN" aninhadas

Nas construções " IF ELSE " e " CASE WHEN ", uma condição define um conjunto de comandos seqüenciais a ser executado. Desse modo, essas construções podem ser aninhadas para executar uma decisão. Esse estilo de descrição pode ser muitas vezes evitado, concatenando-se os sinais envolvidos na decisão em um único sinal. Isso resulta em códigos mais claros e reduzem o esforço de minimização da ferramenta de síntese.

A Figura 4.7.1 apresenta o diagrama de uma unidade lógica aritmética de um bit para discussão do problema. As entradas " d1 " e " d0 " definem a operação lógica executada entre dois bits de entrada " a " e " b ".

No Quadro 4.7.3 são apresentadas duas descrições para o circuito empregando as construções " IF ELSE " com e sem aninhamento, assim como no Quadro 4.7.4 as construções " CASE WHEN ", nessas duas condições. Nos casos sem aninhamento, descrições no lado direito do quadro, os sinais que definem a decisão são concatenados em um único sinal, resultando em descrições mais concisas e claras.



s	d1	d0
$\overline{a}$	0	0
a ⊕ b	0	1
a . b	1	0
a + b	1	1

Figura 4.7.1 Diagrama de uma unidade lógica de um bit.

```
 1  ENTITY t1_if1 IS
 2  PORT (d1,d0 : IN  BIT;
 3        a, b  : IN  BIT;
 4        s     : OUT BIT);
 5  END t1_if1;
 6
 7  ARCHITECTURE teste OF t1_if1 IS
 8  BEGIN
 9   PROCESS (d1,d0,a,b)
10   BEGIN
11    IF d1 ='0'THEN
12     IF d0 ='0'THEN s <= NOT a;
13     ELSE           s <= a XOR b;
14     END IF;
15    ELSE
16     IF d0 ='0'THEN s <= a AND b;
17     ELSE           s <= a OR b;
18     END IF;
19    END IF;
20   END PROCESS;
21  END teste;
```

```
 1  ENTITY t1_if2 IS
 2  PORT (d1,d0 : IN  BIT;
 3        a, b  : IN  BIT;
 4        s     : OUT BIT);
 5  END t1_if2;
 6
 7  ARCHITECTURE teste OF t1_if2 IS
 8    SIGNAL d : BIT_VECTOR(1 DOWNTO 0);
 9  BEGIN
10   d <= d1 & d0;
11   PROCESS (d,a,b)
12   BEGIN
13    IF    d ="00" THEN s <= NOT a;
14    ELSIF d ="01" THEN s <= a XOR b;
15    ELSIF d ="10" THEN s <= a AND b;
16    ELSE               s <= a OR b;
17    END IF;
18   END PROCESS;
19  END teste;
```

Quadro 4.7.3 Descrições " t1_if1 " e " t1_if2 ".

```
 1  ENTITY t1_cas1 IS
 2  PORT (d1,d0 : IN  BIT;
 3        a, b  : IN  BIT;
 4        s     : OUT BIT);
 5  END t1_cas1;
 6
 7  ARCHITECTURE teste OF t1_cas1 IS
 8  BEGIN
 9   PROCESS (d1,d0,a,b)
10   BEGIN
11    CASE d1 IS
12     WHEN '0' =>
13      CASE d0 IS
14       WHEN '0' => s <= NOT a;
15       WHEN '1' => s <= a XOR b;
16      END CASE;
17     WHEN '1' =>
18      CASE d0 IS
19       WHEN '0' => s <= a AND b;
20       WHEN '1' => s <= a OR b;
21      END CASE;
22    END CASE;
23   END PROCESS;
24  END teste;
```

```
 1  ENTITY t1_cas2 IS
 2  PORT (d1,d0 : IN  BIT;
 3        a, b  : IN  BIT;
 4        s     : OUT BIT);
 5  END t1_cas2;
 6
 7  ARCHITECTURE teste OF t1_cas2 IS
 8    SIGNAL d : BIT_VECTOR(1 DOWNTO 0);
 9  BEGIN
10   d <= d1 & d0;
11   PROCESS (d,a,b)
12   BEGIN
13    CASE d IS
14     WHEN "00" => s <= NOT a;
15     WHEN "01" => s <= a XOR b;
16     WHEN "10" => s <= a AND b;
17     WHEN "11" => s <= a OR b;
18    END CASE;
19   END PROCESS;
20  END teste;
```

Quadro 4.7.4 Descrições " t1_cas1 " e " t1_cas2 ".

O resultado da síntese no nível RTL para as quatro descrições é ilustrado pelas Figuras 4.7.2 e 4.7.3. As descrições " t1_cas1 " e " t1_if1 " resultam no mesmo diagrama, assim como as descrições " t1_cas2 " e " t1_if2 ". O resultado final, nível porta lógica, não é mostrado. Ele é idêntico para todas as descrições, graças à etapa de otimização da ferramenta de síntese. Algumas características do processo de síntese podem ser identificadas. Nesses exemplos, nas estruturas aninhadas, cada comando " CASE " ou " IF " leva à inserção de um circuito de seleção. No caso das descrições sem aninhamento, a decisão é tomada com um único circuito de seleção.

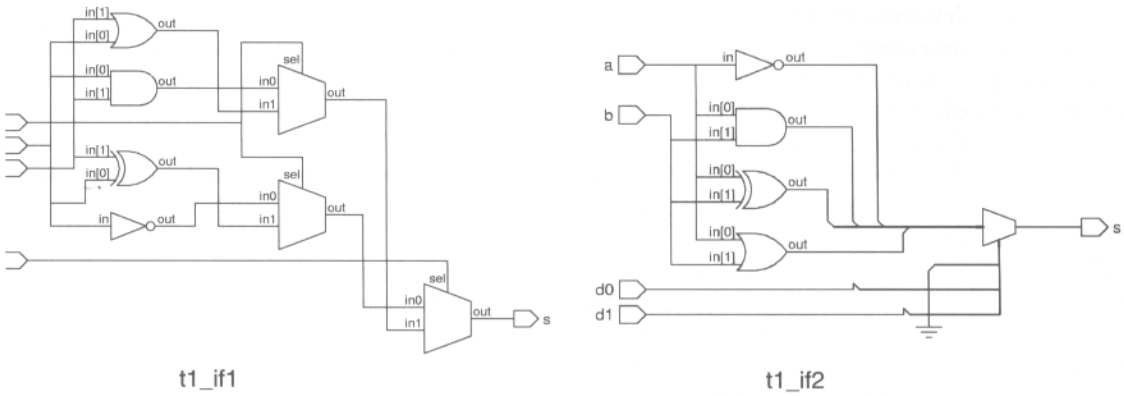


Figura 4.7.2 Descrições " t1_if1 " e " t1_if2 " sintetizadas — nível RTL.

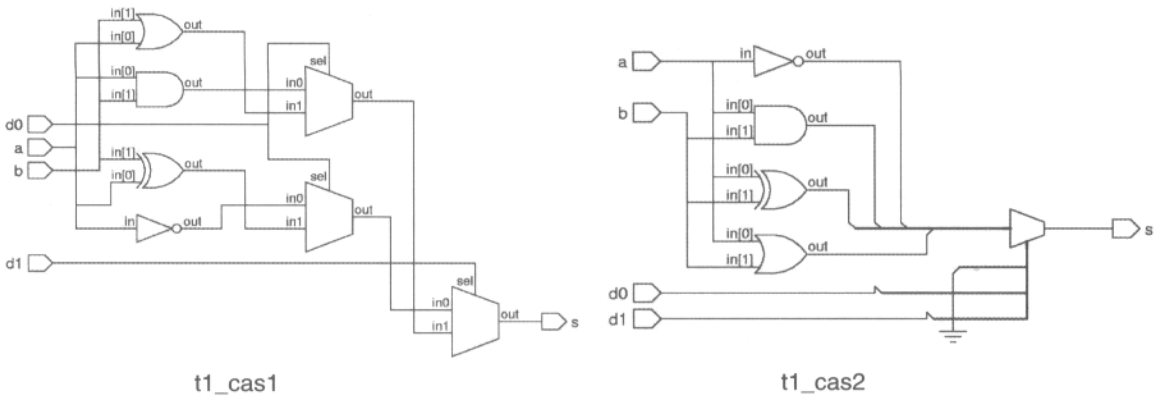
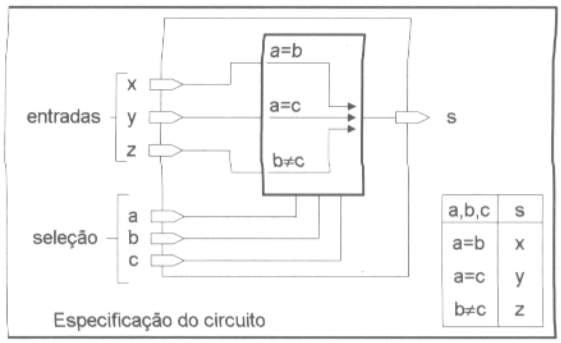


Figura 4.7.3 Descrições " t1_cas1 " e " t1_cas2 " sintetizadas — nível RTL.

O emprego da construção " IF ELSE " e " CASE WHEN "

As restrições impostas pela construção " CASE WHEN " são convenientes para evitar que circuitos seqüenciais sejam criados sem necessidade. Nessa construção, todas as condições possíveis de uma expressão de escolha devem ser testadas. A ausência de uma condição gera uma mensagem de erro na etapa de compilação, advertindo o problema. A construção " IF ELSIF ", por sua vez, não é limitada a uma única expressão de controle, resultando que a cada estágio uma nova expressão pode ser avaliada. Essa liberdade excessiva pode conduzir a erros na descrição quando nem todas as condições são cobertas, por descuido do projetista. A existência de uma condição não-testada pela construção leva à necessidade da inclusão de um elemento de memória para manter a condição anterior. Assim, sempre que possível, é aconselhável o emprego de uma construção " CASE WHEN ".

O diagrama do lado esquerdo da Figura 4.7.4 ilustra como uma especificação imprecisa pode levar a problemas numa descrição. Segundo a proposta do circuito, três condições nas entradas de seleção, " a ", " b " e " c ",



a,b,c	s
a=b	x
a=c	y
b≠c	z

Condições possíveis levantadas

	a	b	c	a=b	a=c	b≠c
0	0	0	0	V	V	F
1	0	0	1	V	F	V
2	0	1	0	F	V	V
3	0	1	1	F	F	F
4	1	0	0	F	F	F
5	1	0	1	F	V	V
6	1	1	0	V	F	V
7	1	1	1	V	V	F

V = verdadeiro
F = falso

Figura 4.7.4 Proposta de um circuito com problemas na especificação.

definem a transferência do valor de uma das entradas, " x ", " y " ou " z ", para a saída " s ". A condição " a=b ", resulta em " s=x ", a condição " a=c " resulta em " s=y ", e a condição " b≠c " resulta em " s=z ".

O levantamento apresentado na tabela da Figura 4.7.4, relacionando as condições especificadas e a combinação de valores nas entradas " a ", " b " e " c ", evidencia dois problemas na especificação. O primeiro deve-se ao fato de que as condições especificadas no diagrama não são mutuamente exclusivas; assim, quando ocorrerem simultaneamente as condições " a=b " e " a=c ", por exemplo, não foi definida a ação do circuito. O segundo problema pode ser observado quando " a,b,c " é igual a " 0,1,1 " ou " 1,0,0 ". Nessas condições de entrada, a operação do circuito é indefinida.

No Quadro 4.7.5 são apresentadas duas possíveis descrições baseadas na especificação proposta, empregando a construção " IF ELSE ". Nessa construção, a ordem das expressões de escolha define a prioridade. Desse modo, para ambas as descrições, no teste da expressão " a=b " são cobertas as condições de número " 0 ", " 1 ", " 6 " e " 7 ", e no teste da expressão " a=c " são cobertas as condições de número " 2 " e " 5 ". Na descrição " dont_if2 " optou-se pela cláusula " ELSE " para cobrir as condições restantes. Analisando a tabela no interior da Figura 4.7.4, observa-se que as condições restantes, número " 3 " e " 4 ", não correspondem à condição " b≠c ". Desse modo, a descrição não corresponde à proposta do circuito.

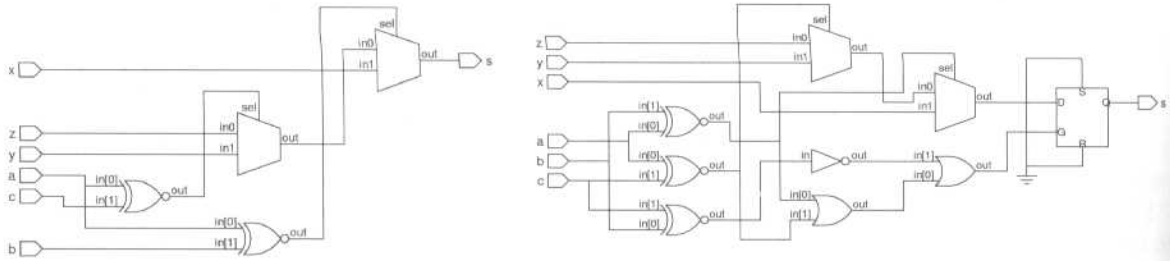
No caso da descrição " dont_if3 ", optou-se pelo teste da expressão " b≠c ". Observando-se a tabela no interior da Figura 4.7.4, fica claro que, uma vez verificadas as expressões " a=b " e " a=c ", a condição " b≠c " nunca é atingida. Assim, nas condições de número " 3 " e " 4 " será mantido o valor na porta de saída " s ", pois a descrição não define nenhuma operação nesse caso.

```
1  ENTITY dont_if2 IS
2  PORT (x,y,z  : IN  BIT;
3        a,b,c  : IN  BIT;
4        s      : OUT BIT);
5  END dont_if2;
6
7  ARCHITECTURE teste OF dont_if2 IS
8  BEGIN
9    PROCESS (a,b,c,x,y,z)
10   BEGIN
11     IF    a=b  THEN s <= x;
12     ELSIF a=c  THEN s <= y;
13     ELSE           s <= z; --b=c
14     END IF;
15   END PROCESS;
16 END teste;
```

```
1  ENTITY dont_if3 IS
2  PORT (x,y,z  : IN  BIT;
3        a,b,c  : IN  BIT;
4        s      : OUT BIT);
5  END dont_if3;
6
7  ARCHITECTURE teste OF dont_if3 IS
8  BEGIN
9    PROCESS (a,b,c,x,y,z)
10   BEGIN
11     IF    a=b  THEN s <= x;
12     ELSIF a=c  THEN s <= y;
13     ELSIF b/=c THEN s <= z; --nunca antingida
14     END IF;
15   END PROCESS;
16 END teste;
```

Quadro 4.7.5 Descrições " dont_if2 " e " dont_if3 ".

O resultado da síntese reflete o problema da especificação e, também, das descrições. No lado esquerdo da Figura 4.7.5 tem-se o diagrama esquemático, nível RTL, resultante da descrição " dont_if2 ". Duas portas "não-ou-exclusivo" detectam as condições de igualdade " a=b " e " a=c ", e a saída dessas portas atua em dois circuitos de seleção. No caso da descrição " dont_if3 ", lado direito da figura, novamente duas portas detectam a condição de igualdade, e uma terceira porta "não-ou-exclusivo" em série com um inversor identifica a condição " b≠c ". Observa-se, também, a inclusão de um *latch* antes da saída " s ". Quando as expressões " a=b " e " a=c " são verdadeiras, a saída do *latch* acompanha o valor de sua entrada " D ", nas condições de número " 3 " e " 4 " o valor é mantido.



" dont_if2 " " dont_if3 "

Figura 4.7.5 Circuitos sintetizados — descrições " dont_if2 " e " dont_if3 ".

Os problemas dessa especificação ficam claros devido à simplicidade da proposta, e ao número reduzido de combinações possíveis nos sinais que definem as expressões. Entretanto, em especificações mais complexas, problemas conforme analisado poderiam passar despercebidos. Neste caso, uma opção pela construção " CASE WHEN " evidenciaria o problema na especificação, e evitaria a inserção de elementos de memória desnecessários.

Expressão de escolha das construções " WITH SELECT " e " CASE WHEN "

Algumas restrições devem ser observadas quanto à expressão de escolha. Dada a característica dessas construções, o compilador deve ser capaz de determinar todos os valores que a expressão de escolha pode assumir. Isso deve ser possível num contexto restrito à expressão.

De uma maneira resumida, a expressão de escolha nas construções " WITH SELECT " e " CASE WHEN " deve:

- ser do tipo discreto, como, por exemplo: " BIT ", " BOOLEAN ", " CHARACTER ", " INTEGER "; tipos " REAL " e " TIME " não são válidos para a expressão de escolha, pois eles não são discretos; ou
- ser um vetor unidimensional composto de caracteres; exemplo, " BIT_VECTOR " ou " STRING " são válidos; um vetor composto de elementos definidos no código não é válido; ou
- ser uma operação predefinida que retorne um valor discreto; exemplo, uma operação de concatenação não é válida, pois retorna um valor do tipo " ARRAY ".

Além disso, o valor de retorno:

- deve ser especificado diretamente na descrição; exemplo, " 101 ", " 7 ", " abc " e constantes; valores especificados por genéricos, sinais ou variáveis não são válidos.

Os Quadros 4.7.6 e 4.7.7 ilustram duas descrições semelhantes empregando a construção " WITH SELECT ". A descrição " cas_sel2 " contém erros, e a descrição " cas_sel3 " ilustra uma possível correção. Os exemplos desses quadros podem ser aplicados, também, para o caso da construção " CASE WHEN ".

```
 1 ENTITY cas_sel2 IS
 2   GENERIC(gen                      : STRING := "eta");
 3   PORT   (i0, i1, b0, b1           : IN  BIT;
 4           c0, c1, c2, c3, sig_chr  : IN  CHARACTER;
 5           t0, t1                   : IN INTEGER;
 6           o0, o1, o2, o3, o4, o5   : OUT BIT);
 7 END cas_sel2;
 8
 9 ARCHITECTURE com_erros OF cas_sel2 IS
10   SIGNAL   sel_stg : STRING(1 TO 3);
11 BEGIN
12 --WITH b1 & b0 SELECT      -- Erro: operacao pre-definida que nao retorna tipo discreto
13 --  o0 <= i0 WHEN "00",
14 --        i1 WHEN OTHERS;
15
16 --WITH c2 & c1 & c0 SELECT -- Erro: operacao pre-definida que nao retorna tipo discreto
17 --  o1 <= i0 WHEN "abc",
18 --        i1 WHEN OTHERS;
19
20   WITH b1 AND b0 SELECT    -- Correto: expressao pre-definida que retorna tipo discreto
21     o2 <= i0 WHEN '0',     -- Bit AND Bit = tipo Bit
22           i1 WHEN '1';
23
24 --sel_stg <= c2 & c1 & c0;
25 --WITH sel_stg SELECT
26 --  o3 <= i0 WHEN gen,     -- Erro: valor definido por generico
27 --        i1 WHEN OTHERS;
28
29 --WITH c3 SELECT
30 --  o4 <= i0 WHEN sig_chr, -- Erro: valor definido por sinal
31 --        i1 WHEN OTHERS;
32
33   WITH t1 + t0 SELECT      -- Correto: expressao pre-definida que retorna tipo discreto
34     o5 <= i0 WHEN 7,       -- Integer + Integer = tipo Integer
35           i1 WHEN OTHERS;
36 END com_erros;
```

Quadro 4.7.6 Descrição com erros: " cas_sel2 ".

```
 1  ENTITY cas_sel3 IS
 2    PORT (i0, i1, b0, b1       : IN  BIT;
 3          c0, c1, c2, c3       : IN  CHARACTER;
 4          o0, o1, o2, o3, o4   : OUT BIT);
 5  END cas_sel3;
 6
 7  ARCHITECTURE correta OF cas_sel3 IS
 8    SIGNAL   sel_bit : BIT_VECTOR(1 DOWNTO 0);
 9    SIGNAL   sel_stg : STRING(1 TO 3);
10    CONSTANT cnt     : STRING := "eta";
11  BEGIN
12    sel_bit <= b1 & b0;
13    WITH sel_bit SELECT      -- Correto: possivel determinar os valores de retorno da expressao
14     o0 <= i0 WHEN "00",     --          no contexto da expressao (00, 01, ... , 11)
15           i1 WHEN OTHERS;
16
17    sel_stg <= c2 & c1 & c0;
18    WITH sel_stg SELECT      -- Correto: possivel determinar os valores de retorno da expressao
19      o1 <= i0 WHEN "abc",   --          no contexto da expressao (... aaa, aab, aac ...)
20            i1 WHEN OTHERS;
21
22    WITH sel_stg SELECT      -- Correto: valor constante
23      o3 <= i0 WHEN cnt,     -- constante cnt := "eta"
24            i1 WHEN OTHERS;
25
26    WITH c3 SELECT
27      o4 <= i0 WHEN 'E',    -- Correto: valor constantte
28            i1 WHEN OTHERS;
29  END correta;
```

Quadro 4.7.7 Descrição corrigida: " cas_sel3 ".

4.8 Principais pontos abordados

- Na construção " IF ELSE ", a seqüência das condições define a prioridade, pois a primeira condição verdadeira encontrada na lista estabelece quais comandos devem ser executados.
- A construção " WHEN ELSE " é uma transferência condicional de sinal concorrente, e a sua correspondente construção seqüencial é " IF ELSE ".
- Na construção " CASE WHEN ", as condições devem ser mutuamente exclusivas; todas as condições devem ser especificadas, e elas possuem a mesma prioridade.
- A construção " WITH SELECT " é uma transferência condicional de sinal concorrente, e a construção seqüencial correspondente é " CASE WHEN ".
- Sempre que possível, deve-se dar preferência a descrições empregando a construção " CASE WHEN " em detrimento da construção " IF ELSE ".
- O comando " WAIT " suspende a execução de um processo. Três tipos de construções podem ser empregados: " WAIT ON ", " WAIT UNTIL ", e " WAIT FOR ".
- O comando " NULL " não realiza nenhuma operação.

4.9 Exercícios

4.9.1 Apresente o comportamento da descrição do Quadro 4.2.1 alterando a linha 8 de duas formas. A primeira substituindo por " PROCESS (sa) " e a segunda por " PROCESS (sa, sb, sc) ". Descreva as iterações que devem ocorrer para cada caso.

4.9.2 A descrição do Quadro 4.9.1 contém erros. Apresente possíveis soluções para os erros e comente as razões.

```
1  ENTITY erro1 IS
2    PORT (a, b :  IN  BIT;
3          c    :  IN  BOOLEAN;
4          x, y, z, k    :  OUT BIT);
5  END erro1;
6
7  ARCHITECTURE teste OF erro1 IS
8  BEGIN
9    abc: PROCESS (a, b, c)
10   BEGIN
11     IF a THEN x <= '1';
12     END IF;
13     IF c THEN z <= '1';
14     END IF;
15     IF a AND b = '1' THEN y <= '0';
16     END IF;
17     IF c AND b = '1' THEN k <= '1';
18     END IF;
19   END PROCESS abc;
20 END teste;
```

Quadro 4.9.1 Código com erros.

4.9.3 A descrição do Quadro 4.9.2 contém erros. Apresente possíveis soluções para os erros e comente os motivos.

```
1  ENTITY erro2 IS
2    PORT (x   :  IN  INTEGER RANGE 0 TO 15;
3          s   :  OUT INTEGER RANGE 0 TO 15);
4  END erro2;
5
6  ARCHITECTURE teste OF erro2 IS
7  BEGIN
8    abc: PROCESS (x)
9    BEGIN
10     CASE x IS
11       WHEN 7 TO 10     => s <= 0;
12       WHEN 3 DOWNTO 0 => s <= 1;
13       WHEN 4 | 5 | 9   => s <= 2;
14       WHEN 15          => s <= 3;
15     END CASE;
16   END PROCESS abc;
17 END teste;
```

Quadro 4.9.2 Código com erros.

4.9.4 A descrição do Quadro 4.9.3 corresponde ao circuito de seleção com uma alteração na lista de sensibilidade. Verifique e comente os problemas acarretados pela alteração.

```
1  ENTITY mux_4b IS
2    PORT (i0, i1, i2, i3  :  IN  BIT;  -- entradas
3          s0, s1          :  IN  BIT;  -- selecao
4          ot              :  OUT BIT); -- saida
5  END mux_4b;
6
7  ARCHITECTURE errada OF mux_4b IS
8  BEGIN
9    abc: PROCESS (i0, i1, i2, i3)  -- sinais s0 e s1 removidos da lista
10   BEGIN
11     IF    s1 & s0 = "00" THEN  ot <= i0;
12     ELSIF s1 & s0 = "01" THEN  ot <= i1;
13     ELSIF s1 & S0 = "10" THEN  ot <= i2;
14     ELSE                       ot <= i3;
15     END IF;
16   END PROCESS abc;
17 END errada;
```

Quadro 4.9.3 Circuito de seleção alterado.

4.9.5 Proponha duas descrições para o esquema da Figura 4.9.1: uma empregando a construção " IF ELSE " e outra com a construção " CASE WHEN ". Três condições estabelecem uma operação lógica entre as entradas " x " e " y ". A condição de maior prioridade, " a=b ", define uma operação "e"; a condição " b=c " define a operação "ou"; e a última condição, " a=c ", de menor prioridade, define uma operação "ou-exclusivo".

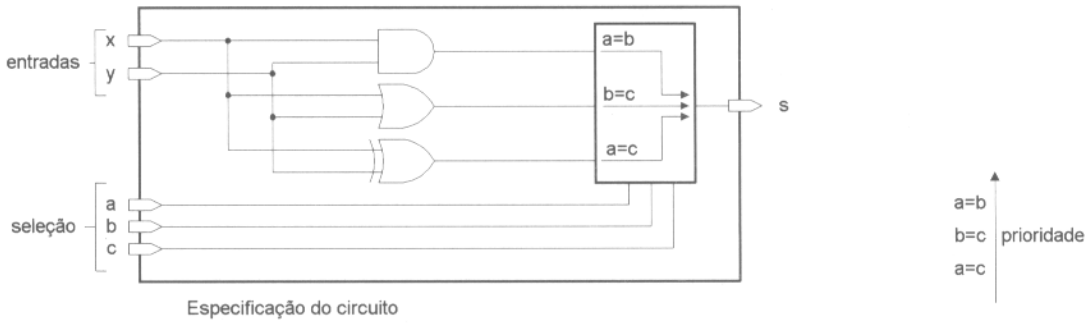


Figura 4.9.1 Proposta de um circuito com problemas na especificação.

4.9.6 Refaça o Exercício 3.10.2 empregando a construção " IF ELSE ".

4.9.7 Refaça o Exercício 3.10.2 empregando a construção " CASE WHEN ".

4.9.8 Refaça o Exercício 3.10.4 empregando a construção" IF ELSE ".

4.9.9 Refaça o Exercício 3.10.4 empregando a construção " CASE WHEN ".

4.9.10 Refaça o Exercício 3.10.6 empregando a construção " IF ELSE ".

4.9.11 Refaça o Exercício 3.10.6 empregando a construção " CASE WHEN ".

4.9.12 Refaça o Exercício 3.10.8 empregando comandos seqüenciais.

4.9.13 Refaça o Exercício 3.10.9 empregando comandos seqüenciais. Não utilize blocos para divisão da descrição; divida a descrição em dois processos: um contendo o decodificador, e o outro contendo o circuito de seleção.

4.9.14 A Figura 4.9.2 apresenta o diagrama de um circuito de deslocamento rápido, *barrel shift*, para discussão do exercício. As entradas " d1 " e " d0 " definem o deslocamento dos bits da entrada " v " para a saída " s ". O deslocamento pode variar entre zero a três bits para a esquerda, conforme descrito na figura.

Apresente duas descrições para a solução empregando as construções " IF ELSE " e " CASE WHEN " aninhadas, e outras duas soluções sem construções aninhadas. Sintetize as descrições e comente os resultados.

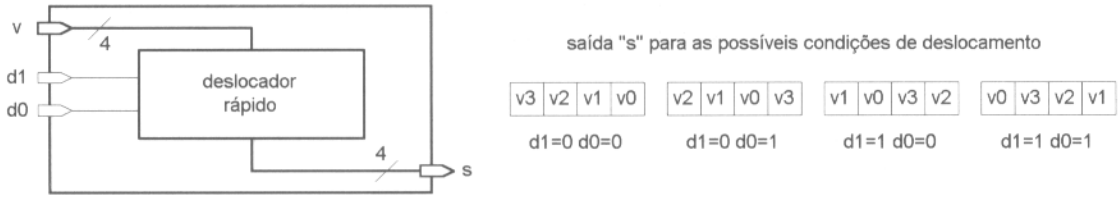


Figura 4.9.2 Diagrama de um deslocador rápido de 4 bits.

4.9.15 Sintetize as descrições dos Quadros 4.3.4 e 4.9.3, entidades " mux_4a " e " mux_4b ", respectivamente. Note que na entidade " mux_4b " dois sinais foram removidos da lista de sensibilidade do processo. Verifique como a ferramenta de síntese trata esse problema. Compare os diagramas gerados no nível RTL e verifique as mensagens geradas pela ferramenta.

4.9.16 A descrição do Quadro 4.9.4 contém erros. Corrija-os e proponha alternativas quando necessário, considerando as restrições quanto à expressão de escolha de uma construção " CASE WHEN ".

```
1  ENTITY cas_sel6 IS
2    GENERIC (gen : INTEGER := 7);
3    PORT (bt     : IN  BIT;
4          btv    : IN  BIT_VECTOR(2 DOWNTO 0);
5          bol    : IN  BOOLEAN;
6          chr    : IN  CHARACTER;
7          int    : IN  INTEGER;
8          rl     : IN  REAL;
9          st     : IN  STRING(1 TO 3);
10         tm     : IN  TIME;
11         o0,o1,o2,o3,o4,o5,o6,o7,o8, o9 : OUT INTEGER);
12 END cas_sel6;
13
14 ARCHITECTURE teste OF cas_sel6 IS
15   CONSTANT cst : INTEGER := 7;
16 BEGIN
17   PROCESS(bt, btv, bol, chr, int, rl, st, tm)
18   BEGIN
19   CASE bt IS
20     WHEN   '0'  => o0 <= 0;
21     WHEN OTHERS => o0 <= 1;
22   END CASE;
23
24   CASE btv IS
25     WHEN "000"  => o1 <= 0;
26     WHEN OTHERS => o1 <= 1;
27   END CASE;
28
29   CASE bol IS
30     WHEN FALSE  => o2 <= 0;
31     WHEN OTHERS => o2 <= 1;
32   END CASE;
33
34   CASE chr IS
35     WHEN 'a'    => o3 <= 0;
36     WHEN OTHERS => o3 <= 1;
37   END CASE;
38
39   CASE int IS
40     WHEN   7    => o4 <= 0;
41     WHEN OTHERS => o4 <= 1;
42   END CASE;
43
44   CASE rl IS
45     WHEN  7.0   => o5 <= 0;
46     WHEN OTHERS => o5 <= 1;
47   END CASE;
48
49   CASE st IS
50     WHEN  "abc" => o6 <= 0;
51     WHEN OTHERS => o6 <= 1;
52   END CASE;
53
54   CASE tm IS
55     WHEN  50 ns => o7 <= 0;
56     WHEN OTHERS => o7 <= 1;
57   END CASE;
58
59   CASE int IS
60     WHEN  gen   => o8 <= 0;
61     WHEN OTHERS => o8 <= 1;
62   END CASE;
63
64   CASE int IS
65     WHEN  cst   => o9 <= 0;
66     WHEN OTHERS => o9 <= 1;
67   END CASE;
68   END PROCESS;
69 END teste;
```

Quadro 4.9.4 Descrição com erros.

CAPÍTULO

5 ATRASO, VARIÁVEL E ATRIBUTOS

O atraso em VHDL é aplicado na geração de formas de onda para teste de descrições ou na modelagem do comportamento dinâmico de um circuito. Os comandos que modelam o atraso são ignorados pelas ferramentas de síntese.

Conforme visto no Capítulo 2, sinal e variável são objetos que contêm um valor armazenado, e o valor armazenado pode ser alterado na execução do código. Esses dois objetos têm características distintas, que serão abordadas com mais detalhes neste capítulo.

O conceito de atributo disponível na linguagem permite extrair informações adicionais de um objeto, tipo e até unidades de projeto. Considerando um sinal, por exemplo, através de atributos, é possível determinar se ocorreu uma troca de valor no objeto, o tempo decorrido após essa mudança e o valor anterior do objeto.

5.1 Atraso

Na linguagem VHDL é possível definir um intervalo de tempo, para modelar o atraso em uma descrição. A modelagem pode ser com inércia ou com transporte. O modelo com inércia é indicado para representar uma informação que se propaga através de um circuito lógico. Normalmente, nesses elementos, pulsos com duração menor do que o atraso de propagação do circuito não são transferidos para a saída. No modelo com transporte a informação é sempre transmitida, não importando a duração do pulso.

A cláusula " AFTER " seguida de um valor de tempo modela um atraso na transferência de uma informação. Caso não seja inserida a opção " TRANSPORT ", será suposto um atraso do modelo com inércia. O Quadro 5.1.1 apresenta alguns exemplos do formato das cláusulas " AFTER " e " TRANSPORT ".

A modelagem do atraso não é sintetizada pelas ferramentas de síntese devido à impossibilidade de implementar circuitos que gerem esse comportamento com precisão. As cláusulas presentes no código são ignoradas pelas ferramentas de síntese.

```
sinal_a <= valor_x AFTER tempo_1;
sinal_b <= valor_y, valor_z AFTER tempo_2, valor_k AFTER tempo_3;
sinal_c <= expressao_z AFTER tempo_4, expressao_k AFTER tempo_5;

sinal_d <= TRANSPORT valor_x AFTER tempo_1;
sinal_e <= TRANSPORT valor_y, valor_z AFTER tempo_2, valor_k AFTER tempo_3;
sinal_f <= TRANSPORT expressao_z AFTER tempo_4;
```

Quadro 5.1.1 Formato das cláusulas " AFTER " e " TRANSPORT ".

No Quadro 5.1.2 é apresentado um código contendo comandos de atraso com inércia e com transporte. A Figura 5.1.1 ilustra o resultado da simulação dessa entidade onde dois pulsos são aplicados: um com duração maior do que o atraso apresentado na cláusula " AFTER ", e outro com duração menor. No caso do sinal " b ", modelagem com inércia, o pulso com duração menor do que o atraso definido não é transferido para a saída. Para o sinal " c ", o pulso é sempre transferido para a saída.

```
1  ENTITY c04_atr1 IS
2    PORT (a    :  IN  BIT;
3          b, c :  OUT BIT);
4  END c04_atr1;
5
6  ARCHITECTURE teste OF c04_atr1 IS
7   BEGIN
8     b <= a AFTER 100 ns;
9     c <= TRANSPORT a AFTER 100 ns;
10 END teste;
```

Quadro 5.1.2 Código VHDL de uma descrição com modelagem de atraso.

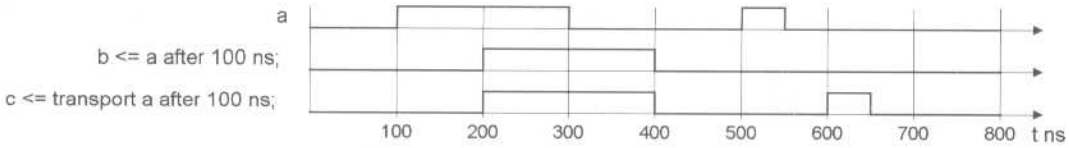


Figura 5.1.1 Simulação da entidade " t_atraso ".

Uma outra opção interessante de emprego da cláusula " AFTER " é a geração de formas de onda. Naturalmente, essa aplicação não é possível para síntese, mas pode auxiliar o teste de uma descrição, ou geração de estímulos de teste para uma outra entidade (ver Capítulo 15). Um exemplo dessa aplicação é ilustrado no Quadro 5.1.3. Nessa descrição, semelhante à descrição anterior, a porta " a " foi substituída por um sinal interno que gera a mesma forma de onda proposta na Figura 5.1.1, (ver linha 8).

```
1  ENTITY c04_atr2 IS
2    PORT (a    :  BUFFER BIT;
3          b, c :  OUT BIT);
4  END c04_atr2;
5
6  ARCHITECTURE teste OF c04_atr2 IS
7   BEGIN
8     a <= '0', '1' AFTER 100 ns, '0' AFTER 300 ns, '1' AFTER 500 ns, '0' AFTER 550 ns;
9     b <= a AFTER 100 ns;
10    c <= TRANSPORT a AFTER 100 ns;
11 END teste;
```

Quadro 5.1.3 Código empregando a cláusula " AFTER " para geração de forma de onda.

5.2 Sinal e variável

Conforme visto, um objeto é um elemento que armazena um valor. Um sinal é um objeto, e pode ser empregado tanto em regiões de código seqüencial como em regiões de código concorrente. Um outro aspecto a ser recordado é quanto à atribuição de valor a um sinal: ela ocorre após um intervalo de tempo de 1Δ, na iteração dos comandos concorrentes.

Variável é um objeto empregado em regiões de código seqüencial, e sua declaração é feita nessas regiões.[1] A transferência de um valor armazenado numa variável para uma região de código concorrente emprega um sinal. Diferentemente dos sinais, os valores das variáveis são assumidos imediatamente após a avaliação do comando.

No Quadro 5.2.1 é apresentado o código de uma entidade contendo dois processos que executam operações similares empregando sinais e variáveis. Observe que os processos não contêm uma lista de sensibilidade; a execução é controlada pelo comando " WAIT " nas linhas 13 e 21. Após uma primeira execução, os processos são repetidamente executados a cada 10 ns. Nos dois processos são realizadas as mesmas operações entre sinais " sa " e " sx ", e a variável " va " e o sinal " vx ". O sinal " sa " e a variável " va " são definidos com valor inicial igual a " 1 ". Como não é possível definir um sinal dentro de um processo, " sa " foi definido externamente.

[1]Na versão VHDL-1993, é possível declarar variáveis na região de declarações de uma arquitetura (ver Capítulo 8.6).

```
 1  ENTITY sg_var8 IS
 2    PORT (sx    :  OUT INTEGER;     -- saida teste sinal
 3          vx    :  OUT INTEGER);    -- saida teste variavel
 4  END sg_var8;
 5
 6  ARCHITECTURE teste OF sg_var8 IS
 7    SIGNAL sa      : INTEGER := 1;
 8  BEGIN
 9    sig: PROCESS
10    BEGIN
11      sa <= sa +sa;
12      sx <= sa;
13      WAIT FOR 10 ns;
14    END PROCESS sig;
15
16    var: PROCESS
17    VARIABLE va      : INTEGER :=1;
18    BEGIN
19      va := va +va;
20      vx <= va;
21      WAIT FOR 10 ns;
22    END PROCESS var;
23  END teste;
```

Quadro 5.2.1 Exemplo do comportamento de sinais e variáveis.

As Figuras 5.2.1 e 5.2.2 ilustram as iterações que ocorrem nos dois processos. Conforme visto, os comandos contidos em um processo são executados seqüencialmente, e a atribuição de um valor para um sinal ocorre 1Δ após, na suspensão do processo.

Considerando a primeira atividade do processo " sig ", na execução da linha 11 é agendada a atribuição do valor " 2 " para " sa " no intervalo de tempo 0 ns +Δ. Na execução da linha 12 o valor de " sa " ainda não foi alterado, e fica agendado para " sx " receber o valor " 1 " no intervalo de tempo 0 ns +Δ. Na execução da linha 13 o processo é suspenso, e ocorre uma iteração, resultando " sa=2 " e " sx=1 ". Após 10 ns, a execução do processo é retomada, repetindo-se as etapas descritas.

No caso da primeira atividade do processo " var ", a variável recebe o valor " 2 " na execução da linha 19, diferentemente de um sinal que receberia o valor somente 1Δ após. Ao ser executada a linha seguinte, o valor " 2 " é agendado para ser transferido no instante de tempo 0 ns +Δ, pois, nesse caso, " vx " é um sinal. Ao ser executada a linha 21, o processo é suspenso e ocorre uma iteração; o sinal " vx " recebe o valor " 2 ". Após 10 ns, a execução do processo é retomada, retornando-se a seqüência exposta.

Figura 5.2.1 Iterações da descrição do Quadro 5.2.1.

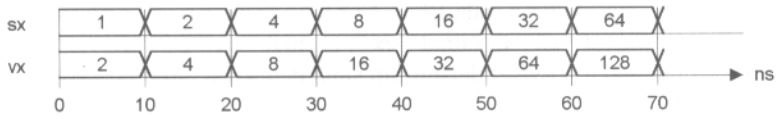


Figura 5.2.2 Carta de tempos da simulação da descrição do Quadro 5.2.1.

5.3 Atributos

Atributos são informações adicionais associadas a tipos, objetos, subprogramas e unidades de projeto. Um objeto, por exemplo, pode conter somente um valor num determinado instante de tempo, mas pode, contudo, possuir vários atributos não necessariamente relacionados com o valor que ele contém. Um atributo pode ser um valor, uma função, um tipo, uma constante, um sinal, ou a extensão de um vetor.

Os atributos podem ser predefinidos ou definidos pelo usuário. Dos atributos predefinidos na linguagem, apenas uma parcela é suportada pelas ferramentas de síntese. Dentre os predefinidos, serão apresentados os atributos para sinais, vetores, tipos escalares e tipos.

O atributo de determinado item é referenciado na forma " prefixo'nome_atributo ". O " prefixo " corresponde ao item, o " nome_atributo " é o atributo desejado do item, e o apóstrofo realiza a separação entre o " prefixo " e o " nome_atributo ".

Antes da definição dos atributos predefinidos, é conveniente a apresentação dos termos relativos a sinais " active ", " quiet " e " event ". O termo " active ", ou ativo, é empregado, caso seja atribuído um novo valor a um sinal, durante um ciclo da simulação, mesmo que o novo valor seja idêntico ao anterior. Os instantes 400 ns, 800 ns, 1200 ns e 1600 ns, na Figura 5.3.1, ilustram esse caso. A condição " quiet ", ou quieto, é oposta à condição " active ", isto é, o sinal não é ativo no ciclo atual da simulação. O evento, ou " event ", em um sinal representa uma mudança no valor a ser assumido (ver instantes 800 ns e 1200 ns na Figura 5.3.1).

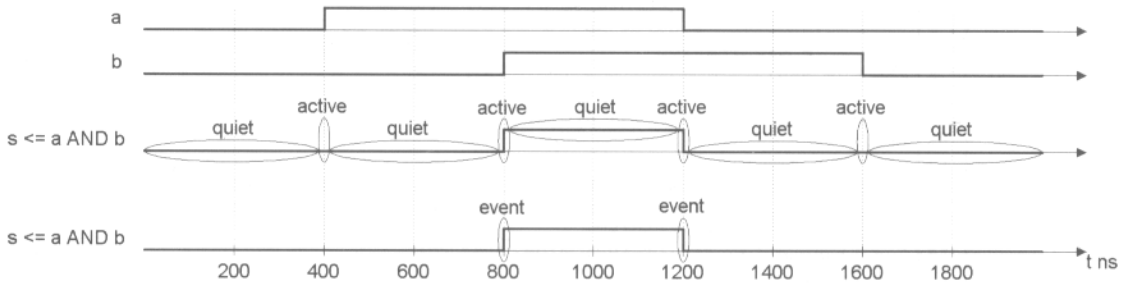


Figura 5.3.1 Condições " active ", " quiet " e " event " de um sinal.

Atributos predefinidos relativos a sinais que resultam em novos sinais

A Tabela 5.3.1 apresenta atributos predefinidos de sinais que têm como resultado novos sinais quando referenciados. Eles podem ser empregados para detectar atribuição de valores ou troca de valores em sinais. O resultado pode ser um sinal do mesmo tipo, do tipo " BOOLEAN " ou do tipo " BIT ", conforme o atributo. O Quadro 5.3.1 apresenta uma descrição para o teste desses atributos, e a Figura 5.3.2 ilustra algumas condições de teste de interesse.

Os atributos relacionados na Tabela 5.3.1 não são normalmente suportados pelas ferramentas de síntese. A síntese de atrasos com valores específicos, conforme determinado pelo atributo " DELAYED ", não é possível de ser sintetizada com precisão. O atributo " QUIET " detecta se não ocorreu a atribuição de um novo valor por um período de tempo. Nesse caso, a síntese envolve dois problemas: a atribuição de um novo valor não representa necessariamente a troca do valor, e exigência do cômputo de um intervalo de tempo. Uma exceção é o atributo " STABLE "; ele pode ser suportado pelas ferramentas de síntese em construções para detectar bordas de subida ou descida em sinais, quando empregado em conjunto com o comando " IF " ou o comando " WAIT ". Deve-se observar, entretanto, que o uso dos atributos não suportados em síntese pode ser de grande valia em descrições para teste de outras entidades.

Tabela 5.3.1 Exemplos de atributos predefinidos para sinais que resultam em novos sinais.

atributo	descrição
s'DELAYED [t]	novo sinal equivalente a " s ", atrasado " t " unidades de tempo
s'STABLE [t]	novo sinal tipo " boolean ", verdadeiro se não ocorreu uma troca de valor por um período de tempo "t"; falso, caso contrário
s'QUIET [t]	novo sinal tipo " boolean ", verdadeiro se nenhum valor foi atribuído por um período de tempo " t "; falso, caso contrário
s'TRANSACTION	novo sinal tipo " bit ", valor complementado com relação ao anterior a cada atribuição de valor
Nota: O parâmetro [t] corresponde ao valor de tempo; se omitido subentende-se " t= 0 s ".	

```
 1  ENTITY c04_at0 IS
 2    PORT (a, b                          : IN  BIT;
 3          s                             : BUFFER BIT;
 4          s_delayed, s_delayed_75       : OUT BIT;
 5          s_stable, s_stable_50         : OUT BOOLEAN;
 6          s_quiet, s_quiet_50           : OUT BOOLEAN;
 7          s_transaction                 : OUT BIT);
 8  END c04_at0;
 9
10  ARCHITECTURE teste OF c04_at0 IS
11  BEGIN
12    s <= a AND b;
13    s_delayed     <= s'DELAYED;    s_delayed_75 <= s'DELAYED(75 ns);
14    s_stable      <= s'STABLE;     s_stable_50  <= s'STABLE(50 ns);
15    s_quiet       <= s'QUIET;      s_quiet_50   <= s'QUIET(50 ns);
16    s_transaction <= s'TRANSACTION;
17  END teste;
```

Quadro 5.3.1 Descrição para teste de atributos relativos a sinais que resultam em novos sinais.

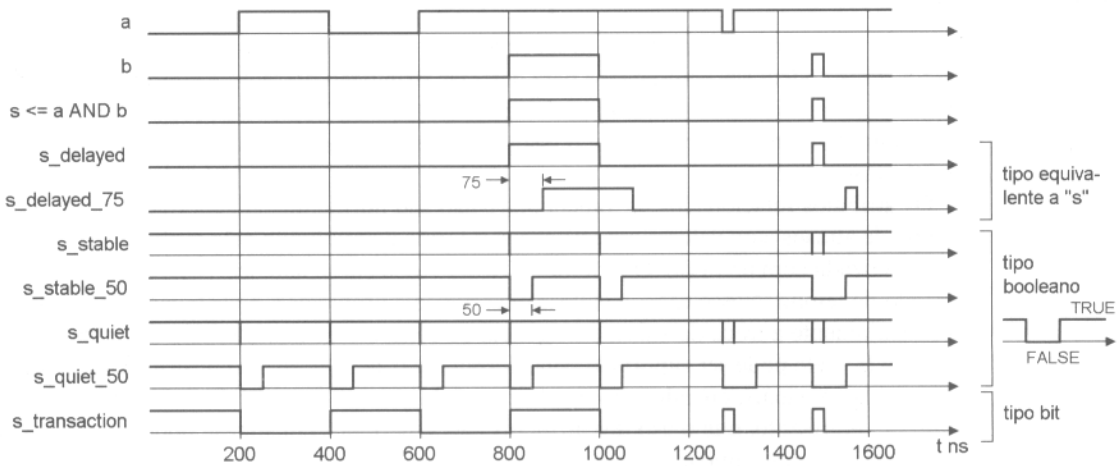


Figura 5.3.2 Atributos de sinais que resultam em novos sinais quando referenciados.

Atributos predefinidos relativos a sinais que não resultam em novos sinais

Na Tabela 5.3.2 são relacionados atributos predefinidos de sinais que não resultam em novos sinais quando referenciados. Assim como os atributos relacionados na Tabela 5.3.1, eles podem ser empregados para detectar atribuição de valores ou troca de valores em sinais. O Quadro 5.3.2 apresenta uma descrição para o teste desses atributos, e a Figura 5.3.3 ilustra algumas condições de teste de interesse.

As ferramentas de síntese suportam unicamente o atributo " EVENT " em conjunto com o comando " IF " ou o comando " WAIT ". Assim como o atributo " STABLE ", ele é empregado na detecção de bordas de subida ou descida de sinais.

Tabela 5.3.2 Exemplos de atributos predefinidos para sinais que não resultam em novos sinais.

atributo	descrição
s'EVENT	verdadeiro, se ocorreu uma troca de valor no ciclo corrente da simulação; falso, caso contrário
s'ACTIVE	verdadeiro, se foi atribuído um valor durante o ciclo corrente da simulação; falso, caso contrário
s'LAST_VALUE	valor do sinal antes do último evento
s'LAST_EVENT	tempo decorrido desde a última troca de valor do sinal

```
 1  ENTITY c04_at1 IS
 2    PORT (a, b                        : IN  BIT;
 3          s                           : BUFFER BIT;
 4          s_event, s_active           : BUFFER BOOLEAN;
 5          s_last_event                : BUFFER TIME;
 6          s_last_value                : BUFFER BIT);
 7  END c04_at1;
 8
 9  ARCHITECTURE teste OF c04_at1 IS
10    SIGNAL a_d, b_d  : BIT;
11  BEGIN
12    s <= a AND b;
13    a_d <= a;  b_d <=b;  -- sinais ativos no mesmo ciclo delta de "s"
14
15    abc: PROCESS(a_d, b_d, s_event)
16    BEGIN
17      IF s'EVENT THEN  s_event <= TRUE;
18      ELSE             s_event <= FALSE;
19      END IF;
20    END PROCESS;
21
22    def: PROCESS (a_d, b_d, s_active)
23    BEGIN
24      IF s'ACTIVE THEN s_active <= TRUE;
25      ELSE             s_active <= FALSE;
26      END IF;
27    END PROCESS;
28
29    ghi: PROCESS(s, s_last_value)
30    BEGIN
31      IF s'LAST_VALUE = '1' THEN s_last_value <= '1';
32      ELSE                       s_last_value <= '0';
33      END IF;
34    END PROCESS;
35
36    jkl: PROCESS
37    BEGIN
38      WAIT FOR 150 ns;
39      s_last_event <= s'LAST_EVENT;
40    END PROCESS;
41  END teste;
```

Quadro 5.3.2 Descrição para teste de atributos relativos a sinais que resultam em novos sinais.

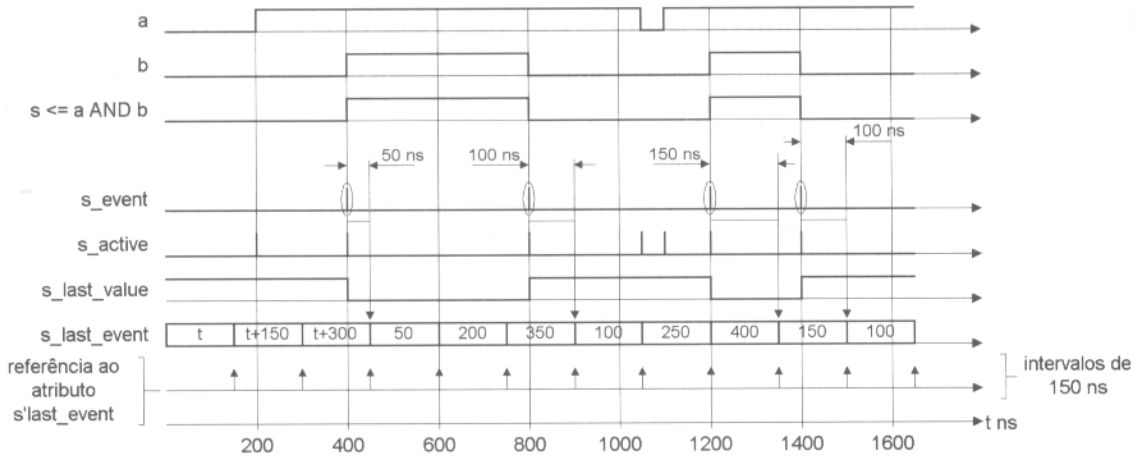


Figura 5.3.3 Atributos de sinais que não resultam em novos sinais quando referenciados.

Atributos predefinidos relativos a vetores

Os atributos predefinidos para vetores permitem determinar características, como número de elementos, limites e faixas em vetores. Os vetores podem conter mais de uma dimensão (ver Capítulo 12). Devido a esse fato, o parâmetro opcional " n " pode ser empregado para especificar a dimensão. O parâmetro é um tipo inteiro; quando omitido, é atribuído o valor " 1 ". Na Tabela 5.3.3 são relacionados os atributos normalmente suportados pelas ferramentas de síntese.

Tabela 5.3.3 Exemplos de atributos predefinidos para vetores.

atributo	descrição
a'HIGH[n]	limite superior da dimensão " [n] " no vetor
a'LOW[n]	limite inferior da dimensão " [n] " no vetor
a'RIGHT[n]	limite direito da dimensão " [n] " no vetor
a'LEFT[n]	limite esquerdo da dimensão " [n] " no vetor
a'LENGTH[n]	número de elementos da dimensão " [n] " no vetor
a'RANGE[n]	faixa do vetor da dimensão " [n] ": (x DOWNTO y) ou (y TO x)
a'REVERSE_RANGE[n]	faixa oposta à obtida com o atributo " RANGE "
Nota: O parâmetro " [n] " é opcional, e indica a dimensão do vetor; quando omitido, subentende-se o valor " 1 ".	

O Quadro 5.3.3 apresenta uma descrição contendo exemplos de referências a atributos relativos a vetores. Note que " a'Left = a'Low " e " a'Right = a'High " para designações crescentes, e " a'Left = a'High " e " a'Right = a'Low " para designações decrescentes. A Figura 5.3.4 ilustra alguns exemplos do Quadro 5.3.3.

```
1  ENTITY c04_at2 IS
2    PORT (a          : IN  BIT_VECTOR(3 TO     8);
3          b          : IN  BIT_VECTOR(9 DOWNTO 2);
4          a_right, b_right, a_left, b_left : OUT INTEGER;
5          a_high,  b_high,  a_low,  b_low  : OUT INTEGER;
6          c_length, d_right                : OUT INTEGER);
7  END c04_at2;
8
9  ARCHITECTURE teste OF c04_at2 IS
10   SIGNAL c : BIT_VECTOR(a'RANGE);              -- a'RANGE = 3 TO 8
11   SIGNAL d : BIT_VECTOR(a'REVERSE_RANGE);      -- a'REVERSE_RANGE = 8 DOWNTO 3
12 BEGIN
13   a_right  <= a'RIGHT;   b_right <= b'RIGHT; -- a'RIGHT =8  b_right =2
14   a_left   <= a'LEFT;    b_left  <= b'LEFT;  -- a'LEFT =3   b_left =9
15   a_high   <= a'HIGH;    b_high  <= b'HIGH;  -- a'HIGH =8   b_high =9
16   a_low    <= a'LOW;     b_low   <= b'LOW;   -- a'LOW =3    b_low=2
17   c_length <= c'LENGTH;                      -- c'LENGTH =6
18   d_right  <= d'RIGHT;                       -- d'RIGHT =3
19 END teste;
```

Quadro 5.3.3 Descrição para teste de atributos relativos a vetores.

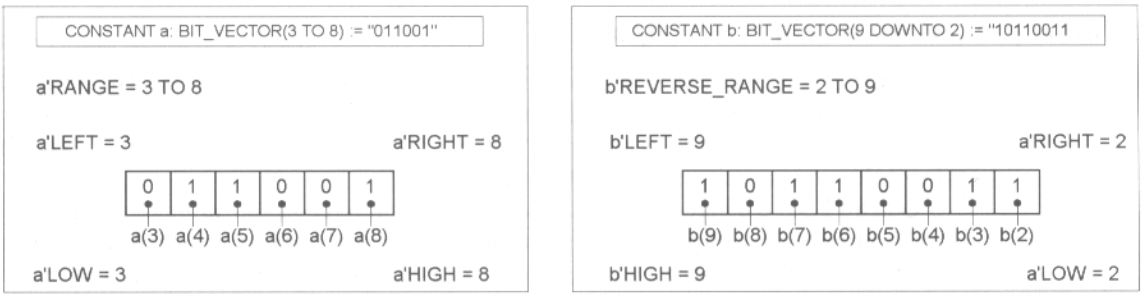


Figura 5.3.4 Exemplos de atributos relativos a vetores.

No Quadro 5.3.4 é apresentada uma descrição com um vetor de três dimensões. Um tipo, denominado " lista ", é definido na linha 9. O sinal " a " é declarado segundo esse tipo, sendo, desse modo, necessário o emprego do parâmetro " [n] " para especificar a dimensão referenciada. Uma visão do vetor criado é apresentada na Figura 5.3.5.

```
1  ENTITY c04_at2a IS
2    PORT (a1_rg, a2_rg, a3_rg, a1_le, a2_le, a3_le : OUT INTEGER;
3          a1_hi, a2_hi, a3_hi, a1_lo, a2_lo, a3_lo : OUT INTEGER;
4          a1_ln, a2_ln, a3_ln                      : OUT INTEGER);
5  END c04_at2a;
6
7  ARCHITECTURE teste OF c04_at2a IS
8    --       dimensao    ...1...    ...2...    ...3...
9    TYPE lista IS ARRAY (10 to 17, 20 to 27, 30 to 37) OF BIT;
10   SIGNAL a : lista;
11 BEGIN
12   a1_rg <= a'RIGHT(1);  a2_rg <= a'RIGHT(2);  a3_rg <= a'RIGHT(3);  -- a'RIGHT_ =17, 27, 37
13   a1_le <= a'LEFT(1);   a2_le <= a'LEFT(2);   a3_le <= a'LEFT(3);   -- a'LEFT_  =10, 20, 30
14   a1_hi <= a'HIGH(1);   a2_hi <= a'HIGH(2);   a3_hi <= a'HIGH(3);   -- a'HIGH_  =17, 27, 27
15   a1_lo <= a'LOW(1);    a2_lo <= a'LOW(2);    a3_lo <= a'LOW(3);    -- a'LOW_   =10, 20, 30
16   a1_ln <= a'LENGTH(1); a2_ln <= a'LENGTH(2); a3_ln <= a'LENGTH(3); -- a'LENGTH_ =8,  8,  8
17 END teste;
```

Quadro 5.3.4 Descrição para teste de atributos relativos a vetores com três dimensões.

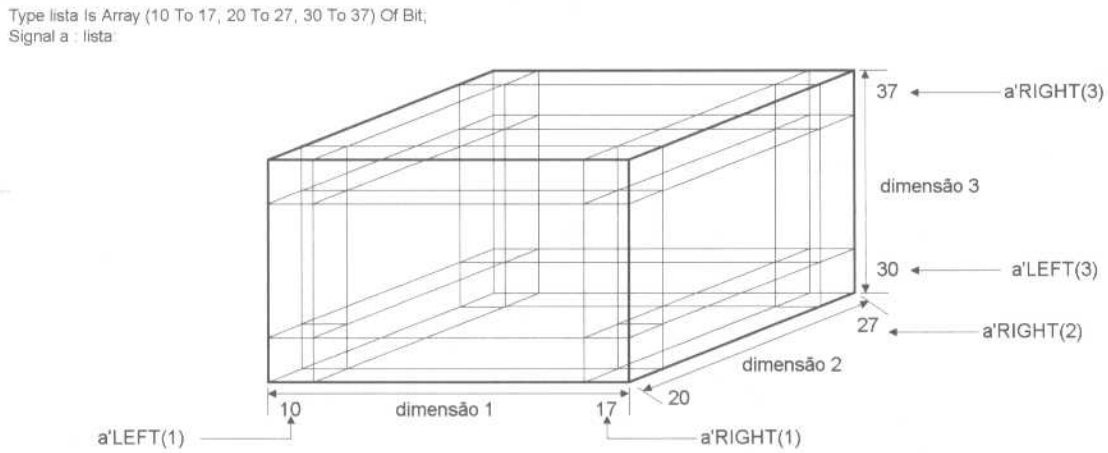


Figura 5.3.5 Ilustração do vetor criada na descrição do Quadro 5.3.4.

Atributos predefinidos relativos a tipos

Os atributos relativos a tipos identificam os elementos de tipos e subtipos escalares (ver Tabela 5.3.4). Normalmente, esses atributos não são suportados pelas ferramentas de síntese; eles podem ser empregados em descrições para teste. Deve-se notar que os atributos " HIGH ", " LOW ", " RIGHT " e " LEFT " têm a mesma designação de atributos de vetores; o significado, entretanto, é outro.

O Quadro 5.3.5 e a Figura 5.3.6 ilustram operações referenciando atributos de tipos relacionados na tabela. A fim de evidenciar a diferença entre os atributos, foram definidos na descrição dois subtipos: um subtipo com elementos na forma ascendente (linha 13) e um subtipo com elementos na forma descendente (linha 14). A definição de subtipos é abordada no Capítulo 12; por ora basta saber que os subtipos podem ser entendidos como um subconjunto de um tipo, e é possível a transferência de valores entre objetos declarados de subtipos originados de um mesmo tipo. Assim, as portas de saída declaradas como tipo " CHARACTER " podem receber valores de sinais declarados pelos subtipos " as " e " ds " definidos na descrição.

```
 1 ENTITY c04_at3 IS
 2   PORT (as_right, ds_right, as_high, ds_high : OUT CHARACTER;
 3         as_left,  ds_left,  as_low,  ds_low  : OUT CHARACTER;
 4         as_succ_G,     ds_succ_G             : OUT CHARACTER;
 5         as_pred_G,     ds_pred_G             : OUT CHARACTER;
 6         as_rightof_G, ds_rightof_G           : OUT CHARACTER;
 7         as_leftof_G,  ds_leftof_G            : OUT CHARACTER;
 8         as_pos_G, c_pos_G                    : OUT INTEGER;
 9         c_val_71                             : OUT CHARACTER);
10 END c04_at3;
11
12 ARCHITECTURE teste OF c04_at3 IS
13   SUBTYPE as IS CHARACTER RANGE 'A' TO 'Z';
14   SUBTYPE ds IS CHARACTER RANGE 'Z' DOWNTO 'A';
15 BEGIN
17   as_right   <= as'RIGHT;     ds_right   <= ds'RIGHT;
18   as_left    <= as'LEFT;      ds_left    <= ds'LEFT;
19   as_high    <= as'HIGH;      ds_high    <= ds'HIGH;
20   as_low     <= as'LOW;       ds_low     <= ds'LOW;
21
22   as_succ_G    <= as'SUCC('G');     ds_succ_G    <= ds'SUCC('G');
23   as_pred_G    <= as'PRED('G');     ds_pred_G    <= ds'PRED('G');
24   as_rightof_G <= as'RIGHTOF('G');  ds_rightof_G <= ds'RIGHTOF('G');
25   as_leftof_G  <= as'LEFTOF('G');   ds_leftof_G  <= ds'LEFTOF('G');
26
27   as_pos_G  <= as'POS('G');
28   c_pos_G   <= CHARACTER'POS('G');
29   c_val_71  <= CHARACTER'VAL(71);
30 END teste;
```

Quadro 5.3.5 Descrição para teste de atributos relativos a tipos.

Tabela 5.3.4 Exemplos de atributos predefinidos para tipos e subtipos.

atributo	descrição
t'HIGH	elemento de posição mais elevada no tipo ou subtipo
t'LOW	elemento de posição mais inferior no tipo ou subtipo
t'SUCC(e)	elemento que sucede o elemento " e " no tipo [2] ou subtipo
t'PRED(e)	elemento que precede o elemento " e " no tipo [2] ou subtipo
t'RIGHT	elemento de posição mais à direita no tipo ou subtipo
t'LEFT	elemento de posição mais à esquerda no tipo ou subtipo
t'RIGHTOF(e)	elemento à direta do elemento " e " no tipo [2] ou subtipo
t'LEFTOF(e)	elemento à esquerda do elemento " e " no tipo [2] ou subtipo
t'VAL(p)	elemento de posição " p " no tipo [2] ou subtipo
t'POS(e)	posição do elemento " e " no tipo [2] ou subtipo
Nota 1: " e " = elemento, " p " = posição referenciada. Nota 2: tipo discreto ou físico.	

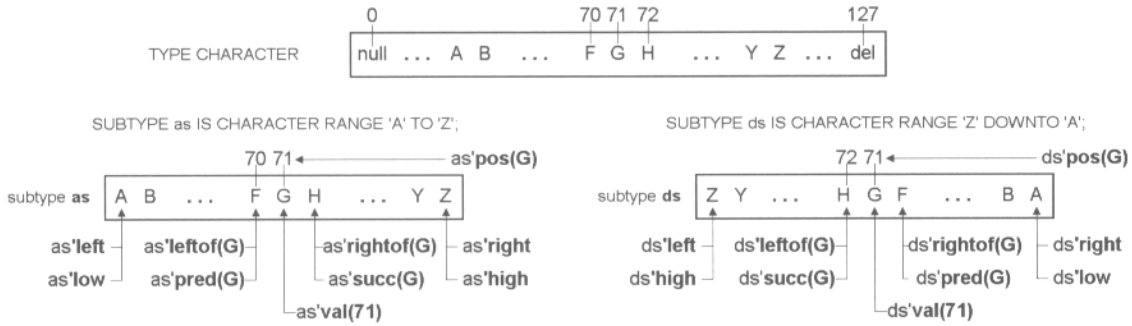


Figura 5.3.6 Exemplos de atributos relativos a tipos.

Conforme visto, " CHARACTER " é um tipo escalar enumerado. Na versão VHDL-1987 o primeiro elemento, " null ", é associado ao número " 0 ", e o último elemento, " del ", é associado ao número " 127 ". Na criação de um subtipo, é preservada a associação dos números que identificam a posição.

Os atributos " LEFT ", " LEFTOF ", " RIGHT " e " RIGHTOF " estão associados à posição física dos elementos, enquanto os atributos " HIGH ", " LOW ", " SUCC " e " PRED " estão associados com a ordem ou enumeração dos elementos. Note que " t'Left = t'Low " e " t'Right = t'High " para designações crescentes, e " t'Left = t'High " e " t'Right = t'Low " para designações decrescentes.

Atributos definidos pelo usuário

A definição de novos atributos permite expandir o leque de opções de atributos fornecido pela linguagem. Com relação à síntese, o suporte a atributos definidos pelo usuário é limitado. As ferramentas de síntese costumam fornecer atributos que podem ser inseridos no código com a finalidade de conduzir e controlar o processo de síntese.

Um novo atributo é definido em duas etapas: inicialmente é feita a declaração, e posteriormente a sua especificação (ver Quadro 5.3.6). Ambas iniciam com a palavra reservada " ATTRIBUTE ". A declaração e a especificação são inseridas na região de declaração de entidades, arquiteturas e pacotes.

A declaração do atributo especifica o nome do atributo e o tipo que retorna na referência ao atributo, respectivamente " nome_atributo " e " tipo " no exemplo do Quadro 5.3.6.

A especificação do atributo associa o atributo definido com um ou mais itens, e estabelece o valor do atributo. A lista de itens, " lista_itens ", pode conter um ou mais nomes que identificam os itens, sendo possível o emprego das palavras reservadas " OTHERS " ou " ALL ". No caso da utilização dessas palavras, os atributos se aplicam aos itens adjacentes já declarados: " OTHERS " para os itens restantes, e " ALL " para todos os itens. A especificação da classe dos itens é apresentada no campo " classe_dos_itens ". Ver exemplos de classes no Quadro 5.3.6. Finalmente, " expressao " indica o valor do atributo.

```
ATTRIBUTE nome_atributo : tipo;                                          -- declaracao do atributo

ATTRIBUTE nome_atributo OF lista_itens : classe_dos_itens IS expressao;  -- especificacao do atributo

-- lista_itens: nome_item_1, nome_item_2, OTHERS
-- lista_itens: ALL

-- classe_itens: CONSTANT VARIABLE SIGNAL TYPE SUBTYPE COMPONENT FUNCTION PROCEDURE UNITS
```

Quadro 5.3.6 Declaração e especificação de atributos.

O Quadro 5.3.7 contém um exemplo de dois atributos especificados pelo usuário denominados " dado_ponta " e " dado_meio ". Observe que a declaração e a especificação foram colocadas na região de declaração da entidade, logo após a declaração das portas.

Ambos os atributos retornam um valor do tipo " BIT " quando referenciados (ver linhas 5 e 8). As especificações são associadas ao item " y ", da classe sinal. O valor de retorno especificado para o atributo " dado_ponta " é o elemento mais à esquerda do vetor " y ", e para " dado_meio " o valor de retorno é o elemento posicionado no meio do vetor " y " (ver linhas 6 e 9).

```
 1  ENTITY atr_usu2b IS
 2    PORT (y  :  IN  BIT_VECTOR(8 DOWNTO 0);
 3          s  :  OUT BIT);
 4
 5    ATTRIBUTE dado_ponta : BIT;                          -- declaracao do atributo: dado_ponta
 6    ATTRIBUTE dado_ponta OF y : SIGNAL IS y(y'LEFT);     -- especificacao do atributo do item: y
 7
 8    ATTRIBUTE dado_meio : BIT;                           -- declaracao do atributo: dado_meio
 9    ATTRIBUTE dado_meio OF y : SIGNAL IS y(y'LENGTH/2); -- especificacao do atributo do item: y
10  END atr_usu2b;
11
12  ARCHITECTURE teste OF atr_usu2b IS
13  BEGIN
14    s <= y'dado_ponta XOR NOT y'dado_meio;    -- equivalente a: s <= y(8) XOR NOT y(4)
15  END teste;
```

Quadro 5.3.7 Declaração e especificação de atributos — exemplo 1.

A descrição do Quadro 5.3.7, submetida a uma ferramenta de síntese, tem como resultado o diagrama esquemático apresentado na Figura 5.3.7. Conforme se pode observar, a atribuição de sinal da linha 14, que emprega os atributos definidos na descrição, foi corretamente sintetizada. A entrada " y(8) " está interligada à porta lógica "ou exclusivo", e a entrada " y(4) " está interligada ao inversor.

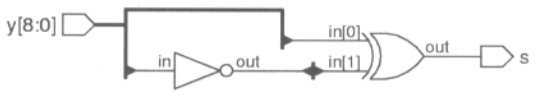


Figura 5.3.7 Diagrama esquemático do circuito gerado a partir da descrição do Quadro 5.3.7.

No exemplo do Quadro 5.3.8 são especificados dois atributos, que têm como objetivo modelar o tempo de propagação na excursão entre níveis lógicos. O atributo " tempo_lh " define o tempo na transição de um nível lógico baixo para um nível lógico alto, e o atributo " tempo_hl " o tempo de transição no sentido inverso. Nesse exemplo, a declaração e a especificação foram colocadas na região de declaração da arquitetura.

Em ambos os atributos o tipo retornado é " TIME " (ver as declarações nas linhas 9 e 14). A especificação do atributo " tempo_lh " associa ao sinal " clk " o valor de " 20 ns ", e o valor " 30 ns " aos itens restantes imediatamente declarados, a saber " rst " e " set ". A especificação do atributo " tempo_hl " associa o valor " 10 ns " a todos os itens imediatamente declarados, no caso " clk ", " rst " e " set ".

O código contém três atribuições de sinal (linhas 19, 22 e 25). Cada atribuição é condicionada à mudança de valor num sinal que resulte em um nível lógico alto após a alteração do valor. Em outras palavras, o teste verifica a ocorrência de uma borda de subida no sinal. Na condição verdadeira, os valores dos sinais das portas de entrada são transferidos com um atraso definido pelo atributo " tempo_lh "; no caso falso, o intervalo de tempo é definido pelo atributo " tempo_hl ". A Figura 5.3.8 ilustra uma simulação da descrição. A detecção de bordas de subida e descida é discutida, com mais detalhes, no Capítulo 6.

```
 1  ENTITY atr_usua4 IS
 2    PORT (clk_ext, rst_ext, set_ext : IN  BIT);  -- sinais externos
 3  END atr_usua4;
 4
 5  ARCHITECTURE teste OF atr_usua4 IS
 6    SIGNAL  clk, rst, set  : BIT;    -- sinais que serao definidos atributos
 7
 8    -- atributo: tempo de propagacao de nivel baixo para nivel alto: ____/----
 9    ATTRIBUTE tempo_lh : TIME;                      -- declaracao do atributo: tempo_lh
10    ATTRIBUTE tempo_lh OF clk    : SIGNAL IS 20 ns; -- atributo do item ck
11    ATTRIBUTE tempo_lh OF OTHERS : SIGNAL IS 30 ns; -- atributo dos itens restantes: rst e set
12
13    -- atributo: tempo de propagacao de nivel alto para nivel baixo: ----\____
14    ATTRIBUTE tempo_hl : TIME;                      -- declaracao do atributo: tempo_hl
15    ATTRIBUTE tempo_hl OF ALL : SIGNAL IS 10 ns;    -- atributo de todos itens: clk, rst e set
17
18  BEGIN                                  --    (_____/------  subida de clk_ext)
19    clk <= clk_ext AFTER clk'tempo_lh WHEN(clk_ext'EVENT AND clk_ext = '1') ELSE
20           clk_ext AFTER clk'tempo_hl;
21                                         --    (_____/------  subida de rst_ext)
22    rst <= rst_ext AFTER rst'tempo_lh WHEN(rst_ext'EVENT AND rst_ext = '1') ELSE
23           rst_ext AFTER rst'tempo_hl;
24                                         --    (_____/------  subida de set_ext)
25    set <= set_ext AFTER set'tempo_lh WHEN(set_ext'EVENT AND set_ext = '1') ELSE
26           set_ext AFTER set'tempo_hl;
27  END teste;
```

Quadro 5.3.8 Declaração e especificação de atributos — exemplo 2.

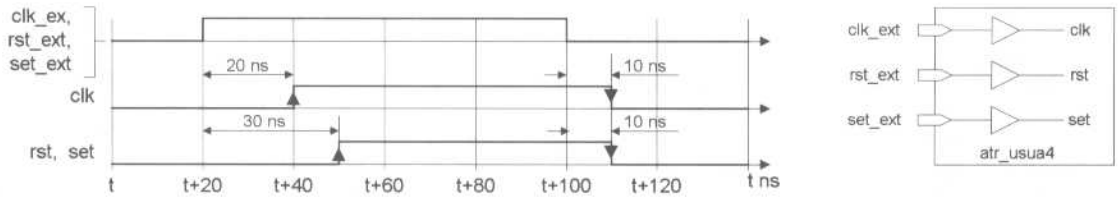


Figura 5.3.8 Simulação da descrição do Quadro 5.3.8.

5.4 Cuidados na descrição

Ordem na atribuição de valores

No Quadro 5.4.1 são apresentadas duas novas versões para a descrição do circuito de seleção visto no Capítulo 3 empregando variáveis. Os sinais que definem as operações lógicas internas, " int0 " a " int3 ", foram substitu-

ídos por variáveis. Das duas descrições, apenas uma descreve adequadamente o comportamento do circuito de seleção, devido à ordem na atribuição de valores nas variáveis e sinais.

Na entidade " mux_vx ", o comando da atribuição de valor para o sinal " ox " está posicionado após as atribuições de valores das variáveis. Como as variáveis recebem o novo valor após a execução do comando, a atribuição do valor para " ox " é avaliada com base nos novos valores das variáveis. Na suspensão do processo ocorre uma iteração, e o sinal " ox " recebe um novo valor.

Na execução do processo da entidade " mux_vy ", a atribuição de valor do sinal " oy " é avaliada com base nos valores anteriores das variáveis " int0 ", " int1 ", " int2 " e " int3 ", pois a atribuição de valor delas ocorre após o comando de atribuição de " oy ". Assim, a entidade " mux_vy " não tem um comportamento combinacional, porque o valor de saída do circuito não depende unicamente da condição atual das entradas. Essa descrição, portanto, não se comporta como o circuito de seleção especificado.

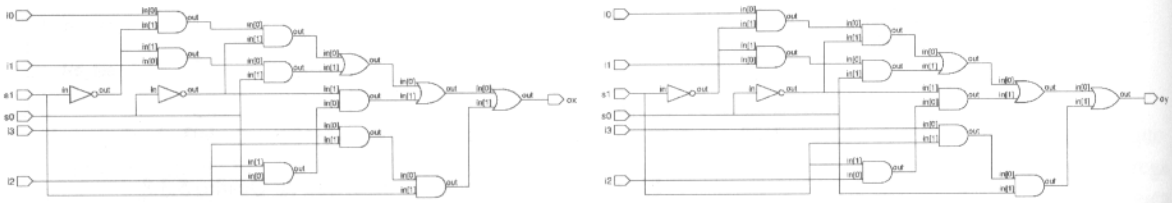
A utilização desse modelo de descrição para armazenar valores deve ser evitada, pois as ferramentas de síntese não inferem a necessidade de armazenamento de valores nessas condições, conforme mostrado a seguir. A inferência de registradores é discutida, com maior detalhamento, no Capítulo 6.

```
 1  -- circuito de selecao: descricao correta
 2  ENTITY mux_vx IS
 3    PORT (i0, i1, i2, i3     :  IN  BIT;
 4          s0, s1             :  IN  BIT;
 5          ox                 :  OUT BIT);
 6  END mux_vx;
 7
 8  ARCHITECTURE teste OF mux_vx IS
 9  BEGIN
10    PROCESS(i0, i1, i2, i3, s0, s1)
11      VARIABLE int0, int1, int2, int3 : BIT;
12    BEGIN
13      int0 := i0 AND NOT s1 AND NOT s0;
14      int1 := i1 AND NOT s1 AND     s0;
15      int2 := i2 AND     s1 AND NOT s0;
16      int3 := i3 AND     s1 AND     s0;
17      ox <= int0 OR int1 OR int2 OR int3;
18    END PROCESS;
19  END teste;
```

```
 1  -- circuito de selecao: descricao incorreta
 2  ENTITY mux_vy IS
 3    PORT (i0, i1, i2, i3     :  IN  BIT;
 4          s0, s1             :  IN  BIT;
 5          oy                 :  OUT BIT);
 6  END mux_vy;
 7
 8  ARCHITECTURE teste OF mux_vy IS
 9  BEGIN
10    PROCESS(i0, i1, i2, i3, s0, s1)
11      VARIABLE int0, int1, int2, int3 : BIT;
12    BEGIN
13      oy <= int0 OR int1 OR int2 OR int3;
14      int0 := i0 AND NOT s1 AND NOT s0;
15      int1 := i1 AND NOT s1 AND     s0;
16      int2 := i2 AND     s1 AND NOT s0;
17      int3 := i3 AND     s1 AND     s0;
18    END PROCESS;
19  END teste;
```

Quadro 5.4.1 Descrições do circuito de seleção: " mux_vx " e " mux_vy ".

A Figura 5.4.1 ilustra o resultado da síntese das descrições do Quadro 5.4.1. Na entidade " mux_vx " o diagrama esquemático contém unicamente portas lógicas. Ele corresponde ao esquema proposto na Figura 3.2.1 do Capítulo 3, onde as portas lógicas com mais de duas entradas foram substituídas por portas lógicas em série realizando a mesma função. Na entidade " mux_vy ", o diagrama esquemático é idêntico ao anterior, indicando problemas na síntese. Não foram inseridos registradores para a manutenção dos valores de " int0 " a " int3 ", resultando que o circuito proposto não tem o mesmo comportamento da descrição.



mux_vx:
"ox <= int0 OR int1 OR int2 OR int3;" linha 17

mux_vy:
"oy <= int0 OR int1 OR int2 OR int3;" linha 13

Figura 5.4.1 Resultado da síntese das duas descrições do Quadro 5.4.1.

5.5 Principais pontos abordados

- A modelagem de atraso é feita através da cláusula " AFTER ". A cláusula " AFTER " é ignorada pelas ferramentas de síntese. Ela encontra aplicação no teste de descrições, e modelagem do desempenho de circuitos sintetizados.
- Variável é um tipo de objeto que permite armazenar valores em regiões de código seqüencial.
- Diferentemente de um sinal, uma variável recebe o valor após a execução do comando. Não existe o intervalo de tempo Δ para a atribuição de um novo valor.
- A transferência de valores armazenados em variáveis para regiões de código concorrente é feita através de sinais.
- Atributos são informações adicionais relacionadas a objetos, subprogramas, tipos, entre outros. Eles são referenciados na forma " prefixo'nome_atributo ". O resultado da referência pode ser um tipo, um valor, um sinal ou uma faixa.
- Apenas alguns atributos predefinidos são suportados pelas ferramentas de síntese. Uma aplicação importante de atributos para síntese é detecção de bordas de subida ou descida em sinais, na descrição de máquinas de estados e *flip flops*.

5.6 Exercícios

5.6.1 Analise a descrição do Quadro 5.6.1 e apresente a carta de tempo das interfaces de saída " x0 ", " x1 " e " x2 ". Comente uma possível aplicação para essa descrição.

```
1  ENTITY c04_e01 IS
2    PORT (x0, x1, x2    : BUFFER BOOLEAN);
3  END c04_e01;
4
5  ARCHITECTURE teste OF c04_e01 IS
6   BEGIN
7     x0 <= NOT x0 AFTER 100 ns;
8     x1 <= NOT x1 AFTER 100 ns WHEN x0 ELSE
9               x1 AFTER 100 ns;
10    x2 <= NOT x2 AFTER 200 ns WHEN x1 ELSE
11              x2 AFTER 200 ns;
12 END teste;
```

Quadro 5.6.1 Descrição do Exercício 5.6.1.

5.6.2 Apresente uma descrição para modelar o circuito proposto na Figura 5.6.1, considerando o atraso das portas lógicas. Esse circuito é empregado na geração de sinais de relógio sem sobreposição, isto é, nunca as saídas $\phi 1$ e $\phi 2$ podem estar em nível alto ao mesmo tempo. Suponha que todas as portas lógicas têm o mesmo atraso de propagação.

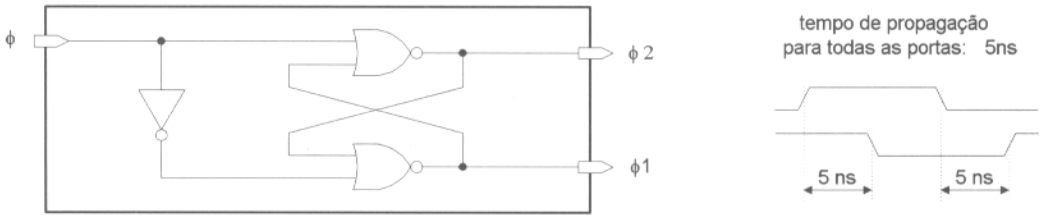


Figura 5.6.1 Exercício 5.6.2.

5.6.3 Apresente a seqüência de iterações que ocorrem no processo " sig " da descrição apresentada no Quadro 5.6.2.

5.6.4 Apresente uma descrição completa para simulação simultânea dos dois processos apresentados no Quadro 5.6.2. Os sinais " sx " e " vx " devem ser duas portas de saídas da entidade. Apresente, também, a seqüência de iterações que deve ocorrer nessa descrição, e o diagrama de tempos contendo os sinais " sx " e " vx ".

5.6.5 Mova o comando " WAIT " nos dois processos do Quadro 5.2.1 para uma linha logo após as palavras reservadas " BEGIN " dos processos. Analise a operação dessa nova entidade, e apresente o diagrama de tempos contendo os sinais " sx " e " vx ".

```
SIGNAL sa     : BIT := '1';
SIGNAL sb, sc : BIT := '0';

sig: PROCESS
  BEGIN
    sb <= sa;
    sc <= sb;
    sa <= NOT sc;
    sx <= NOT sc;
    WAIT FOR 10 ns;
END PROCESS sig;
```

```
var: PROCESS
 VARIABLE va     : BIT :='1';
 VARIABLE vb, vc : BIT :='0';
 BEGIN
   vb := va;
   vc := vb;
   va := NOT vc;
   vx <= NOT vc;
   WAIT FOR 10 ns;
 END PROCESS var;
```

Quadro 5.6.2 Exercícios 5.6.3 e 5.6.4.

5.6.6 Na descrição do Quadro 5.6.3, o comando " IF " emprega o atributo referente a sinal " EVENT ". Complete a carta de tempos da Figura 5.6.2 com base na descrição proposta.

```
 1  ENTITY c04_e03 IS
 2    PORT (ck, d  : IN  BIT;
 3          q1, q2 : OUT BIT);
 4  END c04_e03;
 5
 6  ARCHITECTURE teste OF c04_e03 IS
 7  BEGIN
 8    p_d1: PROCESS(ck)
 9    BEGIN
10      IF ck'EVENT AND ck = '1' THEN  q1 <= d;
11      END IF;
12    END PROCESS;
13
14    p_d2: PROCESS(ck)
15    BEGIN
15      IF ck'EVENT AND ck = '0' THEN  q2 <= d;
17      END IF;
18    END PROCESS;
19  END teste;
```

Quadro 5.6.3 Descrição para o Exercício 5.6.6.

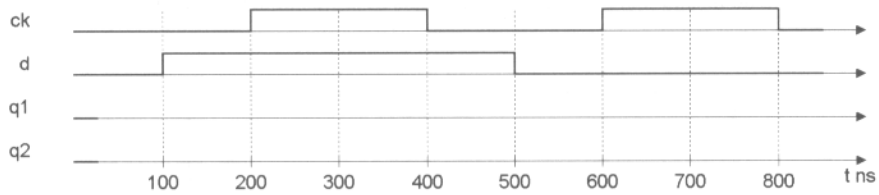


Figura 5.6.2 Exercício 5.6.6.

5.6.7 Proponha uma descrição que tenha o mesmo comportamento do código proposto no Quadro 5.6.3 empregando o comando " WAIT ".

5.6.8 Altere a descrição proposta no Quadro 5.6.3, sem mudar o seu comportamento, de modo a empregar o atributo " STABLE ".

5.6.9 Dados dois subtipos inteiros, um ascendente restrito à faixa " −10 TO +15 ", e outro descendente restrito à faixa " +25 DOWNTO −20 ". Determine o resultado das operações de referência aos atributos relacionados na Tabela 5.6.1 nesses dois subtipos. Apresente, também, uma descrição para teste.

Tabela 5.6.1 Exercício 5.6.9.

HIGH	LOW	SUCC(7)	PRED(7)	RIGHT	LEFT	RIGHTOF(7)	LEFTOF(7)	VAL(7)	POS(7)

5.6.10 Apresente um código que descreva o comportamento apresentado na Figura 5.6.3, com relação ao tempo de propagação de valores. Defina atributos para modelar os diferentes tempos de propagação. Os atributos devem estar associados aos sinais " clk_sai ", " rst_sai " e " set_sai ".

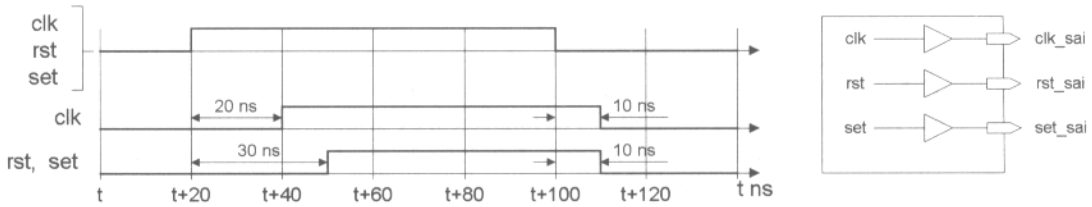


Figura 5.6.3 Comportamento da descrição do Exercício 5.6.10.

5.6.11 Dado o diagrama de blocos da Figura 5.6.4, apresente um código que descreva o comportamento ilustrado. Dois atributos devem ser definidos para o sinal " val ". O atributo " soma_x " retorna com o valor de " val " adicionado ao valor de " x ", e o atributo " soma_y " retorna com o valor de " val " somado ao valor de " y ". A entrada " sel " realiza uma operação de seleção, de modo a selecionar qual valor adicionado a " val " é apresentado na saída " soma ". Para facilitar a descrição empregue o tipo " INTEGER " nos sinais.

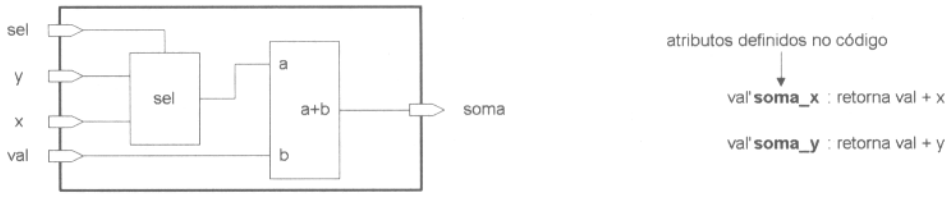


Figura 5.6.4 Comportamento da descrição do Exercício 5.6.11.

5.6.12 Sintetize a descrição proposta para o Exercício 5.6.11 e verifique o resultado.

CAPÍTULO

6 ESTRATÉGIAS DE DESCRIÇÃO DE CIRCUITOS SÍNCRONOS

A descrição de um circuito pode empregar os mais diversos recursos disponíveis na linguagem. Uma descrição visando à síntese, entretanto, nem sempre pode lançar mão de todos esses meios. Os aplicativos de síntese não reconhecem todas as estruturas da linguagem e possuem, normalmente, estilos preferenciais de descrição para inferência das estruturas. Um segundo fator que deve ser considerado são os recursos disponíveis na tecnologia empregada na implementação física do circuito. O conhecimento dos limites da tecnologia leva a descrições que utilizam ao máximo os elementos disponíveis, resultando em circuitos de menor custo e melhor desempenho.

6.1 Conceitos iniciais

Um elemento de memória pode ser inferido quando um valor é atribuído a um sinal ou variável em, pelo menos, uma condição, e nenhum valor é atribuído a esse objeto em, pelo menos, uma condição. A falta de atribuição de um valor acarreta a necessidade de armazenamento do valor quando nenhuma condição de atribuição é atingida.

Em um registrador composto por elementos sensíveis a nível, ou *latches*, a condição para memorização corresponde a um determinado nível lógico do sinal de controle (ver Figura 6.1.1). O sinal de controle habilitado, a saída " q " segue a entrada " d "; na condição desabilitado, o valor em " q " é mantido. Nos registradores contendo elementos sensíveis a borda, ou *flip flops*, a condição de armazenamento é qualificada na transição de um nível lógico no sinal de relógio, ou *clock*. No exemplo da Figura 6.1.1, é apresentado um *latch* habilitado por um nível lógico alto, e um *flip flop* sensível a borda de subida do sinal de relógio.

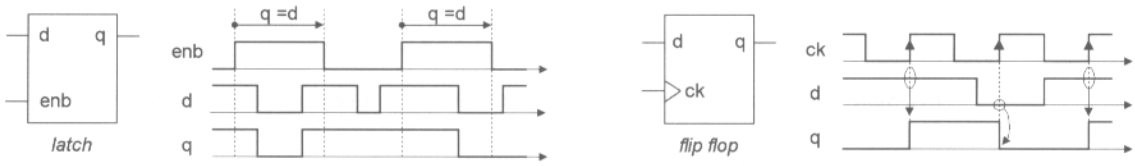


Figura 6.1.1 Operação de um *latch* e um *flip flop*.

Uma transição em sinal pode ser detectada através dos atributos " EVENT " ou " STABLE ". Numa borda de subida, após a ocorrência de um evento, o valor do sinal deve corresponder a um nível lógico alto. No caso de uma borda de descida, o nível do sinal após o evento deve ser baixo. No Quadro 6.1.1, são apresentadas as condições de teste normalmente aceitas pelas ferramentas de síntese para detecção de bordas de subida e descida. Nesses exemplos o sinal é do tipo " BIT ".

```
(ck'EVENT  AND  ck ='1')            -- borda de subida
(ck'EVENT  AND  ck ='0')            -- borda de descida
(NOT ck'STABLE  AND  ck ='1')       -- borda de subida
(NOT ck'STABLE  AND  ck ='0')       -- borda de descida
```

Quadro 6.1.1 Teste da ocorrência de bordas de subida e descida.

A inicialização de um registrador permite levar as saídas a um nível lógico independente do valor da entrada. A operação que leva as saídas ao nível lógico alto costuma ser denominada *set*, e a operação que leva as saídas ao nível lógico baixo pode ser denominada *reset*. As operações de inicialização são denominadas *síncronas,* caso a sua execução dependa do sinal de controle. As operações *assíncronas* podem ser executadas independentemente da condição do sinal de controle. A Figura 6.1.2 ilustra dois exemplos de inicialização em *flip flops*.

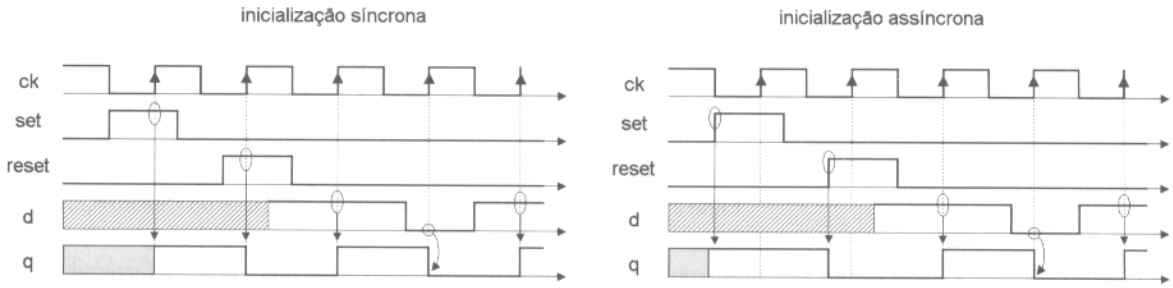


Figura 6.1.2 Exemplos de operações de inicialização em *flip flops*.

6.2 Registrador sensível a nível

O modelo do Quadro 6.2.1 apresenta uma possível descrição de um registrador de três bits sensível a nível, com inicialização assíncrona. O sinal " en " controla o armazenamento da informação presente no sinal " d ": " en " no nível lógico baixo mantém a informação; " en " no nível lógico alto, o sinal " q " segue a entrada " d ". O sinal " rst " estabelece a operação de *reset*, e o sinal " set " a operação de *set*. A construção do tipo " IF ELSE " verifica as condições *set*, *reset* e habilitação. Como qualquer alteração nos sinais de controle, " en ", " rst " e " set ", bem como no sinal " d ", deve sempre ser avaliada, a lista de sensibilidade do processo deve conter todos esses sinais.

```
 1  ENTITY latch3_1 IS
 2    PORT (en   : IN  BIT;                -- habilita
 3          rst  : IN  BIT;                -- rst=1 leva q=000
 4          set  : IN  BIT;                -- set=1 leva q=111
 5          d    : IN  BIT_VECTOR(2 DOWNTO 0);
 6          q    : OUT BIT_VECTOR(2 DOWNTO 0));
 7  END latch3_1;
 8
 9  ARCHITECTURE teste OF latch3_1 IS
10  BEGIN
11    PROCESS (en, d, rst, set)
12    BEGIN
13      IF    (rst ='1') THEN  q <="000";  -- q=000 independente de en
14      ELSIF (set ='1') THEN  q <="111";  -- q=111 independente de en
15      ELSIF (en  ='1') THEN  q <=d;      -- condicao do sinal para habilitar
16      END IF;
17    END PROCESS;
18  END teste;
```

Quadro 6.2.1 Exemplo de uma descrição para um registrador sensível a nível.

A Figura 6.2.1 ilustra o esquema do circuito sintetizado, nível RTL, a partir da descrição. O aplicativo emprega uma primitiva do tipo *latch* com inicialização assíncrona como elemento de memória. O terminal " G " da primitiva habilita a memorização; o terminal " R " leva a saída ao nível lógico baixo; e o terminal " S " leva a saída ao nível lógico alto. As operações de " S " e " R " são ativadas com nível lógico alto. A porta lógica " e " e o inversor realizam os dois primeiros testes contidos na descrição (linhas 13 e 14). O primeiro teste tem precedência sobre os demais, tanto na descrição quanto no circuito sintetizado. A entrada " rst " em nível lógico alto atua diretamente sobre os terminais " R " da primitiva e mantém a entrada " S " em nível lógico baixo via porta lógica " e ". Para " rst =0 " e " set =1 ", tem-se " R =0 " e " S =1 ", condição que leva as saídas das primitivas *latch* para o nível lógico alto. Finalmente, com " rst =0 " e " set =0 ", tem-se " R =0 " e " S =0 ", condição que permite a memorização de um dado através da entrada " en ".

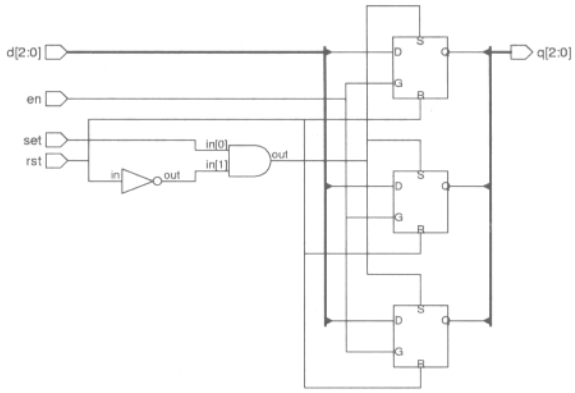


Figura 6.2.1 Circuito sintetizado a partir da descrição do Quadro 6.2.1.

Caso a tecnologia empregada não disponha de elementos sensíveis a nível, a ferramenta de síntese implementa o *latch* com portas lógicas. O mesmo ocorre, se o *latch* disponível na tecnologia não possuir funções de inicialização assíncronas, como *set* e *reset*. No Quadro 6.2.2 é apresentada uma descrição de um registrador sensível a nível com *reset* assíncrono. A título de ilustração, a descrição foi submetida a síntese, tendo como alvo uma tecnologia que não dispõe de *reset* assíncrono nos elementos do tipo *latch*. A ferramenta de síntese notifica esse fato na janela de mensagens, conforme ilustra o Quadro 6.2.3. Na Figura 6.2.2 são apresentados os esquemas nível RTL e nível porta lógica do circuito sintetizado. Conforme se pode observar, a operação de *reset* foi implementada através de portas lógicas, pois a tecnologia escolhida não possui essa função nos elementos disponíveis. Assim, no circuito sintetizado a inicialização é síncrona, diferindo da descrição com respeito à inicialização.

```
1  ENTITY latch1_1 IS
2    PORT (en, rst, d : IN  BIT;
3          q          : OUT BIT);
4  END latch1_1;
5
6  ARCHITECTURE teste OF latch1_1 IS
7  BEGIN
8    PROCESS (en, rst, d)
9    BEGIN
10     IF   (rst = '1') THEN q <= '0';
11     ELSIF (en = '1') THEN q <= d;
12     END IF;
13   END PROCESS;
14 END teste;
```

Quadro 6.2.2 Registrador sensível a nível para teste do circuito gerado - nível de portas lógicas.

```
Warning, asynchronous reset on DLATCH q_EXMPLR replaced by combinational logic.
Warning, Transformations were required because of constraints in target technology max7.
Warning, Design may show simulation differences because of transformations.
```

Quadro 6.2.3 Mensagem de aviso gerada pela ferramenta de síntese.

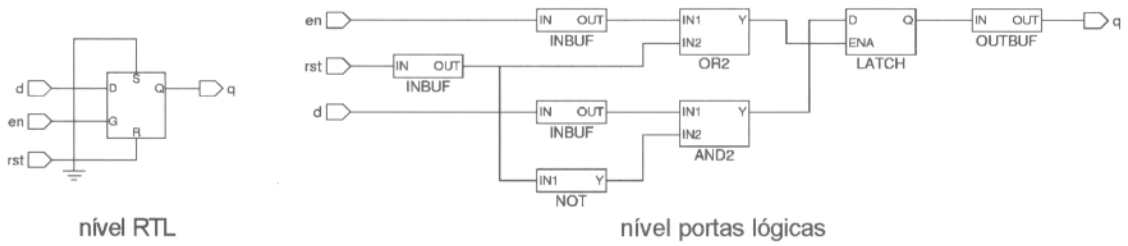


Figura 6.2.2 Resultado da síntese - descrição do Quadro 6.2.2.

6.3 Registrador sensível a borda - inicialização síncrona

O Quadro 6.3.1 apresenta um exemplo de um registrador com três bits sensível a borda de subida do terminal de controle. As operações de *set* e *reset* são síncronas. A descrição emprega duas construções do tipo " IF ELSE ": a primeira detecta a condição do sinal de controle, e a segunda testa sucessivamente as condições para realizar operações de *set*, *reset* ou armazenamento. Como a segunda construção somente é executada na ocorrência de uma borda de subida do sinal de relógio, as operações de *set*, *reset* e armazenamento ficam condicionadas ao sinal de relógio, caracterizando o sincronismo com esse sinal.

```
 1  ENTITY flip3_2 IS
 2    PORT (ck   : IN  BIT;                -- relogio
 3          rst  : IN  BIT;                -- rst=1 leva q=000 sincrono
 4          set  : IN  BIT;                -- set=1 leva q=111 sincrono
 5          d    : IN  BIT_VECTOR(2 DOWNTO 0);
 6          q    : OUT BIT_VECTOR(2 DOWNTO 0));
 7  END flip3_2;
 8
 9  ARCHITECTURE teste OF flip3_2 IS
10  BEGIN
11    PROCESS (ck)
12    BEGIN
13      IF (ck'EVENT AND ck = '1') THEN          -- condicao do sinal relogio
14        IF    (rst = '1') THEN  q <= "000";  -- teste para levar q=000
15        ELSIF (set = '1') THEN  q <= "111";  -- teste para levar q=111
16        ELSE                    q <= d;      -- armazena dado
17        END IF;
18      END IF;
19    END PROCESS;
20  END teste;
```

Quadro 6.3.1 Registrador de 3 bits, sensível a borda, inicialização síncrona.

A Figura 6.3.1 apresenta o resultado da síntese da descrição proposta nível RTL. A ferramenta de síntese emprega um *flip flop* do tipo D como elemento de memória. O sinal de controle é sensível a borda de subida e corresponde ao terminal com o símbolo " > ". Os terminais " S " e " R " dos *flip flops* levam as saídas " Q " para o nível lógico alto ou para o nível lógico baixo, independentemente do sinal de controle, e são ativos no nível lógico alto. Os terminais de " S " e " R " estão fixos no nível lógico baixo, indicando que operações de inicialização assíncronas não foram realizadas no circuito gerado. No caso da entrada " rst " em nível lógico alto, todos os terminais " D " dos *flip flops* recebem um nível lógico baixo, independentemente da condição das entradas " set " e " d ", possibilitando a operação de *reset* síncrona descrita na entidade. A entrada " rst " em nível lógico baixo libera os dados presentes na saída das portas lógicas "ou". Conforme a condição da entrada " set ", os terminais " D " dos *flip flops* podem receber um nível lógico alto, operação *set*, ou a informação contida nas portas

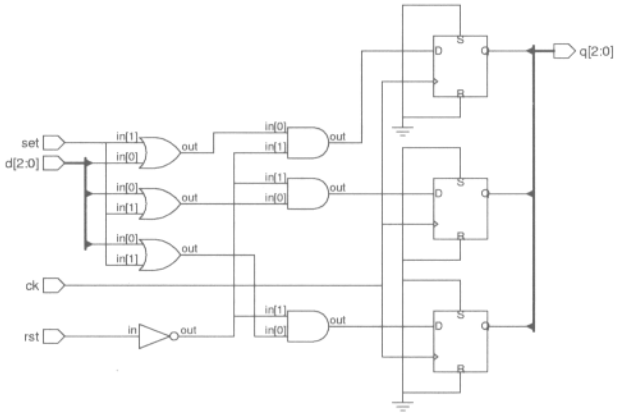


Figura 6.3.1 Circuito gerado a partir da descrição do Quadro 6.3.1.

de entrada " d " da descrição. As portas lógicas operam, portanto, como um circuito de seleção que implementa a construção " IF ELSE " descrita entre as linhas 14 e 17. Pode-se observar, ainda, que o circuito de seleção contém a prioridade prevista na seqüência de decisões tomadas na construção " IF ELSE " das linhas 14 e 17: a entrada " rst " em nível lógico alto possibilita a operação de *reset*, independentemente da condição da entrada " set ". O mesmo pode ser observado quanto ao armazenamento dos dados e à entrada " set ".

6.4 Registrador sensível a borda - inicialização assíncrona

O Quadro 6.4.1 apresenta um exemplo de um registrador, com três bits, sensível a borda de subida do sinal de relógio. As operações para inicialização definidas pelos sinais " set " e " rst " são independentes do sinal de relógio " ck ". A descrição emprega uma construção do tipo " IF ELSE " para testar sucessivamente as condições *set*, *reset* e borda de subida do sinal de relógio. Como os eventos nos sinais não estão mais condicionados ao sinal " ck ", eles devem ser inseridos na lista de sensibilidade do processo. Nesse exemplo, devido à característica da construção " IF ELSE ", a operação *reset* tem precedência sobre a operação *set*, e esta tem precedência sobre a operação de armazenamento.

```
 1 ENTITY flip3_3 IS
 2   PORT (ck   : IN  BIT;                  -- relogio
 3         rst  : IN  BIT;                  -- rst=1 leva q=000 assincrono
 4         set  : IN  BIT;                  -- set=1 leva q=111 assincrono
 5         d    : IN  BIT_VECTOR(2 DOWNTO 0);
 6         q    : OUT BIT_VECTOR(2 DOWNTO 0));
 7 END flip3_3;
 8
 9 ARCHITECTURE teste OF flip3_3 IS
10 BEGIN
11   PROCESS (ck, rst, set)
12   BEGIN
13     IF       (rst = '1')          THEN q <="000"; -- q=000 independente de ck
14     ELSIF    (set = '1')          THEN q <="111"; -- q=111 independente de ck
15     ELSIF (ck'EVENT AND ck ='1') THEN q <=d;     -- condicao do sinal relogio
16     END IF;
17   END PROCESS;
18 END teste;
```

Quadro 6.4.1 Registrador de 3 bits, sensível a borda, inicialização assíncrona.

A Figura 6.4.1 ilustra o esquema do circuito sintetizado, nível RTL, a partir da descrição proposta no Quadro 6.4.1. Nesse circuito, as primitivas *flip flop* tipo D têm os terminais " S " e " R " controlados pelas entradas " set " e " rst " da descrição. Conforme previsto pela análise da descrição, o sinal " rst " tem precedência na definição operação: " rst =1 " leva o sinal " q " ao nível lógico baixo, independentemente da condição dos sinais " set " e " ck ". De modo semelhante à descrição do Quadro 6.2.1, a precedência foi inserida pela ferramenta de síntese através da porta " e ".

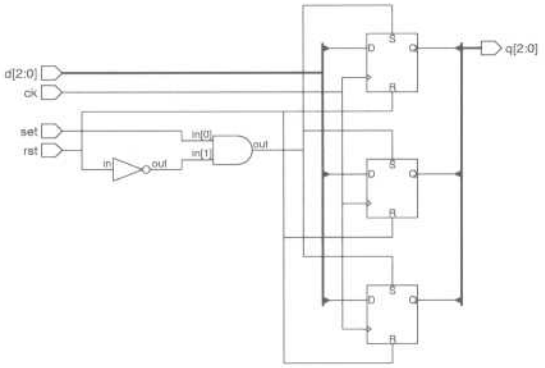


Figura 6.4.1 Circuito gerado a partir da descrição do Quadro 6.4.1 - nível RTL.

No processo de síntese da descrição do Quadro 6.4.1 foi escolhida uma tecnologia que não dispõe de *flip flops* com operação de *set* assincrona. Isso acarreta problemas na etapa final do processo de síntese, conforme ilustra o Quadro 6.4.2. Mensagens de alerta são geradas pela ferramenta, informando que a operação *set* foi reconfigurada para o modo síncrono, e o comportamento do circuito sintetizado deve ser diferente do descrito. Isso é devido ao fato de que a tecnologia-alvo escolhida nesse exemplo possui apenas *flip flops* tipo D com *reset* assíncrono.

A Figura 6.4.2 ilustra o esquema final do circuito, as caixas " F3_LUT " implementam a função combinacional " I2 + (NOT I0 . I1) " para a realização da operação *set* síncrona.

```
-- optimize -target flex6 -effort quick -chip -area -hierarchy=auto
-- Start optimization for design .work.flip3_3.teste
Warning, asynchronous set on DFF q(2)_EXMPLR replaced by synchronous set.
Warning, asynchronous set on DFF q(1)_EXMPLR replaced by synchronous set.
Warning, asynchronous set on DFF q(0)_EXMPLR replaced by synchronous set.
Warning, Transformations were required because of constraints in target technology flex6.
Warning, Design may show simulation differences because of transformations.
```

Quadro 6.4.2 Mensagens de alerta devido à ausência de *flip flops* com *set* assíncrono.

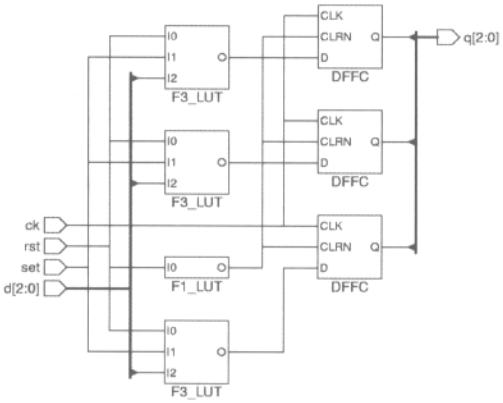


Figura 6.4.2 Circuito gerado a partir da descrição do Quadro 6.4.1 - nível porta lógica.

6.5 Registrador sensível a borda com sinal para habilitação do sinal de relógio

A inferência de um registrador com sinal de habilitação do sinal de relógio pode ser conseguida com o modelo apresentado no Quadro 6.5.1. O registrador é semelhante ao apresentado no item 6.4 com operações de *set* e *reset* assíncronas acrescido do sinal " enb " para habilitar o sinal de relógio.

```
1  ENTITY flip3_4 IS
2    PORT (ck        : IN  BIT;      -- relogio
3          enb       : IN  BIT;      -- habilita sinal de relogio
4          rst, set  : IN  BIT;      -- rst=1 q=000, set=1 q=111 assincronos
5          d     : IN  BIT_VECTOR(2 DOWNTO 0);
6          q     : OUT BIT_VECTOR(2 DOWNTO 0));
7  END flip3_4;
8
9  ARCHITECTURE teste OF flip3_4 IS
10 BEGIN
11   PROCESS (ck, rst, set)
12   BEGIN
13     IF    (rst = '1') THEN q <= "000";    -- q=000 independente de ck
14     ELSIF (set = '1') THEN q <= "111";    -- q=111 independente de ck
15     ELSIF (ck'EVENT AND ck = '1') THEN    -- detecta borda de ck
16       IF  (enb = '1') THEN q <= d;        -- verifica habilitacao
17       END IF;
18     END IF;
19   END PROCESS;
20 END teste;
```

Quadro 6.5.1 Registrador sensível a borda, inicialização assíncrona e sinal de habilitação.

A alteração no código reside na inserção de um teste para verificar a condição de habilitação no sinal " enb ", linha 16. Caso uma borda de subida no sinal de relógio tenha ocorrido, e " enb=1 ", a operação de armazenamento é executada. Como a condição no sinal " enb " deve ser verificada apenas na ocorrência de uma transição no sinal de relógio, não é necessário inserir " enb " na lista de sensibilidade do processo.

A Figura 6.5.1 ilustra o resultado da síntese do circuito nível RTL. A ferramenta de síntese empregou *flip flops* que possuem um terminal de habilitação. Quando não for disponível esse tipo de componente na tecnologia de implementação do circuito, o sintetizador deve criar uma estrutura semelhante ao esquema " a " (Figura 6.5.2). Na condição " enb =1 " a informação presente na entrada " d " é armazenada na borda de subida, e na condição " enb =0 " o valor da saída " q " é transferido para a entrada " d " levando ao armazenamento do mesmo valor na transição do sinal de relógio. Assim, na condição " enb =0 " a informação do valor na saída " q " não é alterada. A solução apresentada no esquema " b " da Figura 6.5.2 não é adequada, e pode levar a um comportamento incorreto. Caso o sinal " enb " sofra uma transição com " ck =1 ", ocorre uma borda de subida na saída da porta lógica " e " levando ao armazenamento de um valor no *flip flop* sem a ocorrência de uma transição no sinal " ck ". Um outro problema pode surgir com " ck =0 ". Dependendo da característica da porta " e ", uma mudança de nível na outra entrada da porta pode gerar pulso de curta duração, ocasionando o armazenamento de um valor no *flip flop* sem a transição do sinal de relógio.

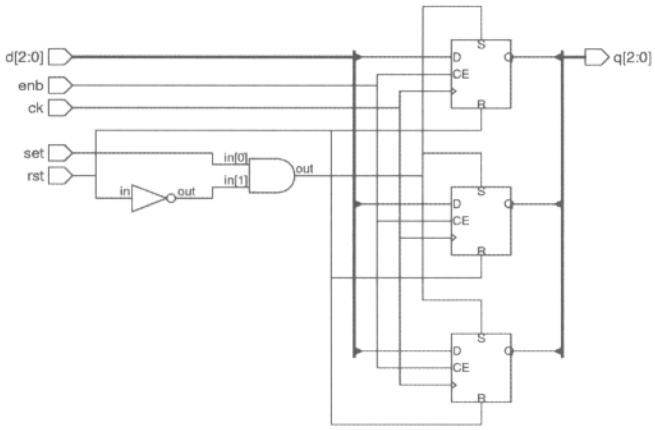


Figura 6.5.1 Circuito gerado a partir da descrição do Quadro 6.5.1- nível RTL.

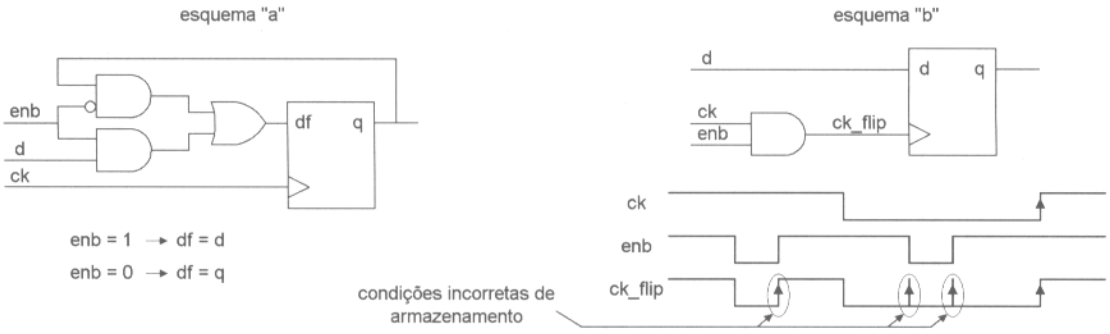


Figura 6.5.2 Implementação correta e implementação incorreta de um sinal de habilitação.

No Quadro 6.5.2 são apresentadas duas opções para descrever uma operação de habilitação do sinal de relógio. Uma descrição conforme o segundo modelo do Quadro 6.5.2, válida com relação à sintaxe da linguagem, leva a uma operação de lógica " e " com a linha de relógio conforme o esquema " b " da Figura 6.5.2. Esse modelo de descrição pode não ser aceito pela ferramenta de síntese, com objetivo de prevenir problemas de falso gatilho, conforme descrito. Nesses casos, é gerada uma mensagem de erro, conforme ilustra o Quadro 6.5.3.

```
-- modo correto
IF (ck'EVENT AND ck = '1') THEN    -- detecta borda de ck
  IF (enb = '1') THEN              -- verifica habilitacao
     q <= d;
  END IF;
END IF;
```

```
-- modo incorreto para descrever um sinal de habilitacao
IF (ck'EVENT AND ck = '1' AND enb ='1') THEN  -- realiza operacao logica "e" com o sinal de relogio
  q <=d;
END IF;
```

Quadro 6.5.2 Descrição do controle de habilitação em um registrador.

```
"/vhdl/flip3_1.vhd",line 15: Error, clock expression should contain only one signal.
```

Quadro 6.5.3 Mensagem de erro gerada pelo segundo modelo proposto no Quadro 6.5.2.

Nas Figuras 6.5.3 e 6.5.4 são apresentados dois esquemas, nível porta lógica, resultantes da síntese da descrição do Quadro 6.5.1. No esquema da Figura 6.5.3, a tecnologia dispõe de *flip flops* com sinal de habilitação, terminal " ENA ", resultando em um circuito equivalente ao esquema nível RTL da Figura 6.5.1. No segundo caso, Figura 6.5.4, a tecnologia não dispõe de *flip flops* com sinal de habilitação. A ausência de um componente desse tipo levou o aplicativo a empregar uma solução semelhante à descrita na Figura 6.5.2, pois as caixas " F3_LUT " realizam a função " $(I0 . I1) + (\overline{I0} . I2)$ ".

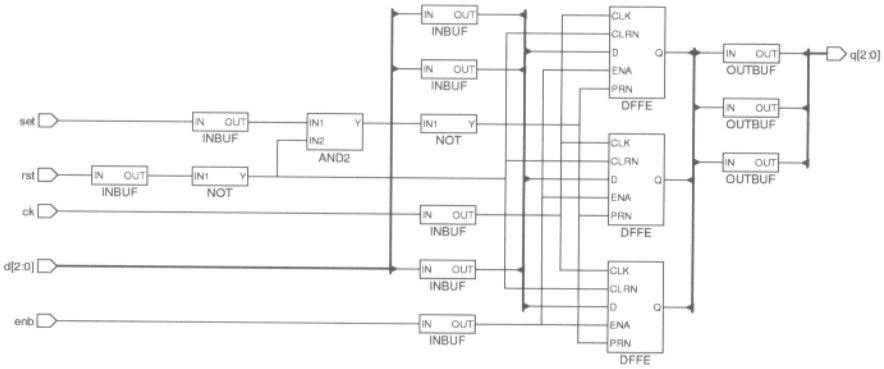


Figura 6.5.3 Tecnologia dispõe de *flip flop* com sinal de habilitação - nível porta lógica.

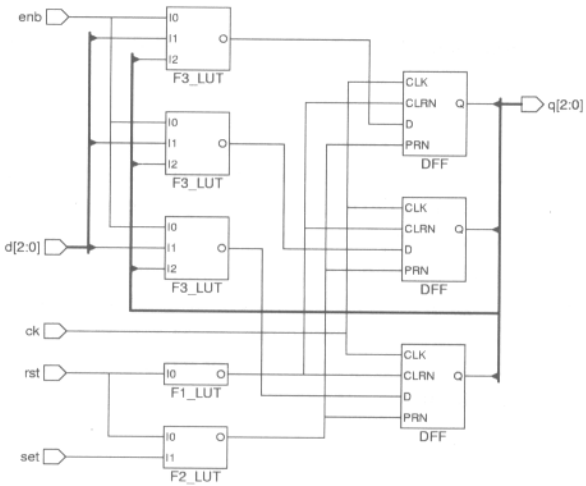


Figura 6.5.4 Tecnologia não dispõe de *flip flop* com sinal de habilitação - nível porta lógica.

6.6 Máquinas de estados

Uma máquina de estados é um circuito seqüencial que transita numa seqüência predefinida de estados. A transição entre os estados é comandada por um sinal de controle.[1] O estado atual é definido por um elemento de memória, e o estado futuro é determinado com base no estado atual e na condição das entradas.

Dois estilos de máquinas são definidos (ver Figura 6.6.1). Na máquina *Mealy*, o valor da saída é função do estado atual e da condição das entradas; na máquina *Moore*, o valor da saída depende exclusivamente do estado atual. A Figura 6.6.2 descreve um ciclo de operação dessas máquinas em que a mudança de estados ocorre na borda de subida do sinal de controle. Na figura é ilustrado, também, o diagrama de estados que representa, na forma de um grafo orientado, a evolução entre os estados. Cada estado é representado por um nó, e a transição entre os estados é indicada por arcos. O nó contém o estado atual, e o arco, a condição de entrada que leva ao próximo estado indicado. Na máquina *Mealy*, o valor da saída é incluído no arco, e, na máquina *Moore*, ele é indicado no nó.

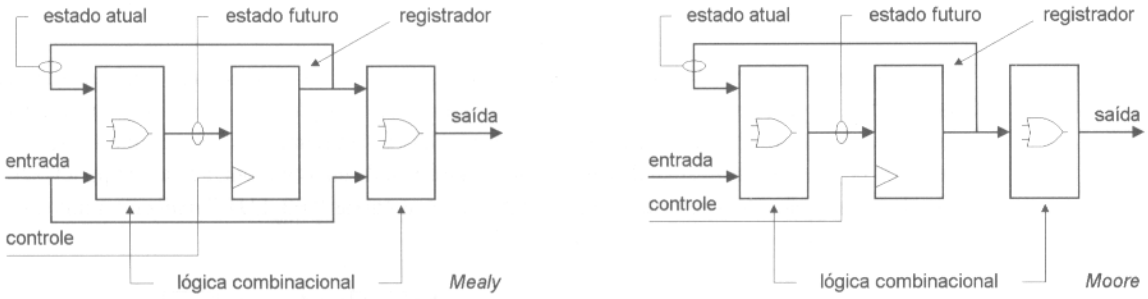


Figura 6.6.1 Diagrama de blocos de máquinas de estados.

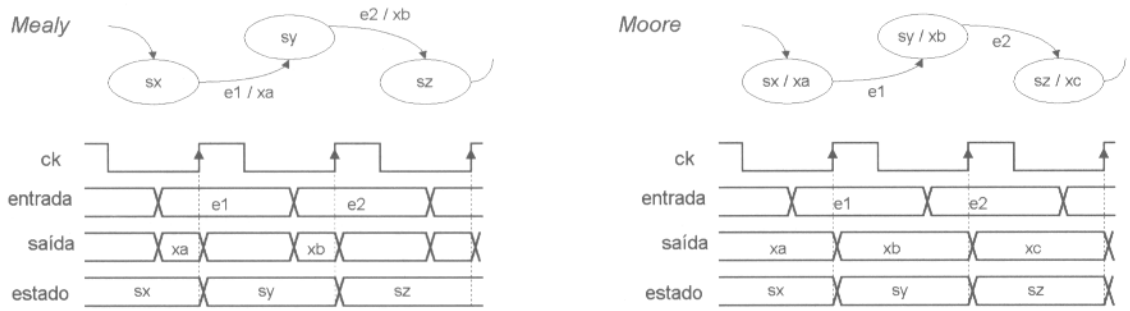


Figura 6.6.2 Exemplo: diagrama de estado e carta de tempo de máquinas *Mealy* e *Moore*.

A ferramenta de síntese pode inferir uma máquina de estados caso a descrição contenha um objeto que armazena o estado atual, uma definição da transição de estados e uma especificação dos valores de saída. A transição entre estados na descrição é controlada por um sinal de relógio, e pode conter condições de inicialização síncronas ou assíncronas.

No exemplo da Figura 6.6.3, é ilustrado o diagrama de estados de uma máquina com quatro estados que realiza uma contagem entre os valores " 00 ", " 01 ", " 11 " e " 10 ". Para simplificação do circuito, o valor que representa o estado atual corresponde ao valor das saídas. A máquina inclui, também, uma entrada para inicialização assíncrona denominada " rst ".

[1]Não estamos considerando máquinas seqüenciais assíncronas.

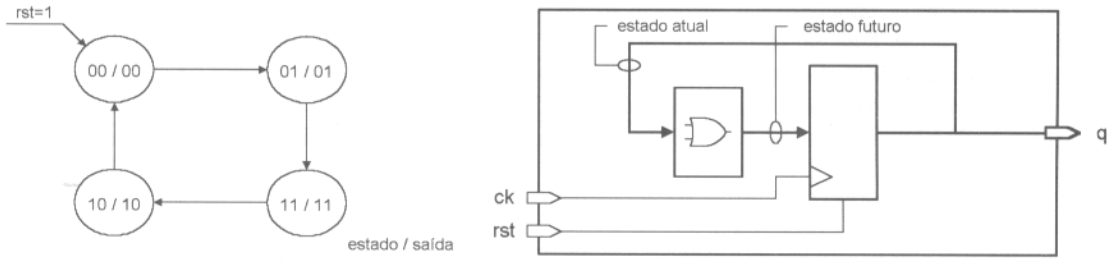


Figura 6.6.3 Máquina de quatro estados.

O Quadro 6.6.1 apresenta uma possível descrição para a máquina proposta. O código contém uma construção do tipo " IF ELSE " que detecta duas condições: sinal " rst " ativo e ocorrência de uma borda de subida no sinal de relógio. O sinal " rst " ativo leva a máquina para o estado " 00 ", e a ocorrência de uma transição entre " 0 " e " 1 " no sinal " ck " leva a uma mudança de estado. O estado futuro é determinado por uma construção " CASE WHEN ", sendo o estado atual o valor das saídas.

A construção " CASE WHEN " é mais adequada para a definição das transições de estado do que uma construção " IF ELSE ". A construção " IF ELSE ", devido à prioridade implícita na escolha das opções, acarreta a geração de um decodificador de prioridade para a tomada de decisão, um circuito mais complexo que o seletor necessário para a construção " CASE WHEN ". Além disso, evita-se a inserção de *latches* desnecessários devido a condições de escolha não-cobertas.

```
 1  ENTITY maq_est1 IS
 2    PORT (ck      :  IN     BIT;                        -- relogio borda subida
 3          rst     :  IN     BIT;                        -- rst=1, q=00
 4          q       :  BUFFER BIT_VECTOR (1 DOWNTO 0)); -- saida codigo Gray
 5  END maq_est1;
 6
 7  ARCHITECTURE teste OF maq_est1 IS
 8
 9  BEGIN
10    abc: PROCESS (ck, rst)
11    BEGIN
12      IF rst = '1' THEN                        -- estado inicial
13        q <= "00";
14      ELSIF (ck'EVENT and ck ='1') THEN        -- ciclo de estados
15        CASE q IS
16          WHEN "00" => q <= "01";
17          WHEN "01" => q <= "11";
18          WHEN "11" => q <= "10";
19          WHEN "10" => q <= "00";
20         END CASE;
21      END IF;
22    END PROCESS abc;
23  END teste;
```

Quadro 6.6.1 Exemplo de máquina de estados finitos.

A Figura 6.6.4 ilustra o esquema do circuito sintetizado a partir da descrição. Como o sinal " rst " leva a máquina ao estado " 00 " de modo assíncrono, a ferramenta de síntese interligou esse sinal às entradas " R " dos *flip flops* que armazenam os estados. A lógica combinacional que determina o estado futuro é apenas um inversor interligado na entrada do *flip flop* associado à saída " q(0) ".

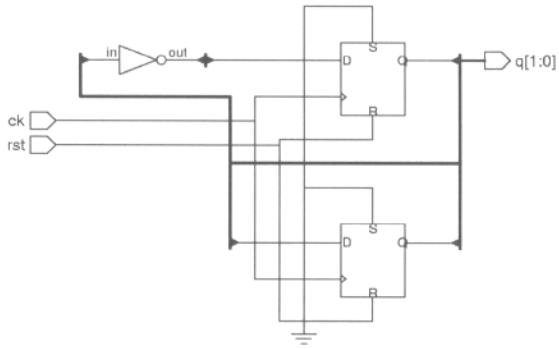


Figura 6.6.4 Circuito, nível RTL, gerado a partir da descrição do Quadro 6.6.1.

Os valores binários assumidos pelos estados podem ser especificados na descrição ou codificados pelo aplicativo de síntese. No segundo caso, é possível criar um tipo enumerado na descrição, composto por valores simbólicos que guardam uma relação com o estado da máquina, permitindo, assim, uma leitura mais fácil do código. O aplicativo de síntese atribui a cada valor simbólico um código binário, segundo um ou mais esquemas. As opções mais comuns são binária, código *Gray* e " único um " ou *one hot*. A Tabela 6.6.1 ilustra os esquemas de atribuição de valores supondo o exemplo de uma máquina com cinco estados.

O esquema " único um " facilita a decodificação do estado futuro, resultando numa redução da lógica combinacional necessária para a implementação, ao custo de um número maior de *flip flops* para o armazenamento dos estados. No caso de dispositivos lógicos programáveis, o esquema " único um " deve ser escolhido quando a tecnologia-alvo disponha de um número elevado *flip flops* em comparação com o número de elementos que realizam as operações combinacionais.

Tabela 6.6.1 Exemplo de codificação de estados.

estado definido no código	binária	código *Gray*	"único um"
reset	000	000	00001
ciclo_rd	001	001	00010
ciclo_wr	010	011	00100
ciclo_int	011	010	01000
ciclo_dma	100	110	10000

O exemplo da Figura 6.6.5 segue o caso mais geral de uma máquina de estados. Nessa máquina, o valor assumido pelo estágio atual não corresponde diretamente ao valor da saída; ele é decodificado a partir do estado atual. O exemplo proposto é um contador crescente / decrescente, com o sentido da contagem definido pela entrada " sobe ". Os estados da máquina são designados como " caso_a ", " caso_b ", " caso_c " e " caso_d ".

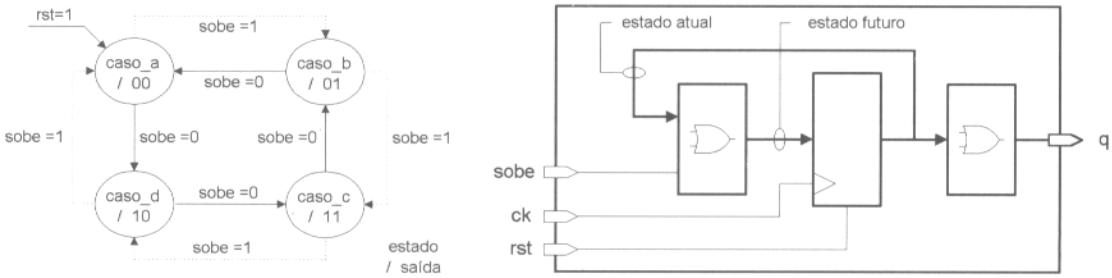


Figura 6.6.5 Outra proposta de máquina de estados.

O código para o exemplo é apresentado no Quadro 6.6.2. A descrição é semelhante ao exemplo anterior, onde a construção " IF ELSE " detecta o sinal " rst " ativo ou a borda de subida do sinal de relógio. O tipo " st " foi definido na linha 10 para representar os estados da máquina, e o estado atual da máquina é armazenado pelo sinal denominado " estado " (ver linha 11). A transição dos estados é estabelecida por uma construção " CASE WHEN " seguida de um teste sobre a condição do sinal " sobe ". O valor do estado atual é decodificado pela construção " WITH SELECT " entre as linhas 39 e 43.

```
 1 ENTITY maq_est2 IS
 2   PORT (ck      :  IN     BIT;                          -- relogio borda subida
 3         sobe    :  IN     BIT;                          -- sobe=1, q=00,01,11,10,00...
 4         iniciar :  IN     BIT;                          -- iniciar=1, q=00
 5         q       :  OUT    BIT_VECTOR (1 DOWNTO 0)); -- saida
 6 END maq_est2;
 7
 8 ARCHITECTURE teste OF maq_est2 IS
 9   --          q=2     q=3     q=1     q=0
10   TYPE st IS (caso_d, caso_c, caso_b, caso_a);  -- novo tipo definido
11   SIGNAL estado : st;                            -- sinal estado tipo "st"
12 BEGIN
13   abc: PROCESS (ck, iniciar)
14   BEGIN
15     IF iniciar = '1' THEN                  -- estado inicial
16       estado <= caso_a;                    -- q=0
17     ELSIF (ck'EVENT and ck ='1') THEN      -- ciclo de estados
18       CASE estado IS
19         WHEN caso_a =>                             -- q=0
20           IF sobe = '1' THEN estado <= caso_b;  -- q=1
21           ELSE               estado <= caso_d;  -- q=2
22           END IF;
23         WHEN caso_b =>                             -- q=1
24           IF sobe = '1' THEN estado <= caso_c;  -- q=3
25           ELSE               estado <= caso_a;  -- q=0
26           END IF;
27         WHEN caso_c =>                             -- q=3
28           IF sobe = '1' THEN estado <= caso_d;  -- q=2
29           ELSE               estado <= caso_b;  -- q=1
30           END IF;
31         WHEN caso_d =>                             -- q=2
32           IF sobe = '1' THEN estado <= caso_a;  -- q=0
33           ELSE               estado <= caso_c;  -- q=3
34           END IF;
35        END CASE;
36     END IF;
37   END PROCESS abc;
38
39   WITH estado SELECT   -- decodifica estado
40     q <= "00" WHEN caso_a, -- q=0
41          "01" WHEN caso_b, -- q=1
42          "11" WHEN caso_c, -- q=3
43          "10" WHEN caso_d; -- q=2
44 END teste;
```

Quadro 6.6.2 Exemplo de máquina de estados finitos.

No Quadro 6.6.3 é apresentada a mensagem gerada pela ferramenta de síntese com relação à atribuição de estados estabelecida para a descrição " maq_est2 ". Conforme se pode observar, a atribuição dos estados definida é binária, e o código atribuído ao primeiro valor relacionado na declaração do tipo, " caso_d ", é " 00 ". Os valores seguintes relacionados na declaração " TYPE " recebem progressivamente os códigos " 01 ", " 10 " e " 11 ".

Na Figura 6.6.6 é apresentado o circuito sintetizado, nível porta lógica, para essa descrição. Observe que a inicialização assíncrona do estado inicial, " caso_a ", é corretamente inferida. O sinal " iniciar " é interligado aos terminais que realizam a operação *set* assíncrono dos *flip flops*. A condição " iniciar =1 " leva a saída dos dois elementos de memória a " 11 ", o que corresponde ao código do estado " caso_a ".

Conforme visto, a ferramenta de síntese atribui o código binário " 00 " para o primeiro valor enumerado da declaração " TYPE ". Assim, como nem todas as tecnologias dispõem de *flip flops* com operação de *set*, sempre que possível o primeiro valor enumerado deve corresponder ao estado inicial. Desse modo, máquinas com inicialização assíncrona são facilmente levadas ao estado inicial, com uma operação de *reset* nos registradores que armazenam o estado.

```
"/vhdl/maq_est2.vhd",line 10: Info, Enumerated type st with 4 elements encoded as binary.
Encodings for st values
        value     st[1-0]
========================
         caso_d   00
         caso_c   01
         caso_b   10
         caso_a   11
```

Quadro 6.6.3 Mensagem da ferramenta de síntese referente à atribuição de estados.

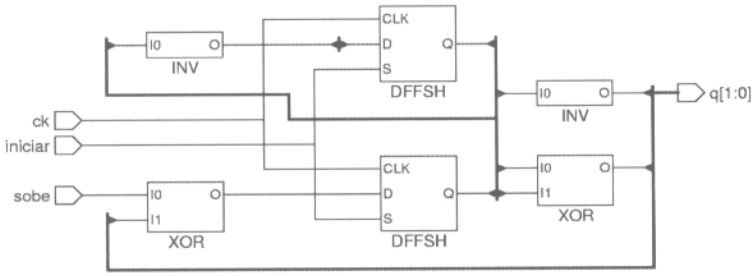


Figura 6.6.6 Descrição " maq_est2 " sintetizada - nível porta lógica.

6.7 Inferência de registrador empregando variável

As ferramentas de síntese podem inferir as estruturas empregando sinais ou variáveis. A diferença no emprego de um ou outro objeto reside no desempenho da simulação. A atribuição de um valor em um sinal não é imediata, e requer a construção de uma tabela para o agendamento da atribuição do valor. No caso de variáveis, a atribuição de um novo valor ocorre na execução do comando, demandando, assim, um menor espaço de memória para simulação.

Nos modelos apresentados, os elementos de memória são inferidos com construções utilizando sinais. Uma variável pode ser empregada, também, para descrever um elemento de memória. A primeira consideração a ser feita é quanto ao fato de uma variável ser visível unicamente no processo em que ela foi declarada. Assim, a transferência do valor armazenado pela variável para fora do processo deve ser feita através de um sinal. A segunda consideração é relativa ao fato de uma variável receber o seu valor imediatamente, enquanto um sinal recebe o valor da expressão que o controla somente 1Δ após. Desse modo, um sinal no interior de um processo tem o seu valor atualizado na suspensão da execução do processo.

O Quadro 6.7.1 contém duas descrições semelhantes, uma empregando sinal e a outra empregando uma variável, para inferência de registradores. As regiões de código de interesse foram alinhadas para facilitar a comparação entre os sinais " s1 " e " s2 ", e as variáveis " v1 " e " v2 ". Note que os sinais " s1 " e " s2 " são visíveis, unicamente, nos blocos em que são definidos, assim como as variáveis " v1 " e " v2 " são visíveis unicamente nos respectivos processos.

Na linha 26 da descrição " flip_1s ", " s2 " recebe um novo valor após a suspensão da execução do processo. Na linha 27, o valor a ser atribuído a " q2 " corresponde ao valor antigo de " s2 ", pois não ocorreu, ainda, uma iteração. Desse modo, são necessários dois *flip flops*: um para armazenar o valor de " d2 " e um outro para armazenar o valor de " s2 ". No processo " ijk " da descrição " flip_1v " ocorrem as seguintes operações: a variável " v2 " recebe o valor de " d2 " imediatamente, e " q2 ", através de " v2 ", recebe o valor de " d2 " após a suspensão da execução do processo. Assim, devido ao fato de a variável " v2 " receber um novo valor antes de ter sido referenciada no processo, não é necessário o armazenamento do valor, e, portanto, somente um *flip flop* deve ser empregado.

No caso do processo " fgh " da descrição " flip_1v ", ocorre algo diferente: a variável " v1 " é referenciada antes de ter recebido um novo valor. Na linha 12 a variável " v1 " é utilizada, e somente na linha 13 ela recebe um novo valor. Assim, para armazenar o valor anterior de " v1 ", é necessário um *flip flop*, bem como para armazenar o valor de " d1 ". Com relação ao processo " abc " da descrição " flip_1s ", as observações são semelhantes ao processo " cde ".

```
1  ENTITY flip_1s IS
2    PORT (ck, d1, d2  : IN  BIT;
3          q1, q2  : OUT BIT);
4  END flip_1s;
5
6  ARCHITECTURE teste OF flip_1s IS
7  BEGIN
8   xyz: BLOCK
9    SIGNAL s1 : BIT;
10   BEGIN
11    abc: PROCESS (ck)
12    BEGIN
13     IF (ck'EVENT AND ck='1') THEN
14      q1 <= s1; -- q1 e s1 recebem os valores
15      s1 <= d1; -- antigos 1 delta apos
16     END IF;
17    END PROCESS abc;
18   END BLOCK xyz;
19
20   wky: BLOCK
21    SIGNAL s2 : BIT;
22   BEGIN
23    cde: PROCESS (ck)
24    BEGIN
25     IF (ck'EVENT AND ck='1') THEN
26      s2 <= d2; -- s2 e q2 recebem os valores
27      q2 <= s2; -- antigos 1 delta apos
28     END IF;
29    END PROCESS cde;
30   END BLOCK wky;
31 END teste;
```

```
1  ENTITY flip_1v IS
2    PORT (ck, d1, d2 : IN  BIT;
3          q1, q2     : OUT BIT);
4  END flip_1v;
5
6  ARCHITECTURE teste OF flip_1v IS
7  BEGIN
8   fgh: PROCESS (ck)
9    VARIABLE v1 : BIT;

10 BEGIN
11   IF (ck'EVENT AND ck='1') THEN
12    q1 <= v1; --v1 utilizada sem receber novo valor
13    v1 := d1; --v1 recebe um novo valor
14   END IF;
15  END PROCESS fgh;
16

17  ijk: PROCESS (ck)
18   VARIABLE v2  : BIT;

19  BEGIN
20   IF (ck'EVENT AND ck='1') THEN
21    v2 := d2; --v2 recebe novo valor sem ser empregada
22    q2 <= v2; --nao e' necessario armazenar o valor
23   END IF;
24  END PROCESS ijk;
25 END teste;
```

Quadro 6.7.1 Descrições empregando sinais e variáveis - inferência de registradores.

Em resumo, não é inferido um registrador, se uma variável recebe um novo valor antes de ser empregada. Isso pode ser entendido, porque o último valor assumido pela variável não é necessário na nova execução do processo. Se a variável é empregada antes de ser atualizada, é inferido um registrador. Nesse caso, o último valor da variável, assumido após a suspensão do processo, é necessário na nova execução do processo, e um registrador deve ser empregado para o armazenamento.

Na Figura 6.7.1 são apresentados os circuitos resultantes da síntese das descrições " flip_1s " e " flip_1v ". Conforme se pode observar, para o circuito correspondente à descrição " flip_1v ", apenas um *flip flop* é inserido entre a entrada " d2 " e a saída " q2 ". Para os casos restantes, dois *flip flops* são necessários entre as entradas e saídas.

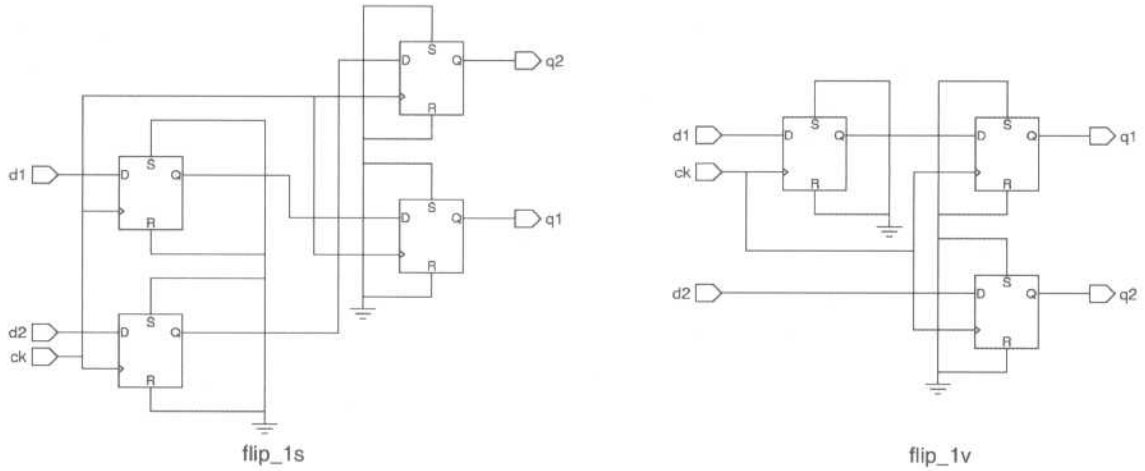


Figura 6.7.1 Circuitos sintetizados, nível RTL, descrições " flip_1s " e " flip_1v ".

6.8 Contadores

Um contador pode ser descrito como uma máquina de estados, onde o código de cada estado corresponde ao valor da contagem. Uma outra opção para descrição de contadores é o emprego de operações aritméticas de soma ou subtração controladas por um sinal de relógio.

O Quadro 6.8.1 apresenta uma descrição que realiza uma contagem entre os valores " 0 " e " 9 " com *reset* assíncrono e carga de dados paralela síncrona. A operação de carga é controlada pelo sinal " ld ". Como o incremento do contador é realizado através de uma operação de soma, o tipo dos sinais " d " e " q ", bem como o da variável " qv " foram declarados como inteiro. O código é similar aos exemplos anteriores, diferindo pelo emprego de uma variável, em vez de um sinal, para definição do registrador que armazena o valor da contagem. O teste executado na linha 20 leva o contador a um estado previsto, um valor entre " 0 " e " 9 ", caso o valor da contagem exceda " 9 ". Isso previne problemas devido à falta de inicialização ou carga de um valor incorreto.

```
 1  ENTITY cnt_dec2 IS
 2    PORT (ck   : IN  BIT;                  -- relogio
 3          ld   : IN  BIT;                  -- carrega dados sincrono
 4          rst  : IN  BIT;                  -- rst=1 leva q=0000 assincrono
 5          d    : IN  INTEGER RANGE 15 DOWNTO 0;
 6          q    : OUT INTEGER RANGE 15 DOWNTO 0);
 7  END cnt_dec2;
 8
 9  ARCHITECTURE teste OF cnt_dec2 IS
10
11  BEGIN
12    abc: PROCESS (ck, rst)
13      VARIABLE qv : INTEGER RANGE 15 DOWNTO 0;
14    BEGIN
15      IF    (rst = '1') THEN                 -- q=0000 independente de ck
16        qv := 0;
17      ELSIF (ck'EVENT AND ck = '1') THEN     -- condicao do sinal relogio
18        IF (ld = '1') THEN qv := d;          -- carrega dados
19        ELSE
20          IF qv >= 9  THEN qv := 0;          -- retorna a zero
21          ELSE             qv := qv +1;      -- incrementa
22          END IF;
23        END IF;
24      END IF;
25      q <= qv;
26    END PROCESS abc;
27  END teste;
```

Quadro 6.8.1 Exemplo de um contador.

Uma das mensagens geradas pela ferramenta de síntese é apresentada no Quadro 6.8.2. Ela relata que foi inferida a necessidade de um contador na linha 21 do código. Nessa linha, o valor da variável " qv " é incrementado de um, e a ferramenta emprega a primitiva denominada " counter_up_sload_sclear_aclear_clock4 " para realizar essa operação.

```
"/vhdl/cnt_dec2.vhd", line 21:Info, Inferred counter instance 'q' of type
'counter_up_sload_sclear_aclear_clock_4'
```

Quadro 6.8.2 Mensagem da ferramenta de síntese.

A Figura 6.8.1 ilustra o circuito gerado a partir da descrição no nível RTL. A primitiva que realiza a contagem, " counter_up_sload_sclear_aclear_clock4 ", é um módulo contador com opções para carga síncrona, *reset* síncrono, *reset* assíncrono, contagem crescente ou decrescente, entre outras. O bloco representado por uma circunferência contendo o símbolo " > " realiza o teste da linha 20 do código. Caso a contagem tenha atingido um valor maior ou igual a nove, é realizada uma operação de *reset* síncrona no contador via porta lógica " e " ligada ao terminal " sclear " do módulo contador.

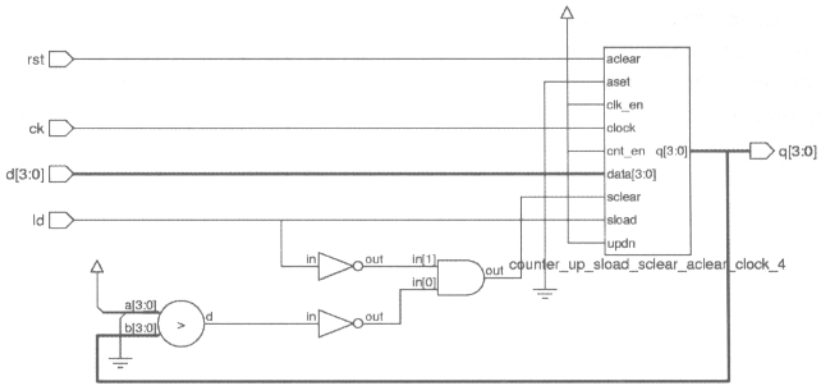


Figura 6.8.1 Contador sintetizado - nível RTL.

6.9 Alternativas para descrição de registradores

Os modelos apresentados para descrição de registradores empregam a construção " IF ELSE ". Registradores podem, também, ser inferidos em regiões de código concorrente e em regiões de código seqüencial empregando o comando " WAIT ". Deve-se observar, entretanto, que esses modelos nem sempre são suportados pelas ferramentas de síntese, e sofrem restrições quando aceitos.

O Quadro 6.9.1 ilustra uma descrição contendo outras opções para descrição de um *flip flop*. O primeiro modelo (linhas 10 a 12) emprega a construção " WAIT UNTIL " que suspende a execução do processo até a ocorrência de uma borda de subida no sinal de relógio. Deve-se lembrar que um processo com um comando" WAIT " não deve conter lista de sensibilidade. Operações de inicialização assíncrona não são, normalmente, suportadas pelas ferramentas de síntese nesse modelo.

O segundo modelo (linhas 14 e 15) emprega a construção concorrente " WHEN ELSE ". Uma opção semelhante foi apresentada no Capítulo 3 para o caso de um *latch* transparente. Finalmente, o último modelo (linhas 17 a 20) descreve um *flip flop* através de um bloco e uma expressão de guarda. A expressão de guarda (linha 17) detecta a ocorrência de uma borda de subida, e o comando da linha 19 fica desabilitado enquanto não ocorrer essa condição. Deve-se observar que blocos contendo o comando " GUARDED " nem sempre são suportados pelas ferramentas de síntese. Na Figura 6.9.1 é apresentado o circuito sintetizado a partir da descrição " flip1_5 ", e, conforme se pode observar, todos os *flip flops* foram inferidos corretamente.

```
 1  ENTITY flip1_5 IS
 2    PORT (ck, d1, d2, d3 : IN     BIT;
 3              q1, q2, q3 : BUFFER BIT);
 4  END flip1_5;
 5
 6  ARCHITECTURE teste OF flip1_5 IS
 7  BEGIN
 8    PROCESS
 9    BEGIN
10     WAIT UNTIL (ck'EVENT AND ck='1');
11      q1 <= d1;
12    END PROCESS;
13
14    q2 <= d2 WHEN (ck'EVENT AND ck='1') ELSE
15          q2;
16
17    xyz: BLOCK (ck'EVENT AND ck='1')
18    BEGIN
19      q3 <= GUARDED d3;
20    END BLOCK xyz;
21
22  END teste;
```

Quadro 6.9.1 Outras opções para descrição de registradores.

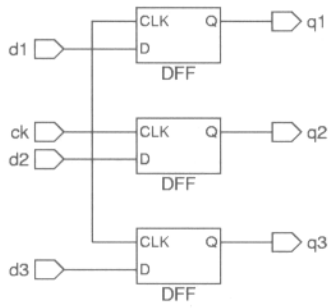


Figura 6.9.1 Circuito sintetizado - nível porta lógica - descrição " flip1_5 ".

6.10 Cuidados na descrição

Sinais na lista de sensibilidade

Os sinais de *reset* e *set* assíncronos devem, obrigatoriamente, estar na lista de sensibilidade do processo. A ausência leva a uma dependência na execução do processo com os sinais relacionados na lista de sensibilidade, não caracterizando, assim, uma operação assíncrona desses sinais. Deve-se notar, entretanto, que a ausência desses sinais não caracteriza um erro de linguagem; apenas o comportamento não será assíncrono.

As ferramentas de síntese, por sua vez, possuem limitações quanto aos circuitos possíveis de serem sintetizados. Uma descrição com inicialização assíncrona, sem a inserção dos sinais responsáveis na lista de sensibilidade, é uma dessas limitações. A ferramenta, não sendo capaz de gerar um circuito que corresponda à descrição segundo a semântica da linguagem, pode alertar a ausência dos sinais assíncronos na lista de sensibilidade e implementar o circuito da melhor forma possível.

O Quadro 6.10.1 contém uma descrição de dois *flip flops*. O primeiro, processo " abc ", segue um modelo que caracteriza uma operação assíncrona do *reset*, mas o sinal responsável por essa operação, no caso " rst ", não foi incluído na lista de sensibilidade. O segundo segue o modelo de operação do *reset* síncrono. Nesse caso, como a operação do sinal " rst " é síncrona, não é necessário incluí-lo na lista de sensibilidade.

A descrição, submetida à ferramenta de síntese, gera uma mensagem de aviso. A mensagem notifica que, no processo " abc ", o sinal " rst " não está incluído na lista de sensibilidade (ver Quadro 6.10.2). No processo " cde " uma mensagem semelhante não é gerada, pois, nesse caso, a operação deste sinal é síncrona com " ck ". A mensagem alerta, portanto, que a ferramenta é incapaz de sintetizar a descrição contida no processo " abc " conforme a semântica da linguagem.

```
 1  ENTITY flip1_x IS
 2    PORT (ck, rst, d : IN  BIT;
 3          q1, q2     : OUT BIT);
 4  END flip1_x;
 5
 6  ARCHITECTURE errada OF flip1_x IS
 7
 8  BEGIN
 9    abc: PROCESS (ck)          -- set e reset removidos da lista
10    BEGIN                      -- descricao segue modelo assincrono para reset
11      IF    (rst = '1') THEN              q1 <= '0';
12      ELSIF (ck'EVENT AND ck = '1') THEN  q1 <= d;
13      END IF;
14    END PROCESS abc;
15
16    def: PROCESS (ck)         -- nao e' necessario incluir rst na lista
17    BEGIN                     -- descricao reset sincrono
18      IF (ck'EVENT AND ck = '1') THEN
19        IF   (rst = '1') THEN  q2 <= '0';
20        ELSE                   q2 <= d;
21        END IF;
22      END IF;
23    END PROCESS def;
24  END errada;
```

Quadro 6.10.1 Descrição para verificar a influência da lista de sensibilidade na síntese.

O resultado da síntese é apresentado na Figura 6.10.1. O *flip flop* associado à porta de saída " q1 " corresponde ao processo " abc ", e o *flip flop* associado à porta de saída " q2 " corresponde ao processo " cde ". Conforme se pode observar, o fato de o sinal " rst " não estar incluído na lista de sensibilidade do processo " abc " foi detectado pela ferramenta de síntese; entretanto, devido à incapacidade de gerar um circuito que corresponda à descrição, foi implementado um *flip flop* com " reset " assíncrono.

```
-- Compiling root entity flip1_x(errada)
"/vhdl/flip1_x.vhd",line 11: Warning, rst should be declared on the sensitivity list of the process.
```

Quadro 6.10.2 Mensagens de aviso - sinal não-incluído na lista de sensibilidade.

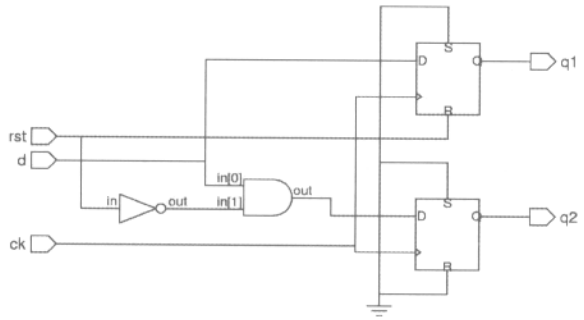


Figura 6.10.1 Circuito sintetizado nível RTL - descrição do Quadro 6.10.1.

Ciclo de inicialização

A descrição, assim como o projeto de um circuito síncrono, deve sempre prever um ciclo de inicialização. Isso é necessário porque o valor dos registradores logo após a energização do circuito é desconhecido, diferindo dos valores iniciais escolhidos pela ferramenta de simulação. Não incluir um ciclo de inicialização pode levar a um comportamento não previsto pela simulação, como iniciar em um estado diferente ou num estado não previsto na descrição. Desse modo, operações de inicialização assíncronas ou síncronas devem ser sempre previstas no projeto.

Inserção desnecessária de elementos de memória

Conforme visto, na linguagem VHDL a falta de atribuição de um valor em um objeto implica a manutenção do valor nesse objeto. Assim, uma transição de estados especificada incompletamente, ou a falta de atribuição de um valor a uma saída em uma determinada condição, leva à inserção de um *latch* desnecessário.

O emprego da construção " CASE WHEN " na especificação da transição de estados é conveniente para esse tipo de problema, pois todas as condições devem ser obrigatoriamente cobertas. Deve-se observar, entretanto, que o uso da cláusula " OTHERS " para cobertura das condições restantes nem sempre é vantajoso com relação à síntese do circuito. Uma lógica combinacional extra pode ser gerada para detectar os estados restantes englobados pela cláusula " OTHERS ". Uma opção para contornar esse problema é a criação de um tipo enumerado definindo todos os estados previstos para a operação da máquina, evitando assim o uso da cláusula " OTHERS ".

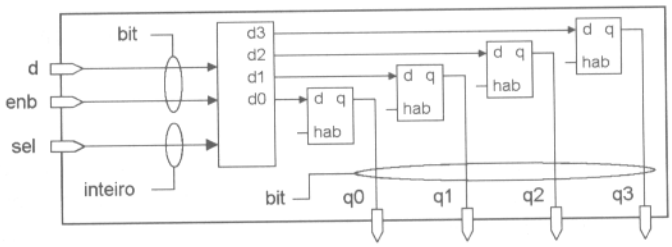
6.11 Principais pontos abordados

- Registradores sensíveis a nível e registradores sensíveis a borda podem ser descritos com estruturas semelhantes. O teste que habilita a memorização, no caso de um registrador sensível a nível, verifica o nível do sinal de controle. No caso de um registrador sensível a borda, o teste que qualifica o armazenamento detecta duas condições: a ocorrência de uma transição e o nível lógico final, no sinal de controle.
- A descrição de um circuito deve, sempre que possível, considerar as estruturas disponíveis na tecnologia-alvo. Com isso, o circuito final gerado terá um menor custo de implementação e melhor desempenho. No caso de registradores, a descrição de um comportamento que não possua correspondência com os elementos disponíveis na tecnologia-alvo pode levar a condições que impossibilitem a ferramenta de síntese gerar o circuito descrito.

- Máquinas de estados podem ser descritas de modo semelhante a registradores. Uma construção " CASE WHEN " define transição de estados, e uma construção " IF ELSE " controla a transição de estados.
- A definição de um tipo enumerado é um modo prático para a descrição dos estados de uma máquina. Cada valor definido no tipo corresponde a um dos estados da máquina. As ferramentas de síntese suportam esse tipo de descrição, e a atribuição dos códigos para cada estado é realizada no momento da síntese. Diferentes modos de codificação são disponíveis nas ferramentas.
- Registradores podem ser inferidos através de sinais ou variáveis. Para a inferência de registradores com variáveis, é necessário que ela seja empregada antes de receber um novo valor.
- Sinais de inicialização assíncrona devem estar, obrigatoriamente, incluídos na lista de sensibilidade do processo. A não-inclusão leva a um comportamento não corretamente sintetizado pelas ferramentas de síntese.
- Circuitos síncronos devem ter um ciclo de inicialização. A falta desse procedimento pode levar a condições não previstas na simulação, pois o conteúdo dos registradores implementados é indeterminado no momento da energização do circuito.

6.12 Exercícios

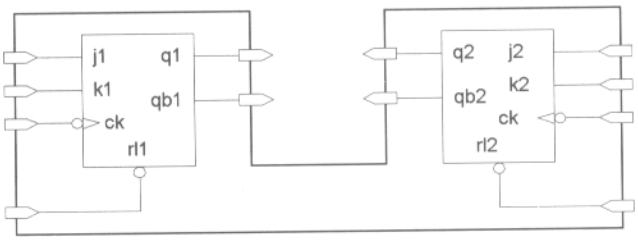
6.12.1 Apresente a descrição de um registrador sensível a nível de quatro bits. A entrada dos dados é feita por uma única entrada " d ". A seleção do elemento de memória é feita pela entrada " sel " do tipo inteiro. Uma vez selecionado o elemento, a saída correspondente segue a entrada " d " com " enb =1 ". A informação é mantida nas saídas com " enb =0 ". A Figura 6.12.1 ilustra o circuito proposto.



sel	enb	d	q0	q1	q2	q3
X	0	X	q0	q1	q2	q3
0	1	d_a	d_a	q1	q2	q3
1	1	d_b	q0	d_b	q2	q3
2	1	d_c	q0	q1	d_c	q3
3	1	d_d	q0	q1	q2	d_d

Figura 6.12.1 Ilustração para o Exercício 6.12.1.

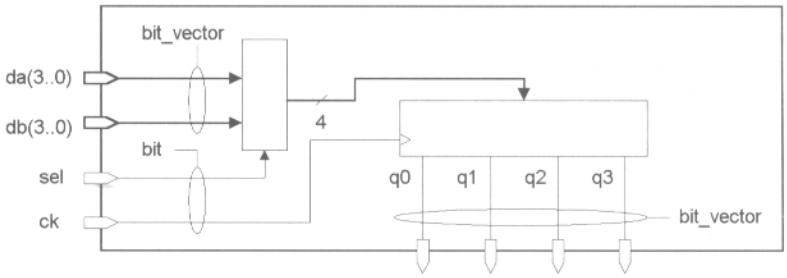
6.12.2 Apresente a descrição de um *flip flop* tipo JK equivalente ao integrado 7473 da família TTL. A entrada " rl " realiza uma operação de *reset* assíncrona, a entrada " ck " é o sinal de relógio sensível a borda de descida, e as entradas " j " e " k " definem a operação no *flip flop*. Todas as entradas e saídas são do tipo " bit ". Veja a Figura 6.12.2.



	rl	ck	j	k	q	qb
reset ass.	0	x	x	x	0	1
troca	1	↘	1	1	$\overline{q}$	$\overline{qb}$
reset	1	↘	0	1	0	1
set	1	↘	1	0	1	0
mantém	1	↘	0	0	q	qb

Figura 6.12.2 Ilustração para o Exercício 6.12.2.

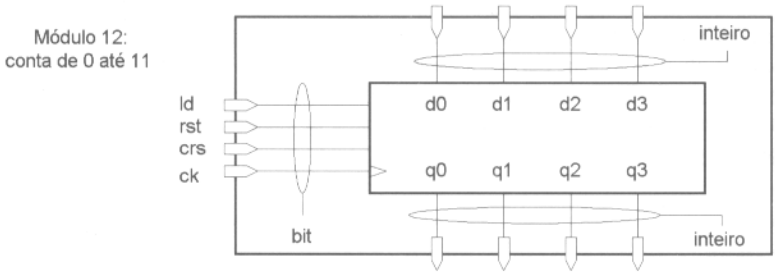
6.12.3 Apresente a descrição de um registrador com 4 bits e duas entradas de 4 bits, " da " e " db ". O sinal de seleção " sel " define de qual entrada o dado será armazenado na borda de descida do sinal de relógio " ck ". A operação do circuito é similar ao registrador 74298 da família TTL. Veja a Figura 6.12.3.



ck	sel	da	db	q
0	X	X	X	q
1	X	X	X	q
↗	1	d_a	X	d_a
↗	0	X	d_b	d_b

Figura 6.12.3 Ilustração para o Exercício 6.12.3.

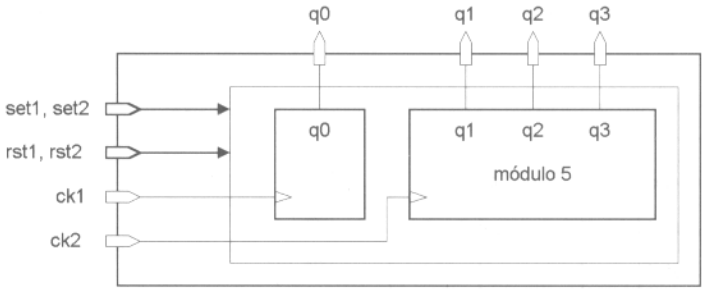
6.12.4 Apresente a descrição de um contador com 4 bits crescente/decrescente com opção para carga de dados paralela e *reset* assíncrono. O contador deve contar entre os valores " 0 " a " 11 ". A Figura 6.12.4 ilustra o contador. O terminal " ld " permite o armazenamento do dado " d " na borda de subida do sinal de relógio " ck ". O terminal " crs " define se a contagem é crescente ou decrescente, e " rst " executa uma operação de *reset* assíncrona. Os sinais " d " e " q " são do tipo inteiro, e os sinais restantes do tipo " bit ".



crs	rst	ld	q3	q2	q1	q0
X	1	X	0	0	0	0
X	0	1	d3	d2	d1	d0
1	0	0	cont. decresc.			
0	0	0	cont. cresc.			

Figura 6.12.4 Ilustração para o Exercício 6.12.4.

6.12.5 Apresente a descrição de um contador decimal equivalente ao circuito integrado da família TTL tipo 7490. A tabela contendo os modos de operação do componente está apresentada na Figura 6.12.5. Os terminais " rst1 " e " rst2 " levam as saídas ao nível lógico " 0000 ", e os terminais " set1 " e " set2 " levam as saídas ao nível lógico " 1001 ". Ambas as operações são executadas de maneira assíncrona. O circuito é composto por um contador módulo 5 e um *flip flop*; a contagem módulo 10 é atingida com uma ligação pseudo-síncrona entre o *flip flop* e o sinal de relógio do contador módulo 5. A saída do contador deve ser do tipo " BIT_VECTOR ", e os demais sinais são do tipo " bit ".

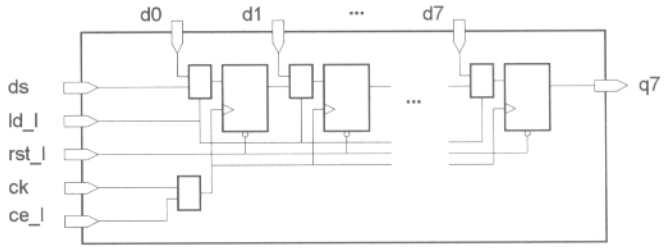


rst2	rst1	set2	set1	q3	q2	q1	q0
1	1	0	X	0	0	0	0
1	1	X	0	0	0	0	0
X	X	1	1	1	0	0	1
0	X	0	X	contagem			
X	0	X	0	contagem			
0	X	X	0	contagem			
1	0	0	X	contagem			

Figura 6.12.5 Ilustração para o Exercício 6.12.5.

6.12.6 Apresente a descrição de um registrador de deslocamento de 4 bits. A entrada serial dos dados é feita pelo terminal " de ", e a saída paralela dos dados é feita pelos terminais " qs(3) ", " qs(2) ", " qs(1) " e " qs(0) ". O deslocamento é processado na borda de subida do sinal " ck ". Empregue variáveis para armazenar os dados.

6.12.7 Apresente a descrição de um registrador de deslocamento equivalente ao circuito integrado 74166 da família TTL. O registrador contém 8 bits, possibilidade de carga paralela através das entradas " d0 " a " d7 " e uma entrada serial, terminal " ds ". O terminal " rst_l " leva a saída de todos os registradores ao nível lógico baixo de modo assíncrono. O terminal " ld_l " permite a carga dos dados presentes nas entradas de " d0 " a " d7 " na borda de subida do sinal " ck ". A operação de deslocamento é executada, também, na borda de subida do sinal " ck ". O terminal " ce_l " habilita as operações síncronas: deslocamento e carga paralela. A Figura 6.12.6 ilustra simplificadamente o componente e uma tabela contendo os modos de operação do registrador.



rst_l	ld_l	ce_l	ck	operação
0	X	X	X	reset assíncrono
1	0	0	↗	carga paralela
1	1	0	↗	desloca
1	X	1	X	desabilitado

Figura 6.12.6 Ilustração para o Exercício 6.12.7.

6.12.8 Projete um controlador de semáforo para trânsito composto por três lâmpadas: vermelha, amarela e verde. O sinal " tempo " fornece a temporização do sistema, de modo que a cada pulso desse sinal é feita a transição para um novo estado. O sinal " inicio " leva o semáforo imediatamente para o estado inicial, causando o acendimento da lâmpada vermelha. O sinal " espera " mantém o estado atual, nos casos em que a lâmpada vermelha ou a lâmpada verde estão acesas. Esse sinal permite que as lâmpadas vermelha e verde fiquem acesas por um período maior de tempo. O sinal " noite " pisca a lâmpada amarela simulando a operação de um semáforo na madrugada. O diagrama de estados completo é apresentado na Figura 6.12.7. Os tipos dos sinais são indicados na figura.

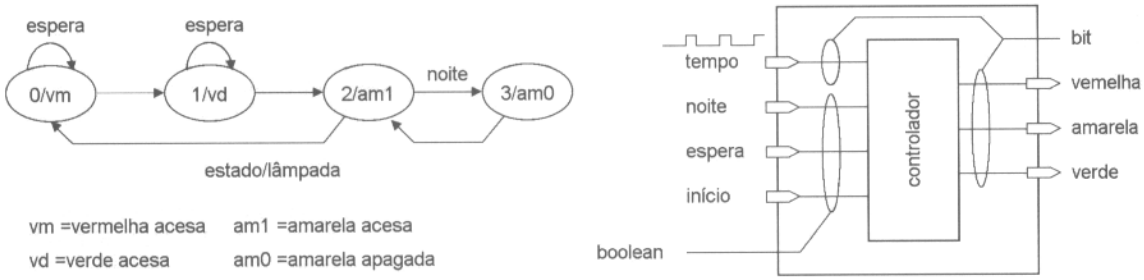


Figura 6.12.7 Ilustração para o Exercício 6.12.8.

6.12.9 Projete um controlador de endereços para uma memória RAM dinâmica. A memória contém 8 entradas para seleção de endereço e duas linhas denominadas " RAS " e " CAS " para controle. O controlador, para executar um ciclo de endereçamento, deve seguir as operações descritas na Figura 6.12.8. O sinal " ce_l =0 ", na borda de subida da base de tempo " ck ",

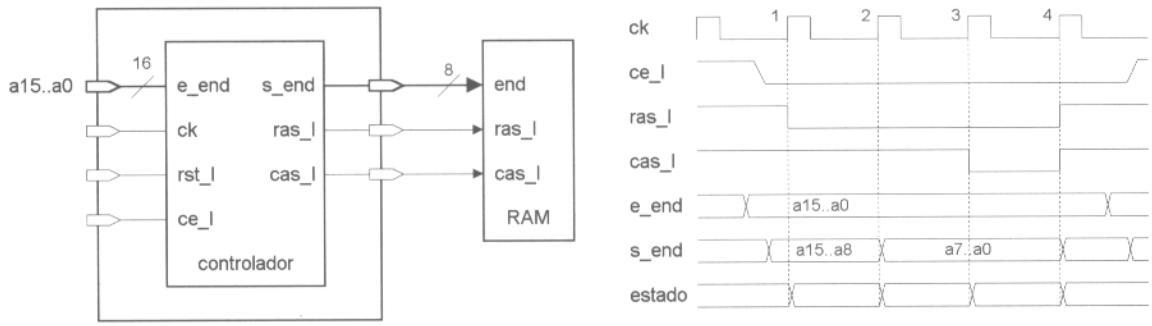


Figura 6.12.8 Ilustração para o Exercício 6.12.9.

inicia o ciclo ativando o sinal de seleção de coluna " ras_l ". Nesse estado, as linhas de endereço " s_end " correspondem aos 8 bits mais significativos das linhas de endereço da entrada. No segundo ciclo de relógio, as linhas " s_end " recebem os 8 bits menos significativos das linhas de endereço da entrada. No terceiro ciclo, o sinal " cas_l " é ativado, e no quarto ciclo de relógio a operação é terminada com os sinais " ral_l " e " cas_l " desativados.

6.12.10 O Quadro 6.7.1 contém uma descrição para inferência de *flip flops* empregando variáveis, entidade " flip_1v ". Altere a descrição de modo que os elementos inferidos sejam *latches*, em vez de *flip flops*.

6.12.11 Apresente o código para descrever o comportamento da máquina de estados ilustrada na Figura 6.12.9. Note que esta é uma máquina *Mealy*, onde o valor da saída depende do valor da entrada.

6.12.12 Sintetize a descrição proposta para o Exercício 6.12.11. Imponha para a ferramenta de síntese uma codificação binária para a atribuição de estados. Verifique o resultado, e observe o número de *flip flops* empregado para armazenar os estados.

6.12.13 Repita o Exercício 6.12.12 impondo, agora, uma codificação "único um" na atribuição de estados. Verifique o resultado, e compare o número de *flip flops* empregado para armazenar os estados com relação ao exercício anterior.

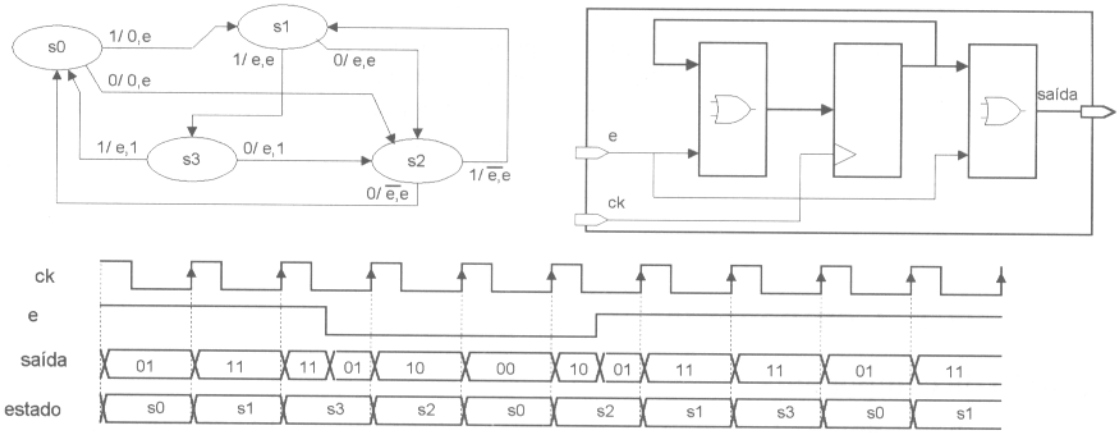


Figura 6.12.9 Ilustração para o Exercício 6.12.11.

CAPÍTULO

7 COMPONENTES E ESQUEMAS DE ITERAÇÃO

Um componente é uma entidade de projeto empregada na arquitetura de uma outra entidade. A utilização desses elementos permite a interligação de múltiplas entidades de projeto, de modo a formar uma entidade mais complexa em um projeto hierárquico. A interligação dos componentes é similar à definição de uma rede de ligações em um diagrama esquemático, onde todos os pontos interligados recebem uma identificação.

A linguagem possui construções que permitem repetir os comandos concorrentes e seqüenciais dentro de um esquema iterativo. Como a solicitação de um componente é um comando concorrente, é possível definir a geração automática de circuitos regulares através da interligação desses elementos. No caso de comandos seqüenciais, rotinas de conversão, e até mesmo estruturas regulares podem ser criadas.

7.1 Componentes

Declaração de componente

Para a utilização de um componente em uma arquitetura, é necessário que se faça uma declaração do componente. A declaração é semelhante a uma declaração de entidade, empregando a palavra reservada " COMPONENT " no lugar da palavra reservada " ENTITY ". Nela podem ser apresentadas uma lista de portas e uma lista de genéricos. O Quadro 7.1.1 ilustra o exemplo de uma declaração de um componente.

```
COMPONENT nome_componente_x
  GENERIC (n          : tipo_n   := valor);          -- lista de genericos
  PORT  (sinal_a      : modo_a   tipo_sinal_a;       -- lista de portas
         sinal_b      : modo_b   tipo_sinal_b;
         sinal_c      : modo_c   tipo_sinal_c);
END COMPONENT;
```

Quadro 7.1.1 Declaração de um componente.

A declaração de um componente pode ser inserida no corpo da arquitetura ou em pacotes. Neste capítulo, os exemplos abordam unicamente a primeira opção; a inserção da declaração em um pacote é apresentada no Capítulo 9. A inclusão da declaração de um componente em um pacote é conveniente nos casos em que um mesmo componente é empregado por diferentes arquiteturas, pois esse recurso evita a repetição da declaração em cada arquitetura. Basta, apenas, tornar o pacote visível para a arquitetura.

Solicitação de componente

O Quadro 7.1.2 apresenta exemplos para solicitação de um componente numa descrição. A solicitação é um comando concorrente, e consiste em um rótulo de denominação qualquer, o nome de um componente declarado e o mapa de portas, ou " PORT MAP ". O mapa contém uma lista que permite estabelecer uma associação entre os sinais definidos na declaração do componente e os sinais definidos na entidade que solicita o componente.

A lista de sinais no mapa pode seguir a mesma ordem estabelecida na declaração do componente, notação posicional, ou uma nova seqüência definida no mapa, notação nomeada. A palavra reservada " OPEN " indica

que o sinal do componente solicitado não é conectado a nenhum sinal da arquitetura. Algumas restrições se aplicam a esse caso. Considerando, por exemplo, a solicitação " x3 " do Quadro 7.1.2. Se " sinal_y " for do modo " IN ", ele deve ser declarado com um valor inicial na declaração componente. Para os outros modos é possível que nenhuma associação seja estabelecida, excetuando o caso de vetores sem limites definidos.

```
--rotulo    nome          mapa      lista
   x1:  componente_x  PORT MAP( a(1), b(1), c(1));                                  -- posicional
   x2:  componente_x  PORT MAP(sinal_b => b(1), sinal_a => a(1), sinal_c => c(1)); -- nomeada
   x3:  componente_y  PORT MAP(sinal_x => z(1), sinal_y => OPEN);                   -- nomeada
```

Quadro 7.1.2 Solicitação de componentes na descrição.

Na Figura 7.1.1, a arquitetura " final " da entidade " topo " solicita dois componentes com rótulos " u1 " e " u2 ". No mapa de portas de " u1 " é estabelecida a conexão entre os sinais " k1 ", " k2 " e " j1 ", e as portas " x ", " y " e " z " do componente " soma ". De modo semelhante, é estabelecida a interligação entre " j1 ", " j2 ", " js " e " jr ", e as portas " a ", " b ", " s " e " o " do componente " alu ". Note que o sinal " j1 " define a ligação entre as portas " z " e " a " das solicitações de componentes " u1 " e " u2 ".

Os nomes dos componentes declarados são idênticos aos nomes das entidades associadas aos componentes. O mesmo se aplica aos nomes das portas. Essa é a maneira mais simples de se estabelecer um elo de correspondência entre uma entidade e uma declaração de componente. No Capítulo 14, tópico de configurações, é apresentado como especificar essas associações em casos de nomes diferentes, entidades com mais de uma arquitetura e bibliotecas distintas.

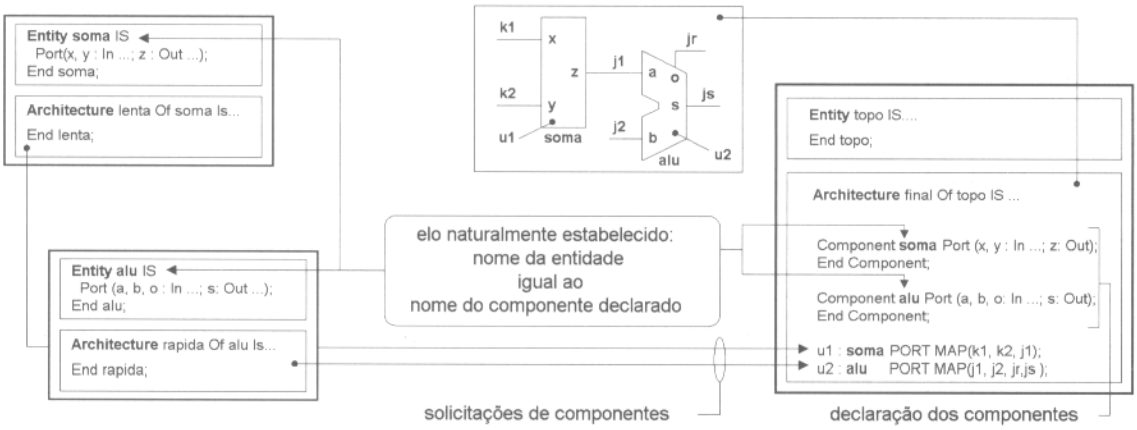


Figura 7.1.1 Exemplo de elo entre componente solicitado e entidade.

A Figura 7.1.2 ilustra o exemplo de um somador de 4 bits composto por quatro somadores completos de 1 bit. As entradas " x " e " y " correspondem aos operandos, e a saída " s " é o resultado da soma. A saída " zs " e a

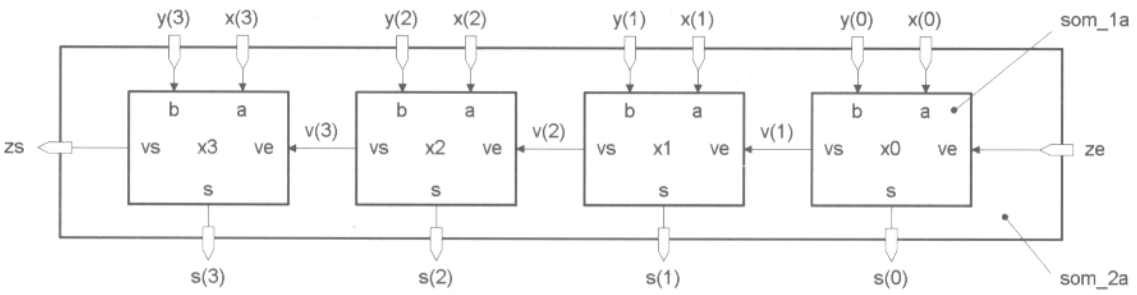


Figura 7.1.2 Somador de 4 bits.

entrada " ze " permitem a expansão do módulo, onde " zs " corresponde ao " vai um " da operação de soma dos quatro bits, e " ze " corresponde ao " vem um " de um outro módulo. Internamente, a soma de cada bit é realizada por um somador completo que avalia o valor dos bits " x(i) " e " y(i) ", bem como o valor do " vai um " de um módulo anterior.

A descrição do somador completo de 1 bit, denominado " som_1a ", é apresentada no Quadro 7.1.3. Na entidade " som_2b ", descrita no Quadro 7.1.4, a declaração do componente " som_1a " é feita entre as linhas 9 e 11. O " vai um " interno é declarado como um vetor de bits na linha 13, e a solicitação dos componentes é feita entre as linhas 15 e 18. Note que as ligações na solicitação " x2 " não seguem a notação posicional.

```
 1 ENTITY som_1a IS
 2   PORT   (a, b, ve  : IN  BIT;
 3           s, vs     : OUT BIT);
 4 END som_1a;
 5
 6 ARCHITECTURE teste OF som_1a IS
 7 BEGIN
 8   s  <=  a XOR b  XOR ve;                        -- soma
 9   vs <= (a AND b) OR (a AND ve) OR (b AND ve); -- vai um
10 END teste;
```

Quadro 7.1.3 Somador completo de 1 bit.

```
 1 ENTITY som_2b IS
 2   PORT (x, y : IN  BIT_VECTOR(3 DOWNTO 0);  -- entradas do somador
 3         ze   : IN  BIT;                     -- entrada  vem um
 4         s    : OUT BIT_VECTOR(3 DOWNTO 0);  -- soma
 5         zs   : OUT BIT);                    -- vai um
 6 END som_2b;
 7
 8 ARCHITECTURE estrutural OF som_2b IS
 9   COMPONENT som_1a
10     PORT   (a, b, ve : IN  BIT;  s, vs : OUT BIT);
11   END COMPONENT;
12
13   SIGNAL v   : BIT_VECTOR (3 DOWNTO 1);  -- vai um interno
14 BEGIN
15   x0: som_1a PORT MAP(     x(0),      y(0),      ze,        s(0),       v(1));
16   x1: som_1a PORT MAP(     x(1),      y(1),      v(1),      s(1),       v(2));
17   x2: som_1a PORT MAP(b => y(2), a => x(2), s => s(2), ve => v(2), vs => v(3));
18   x3: som_1a PORT MAP(     x(3),      y(3),      v(3),      s(3),       zs);
19 END estrutural;
```

Quadro 7.1.4 Somador de 4 bits formado a partir de componentes.

Solicitação direta de componente

A versão VHDL-1993 permite omitir a declaração de um componente, e solicitar uma entidade de projeto diretamente. Nessa opção, é especificado o nome expandido da entidade e arquitetura (ver Quadro 7.1.5). O nome expandido identifica a biblioteca em que a entidade foi compilada, " WORK " no exemplo, e o da entidade. A arquitetura pode ser omitida, se desejado. O Quadro 7.1.6 contém a descrição do somador de quatro bits empregando a solicitação direta.

```
x1: ENTITY WORK.componente_x(arquitetura_abc) PORT MAP(a(1), b(1), c(1));
x2: ENTITY WORK.componente_x  PORT MAP(a(1), b(1), c(1));                -- arquitetura omitida
```

Quadro 7.1.5 Solicitação direta de componentes - VHDL-1993.

```
 1  -- VHDL-1993
 2  ENTITY som_2c IS
 3    PORT (x, y : IN  BIT_VECTOR(3 DOWNTO 0);  -- entradas do somador
 4          ze   : IN  BIT;                     -- entrada  vem um
 5          s    : OUT BIT_VECTOR(3 DOWNTO 0);  -- soma
 6          zs   : OUT BIT);                    -- vai um
 7  END som_2c;
 8
 9  ARCHITECTURE estrutural OF som_2c IS
10    SIGNAL v   : BIT_VECTOR (3 DOWNTO 1);  -- vai um interno
11  BEGIN
12    x0: ENTITY WORK.som_1a(teste) PORT MAP(x(0), y(0), ze,   s(0), v(1));
13    x1: ENTITY WORK.som_1a(teste) PORT MAP(x(1), y(1), v(1), s(1), v(2));
14    x2: ENTITY WORK.som_1a(teste) PORT MAP(x(2), y(2), v(2), s(2), v(3));
15    x3: ENTITY WORK.som_1a        PORT MAP(x(3), y(3), v(3), s(3), zs); -- arquitetura omitida
16  END estrutural;
```

Quadro 7.1.6 Somador com solicitação direta de componentes - VHDL-1993.

7.2 Genéricos

Definição de genéricos

Genéricos provêem um meio de levar informações externas estáticas para entidades de projeto e blocos. As informações passadas podem ser características de desempenho a parâmetros que definem a dimensão e a lei de formação na interligação de componentes. O emprego de genéricos possibilita, assim, a reconfiguração de um circuito pela simples alteração de seus valores, um recurso muito importante que permite a utilização de um mesmo código para diversas necessidades.

Os genéricos podem ser definidos em entidades e em blocos (ver Quadro 7.2.1). O valor de um genérico pode ser especificado em:

- declarações de entidades
- declarações de componentes
- solicitação de componentes
- configurações (ver Capítulo 14)
- blocos.

Se necessário, o valor inicial de um genérico pode ser sobreposto por um outro genérico. O Quadro 7.2.2 ilustra a definição de valores de genéricos na declaração de um componente.

```
-- genericos na declaracao de entidade: definicao e valor
ENTITY nome_entidade IS                                    --
  GENERIC (generico_1  : tipo_do_generico_1 := valor_inicial;   -- definicao e valor
           generico_2  : tipo_do_generico_2);                   -- apenas definicao
  PORT    (......);
END nome_entidade;
```

```
-- generico em bloco: declaracao e definicao do valor
abc: BLOCK
  GENERIC(generico_3  : tipo_do generico_3 := valor_inicial;   -- definicao e valor
BEGIN
  -- comandos concorrentes ....
END BLOCK;
```

Quadro 7.2.1 Exemplos: definição e especificação do valor de genéricos.

```
-- definicao de valores na declaracao de componente
COMPONENT nome_componente
  GENERIC (generico_4  : tipo_do_generico_4 := valor_inicial;   -- definido valor do generico
           generico_5  : tipo_do_generico_5);                   -- valor do generico omitido
  PORT    (.....);
END COMPONENT;
```

Quadro 7.2.2 Exemplo: definição de valores de genéricos na declaração de um componente.

O Quadro 7.2.3 ilustra a descrição de um registrador cujo número de bits é estabelecido pelo genérico " n ". Note que após a definição do genérico, linha 2, é possível empregá-lo para especificar as dimensões dos vetores " d " e " q ". Não seria um erro omitir o valor do genérico; isto é, a descrição poderia ser compilada sem problemas. Entretanto, no momento da simulação seria gerada uma mensagem de erro devido à falta desse valor. Nesse caso, o valor do genérico " n " deve ser estabelecido de uma outra forma, como, por exemplo, na solicitação da entidade na forma de um componente, conforme será explorado no item a seguir.

```
 1 ENTITY flipn_3 IS
 2   GENERIC(n    : INTEGER := 3);                    -- definicao e valor do generico
 3   PORT    (ck  : IN  BIT;                          -- relogio
 4            rst : IN  BIT;                          -- rst=1 leva q=000 assincrono
 5            d   : IN  BIT_VECTOR(n-1 DOWNTO 0);     -- numero de bits definidos pelo generico 'n'
 6            q   : OUT BIT_VECTOR(n-1 DOWNTO 0));    -- numero de bits definidos pelo generico 'n'
 7 END flipn_3;
 8
 9 ARCHITECTURE teste OF flipn_3 IS
10 BEGIN
11   PROCESS (ck, rst)
12   BEGIN
13     IF       (rst = '1')          THEN q <= (OTHERS => '0'); -- q=00...0 independente de ck
14     ELSIF (ck'EVENT AND ck ='1') THEN q <= d;               -- armazena dado
15     END IF;
16   END PROCESS;
17 END teste;
```

Quadro 7.2.3 Descrição de um registrador com o número de bits definido por um genérico.

Mapa de genéricos na solicitação de componentes

O mapa de genéricos, ou " GENERIC MAP ", contém uma lista que possibilita a associação entre os genéricos definidos na declaração do componente e os valores estabelecidos na entidade que solicita o componente (ver Quadro 7.2.4). De modo semelhante ao mapa de portas, a lista de associações pode adotar a notação posicional ou nomeada, e a palavra reservada " OPEN " pode ser empregada.

```
x1: componente_y GENERIC MAP(valor_a, valor_b, valor_c) PORT MAP(sinal_1, sinal_etc);
x2: componente_x GENERIC MAP(generico_a => valor_a)     PORT MAP(sinal_etc);
x3: componente_z GENERIC MAP(gen_1 => 7, gen_2 => OPEN) PORT MAP(sig_a, sig_b, sig_c);
```

Quadro 7.2.4 Exemplo de mapa de genéricos.

A Figura 7.2.1 apresenta algumas opções para a especificação do valor de um genérico, e como um valor pode se sobrepor a um outro valor definido. Quatro arquiteturas empregam a entidade " gen_a " que contém a definição do genérico " tp " declarado com o valor " 7 ns ". No primeiro caso, na solicitação do componente " u1 ", um novo valor para o genérico é definido na solicitação do componente através de " tp_ex ". Os valores " 7 ns " e " 14 ns ", definidos na declaração da entidade " gen_a " e na declaração do componente, são sobrepostos pelo valor " 21 ns ". De modo semelhante, no quarto caso, é empregado o valor " 3 ns " definido diretamente na solicitação do componente. No segundo caso, a solicitação do componente não contém um mapa de genéricos, e o valor empregado é o valor definido na declaração do componente, " 14 ns ". No terceiro caso, tanto na solicitação quanto na declaração do componente não é feita nenhuma menção ao genérico. Na falta de outro valor, é empregado o valor " 7 ns " para " tp ".

Os dois quadros que seguem permitem verificar as observações feitas na Figura 7.2.1. Algumas alterações foram introduzidas, com o objetivo de reduzir o código apresentado. O código da entidade " gen_a " da Figura 7.2.1 foi copiado em quatro entidades idênticas: " gen_1a ", " gen_1b ", " gen_1c " e " gen_1d ", que diferem unicamente de seus nomes. No Quadro 7.2.5 é apresentada unicamente a entidade " gen_1a ". Os quatro casos foram inseridos em uma única entidade, " gen_3x ", conforme ilustra o Quadro 7.2.6.

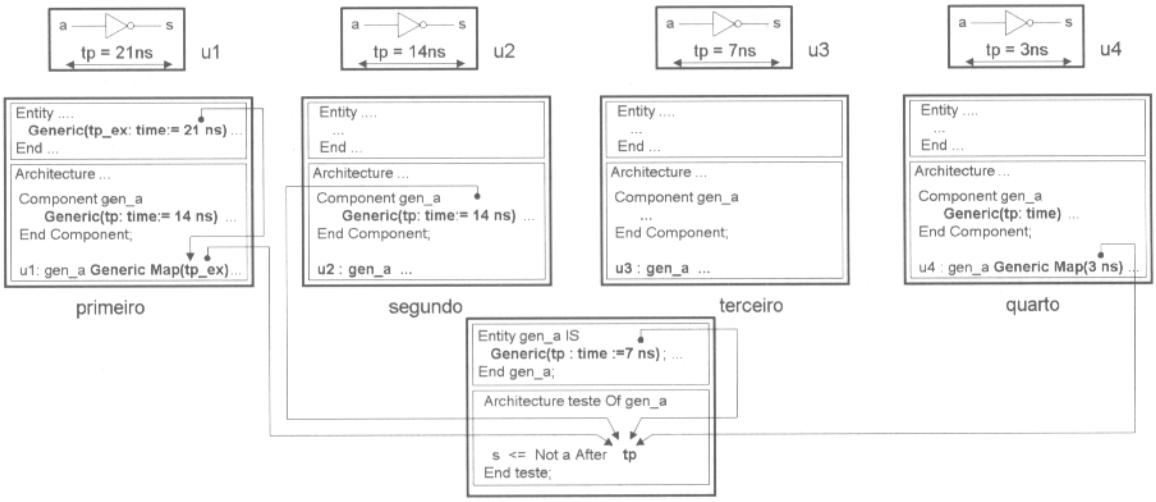


Figura 7.2.1 Exemplo de genéricos: valor empregado.

```
1  ENTITY gen_1a IS
2    GENERIC (tp : TIME := 7 ns);
3    PORT    (a : IN  BIT; s : OUT BIT);
4  END gen_1a;
5
6  ARCHITECTURE teste OF gen_1a IS
7  BEGIN
8    s  <=  NOT a AFTER tp;
9  END teste;
```

Quadro 7.2.5 Código da entidade de projeto " gen_1a ".

```
1  ENTITY gen_3x IS
2    GENERIC (tp_ex : TIME := 21 ns);
3    PORT  (x  : IN  BIT_VECTOR (3 DOWNTO 0);
4           s  : OUT BIT_VECTOR (3 DOWNTO 0));
5  END gen_3x;
6
7  ARCHITECTURE estrutural OF gen_3x IS
8    COMPONENT gen_1a   -- lista de associacoes: generico com valor inicial (equivalente ao caso 1)
9      GENERIC (tp : TIME := 14 ns);
10     PORT    (a : IN  BIT;  s : OUT BIT);
11   END COMPONENT;
12
13    COMPONENT gen_1b  -- lista de associacoes: generico com valor inicial (equivalente ao caso 2)
14     GENERIC (tp : TIME := 14 ns);
15     PORT    (a : IN  BIT;  s : OUT BIT);
16   END COMPONENT;
17
18   COMPONENT gen_1c  -- sem lista de associacao de generico              (equivalente ao caso 3)
19     PORT    (a : IN  BIT;  s : OUT BIT);
20   END COMPONENT;
21
22   COMPONENT gen_1d  -- lista de associacoes: generico sem valor inicial  (equivalente ao caso 4)
23     GENERIC (tp : TIME);
24     PORT    (a : IN  BIT;  s : OUT BIT);
25   END COMPONENT;
26 BEGIN
27   x0: gen_1a GENERIC MAP(tp_ex) PORT MAP(x(0), s(0)); -- valor empregado =21ns
28   x1: gen_1b                    PORT MAP(x(1), s(1)); -- valor empregado =14ns
29   x2: gen_1c                    PORT MAP(x(2), s(2)); -- valor empregado =7ns
30   x3: gen_1d GENERIC MAP(3 ns)  PORT MAP(x(3), s(3)); -- valor empregado =3ns
31 END estrutural;
```

Quadro 7.2.6 Código para verificação da Figura 7.2.1.

7.3 Comando " GENERATE "

O comando " GENERATE " permite repetir comandos concorrentes segundo dois esquemas de geração: " FOR " e " IF ". Esse comando encontra aplicação na geração automática de circuitos que observam uma regularidade na sua estrutura, ou que podem ser definidos a partir de um conjunto de células interligadas segundo uma lei de formação estabelecida. Nessa categoria de circuitos se enquadram, por exemplo, as unidades lógicas aritméticas, os somadores e os multiplicadores.

Esquema de geração " FOR "

O esquema de geração " FOR " contém um identificador local e os limites da geração. O identificador não necessita ser declarado, e os limites da geração são definidos por uma faixa discreta estática. O tipo do identificador é estabelecido pela faixa de valores.

O Quadro 7.3.1 ilustra o formato geral do comando com diferentes especificações dos limites da geração. Os limites podem ser definidos diretamente por uma faixa discreta de valores na forma " 0 TO n-1 " ou " n-1 DOWNTO 0 ". Uma outra forma de definição é através da referência de atributos que retornam uma faixa de valores, como os atributos " RANGE " e " REVERSE_RANGE ".

Comandos " GENERATE " podem ser aninhados permitindo a criação de estruturas com mais de uma dimensão (ver exemplo no Quadro 7.3.2).

```
abc: FOR indentificador IN limites_da_geracao GENERATE
   -- comando concorrente
END GENERATE abc;
```

```
def: FOR i IN sinal'RANGE GENERATE
    -- comando concorrente
END GENERATE def;
```

```
ghi: FOR i IN valor_final DOWNTO valor_inicial GENERATE
    -- comando concorrente
END GENERATE;                    -- ultimo rotulo opcional
```

```
xyz: FOR i IN sinal'REVERSE_RANGE GENERATE
    -- comando concorrente
END GENERATE;                   -- ultimo rotulo opcional
```

Quadro 7.3.1 Formato do comando " GENERATE " - esquema de geração " FOR ".

```
linha: FOR i IN 0 TO 3 GENERATE
  coluna: FOR j IN 0 TO 3 GENERATE
      x: componente_abc PORT MAP (entrada_a(i),entrada_b(j),saida(i+1)) ;
  END GENERATE coluna;
END GENERATE linha;
```

Quadro 7.3.2 Comando " GENERATE " aninhado.

Os Quadros 7.3.3 e 7.3.4 apresentam a descrição de um somador empregando o comando " GENERATE " no esquema de geração " FOR ". No primeiro é solicitado um conjunto de componentes, e no segundo são utilizadas atribuições de sinais. Em ambas as descrições, o tamanho do somador é definido pela cláusula " GENERIC " da linha 2. A Figura 7.3.1 ilustra o circuito resultante da síntese da descrição do Quadro 7.3.3, nível RTL. Observe que cada retângulo representa um componente, entidade " som_1a ", com as respectivas portas de entrada e saída.

```
1  ENTITY som_6aa IS
2    GENERIC(n      : INTEGER  := 3 );                  -- numero de bits
3    PORT   (x, y   : IN  BIT_VECTOR (n-1 DOWNTO 0);    -- entradas do somador
4            ze     : IN  BIT;                          -- vem um
5            s      : OUT BIT_VECTOR (n-1 DOWNTO 0);    -- saida
6            zs     : OUT BIT);                         -- vai um
7  END som_6aa;
8
9  ARCHITECTURE estrutural OF som_6aa IS
10   COMPONENT som_1a
11     PORT   (a, b, ve : IN  BIT;  s, vs  : OUT BIT);
12   END COMPONENT;
13   SIGNAL v : BIT_VECTOR (n DOWNTO 0);  -- vai um interno
14 BEGIN
15   v(0) <= ze;
16   zs   <= v(n);
17   abc: FOR i IN 0 TO n-1 GENERATE
18     centro: som_1a PORT MAP (x(i),  y(i),   v(i),    s(i),   v(i+1));
19   END GENERATE abc;
20 END estrutural;
```

Quadro 7.3.3 Somador empregando o comando " GENERATE " - esquema de geração " FOR ".

```
1  ENTITY som_6c IS
2    GENERIC(n      : INTEGER  := 3 );                  -- numero de bits
3    PORT   (x, y   : IN  BIT_VECTOR (n-1 DOWNTO 0);    -- entradas do somador
4            ze     : IN  BIT;                          -- vem um
5            s      : OUT BIT_VECTOR (n-1 DOWNTO 0);    -- saida
6            zs     : OUT BIT);                         -- vai um
7  END som_6c;
8
9  ARCHITECTURE teste OF som_6c IS
10   SIGNAL v : BIT_VECTOR (n DOWNTO 0);  -- vai um interno
11 BEGIN
12   v(0) <= ze;
13   zs   <= v(n);
14   abc: FOR i IN 0 TO n-1 GENERATE
15     s(i)   <=  x(i) XOR y(i)  XOR v(i);
16     v(i+1) <= (x(i) AND y(i)) OR (x(i) AND v(i)) OR (y(i) AND v(i));
17   END GENERATE abc;
18 END teste;
```

Quadro 7.3.4 Somador: comando " GENERATE " - esquema de geração " FOR ".

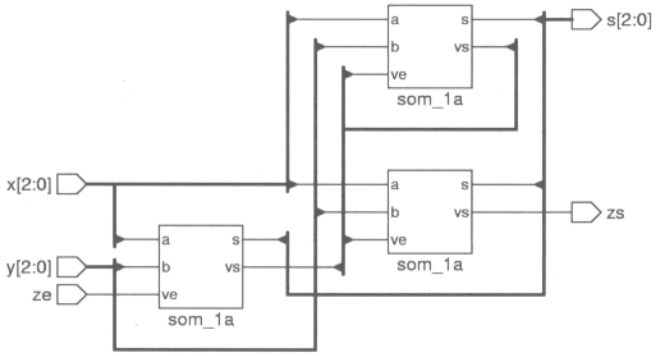


Figura 7.3.1 Circuito somador sintetizado a partir da descrição do Quadro 7.3.3 - nível RTL.

Esquema de geração " IF "

O comando " GENERATE ", no esquema de geração " IF ", introduz uma réplica dos comandos contidos, caso a condição contida após a palavra reservada " IF " seja satisfeita. A condição é verificada por uma expressão estática. O Quadro 7.3.5 ilustra o formato da construção.

```
abc: IF condicao  GENERATE
  -- comando concorrente
  -- comando concorrente
END GENERATE abc;
```

Quadro 7.3.5 Formato do comando " GENERATE " - modo condicional.

No Quadro 7.3.6 é apresentada uma nova descrição para um somador empregando a construção " GENERATE " com o esquema de geração " IF ". O código é semelhante à descrição do Quadro 7.3.3, e tem o objetivo de ilustrar a nova construção. Não é, com certeza, a melhor opção de código para o exemplo. Três esquemas de geração são previstos. O modo entre as linhas 16 e 18 posiciona a célula associada ao bit menos significativo do somador, e o esquema entre as linhas 22 e 24 posiciona a célula associada ao bit mais significativo. Entre as linhas 19 e 21 são definidas as células restantes do somador.

```
 1  ENTITY som_7a IS
 2    GENERIC(n      : INTEGER  := 3 );                  -- numero de bits
 3    PORT   (x, y   : IN  BIT_VECTOR (n-1 DOWNTO 0);    -- entradas so somador
 4            ze     : IN  BIT;                          -- vem um
 5            s      : OUT BIT_VECTOR (n-1 DOWNTO 0);    -- saida
 6            zs     : OUT BIT);                         -- vai um
 7  END som_7a;
 8
 9  ARCHITECTURE estrutural OF som_7a IS
10    COMPONENT som_1a
11      PORT   (a, b, ve : IN  BIT;  s, vs  : OUT BIT);
12    END COMPONENT;
13    SIGNAL v : BIT_VECTOR (n-1 DOWNTO 1);  -- vai um interno
14  BEGIN
15    global: FOR i IN 0 TO n-1 GENERATE
16      def: IF i=0 GENERATE
17        primeira: som_1a PORT MAP (x(i), y(i), ze,  s(i), v(i+1));  -- 1a celula
18      END GENERATE def;
19      abc: IF (i > 0) AND (i < n-1) GENERATE
20        centro: som_1a PORT MAP  (x(i), y(i), v(i), s(i), v(i+1));  -- celulas centro
21      END GENERATE abc;
22      ghi: IF i=n-1 GENERATE
23        ultima: som_1a PORT MAP  (x(i), y(i), v(i), s(i), zs);      -- ultima celula
24      END GENERATE ghi;
25    END GENERATE global;
26  END estrutural;
```

Quadro 7.3.6 Somador: comando " GENERATE " - modo condicional e comandos concorrentes.

7.4 Comando " LOOP "

O comando " LOOP " permite repetir a execução de um conjunto de comandos seqüenciais. Dois esquemas de iteração são disponíveis: o modo " FOR " e o modo " WHILE ". Assim como no comando " GENERATE ", esse comando pode ser empregado na formação de circuitos regulares através da reprodução programada de um conjunto de comandos. As iterações realizadas pelo comando " LOOP " podem ser também empregadas em rotinas de conversão de tipos e entidades que geram vetores de teste. O comando " LOOP ", entretanto, nem sempre é suportado pelas ferramentas de síntese.

Esquema de iteração " FOR "

O esquema de iteração " FOR " do comando " LOOP " é semelhante ao esquema de geração de mesmo nome do comando " GENERATE ". Ele contém um identificador local e os limites da iteração. O identificador não necessita

ser declarado, e os limites da iteração são definidos por uma faixa discreta de valores. O tipo do identificador é estabelecido pela faixa de valores, e nenhuma operação dentro do laço pode atribuir um valor ao identificador.

O Quadro 7.4.1 ilustra o formato geral do comando com diferentes especificações dos limites da iteração. Os limites podem ser definidos diretamente por uma faixa discreta de valores na forma " 0 TO n-1 " ou " n-1 DOWNTO 0 ", bem como através da referência de atributos que retornam uma faixa de valores, como os atributos " RANGE " e " REVERSE_RANGE ". Uma outra opção é delimitar a faixa através de um tipo ou subtipo discreto. Neste caso, o primeiro elemento da iteração corresponde ao valor de posição mais à esquerda do tipo, e o último elemento ao valor de posição mais à direita.

```
abc: FOR identificador IN limites_da_iteracao LOOP
   -- comando sequencial
END LOOP abc;
```

```
FOR i IN valor_final DOWNTO valor_inicial LOOP          -- faixa decrescente
   -- comando sequencial
END LOOP;                                               -- rotulo opcional removido
```

```
ghi: FOR i IN x'RANGE LOOP        -- faixa: cobre os elementos na faixa definida, exemplo
   -- comando sequencial          --        (final DOWNTO inicial), (inicial TO final)
END LOOP ghi;
```

```
jkl: FOR i IN tipo_discreto LOOP        -- tipo: cobre o valor mais a esquerda
   -- comando sequencial                --       ao valor mais a direita do tipo
END LOOP jkl;
```

Quadro 7.4.1 Formato do comando " LOOP " - esquema de geração " FOR ".

No Quadro 7.4.2 é apresentado um código para o circuito somador empregando o comando " LOOP " com esquema de iteração " FOR ". Ele é bem semelhante ao código do somador empregando o comando " GENERATE " descrito no Quadro 7.3.4. O " vai um " interno é agora uma variável, e a transferência do valor do último " vai um " para o sinal externo " zs " deve ser realizada após o término do laço.

```
 1 ENTITY som_8a IS
 2   GENERIC(n      : INTEGER  := 3 );                -- numero de bits
 3   PORT   (x, y   : IN  BIT_VECTOR (n-1 DOWNTO 0);  -- entradas do somador
 4           ze     : IN  BIT;                        -- vem um
 5           s      : OUT BIT_VECTOR (n-1 DOWNTO 0);  -- saida
 6           zs     : OUT BIT);                       -- vai um
 7 END som_8a;
 8
 9 ARCHITECTURE teste OF som_8a IS
10 BEGIN
11   abc: PROCESS (x, y, ze)
12     VARIABLE v : BIT_VECTOR (n DOWNTO 0);  -- vai um interno
13   BEGIN
14     v(0) := ze;
15     abc: FOR i IN 0 TO n-1 LOOP
16       s(i)   <=  x(i) XOR y(i)  XOR v(i);
17       v(i+1) := (x(i) AND y(i)) OR (x(i) AND v(i)) OR (y(i) AND v(i));
18     END LOOP abc;
19     zs   <= v(n);
20   END PROCESS;
21 END teste;
```

Quadro 7.4.2 Somador: comando " LOOP " - esquema de iteração " FOR ".

No exemplo do Quadro 7.4.3, dois laços executam uma operação que reverte a posição dos elementos de um vetor. Nessa descrição o elemento de posição mais à esquerda de " x " é transferido para a posição mais à direita de " s " e " t ". No laço " abc " a faixa de iteração é definida do valor inicial ao valor final. No laço " def " os limites da iteração são identificados pela referência ao atributo " RANGE " do sinal " x ".

```
1  ENTITY rev_bit0 IS
2    GENERIC(n      : INTEGER  := 4 );
3    PORT   (x      : IN  BIT_VECTOR (n-1 DOWNTO 0);
4            s, t   : OUT BIT_VECTOR (n-1 DOWNTO 0));
5  END rev_bit0;
6
7  ARCHITECTURE teste OF rev_bit0 IS
8  BEGIN
9    abc: PROCESS (x)
10   BEGIN
11     abc: FOR i IN n-1 DOWNTO 0 LOOP
12       t(t'LEFT -i) <= x(i);
13     END LOOP abc;
14     def: FOR i IN x'RANGE LOOP       -- faixa: x'RANGE =(n-1 DOWNTO 0)
15       s(s'LEFT -i) <= x(i);
16     END LOOP def;
17   END PROCESS;
18 END teste;
```

Quadro 7.4.3 Comando " LOOP " esquema de iteração " FOR " - exemplo 1.

No Quadro 7.4.4 é apresentado um outro exemplo, onde os limites de iteração são definidos por um tipo. Nesse caso, para a versão VHDL-1987, o primeiro elemento da iteração corresponde ao valor mais à esquerda definido no tipo " CHARACTER ", o valor " nul ", e o último elemento da iteração corresponde ao valor mais à direita, valor " del ". A descrição, sem aplicação para síntese, retorna a posição no tipo " CHARACTER " de um valor fornecido pelo sinal " ch ". A cada alteração de valor em " ch ", o processo sai do estado suspenso, e o comando " LOOP " executa uma busca, percorrendo por todos os valores definidos no tipo " CHARACTER ". Quando o valor de " ch " for igual ao valor de " ch_loop ", o dado da variável " posicao " é transferido para o sinal " posicao_loop ". Para efeito de comparação, o mesmo valor é obtido com a referência ao atributo " POS " do tipo " CHARACTER " (ver linha 23).

```
1  ENTITY para_gr1 IS
2    PORT (ch                          : IN CHARACTER;
3          posicao_loop, posicao_atrib : OUT INTEGER);
4  END para_gr1;
5
6  ARCHITECTURE teste OF para_gr1 IS
7  BEGIN
8    -- determinacao da posicao do caractere pelo comando  LOOP
9    xyz: PROCESS (ch)
10     VARIABLE posicao : INTEGER;
11   BEGIN
12    posicao := 0;
13    abc: FOR ch_loop IN CHARACTER LOOP -- limites: primeiro ao ultimo elemento
14      IF ch = ch_loop THEN
15        posicao_loop <= posicao;
16      ELSE
17        posicao := posicao +1;
18      END IF;
19    END LOOP abc;
20   END PROCESS;
21
22   -- determinacao da posicao do caractere pelo atributo POS no tipo CHARACTER
23   posicao_atrib <= CHARACTER'POS(ch);
24 END teste;
```

Quadro 7.4.4 Comando " LOOP " esquema de iteração " FOR " - exemplo 2.

O Quadro 7.4.5 ilustra a conversão de um valor contido em um objeto do tipo vetor de bits para um valor do tipo inteiro. A operação consiste em testar o valor do bit de posição " i ". Se o bit for igual a um, o valor " 2^i " é somado a uma variável temporária " temp ". Ao término das iterações, os valores temporários são transferidos para o sinal " inteiro_for ". Observe que o controle das iterações é determinado pelo número de elementos do vetor; assim, vetores de qualquer extensão podem ser convertidos.

A operação de exponenciação com o valor " 2 " costuma ser aceita pelas ferramentas de síntese. Apesar do relativo distanciamento das operações realizadas com um circuito real, algumas ferramentas de síntese conse-

guem inferir o circuito correspondente à descrição. A Figura 7.4.1 mostra o circuito sintetizado corretamente: a porta " e_bit " está interligada diretamente à porta " inteiro_for ".

```
1  ENTITY loop_f1 IS
2    PORT (e_bit          : IN  BIT_VECTOR (3 DOWNTO 0); -- entrada tipo vetor de bits
3          inteiro_for    : OUT INTEGER RANGE 0 TO 15);  -- saida convertida para inteiro
4  END loop_f1;
5
6  ARCHITECTURE teste OF loop_f1 IS
7  BEGIN
8    abc_for: PROCESS (e_bit)
9      VARIABLE temp : INTEGER;
10   BEGIN
11     temp := 0;
12     FOR i IN e_bit'RANGE LOOP  -- numero de iteracoes: tamanho do vetor entrada
13       IF e_bit(i) = '1' THEN temp := temp + 2**i;
14       END IF;
15     END LOOP;
16     inteiro_for <= temp;
17   END PROCESS;
18 END teste;
```

Quadro 7.4.5 Conversão de um tipo " bit_vector " para o tipo " integer " - esquema " FOR ".

Figura 7.4.1 Circuito inferido a partir da descrição da rotina de conversão.

Esquema de iteração " WHILE "

No esquema de iteração " WHILE " do comando " LOOP ", um conjunto de comandos seqüenciais são executados se uma condição for verdadeira. O teste é realizado antes da execução dos comandos contidos no laço, e a iteração termina se a condição de teste for falsa. Ele segue o formato apresentado no Quadro 7.4.6.

```
abc: WHILE condicao LOOP                -- executado enquanto condicao = verdadeira
  -- comando sequencial
  -- comando sequencial
  -- atualizacao variavel de indice
END LOOP abc;
```

Quadro 7.4.6 Formato do comando " LOOP " - esquema " WHILE ".

O Quadro 7.4.7 ilustra uma nova descrição para um circuito somador empregando o comando " LOOP " no esquema de iteração " WHILE ". A cada execução do processo, é realizado um conjunto de iterações no laço sob o controle da variável " i ", definida na linha 12. Ao final do laço, ela é incrementada, e as operações são repetidas enquanto o valor da variável " i " for menor ou igual a " n-1 ".

O Quadro 7.4.8 apresenta a rotina de conversão de um valor do tipo vetor de bits para o tipo inteiro, empregando o esquema de iteração " WHILE ". Nesse caso, o esquema é desvantajoso em comparação com o modo " FOR ", pois é necessário declarar e inicializar a variável controle, bem como incrementá-la a cada iteração. Assim como no exemplo do Quadro 7.4.5, o controle da iteração é determinado pelo número de elementos do vetor, através da referência ao atributo " LENGTH " do sinal " e_bit ".

```
1  ENTITY som_9a IS
2    GENERIC(n      : INTEGER  := 3 );                  -- numero de bits
3    PORT   (x, y   : IN  BIT_VECTOR (n-1 DOWNTO 0);    -- entradas so somador
4            ze     : IN  BIT;                          -- vem um
5            s      : OUT BIT_VECTOR (n-1 DOWNTO 0);    -- saida
6            zs     : OUT BIT);                         -- vai um
7  END som_9a;
8
9  ARCHITECTURE teste OF som_9a IS
10 BEGIN
11   abc: PROCESS (x, y, ze)
12     VARIABLE i : INTEGER := 0;
13     VARIABLE v : BIT_VECTOR (n DOWNTO 0);  -- vai um interno
14   BEGIN
15     i := 0;        -- deve ser atualizado a cada iteracao
16     v(0) := ze;
17     abc: WHILE i <= n-1 LOOP    -- executado enquanto verdadeiro
18       s(i)   <=  x(i) XOR y(i)  XOR v(i);
19       v(i+1) := (x(i) AND y(i)) OR (x(i) AND v(i)) OR (y(i) AND v(i));
20       i := i+1;
21     END LOOP abc;
22     zs   <= v(n);
23   END PROCESS;
24 END teste;
```

Quadro 7.4.7 Somador: comando " LOOP " - esquema de iteração " WHILE ".

```
1  ENTITY loop_w1 IS
2    GENERIC (n           : INTEGER := 4);
3    PORT (e_bit          : IN  BIT_VECTOR (n-1 DOWNTO 0); -- entrada tipo vetor de bits
4          inteiro_while  : OUT INTEGER RANGE 0 TO 2**n-1);-- saida convertida para inteiro
5  END loop_w1;
6
7  ARCHITECTURE teste OF loop_w1 IS
8
9  BEGIN
10   abc_while: PROCESS (e_bit)
11     VARIABLE temp, i : INTEGER;
12   BEGIN
13     temp := 0;
14     i := 0;
15     WHILE (i /= e_bit'LENGTH) LOOP  -- numero de iteracoes: tamanho do vetor entrada
16       IF e_bit(i) = '1' THEN temp := temp + 2**i;
17       END IF;
18       i := i +1;
19     END LOOP;
20     inteiro_while <= temp;
21   END PROCESS;
22 END teste;
```

Quadro 7.4.8 Conversão de " bit_vector " para o tipo " integer " - esquema de iteração " WHILE ".

A Figura 7.4.2 ilustra o resultado da síntese das descrições dos Quadros 7.4.7 e 7.4.8. Conforme se pode observar, tanto o somador, quanto a conversão de tipos foram sintetizados corretamente pela ferramenta.

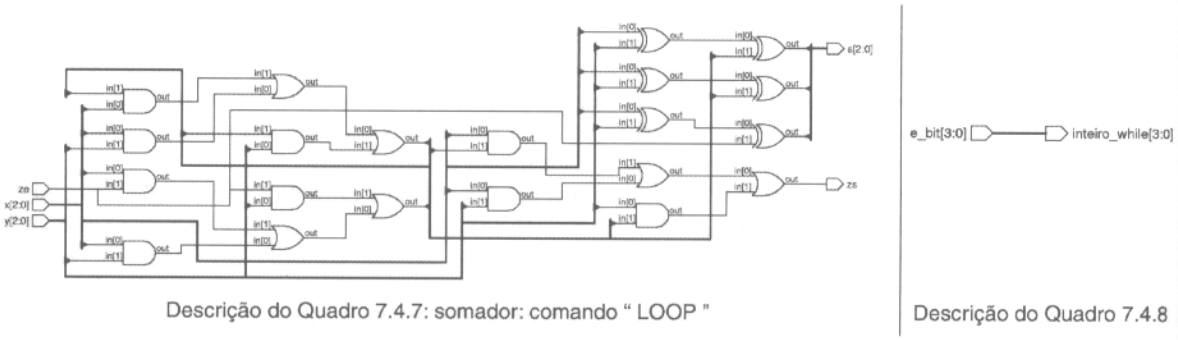


Figura 7.4.2 Resultado da síntese - descrições dos Quadros 7.4.7 e 7.4.8.

Comandos " NEXT " e " EXIT "

A seqüência das operações executadas em um comando " LOOP " pode ser alterada através dos comandos " NEXT " e " EXIT ". A execução do comando " NEXT " causa um salto para o final do laço, omitindo a execução dos comandos restantes no esquema de iteração. A execução do comando " EXIT " leva ao término das operações do laço em execução. O Quadro 7.4.9 apresenta o formato dos comandos.

Ambos os comandos podem ser subordinados a uma condição estabelecida após a palavra reservada " WHEN ". Após os comandos " NEXT " e " EXIT ", pode ser inserido um rótulo identificando um outro laço para a retomada das operações (ver Figura 7.4.3).

```
NEXT;                        -- pula para a proxima iteracao
NEXT WHEN condicao_1;        -- pula para a proxima iteracao caso condicao_1 for verdadeira
NEXT loop_1 WHEN condicao_3; -- pula para loop_1
abc: NEXT WHEN condicao_2;   -- idem, rotulo abc opcional
```

```
EXIT;                        -- termina a iteracao
EXIT WHEN condicao_1;        -- termina a iteracao caso condicao_1 for verdadeira
EXIT loop_1 WHEN condicao_3; -- termina a iteracao e salta para loop_1
abc: EXIT WHEN condicao_2;   -- idem, rotulo abc opcional
```

Quadro 7.4.9 Exemplos de comandos dos " NEXT " e " EXIT ".

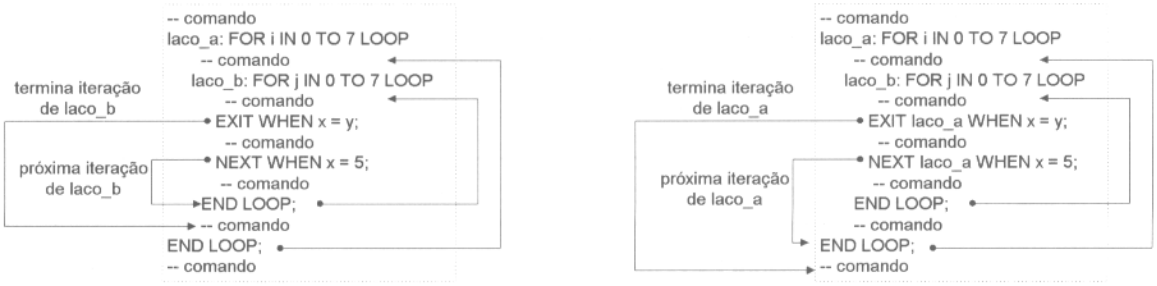


Figura 7.4.3 Ilustração da operação dos comandos " NEXT " e " EXIT ".

Laços infinitos

Laços executados indefinidamente podem ser definidos de duas maneiras. A primeira emprega uma condição de teste sempre verdadeira, como " WHILE TRUE LOOP ", e a segunda emprega o comando " LOOP " sem uma condição de teste (ver Quadro 7.4.10). Nesses casos, deve ser previsto um mecanismo para abortar a execução do laço. O comando " EXIT " é uma opção para esse término. Laços executados indefinidamente não são suportados pelas ferramentas de síntese.

```
def: WHILE TRUE LOOP  -- executado sempre
  -- comando sequencial
  -- comando sequencial
END LOOP def;
```

```
ghi: LOOP             -- executado sempre
  -- comando sequencial
  -- comando sequencial
END LOOP ghi;
```

Quadro 7.4.10 Laços infinitos.

Na descrição do Quadro 7.4.11, a rotina de conversão de um valor contido num objeto do tipo vetor de bits para um tipo inteiro foi alterada para ilustrar o emprego do comando " EXIT ". Nesse caso, foi empregado um laço infinito, onde o fim da execução é definido pelo comando " EXIT " na linha 19.

```
 1 ENTITY loop_e1 IS
 2   GENERIC (n          : INTEGER := 4);
 3   PORT (e_bit         : IN  BIT_VECTOR (n-1 DOWNTO 0); -- entrada tipo vetor de bits
 4         inteiro_exit  : OUT INTEGER RANGE 0 TO 2**n-1);-- saida convertida para inteiro
 5 END loop_e1;
 6
 7 ARCHITECTURE teste OF loop_e1 IS
 8
 9 BEGIN
10   abc_for: PROCESS (e_bit)
11     VARIABLE temp, i : INTEGER;
12   BEGIN
13     temp := 0;
14     i := 0;
15     LOOP
16       IF e_bit(i) = '1' THEN temp := temp + 2**i;
17       END IF;
18       i := i +1;
19       EXIT WHEN i = e_bit'LENGTH;
20     END LOOP;
21     inteiro_exit <= temp;
22   END PROCESS;
23 END teste;
```

Quadro 7.4.11 Conversão " bit_vector " para o tipo " integer " - laço infinito.

A descrição do Quadro 7.4.11 submetida à ferramenta de síntese gerou uma mensagem de erro (ver Quadro 7.4.12). A ferramenta de síntese não suporta laços sem uma condição de teste, conforme apresentado na descrição.

```
"/vhdl/loop_e1.vhd",line 15: Error, infinite loops are not supported.
```

Quadro 7.4.12 Mensagem de erro: laço da descrição do Quadro 7.4.11 não é suportado.

7.5 Principais pontos abordados

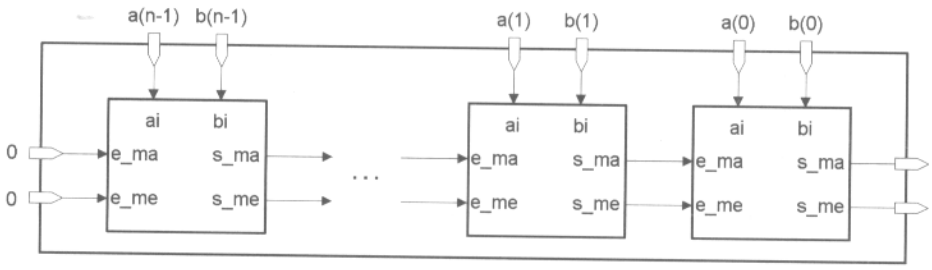
- Componente é uma entidade de projeto (declaração e arquitetura) empregada por uma arquitetura. Os componentes são empregados em projetos hierárquicos.
- A declaração de um componente é necessária para ele ser solicitado no interior de uma descrição. O formato da declaração é semelhante à declaração de uma entidade, onde são definidas as portas de entrada e saída.
- Na versão VHDL-1993 é permitida a solicitação de um componente sem uma correspondente declaração.
- Genéricos são empregados na transferência de informações estáticas para entidades de projeto e blocos.
- O comando " GENERATE " permite repetir comandos concorrentes. Dois esquemas de geração são possíveis: esquema " FOR " e esquema " IF ".
- O comando " LOOP " permite repetir a execução de comandos seqüenciais. Os esquemas de iteração disponíveis são: " FOR " e " WHILE ". A execução das iterações pode ser alterada pelos comandos " NEXT " e " EXIT ".

7.6 Exercícios

7.6.1 O objetivo do exercício é propor o código de um circuito comparador de dois números " a " e " b ". O circuito deve detectar se o valor de " a " é maior, menor ou igual ao valor de " b ". O número de bits de " a " e " b " é definido pelo genérico " n ".

O circuito comparador deve ser composto por células conforme ilustra a Figura 7.6.1. Cada célula possui quatro entradas " ai ", " bi ", " e_ma " e " e_me " e duas saídas " s_ma " e " s_me ". A saída " s_ma " em nível alto significa " a " maior

que " b "; a saída " s_me " em nível alto significa " a " menor que " b "; e as duas saídas em nível lógico baixo significa " a " igual a " b ". As entradas " e_ma " e " e_me " são interligadas à célula que avalia o próximo par de bits mais significativo. A comparação procede do par de bits mais significativo para o par de bits menos significativo. Apresente as descrições da célula e do comparador. O código do comparador deve solicitar a célula na forma de componente.



	s_ma	s_me
a > b	1	0
a < b	0	1
a = b	0	0

Figura 7.6.1 Diagrama de blocos do Exercício 7.6.1.

7.6.2 Sintetize a descrição proposta para o Exercício 7.6.1 no caso de um comparador com três bits. Observe o resultado da síntese, e compare os recursos empregados no componente que trata do bit mais significativo com os componentes restantes. Verifique se a ferramenta de síntese reduziu o número de componentes de alguma célula.

7.6.3 Altere a descrição proposta no Quadro 7.1.4 de modo a implementar um somador de quatro bits sem a entrada "vem um", e a saída "vai um", conforme ilustrado na Figura 7.6.2. Empregue a palavra reservada " OPEN " para os sinais desconectados dos componentes.

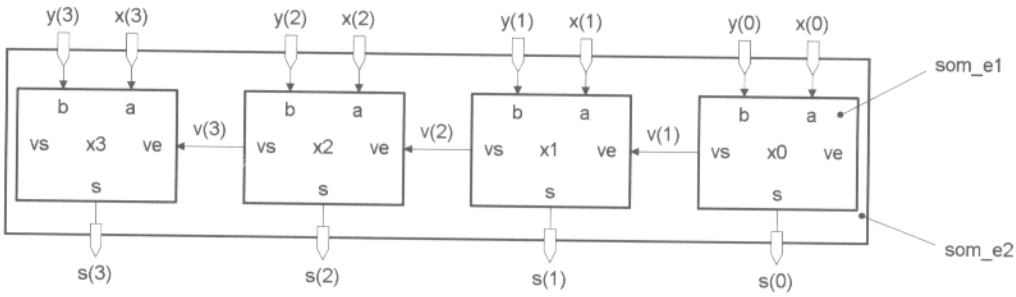


Figura 7.6.2 Somador de 4 bits sem entrada "vem um" e saída "vai um".

7.6.4 Sintetize a descrição proposta para o Exercício 7.6.3, e verifique se o resultado da síntese corresponde ao desejado. Verifique, também, como a ferramenta de síntese trata os blocos com sinais sem conexão. Ela elimina as portas lógicas desnecessárias nos componentes com sinais desconectados?

7.6.5 Apresente a descrição de um circuito contendo 3 contadores na forma de componentes. Os circuitos realizam contagens entre os valores 0 e 9, e a operação de *reset* é realizada de modo síncrono. As Figuras 7.6.3 e 7.6.4 ilustram a operação do conjunto. O valor na saída do contador é incrementado, se o sinal de habilitação " enb " estiver em nível alto, e o sinal " rst_s " estiver em nível baixo. A saída " cnt_9 " indica ao próximo contador que o final da contagem foi atingida, e este deve incrementar o seu valor no próximo ciclo de relógio. Observe, na Figura 7.6.4, que o sinal " cnt_9 " não pode apenas detectar o valor " 9 " do próprio contador. Considere o sinal de *reset*, " rst_s ", ativo em nível alto, todos os sinais do tipo bit, e as saídas " q " dos contadores como tipo inteiro.

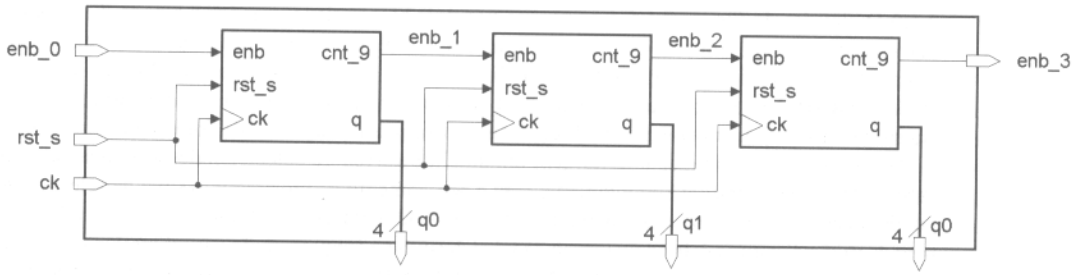


Figura 7.6.3 Contadores decimais síncronos - Exercício 7.6.5.

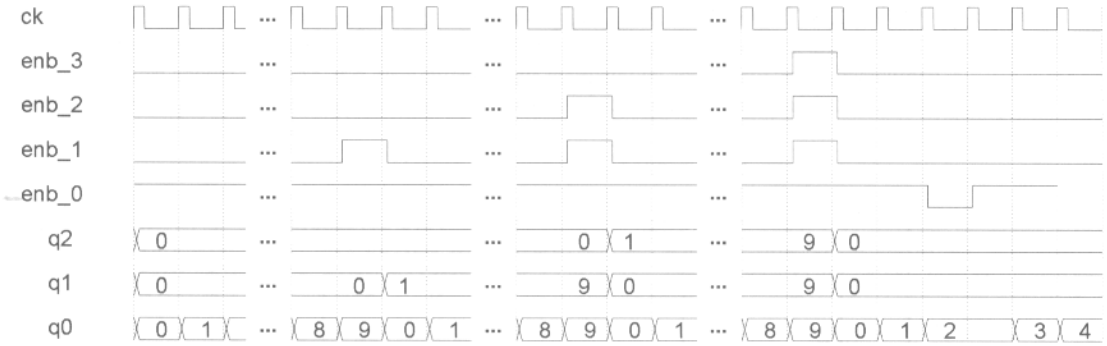


Figura 7.6.4 Carta de tempos da operação dos contadores síncronos - Exercício 7.6.5.

7.6.6 A operação de multiplicação entre dois números inteiros positivos pode ser implementada por um conjunto de somadores completos. A Figura 7.6.5 ilustra o caso de um multiplicador com 3 bits, onde " a " e " b " são os sinais de entrada, e " p " é o sinal de saída. Apresente a descrição do multiplicador proposto.

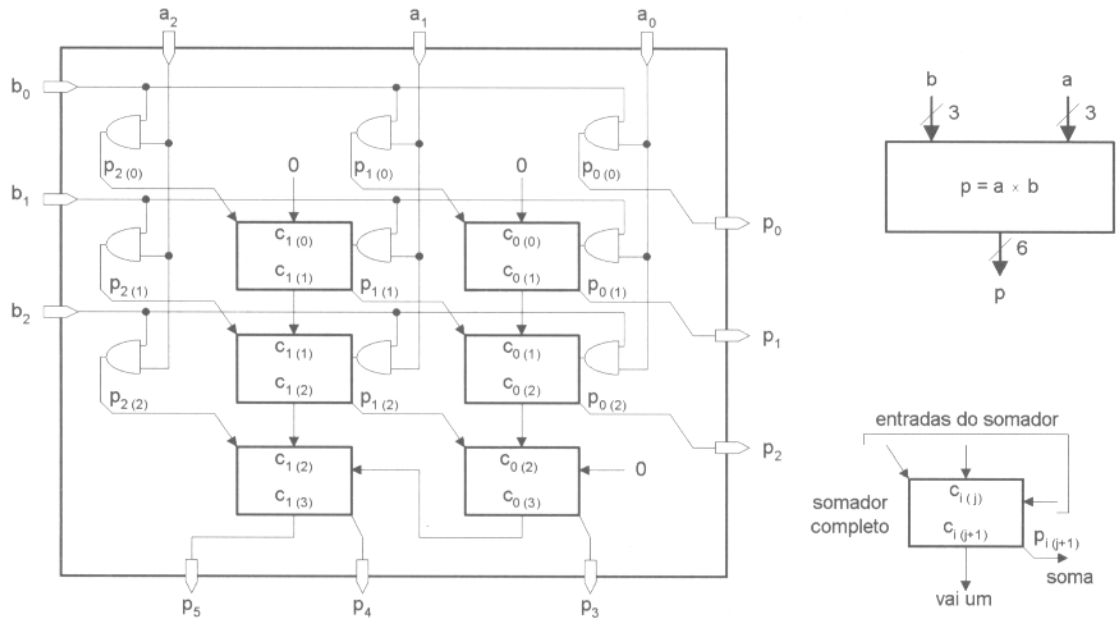


Figura 7.6.5 Multiplicador celular para operandos de três bits.

7.6.7 Altere a descrição " rev_bit0 " (Quadro 7.4.3), para empregar o comando " LOOP " com o esquema de iteração " WHILE ".

7.6.8 Apresente uma rotina de conversão de um valor do tipo inteiro para o tipo vetor de bits empregando o comando " NEXT ".

7.6.9 O circuito da Figura 7.6.6 é composto por dois blocos denominados " bloco1 " e " bloco2 ". O bloco 1 permite armazenar 4 dados em um conjunto de registradores, e o bloco 2 apresenta o valor dos registradores em 4 mostradores de 7 segmentos. A entrada dos dados é feita pela linha " de ", e a seleção do registrador que armazena os dados é feita pela entrada " end_de ". Um dado é armazenado na borda de subida do sinal " wr_l ", conforme o endereço de seleção " end_de ". Na saída " qs " do bloco 1 é possível ler o valor armazenado nos registradores conforme o sinal de seleção " end_qs ".

No bloco 2, um contador interno gera um sinal de varredura selecionando periodicamente um dos quatro mostradores através das linhas " mst ". Conjuntamente com o acionamento do mostrador, o sinal de seleção endereça o registrador correspondente ao mostrador através das linhas " end_qs ". O valor do registrador é decodificado para 7 segmentos e transferido para as saídas " sgm ". A base de tempo " ck " determina a freqüência de varredura dos mostradores.

Considere todos os sinais do tipo " bit " ou " bit_vector " e o sinal " end_qs " do tipo inteiro. Apresente a descrição do circuito completo contendo os dois blocos na forma de componentes.

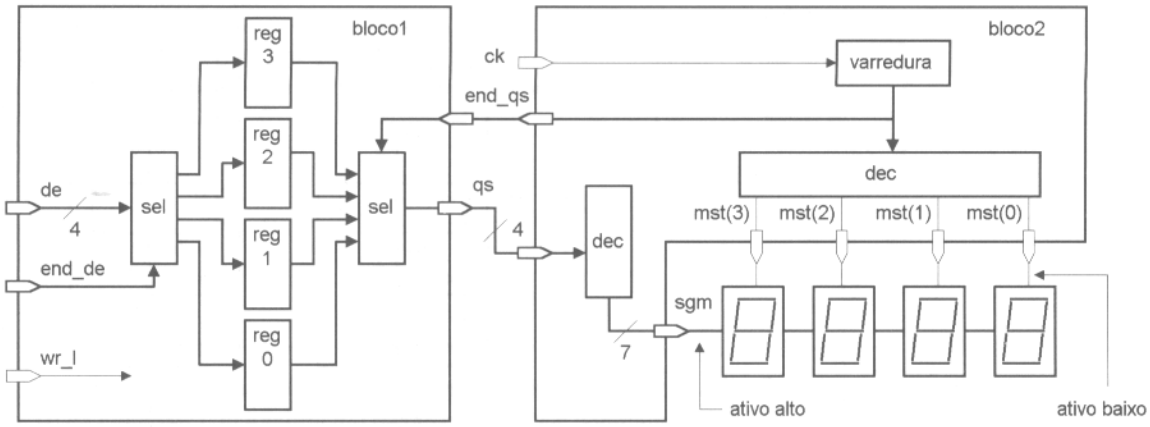


Figura 7.6.6 Diagrama de blocos do Exercício 7.6.9.

7.6.10 O objetivo deste exercício é gerar um controlador para um conversor analógico digital do tipo dupla rampa. A Figura 7.6.7 ilustra a operação do sistema. O valor da tensão a ser medido, " Vm ", é aplicado a um circuito integrador por um período de tempo fixo " tx ". Após esse período de tempo, uma tensão de referência, " Vref ", de polaridade oposta é aplicada à entrada do integrador, e um ciclo de contagem é iniciado. Quando a tensão na saída do integrador, " Vint ", atingir o valor zero, o processo de contagem é interrompido. O valor resultante dessa contagem é proporcional à tensão medida.

O diagrama do controlador a ser projetado é apresentado na Figura 7.6.8. Ele é composto por uma máquina de estados que gera os sinais de controle, contadores decimais, registradores e conversores de código " bdc " para 7 segmentos. Os contadores executam a contagem dos períodos " tx " e " tm ", e são equivalentes aos propostos no Exercício 7.6.5. Os registradores armazenam o valor da última contagem, permitindo, assim, a visualização da última medida nos mostradores de 7 segmentos. A operação deve iniciar com o acionamento da chave " ch_vm ", e a máquina de estados deve aguardar o final da contagem, sinal " enb_3=1 ". Após esse período de tempo, " tx ", a chave " ch_ref " deve ser acionada, e a máquina de estados deve aguardar a tensão na saída do integrador atingir o valor zero, sinal " Vint_z=1 ". Uma vez " Vint_z=1 ", o dado dos contadores deve ser transferido para os registradores no próximo ciclo de relógio, atualizando os mostradores com um novo valor de tensão. A operação termina com o acionamento da chave " ch_zr " para levar a saída do integrador a zero, e a inicialização dos contadores com o valor zero. Observe que, ao longo de toda a conversão, apenas uma das chaves é acionada.

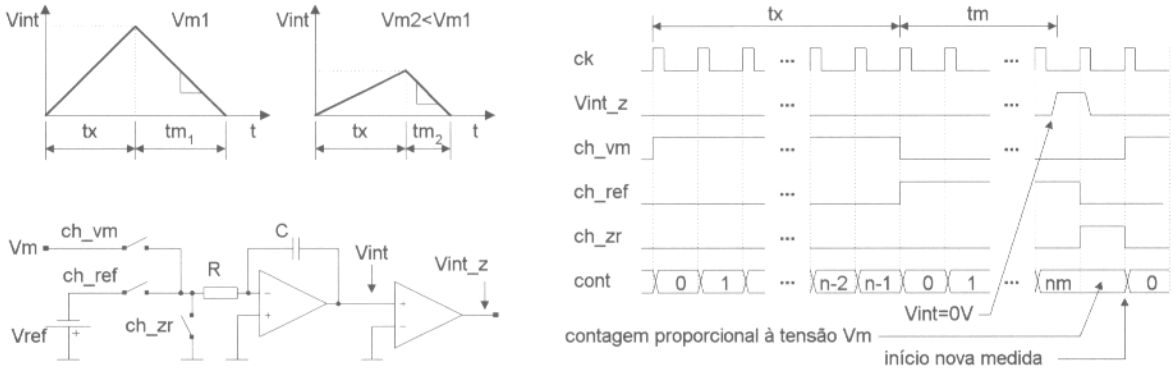


Figura 7.6.7 Conversor analógico digital dupla rampa - Exercício 7.6.10.

Considere a implementação com três níveis de hierarquia. O nível mais baixo contém a descrição do contador síncrono, registrador e decodificador " BCD " para 7 segmentos. O segundo nível, composto por um contador, um registrador e um decodificador, corresponde ao " bloco1 " da Figura 7.6.8. O nível mais elevado contém a máquina de estados finitos e três unidades " bloco1 ". No esquema proposto foi inserido o sinal " inicio " para inicializar as operações da máquina de estados. Como sugestão, operação de carga dos registradores é comandada pelo sinal " vint_z ". Isso é possível, considerando-se que, no ciclo de relógio após a condição " vint_z=1 " ser detectada, a máquina deve acionar a chave " ch_zr ", levando a tensão na saída do integrador a zero, e, portanto, o sinal " vint_z " deve, também, ir a zero. Os elementos contidos nos retângulos pontilhados são externos.

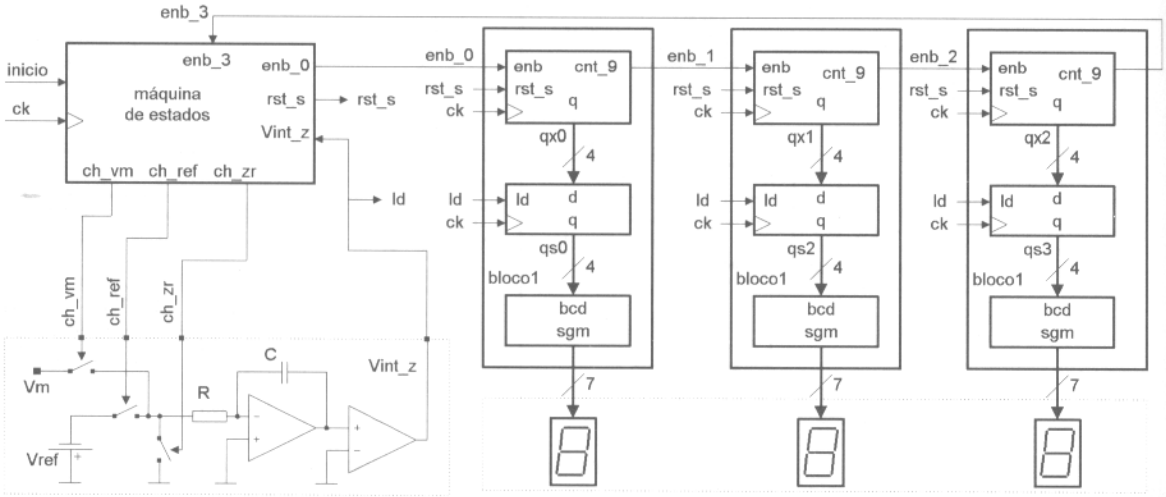
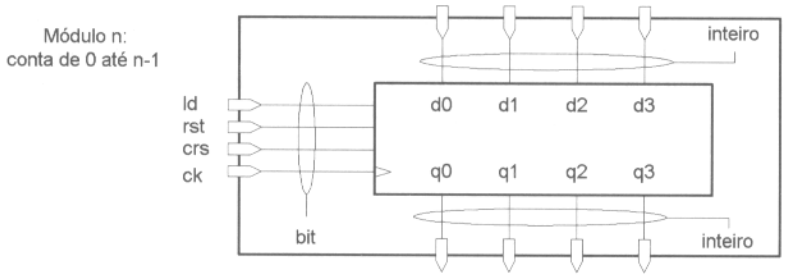


Figura 7.6.8 Diagrama de blocos do controlador para o conversor - Exercício 7.6.10.

7.6.11 Apresente a descrição de um contador com 4 bits crescente/decrescente com opção para carga de dados paralela e *reset* assíncrono. O módulo do contador deve ser definido por uma cláusula " GENERIC ", permitindo, assim, a alteração do valor máximo atingido pelo contador. O valor máximo de contagem pode estar na faixa de oito a quinze. A Figura 7.6.9 ilustra o contador. O terminal " ld " carrega o dado " d " na borda de subida do sinal de relógio " ck ". O terminal " crs " define se a contagem é crescente ou decrescente, e " rst " executa uma operação de *reset* assíncrona. Os sinais " d " e " q " são do tipo inteiro, e os sinais restantes, do tipo " bit ".



crs	rst	ld	q3	q2	q1	q0
X	1	X	0	0	0	0
X	0	1	d3	d2	d1	d0
1	0	0	cont. decresc.			
0	0	0	cont. cresc.			

Figura 7.6.9 Ilustração para o Exercício 7.6.11.

CAPÍTULO

8 SUBPROGRAMAS

Um subprograma é um recurso para isolar um conjunto de comandos freqüentemente utilizados, com objetivo de melhorar o entendimento da descrição. Esse conjunto de comandos pode realizar uma operação mais abstrata na descrição, como uma rotina de conversão de tipos, ou descrever o comportamento de um circuito. Funções e procedimentos são subprogramas que contêm comandos seqüenciais. Uma função retorna um único valor, enquanto um procedimento pode retornar vários valores.

Deve-se observar que o emprego de subprogramas para descrever um módulo de um circuito não resulta em uma economia de recursos no circuito final sintetizado. Cada chamada de subprograma pode levar à criação de uma unidade funcional para realizar as operações descritas no subprograma.

8.1 Subprogramas

Os subprogramas podem ser invocados em regiões de código concorrente ou seqüencial, sendo permitida, também, a chamada recursiva. Normalmente, a chamada recursiva de subprogramas não é aceita pelas ferramentas de síntese. Todos os objetos declarados localmente são inicializados quando um subprograma é invocado e são descartados ao final da execução. Isso significa que o valor assumido por uma variável declarada em um subprograma não é mantido após o término de sua execução.

A troca de informação em um subprograma é feita através de parâmetros que podem ser da classe constante, sinal, variável ou arquivo.[1] Eles são declarados na lista de interface do subprograma. Essa lista identifica a classe, o nome e o tipo dos objetos, bem como a direção com que a informação é transmitida (ver Figura 8.1.1).

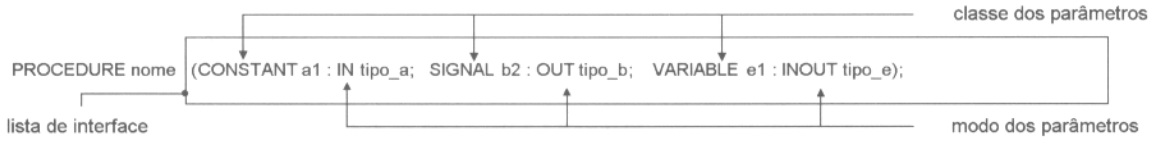


Figura 8.1.1 Lista de interface de um subprograma.

No caso de parâmetro declarado como constante ou variável, apenas o valor é transferido para o subprograma. Um parâmetro declarado como sinal permite que as informações de atributo sejam referenciadas. Assim, o atributo " EVENT " pode, por exemplo, ser referenciado em um subprograma, se a classe do parâmetro for declarada como sinal.

Os atributos " STABLE ", " QUIET ", " TRANSACTION ", e " DELAYED " não podem ser empregados em um subprograma, porque eles resultam em novos sinais (ver Cap. 5). Como sinais são criados numa fase anterior à simulação, e um subprograma deixa de existir após a sua execução, não seria possível a criação de um sinal no momento da execução de um subprograma. Pelo mesmo motivo, não é permitido declarar sinais em subprogramas.

Devem ser respeitadas algumas regras com relação à classe do parâmetro declarado no subprograma (parâmetro formal) e à classe do parâmetro que é passado ou retorna do subprograma (parâmetro real) (ver Figuras 8.1.2, 8.1.3 e 8.1.4). Um parâmetro formal da classe constante pode ser associado a parâmetros reais das classes

[1]O emprego da classe " FILE " em subprogramas não é abordado.

constante, sinal e variável. Os parâmetros da classe sinal ou variável podem ser associados exclusivamente a um parâmetro da mesma classe. Caso não seja necessária uma referência ao atributo de um parâmetro real, é mais conveniente definir o parâmetro formal da classe constante. Assim, o subprograma pode aceitar parâmetros reais das classes constante, sinal e variável, tornando o código mais flexível.

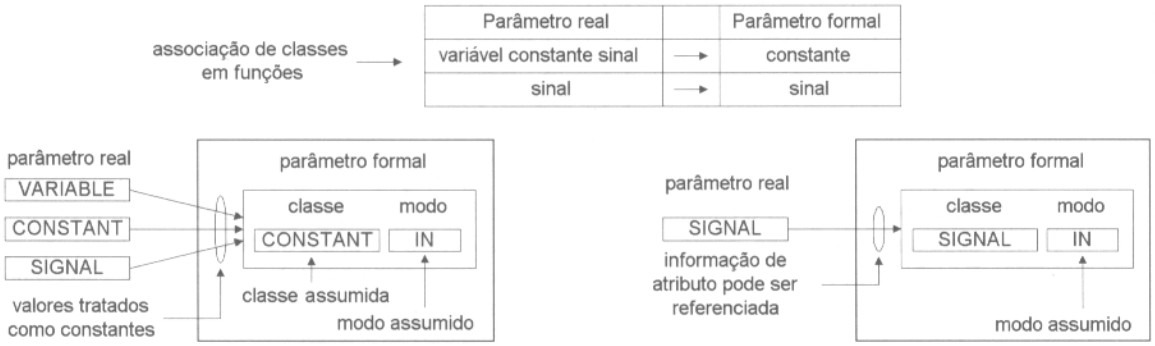


Figura 8.1.2 Passagem de dados para funções.

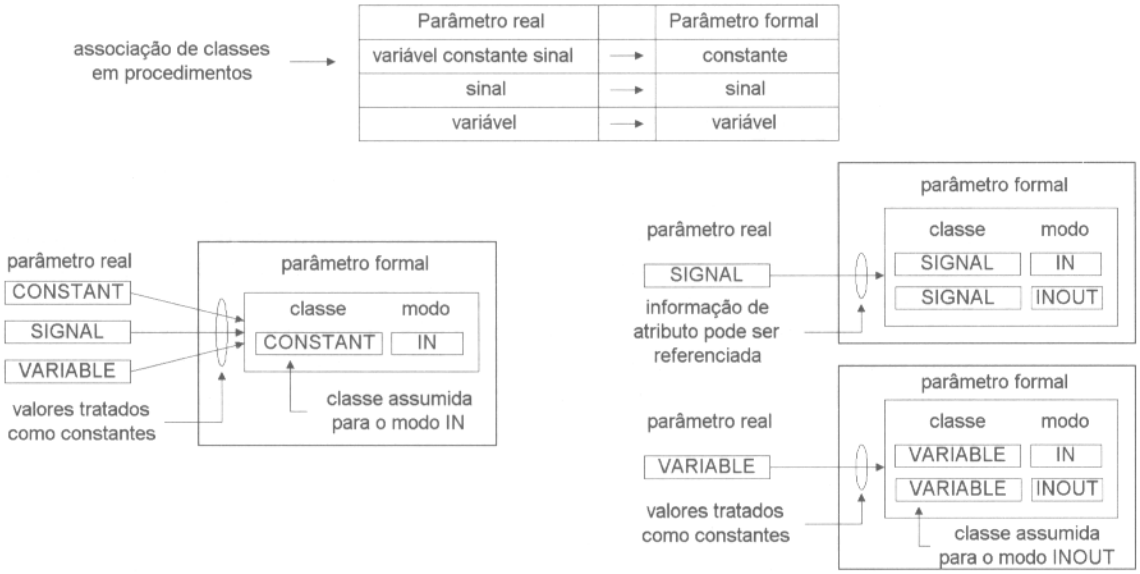


Figura 8.1.3 Passagem de dados para procedimentos.

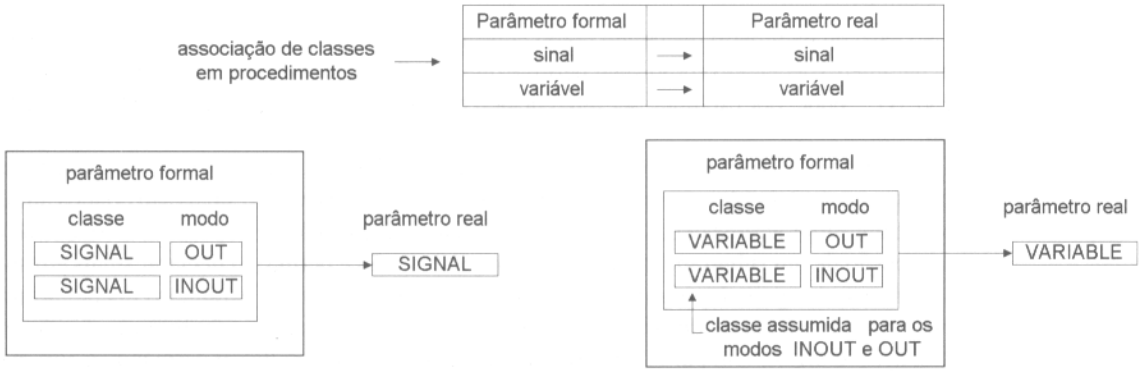


Figura 8.1.4 Passagem de dados de procedimentos.

Quando a classe de um parâmetro do modo " IN " não é especificada, pressupõe-se a classe constante. A classe variável é assumida no caso de parâmetros do modo " INOUT " ou " OUT ", quando a especificação de classe é omitida. Os parâmetros modo " INOUT " ou " OUT " aplicam-se unicamente nos procedimentos, tendo em vista que uma função pode retornar um único argumento. No caso particular de uma função, é possível omitir, também, o modo de um parâmetro, considerando que ele é obrigatoriamente " IN ".

Um subprograma consiste em duas partes: uma declaração e um corpo. A declaração define o nome do subprograma e a lista dos parâmetros formais. O corpo do subprograma contém um cabeçalho (cópia das informações contidas na declaração do subprograma) e a descrição do comportamento do subprograma.

A declaração de um subprograma é opcional. O cabeçalho do corpo do subprograma provê as informações necessárias, se a declaração for omitida. O corpo de um subprograma pode ser inserido:

- na declaração de uma entidade;
- no corpo de uma arquitetura;
- no corpo de um subprograma;
- no corpo de um pacote.

8.2 Função

As funções são invocadas por uma expressão, e retornam um único valor. A classe dos objetos declarada na lista de parâmetros pode ser " SIGNAL ", " CONSTANT " ou " FILE ", e o modo do objeto deve ser obrigatoriamente do tipo " IN ". A classe " VARIABLE " não é permitida. Uma função não altera os parâmetros que são passados a ela.

O argumento de retorno é identificado pelo comando " RETURN ", e uma função pode conter mais de um comando " RETURN ". Como não é possível declarar um sinal em um subprograma, atribuições de valores empregam sempre variáveis. O comando " WAIT " não é permitido em uma função. O Quadro 8.2.1 ilustra a declaração e corpo de uma função.

A chamada de uma função pode ocorrer em regiões de código concorrente ou seqüencial. Chamadas recursivas são permitidas, mas não são suportadas por ferramentas de síntese, devido à dificuldade em sintetizar um circuito correspondente.

```
-- declaracao da funcao
FUNCTION nome_funcao (SIGNAL   a : IN tipo_a;
                      SIGNAL   b :    tipo_b;
                      CONSTANT c : IN tipo_c;
                               d :    tipo_c)
                      RETURN          tipo_r;
```

```
-- corpo da funcao
FUNCTION nome_funcao (SIGNAL    a : IN tipo_a;
                      SIGNAL    b :    tipo_b;    -- assumido modo IN
                      CONSTANT  c : IN tipo_c;
                                d :    tipo_d)    -- assumida classe CONSTANT, assumido modo IN
                      RETURN           tipo_r IS
 --
 -- declaracao de tipo, constante, variavel
BEGIN
  -- regiao de codigo sequencial
  --
  RETURN expressao;                                -- valor de retorno
END nome_funcao;
```

Quadro 8.2.1 Declaração e corpo de uma função.

No Quadro 8.2.2 é apresentada uma função invocada de uma região de código concorrente. Os parâmetros passados para a função (parâmetros reais) pertencem à classe " SIGNAL ", e os parâmetros formais pertencem à classe " CONSTANT ". Como os parâmetros declarados na função pertencem à classe constante, as informações referentes aos atributos dos sinais não podem ser referenciadas. Uma variável foi declarada na função, a título de ilustração (ver linha 9).

```
1  ENTITY sb_fct0 IS
2    PORT (a_i, b_i   : IN  INTEGER RANGE 0 TO 15;
3          s_conc     : OUT INTEGER RANGE 0 TO 15);
4  END sb_fct0;
5
6  ARCHITECTURE exemplo OF sb_fct0 IS
7
8  FUNCTION soma (a : INTEGER; b : INTEGER) RETURN INTEGER IS
9    VARIABLE s : INTEGER;
10 BEGIN
11   s :=  a + b;
12   RETURN s;
13 END soma;
14
15 BEGIN
16    s_conc <= soma(a_i, b_i);
17 END exemplo;
```

Quadro 8.2.2 Exemplo de uma função invocada de uma região concorrente.

No Quadro 8.2.3 é apresentado um código semelhante ao apresentado no Quadro 8.2.2, sendo a função invocada de uma região de código seqüencial. Os parâmetros reais pertencem à classe " VARIABLE ", e os parâmetros formais pertencem à classe " CONSTANT ". Nesse caso, os atributos dos sinais não podem ser referenciados.

```
1  ENTITY sb_fct1 IS
2    PORT (a_i, b_i    : IN  INTEGER RANGE 0 TO 15;
3          s_sequ      : OUT INTEGER RANGE 0 TO 15);
4  END sb_fct1;
5
6  ARCHITECTURE exemplo OF sb_fct1 IS
7
8  FUNCTION soma (a : INTEGER; b : INTEGER) RETURN INTEGER IS
9  BEGIN
10   RETURN a + b;
11 END soma;
12
13 BEGIN
14   abc: PROCESS(a_i, b_i)
15   VARIABLE a, b: INTEGER RANGE 0 TO 15;
16   BEGIN
17     a := a_i;  b := b_i;
18     s_sequ <= soma(a, b);
19   END PROCESS;
20 END exemplo;
```

Quadro 8.2.3 Exemplo de uma função invocada de uma região seqüencial.

A função descrita no Quadro 8.2.4 detecta a borda de subida de um sinal, retornando um valor verdadeiro se essa condição for detectada. A operação da função necessita das informações referentes aos atributos do sinal; por isso, é necessário que o parâmetro formal pertença à classe " SIGNAL ". Note que nesse exemplo, o corpo da função foi inserido na declaração da entidade.

```
1  ENTITY sb_fct9 IS
2    PORT (ck, d : IN  BIT;
3          q_fnc : OUT BIT);
4
5    FUNCTION subida (SIGNAL ck : IN BIT) RETURN BOOLEAN IS
6     BEGIN
7       RETURN (ck'EVENT) AND (ck = '1');
8    END subida;
9  END sb_fct9;
10
11 ARCHITECTURE teste OF sb_fct9 IS
12 BEGIN
13   abc: PROCESS (ck)
14   BEGIN
15     IF subida(ck) THEN q_fnc <= d;
16     END IF;
17   END PROCESS abc;
18 END teste;
```

Quadro 8.2.4 Exemplo de uma função que necessita dos atributos de um dos parâmetros.

A Figura 8.2.1 ilustra o resultado da síntese das três descrições propostas como exemplos de funções. Os dois somadores são resultantes de funções invocadas em regiões de código concorrente e seqüencial. O *flip flop* é resultado de uma função em que os atributos do parâmetro real foram referenciados. Nos três casos, as descrições foram corretamente sintetizadas.

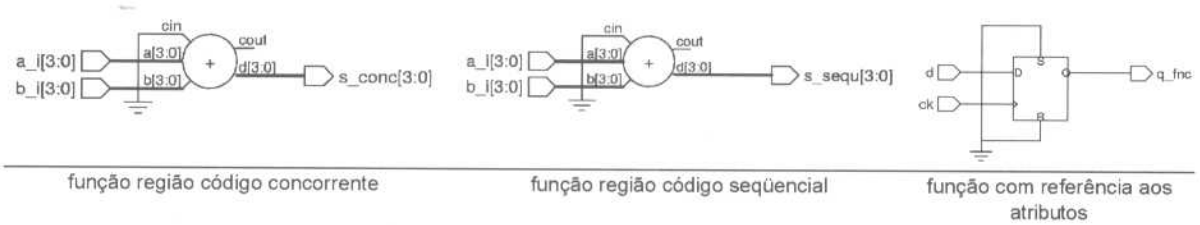


Figura 8.2.1 Resultado da síntese das funções propostas - nível RTL.

8.3 Procedimento

Os procedimentos são invocados por um comando, e permitem o retorno de mais de um valor. A classe dos objetos declarados na lista de parâmetros pode ser " SIGNAL ", " VARIABLE ", " CONSTANT " ou " FILE ", sendo permitidos os modos " IN ", " OUT " ou " INOUT ". Como parâmetros formais podem ser do modo " INOUT ", um procedimento pode alterar o valor dos parâmetros passados a ele. O comando " RETURN " não é obrigatório e causa o término da execução do procedimento. O Quadro 8.3.1 ilustra o exemplo de uma declaração e um corpo de um procedimento.

Apesar de não ser permitido declarar um sinal em um subprograma, é possível atribuir valor para sinal no interior de um procedimento. Isso pode ocorrer na atribuição de valores em parâmetros formais da classe sinal, modo " OUT " ou modo " INOUT ".

A chamada de um procedimento, assim como em funções, pode ocorrer em regiões de código concorrente ou seqüencial. Na região de código concorrente, a execução ocorre na alteração de valor em um dos parâmetros de modo " IN " ou " INOUT ". Em uma região de código seqüencial, o procedimento é tratado como um comando seqüencial, isto é, o procedimento é executado sempre que encontrado no código.

```
-- declaracao do procedimento
PROCEDURE nome_proced (SIGNAL   a : IN    tipo_a;
                       VARIABLE c : OUT   tipo_c;
                       VARIABLE e : INOUT tipo_e);
```

```
-- corpo do procedimento
PROCEDURE nome_proced (SIGNAL   a : IN    tipo_a;
                       VARIABLE c : OUT   tipo_c;
                       VARIABLE e : INOUT tipo_e) IS
  --
  -- declaracao de tipo, constante, variavel
BEGIN
   -- regiao de codigo sequencial
   --
END nome_proced;
```

Quadro 8.3.1 Declaração e corpo de um procedimento.

No Quadro 8.3.2 é apresentado um procedimento invocado de uma região de código concorrente. Como a classe dos parâmetros formais modo " IN " não foi especificada, fica subentendido que eles pertencem à classe " CONSTANT ". Os parâmetros formais do modo " OUT " devem pertencer à classe " SIGNAL ", porque eles são empregados em uma área de código concorrente. Não é permitido um objeto da classe variável numa região de código concorrente.

```
1  ENTITY sb_prc0 IS
2    PORT (a_i, b_i, c_i     : IN  INTEGER RANGE 0 TO 15;
3          som_con, sub_con : OUT INTEGER RANGE 0 TO 15);
4  END sb_prc0;
5
6  ARCHITECTURE teste OF sb_prc0 IS
7
8  PROCEDURE soma_sub( a, b, c          : IN  INTEGER RANGE 0 TO 15;
9                      SIGNAL som, sub : OUT INTEGER RANGE 0 TO 15) IS
10 BEGIN
11   som <= a + b;
12   sub <= a - c;
13 END soma_sub;
14
15 BEGIN
16   soma_sub(a_i, b_i, c_i, som_con, sub_con);
17 END teste;
```

Quadro 8.3.2 Exemplo de um procedimento invocado de uma região concorrente.

O Quadro 8.3.3 apresenta um procedimento semelhante ao descrito no Quadro 8.3.2 invocado de uma região de código seqüencial. Várias opções de parâmetros foram apresentadas a título de ilustração. No caso da linha 8, não é necessário especificar a classe, pois para o modo " IN " é assumida a classe constante. O mesmo ocorre com a linha 11, pois para o modo " OUT " é assumida a classe variável. Note ainda que o parâmetro real associado ao parâmetro formal " b " deve ser uma variável. Por esse motivo, o sinal " b_i " foi transferido para a variável " b_v " (linha 22 do código).

```
1  ENTITY sb_prc1 IS
2    PORT (a_i, b_i, c_i     : IN  INTEGER RANGE 0 TO 15;
3          som_con, sub_con : OUT INTEGER RANGE 0 TO 15);
4  END sb_prc1;
5
6  ARCHITECTURE teste OF sb_prc1 IS
7
8  PROCEDURE soma_sub(CONSTANT a   : IN  INTEGER RANGE 0 TO 15;
9                     VARIABLE b   : IN  INTEGER RANGE 0 TO 15;
10                    SIGNAL   c   : IN  INTEGER RANGE 0 TO 15;
11                    VARIABLE som : OUT INTEGER RANGE 0 TO 15;
12                    SIGNAL   sub : OUT INTEGER RANGE 0 TO 15) IS
13 BEGIN
14   som := a + b;
15   sub <= a - c;
16 END soma_sub;
17
18 BEGIN
19   abc: PROCESS(a_i, b_i, c_i)
20     VARIABLE b_v, som_v : INTEGER RANGE 0 TO 15;
21   BEGIN
22     b_v := b_i;
23     soma_sub(a_i, b_v, c_i, som_v, sub_con);
24     som_con <= som_v;
25   END PROCESS;
26 END teste;
```

Quadro 8.3.3 Exemplo de um procedimento invocado de uma região seqüencial.

O procedimento descrito no Quadro 8.3.4 implementa um *flip flop* com duas saídas. Observe que a chamada ao procedimento emprega uma associação nomeada. Como é detectada a borda de subida do sinal de relógio, é necessária uma referência aos atributos desse sinal. Assim, deve-se declarar o parâmento formal de " ck " como da classe sinal.

```
1  ENTITY sb_prc12 IS
2    PORT (ck, d : IN  BIT;
3          q, q_b: BUFFER BIT);
4  END sb_prc12;
5
6  ARCHITECTURE teste OF sb_prc12 IS
7    PROCEDURE flip_d(SIGNAL relogio, dado:  IN BIT;
8                     SIGNAL valor, valor_b: OUT BIT) IS
9    BEGIN
10     IF (relogio'EVENT AND relogio = '1') THEN
11       valor   <= dado;
12       valor_b <= NOT dado;
13     END IF;
14   END flip_d;
15 BEGIN
16   abc: PROCESS (ck)
17   BEGIN
18     flip_d(relogio => ck, dado => d, valor => q, valor_b => q_b);
19   END PROCESS abc;
20 END teste;
```

Quadro 8.3.4 Procedimento com associação nomeada e referência aos atributos de parâmetro.

A Figura 8.3.1 ilustra o resultado da síntese das três descrições propostas como exemplos de procedimentos. Os somadores e subtratores são resultantes de procedimentos invocados em regiões de código concorrente e seqüencial. O *flip flop* é resultado de um procedimento em que os atributos foram referenciados pelo procedimento do subprograma.

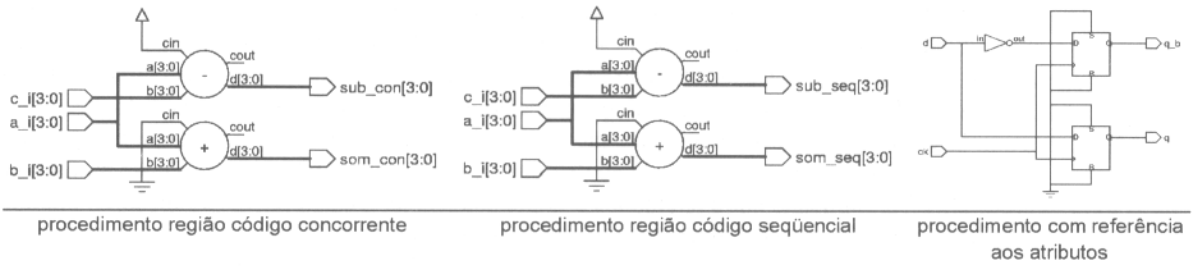


Figura 8.3.1 Resultado da síntese dos procedimentos propostos - nível RTL.

8.4 Sobrecarregamento de subprogramas e operadores

Uma mesma designação pode ser empregada para diferentes subprogramas. Nesse caso, o subprograma e o seu nome estão sobrecarregados, ou " overloaded ". O sobrecarregamento permite o emprego do mesmo nome para subprogramas funcionalmente semelhantes, mas que diferem na lista de parâmetros. A seleção do subprograma é feita com base:

- no número de parâmetros;
- no tipo dos parâmetros;
- na ordem dos parâmetros;
- no nome dos parâmetros formais (associação nomeada);
- no tipo do resultado (caso de funções).

A impossibilidade de se identificar o subprograma constitui uma condição de erro.

No Quadro 8.4.1 é apresentada uma descrição contendo três subprogramas com a mesma designação. A função empregada, no caso do sinal " si ", é a primeira, pois ela é única com dois operandos do tipo inteiro. O valor do sinal " ti ", por sua vez, é definido pela segunda função, por ser ela a única que opera com três parâmetros. Finalmente, no caso do sinal " sr ", é empregada a terceira função, porque os parâmetros são do tipo " REAL ".

```
1  ENTITY sb_over1 IS
2    PORT (ai, bi, ci : IN  INTEGER RANGE 0   TO 15;
3          ar, br     : IN  REAL    RANGE 0.0 TO 15.0;
4          si, ti     : OUT INTEGER RANGE 0   TO 31;
5          sr         : OUT REAL    RANGE 0.0 TO 31.0);
6  END sb_over1;
7
8  ARCHITECTURE teste OF sb_over1 IS
9    FUNCTION soma (a, b : INTEGER) RETURN INTEGER IS     -- 1a funcao
10   BEGIN
11     RETURN a + b;
12   END soma;
13
14   FUNCTION soma (a, b, c : INTEGER) RETURN INTEGER IS  -- 2a funcao
15   BEGIN
16     RETURN a + b + c;
17   END soma;
18
19   FUNCTION soma (x, y : REAL) RETURN REAL IS           -- 3a funcao
20   BEGIN
21     RETURN x + y + 1.0;
22   END soma;
23 BEGIN
24   si <= soma(ai,bi);     -- ai,bi operandos tipo integer    -> 1a funcao
25   ti <= soma(ai,bi,ci); -- ai,bi,ci operandos tipo integer -> 2a funcao
26   sr <= soma(ar,br);     -- ar,br operandos tipo real       -> 3a funcao
27 END teste;
```

Quadro 8.4.1 Exemplo de subprogramas sobrecarregados - " overloaded ".

A linguagem VHDL permite, também, o sobrecarregamento de operadores lógicos, relacionais e aritméticos predefinidos, com o objetivo de facilitar a leitura do código. Isso se deve ao fato de que operadores predefinidos, como, por exemplo, " AND ", " + ", " = ", são funções. As mesmas regras observadas à sobrecarga de uma função são aplicadas nesses casos, exceção feita ao nome da função, que deve ser declarado entre aspas. A sobrecarga de operadores é restrita aos operadores predefinidos (ver Tabela 2.4.1, no Capítulo 2).

O Quadro 8.4.2 apresenta o exemplo de um código que emprega a sobrecarga no operador de soma " + ". A nova função permite realizar a adição de dois valores do tipo " bit_vector ". Desse modo, toda vez que o operador " + " for aplicado entre operandos do tipo " bit_vector " é empregada a função " + " contida na descrição. A Figura 8.4.1 ilustra o resultado obtido com a síntese da descrição.

O exemplo mostra como o escopo de uma operação predefinida pode ser expandido para melhorar a leitura de um código. A descrição, entretanto, não é o modo mais eficiente para realização de somas entre tipos " bit_vector ". No Capítulo 11 será descrito o padrão numérico IEEE 1076.3 que contém uma série de operações aritméticas para vetores.

```
1  ENTITY sb_over2 IS
2    GENERIC(n : INTEGER := 3);
3    PORT (a, b : IN  BIT_VECTOR(n-1 DOWNTO 0);
4          s    : OUT BIT_VECTOR(n-1 DOWNTO 0));
5  END sb_over2;
6
7  ARCHITECTURE teste OF sb_over2 IS              -- retorna tipo
8    FUNCTION "+" (x, y : BIT_VECTOR(n-1 DOWNTO 0)) RETURN BIT_VECTOR IS
9      VARIABLE v : BIT := '0';
10     VARIABLE s : BIT_VECTOR(n-1 DOWNTO 0);
11   BEGIN
12     FOR i IN 0 TO n-1 LOOP
13       s(i) :=  x(i) XOR y(i) XOR v;
14       v    := (x(i) AND y(i)) OR (v AND (x(i) OR y(i)));
15     END LOOP;
16     RETURN s;
17   END "+";
18 BEGIN
19   s <= a + b;  -- a,b operandos tipo bit-vector
20 END teste;
```

Quadro 8.4.2 Exemplo de sobrecarga de operador.

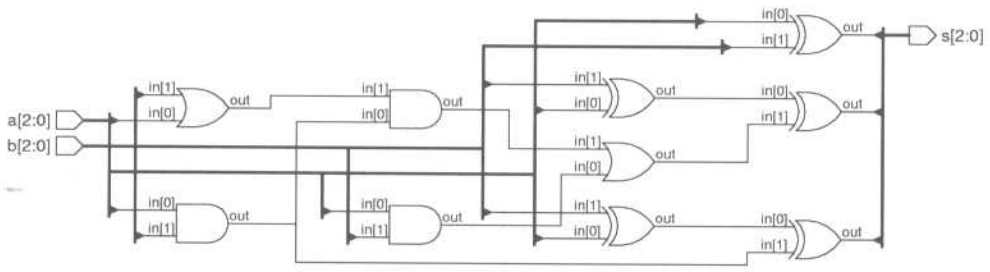


Figura 8.4.1 Resultado da síntese de uma descrição com sobrecarga do operador " + ".

8.5 Vetores sem especificação de limites em subprogramas

A linguagem VHDL permite que os limites de vetores em parâmetros formais sejam deixados em aberto. A dimensão dos vetores, nesses casos, é determinada pelo tamanho do parâmetro real que realiza a chamada do subprograma. Isso possibilita a criação de funções e procedimentos de uso mais amplo, reduzindo a extensão do código através da reutilização dos subprogramas para diferentes condições. Cuidados devem ser tomados com relação ao sentido dos limites dos vetores. Uma boa descrição deve prever que o sentido definido para um vetor pode ser tanto crescente, " (a TO b) ", como decrescente, " (b DOWNTO a) ". Para ambos os casos, o comportamento do código deve ser previsto.

O Quadro 8.5.1 ilustra um novo código para a função que realiza a soma de valores do tipo " bit_vector ", onde o bit mais significativo ocupa a posição esquerda do vetor. Os limites dos parâmetros " x " e " y " não foram especificados. Note que, numa função, é especificado apenas o tipo do parâmetro de retorno. Seria um erro, por exemplo, especificar o tamanho do vetor após a palavra reservada " RETURN " na linha 9. O tamanho da variável " s " no interior do subprograma é determinado pela dimensão do argumento real associado ao parâmetro formal " x ". O número de iterações no laço é estabelecido pela faixa discreta de valores resultante da referência ao atributo " REVERSE_RANGE " de " s ". Assim, o esquema de iteração inicia pelo bit de posição menos significativa. Na descrição, duas chamadas a sub-rotina são executadas, uma com vetores de três bits e outra com vetores de quatro bits.

Na Figura 8.5.1 é apresentado o resultado da síntese da descrição do Quadro 8.5.1. Conforme se pode observar, os dois somadores necessários para a implementação do circuito foram corretamente inferidos, um de três bits e outro de quatro bits.

```
 1  ENTITY sb_over4 IS
 2    PORT (a_dec, b_dec : IN  BIT_VECTOR(3 DOWNTO 0);    -- bit3= +significativo  bit0= -sig.
 3          a_cre, b_cre : IN  BIT_VECTOR(0 TO 2);        -- bit0= +significativo  bit2= -sig.
 4          s_dec        : OUT BIT_VECTOR(3 DOWNTO 0);
 5          s_cre        : OUT BIT_VECTOR(0 TO 2));
 6  END sb_over4;
 7
 8  ARCHITECTURE teste OF sb_over4 IS
 9    FUNCTION "+" (x, y : BIT_VECTOR) RETURN BIT_VECTOR IS
10      VARIABLE v : BIT := '0';                --          +sig.    -sig.    +sig.   -sig.)
11      VARIABLE s : BIT_VECTOR(x'RANGE);       -- x'RANGE= ( 3 DOWNTO 0 ) ou ( 0   TO  2 )
12
13    BEGIN                                     --     -sig. +sig.          -sig.    +sig
14      FOR i IN x'REVERSE_RANGE LOOP           -- (i IN 0  TO 3)   ou (i IN 2 DOWNTO 0)
15        s(i) :=  x(i) XOR y(i) XOR v;
16        v    := (x(i) AND y(i)) OR (v AND (x(i) OR y(i)));
17      END LOOP;
18      RETURN s;
19    END "+";
20  BEGIN
21    s_dec <= a_dec + b_dec;  -- a_dec, b_dec operandos 4 bits tipo bit-vector
22    s_cre <= a_cre + b_cre;  -- a_cre, b_cre operandos 3 bits tipo bit-vector
23  END teste;
```

Quadro 8.5.1 Exemplo de sobrecarga de operador.

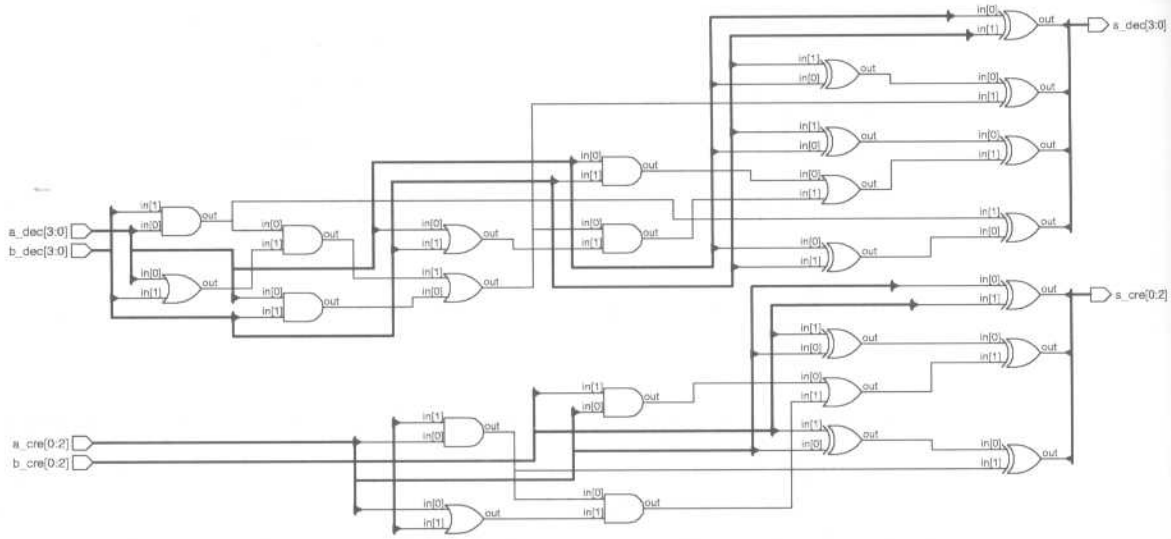


Figura 8.5.1 Síntese da descrição do Quadro 8.5.1.

8.6 Cuidados na descrição

Comportamento de variáveis declaradas em subprogramas

O comportamento de variáveis declaradas em subprogramas difere do comportamento de variáveis em processos. Uma variável declarada em um subprograma perdura até o término do subprograma, enquanto uma variável declarada em um processo persiste até o término da simulação. Isso representa que, a cada chamada de um subprograma, a variável é criada e inicializada. O seu valor, portanto, não é mantido, como ocorre em uma variável declarada em um processo.

Os Quadros 8.6.1 e 8.6.2 apresentam duas descrições semelhantes que implementam contadores. Para efeito de comparação, o valor da contagem é armazenado em variáveis declaradas em processos e subprogramas. O resultado da simulação para ambos os casos é apresentado na Figura 8.6.1. O sinal associado ao processo, " cnt_process ", é incrementado a cada borda de subida de " ck ", pois a variável " a " é mantida durante toda a simulação. No caso dos sinais correspondentes aos subprogramas, " cnt_function " e " cnt_process ", o valor da

```
1  ENTITY sb_fct6 IS
2    PORT (ck : IN  BIT;
3          cnt_process, cnt_function : OUT INTEGER RANGE 0 TO 15);
4  END sb_fct6;
5
6  ARCHITECTURE exemplo OF sb_fct6 IS
7
8  FUNCTION conta_f (SIGNAL ck : IN BIT) RETURN INTEGER IS
9    VARIABLE af : INTEGER RANGE 0 TO 15;
10 BEGIN
11   IF (ck'EVENT) AND (ck = '1') THEN af := af +1;
12   END IF;
13   RETURN af;
14 END conta_f;
15
16 BEGIN
17   abc: PROCESS (ck)
18     VARIABLE a : INTEGER RANGE 0 TO 15;
19   BEGIN
20     IF (ck'EVENT) AND (ck = '1') THEN  a := a +1;
21     END IF;
22     cnt_process <= a;
23    END PROCESS;
24
25   cnt_function <= conta_f(ck);
26 END exemplo;
```

Quadro 8.6.1 Exemplo de uma função não-válida.

contagem fica variando entre " 0 " e " 1 ". Como as variáveis " af " e " ap " são criadas a cada chamada do subprograma, na borda de subida ela é incrementada, passando de " 0 " para " 1 ", e na borda de descida é mantido o seu valor inicial. Note que, para ambos os casos, o valor inicial das variáveis, " 0 ", corresponde ao elemento mais à esquerda da faixa de valores definidos, no caso " 0 TO 15 ".

```
 1 ENTITY sb_prc6 IS
 2   PORT (ck    : IN  BIT;
 3         cnt_process, cnt_procedure : OUT INTEGER RANGE 0 TO 15);
 4 END sb_prc6;
 5
 6 ARCHITECTURE exemplo OF sb_prc6 IS
 7
 8 PROCEDURE conta_p (SIGNAL ck  : IN BIT;  SIGNAL a_o : OUT INTEGER RANGE 0 TO 15) IS
 9   VARIABLE ap : INTEGER RANGE 0 TO 15;
10 BEGIN
11   IF (ck'EVENT) AND (ck = '1') THEN  ap := ap +1;
12   END IF;
13   a_o <= ap;
14 END conta_p;
15
16 BEGIN
17   abc: PROCESS (ck)
18     VARIABLE a : INTEGER RANGE 0 TO 15;
19   BEGIN
20     IF (ck'EVENT) AND (ck = '1') THEN a := a +1;
21     END IF;
22     cnt_process <= a;
23   END PROCESS;
24
25   conta_p(ck, cnt_procedure);
26 END exemplo;
```

Quadro 8.6.2 Exemplo de um procedimento não-válido.

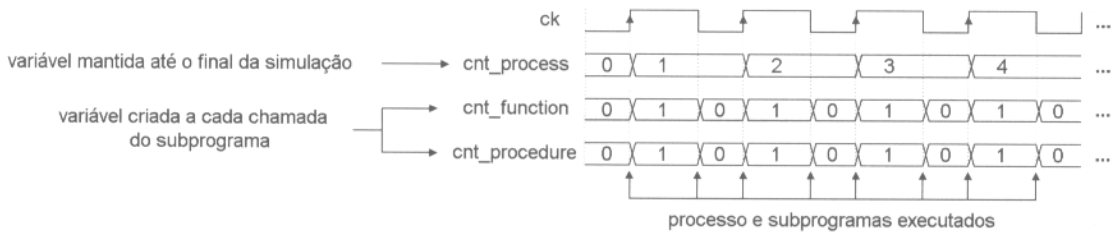


Figura 8.6.1 Problemas encontrados nas descrições dos Quadros 8.6.1 e 8.6.2.

Chamada de procedimentos: regiões concorrentes

A chamada de um procedimento é um comando seqüencial; entretanto, é possível invocar um procedimento de uma região concorrente de código, conforme mostrado no exemplo do Quadro 8.3.2. Nesses casos, o comando leva à criação de um processo equivalente, contendo a chamada ao processo e um comando " WAIT " no seu final. Cada parâmetro formal do modo " IN " ou " INOUT " é incluído no comando " WAIT " do processo equivalente. Assim, o processo fica aguardando a ocorrência de um evento nos sinais associados a parâmetros formais do modo " IN " ou " INOUT ". O Quadro 8.6.3 ilustra o processo equivalente resultante de uma chamada de um

```
-- modo  IN   IN   IN    OUT    OUT
soma_sub(a_i, b_i, c_i, som_o, sub_o);   -- regiao de codigo concorrente
```

```
-- processo equivalente
PROCESS
BEGIN
  soma_sub(a_i, b_i, c_i, som_o, sub_o);
  WAIT ON a_i, b_i, c_i;
END PROCESS;
```

Quadro 8.6.3 Processo equivalente à chamada de um procedimento em uma região concorrente.

processo em uma região de código concorrente. Nesse exemplo, a cada evento nos sinais " a_i ", " b_i " ou " c_i ", o procedimento é executado.

Parâmetros formais da classe variável acarretam problemas no caso de processos com chamada em regiões concorrentes. Isso ocorre porque um parâmetro formal da classe variável deve estar associado a um parâmetro real da mesma classe. Para tal condição ser satisfeita, seria necessária a existência de uma variável fora de uma região de código seqüencial.

No Quadro 8.6.4 é apresentado um exemplo que ilustra esse problema. Como o parâmetro formal " qv " do procedimento " flip_v " pertence à classe variável, é necessário que o parâmetro real seja da mesma classe. Desse modo, uma chamada do procedimento em uma região de código concorrente, conforme apresentada na linha 22, levaria à existência de uma variável em uma região de código concorrente, o que não é permitido. A mesma observação pode ser feita ao parâmetro formal " dv ".

```
 1 ENTITY sb_prc10 IS
 2   PORT (ck, d  : IN  BIT;
 3         q      : OUT BIT);
 4 END sb_prc10;
 5
 6 ARCHITECTURE teste OF sb_prc10 IS
 7
 8 PROCEDURE flip_v(SIGNAL ck : IN BIT; VARIABLE dx : IN BIT; VARIABLE qx : OUT BIT) IS
 9 BEGIN
10   IF ck'EVENT AND ck ='1' THEN qx := dx;
11   END IF;
12 END flip_v;
13
14 BEGIN
15   abc: PROCESS(ck)
16     VARIABLE dv, qv : BIT;
17   BEGIN                  -- correto                ck      dx      qx
18    dv := d;              -- parametros formais: sinal variavel variavel
19    flip_v(ck, dv, qv);   -- parametros reais:   sinal variavel variavel
20    q <= qv;
21   END PROCESS;           -- errado                 ck      dx      qx
22 --flip_v(ck, d, q);      -- parametros formais: sinal variavel variavel
23 END teste;               -- parametros reais:   sinal   sinal    sinal
```

Quadro 8.6.4 Chamada de procedimento em região de código concorrente - cuidados.

Variáveis compartilhadas

Na versão VHDL-1993 é possível declarar variáveis na região da arquitetura. Elas são denominadas variáveis compartilhadas, " SHARED VARIABLE ", e têm como característica a possibilidade de serem utilizadas por mais de um processo. Variáveis compartilhadas fornecem uma opção de comunicação entre dois processos sem o emprego de um sinal. Esse tipo de variáveis deve ser empregado com cautela, pois elas, também, recebem o valor instantaneamente. Assim, o resultado de dois processos atribuindo valores a uma mesma variável pode ser imprevisível. A linguagem não define o comportamento nesses casos; o resultado depende da implementação.

O Quadro 8.6.5 ilustra uma aplicação desse tipo de variável. A variável compartilhada " qv " é declarada na região da arquitetura (ver linha 8). Ela estabelece apenas a comunicação entre o processo " abc " e " def ". A descrição corresponde a um simples *flip flop*, e pode ser sintetizada, conforme ilustra a Figura 8.6.2.

```
 1  -- VHDL-1993 somente
 2  ENTITY share_v IS
 3    PORT (ck, d  : IN  BIT;
 4          q      : OUT BIT);
 5  END share_v;
 6
 7  ARCHITECTURE teste OF share_v IS
 8    SHARED VARIABLE qv : BIT;
 9  BEGIN
10    abc: PROCESS(ck)
11    BEGIN
12      IF ck'EVENT AND ck ='1' THEN qv := d;
13      END IF;
14    END PROCESS;
15
16    def: PROCESS(ck)
17    BEGIN
18      q <= qv;
19    END PROCESS;
20  END teste;
```

Quadro 8.6.5 Exemplo do uso de variáveis compartilhadas - somente versão VHDL-1993.

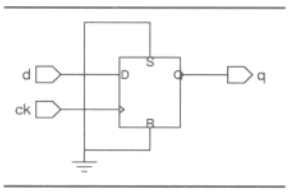


Figura 8.6.2 Circuito sintetizado empregando variáveis compartilhadas.

8.7 Principais pontos abordados

- As seguintes associações entre parâmetros que possuem modo podem ser observadas em subprogramas:
  - par. formal, classe constante ↔ par. real, classe constante, variável ou sinal;
  - par. formal, classe sinal ↔ par. real, classe sinal;
  - par. formal, classe variável ↔ par. real, classe variável.

  Somente os parâmetros formais da classe sinal permitem que as informações de atributo sejam referenciadas. Parâmetros da classe arquivo não possuem modo.

- Funções:
  - invocadas por uma expressão;
  - retornam um único valor;
  - classe dos parâmetros formais " SIGNAL " ou " CONSTANT ";
  - modo dos parâmetros formais obrigatoriamente do tipo " IN ";
  - argumento de retorno é identificado pelo comando " RETURN ".

- Procedimentos:
  - invocados por um comando;
  - permitem o retorno de mais de um valor;
  - classe dos parâmetros formais " SIGNAL ", " VARIABLE " ou " CONSTANT ";
  - modo dos parâmetros formais " IN ", " OUT " ou " INOUT ";
  - podem alterar o valor dos parâmetros passados.

- O sobrecarregamento, ou " overloading ", encontra aplicação nos casos de subprogramas que diferem nos elementos da lista de parâmetro, mas são funcionalmente semelhantes.

- Variáveis declaradas em subprogramas são mantidas até o término do subprograma; variáveis declaradas em processos são mantidas até o término da simulação.

8.8 Exercícios

8.8.1 No Quadro 8.8.1 são apresentadas duas funções que realizam as operações de um somador completo. Apresente a descrição de um somador de 4 bits empregando essas duas funções, sendo a chamada das funções executada de uma região de código concorrente.

8.8.2 Altere o código da solução do Exercício 8.8.1, realizando agora a chamada da função de uma região de código seqüencial.

8.8.3 No Quadro 8.8.2 é apresentada a proposta de uma função para detectar o menor de dois valores. A descrição contém erros. Determine os erros e apresente o código correto.

```
1  FUNCTION soma (a : IN BIT; b : IN BIT; c : IN BIT) RETURN BIT IS
2    BEGIN
3      RETURN a XOR b XOR c;
4  END soma;
5
6  FUNCTION vai_um (a: BIT; b: BIT; c: BIT) RETURN BIT IS
7    BEGIN
8      RETURN (a AND b) OR (a AND c) OR (b AND c);
9  END vai_um;
```

Quadro 8.8.1 Funções para um somador completo.

```
1  ENTITY som_fct7 IS
2    PORT (a_i, b_i    : IN  INTEGER RANGE 0 TO 15;
3          s_o         : OUT INTEGER RANGE 0 TO 15);
4  END som_fct7;
5
6  ARCHITECTURE falha OF som_fct7 IS
7
8  FUNCTION minimo (a : INTEGER;  b : INTEGER) RETURN INTEGER IS
9    SIGNAL min  : INTEGER RANGE 0 TO 15;
10 BEGIN
11   IF (a < b)
12     THEN min <= a;
13     ELSE min <= b;
14   END IF;
15   RETURN min;
16 END minimo;
17
18 BEGIN
19   s_o <= minimo(a_i, b_i);
20 END falha;
```

Quadro 8.8.2 Função com erro.

8.8.4 Apresente o código de um somador de 4 bits empregando unidades de um somador completo de 1 bit. Cada somador completo é implementado por um procedimento contendo três entradas e duas saídas: soma e vai um. A arquitetura deve realizar a chamada dos procedimentos de uma região de código concorrente.

8.8.5 Repita o Exercício 8.8.4 de modo que a chamada do procedimento seja executada de uma região de código seqüencial.

8.8.6 Apresente o código de dois subprogramas que permitam realizar as seguintes operações de conversão: tipo vetor de bits para um tipo inteiro, e tipo inteiro para vetor de bits. Considere que o tamanho do vetor de bits e a faixa de valores para o tipo inteiro são definidos quando o subprograma é invocado.

8.8.7 Apresente a descrição de um subprograma de *flip flop* tipo D, operando conforme a Tabela 8.8.1. Os sinais de " set " e " clr " são ativos no nível baixo. A transferência do dado da entrada " d " para as saídas " q " e " q_b " é feita na borda de subida do sinal de relógio " ck ".

Tabela 8.8.1 Definição do *flip flop* para o Exercício 8.8.7.

operação	entradas				saídas	
	set	clr	ck	d	q	q_b
set - assíncrono	0	1	X	X	1	0
clear - assíncrono	1	0	X	X	0	1
não definido	0	0	X	X	1	1
transfere dado	1	1	0 → 1	0	0	1
	1	1	0 → 1	1	1	0

8.8.8 Empregando o *flip flop* tipo D desenvolvido no Exercício 8.8.7, apresente uma descrição para implementar o circuito da Figura 8.8.1. Cada *flip flop* deve corresponder a uma chamada de um subprograma. O circuito descrito na figura implementa um gerador de padrões pseudo-aleatórios por meio de um registrador de deslocamento com realimentação linear " LFSR ", ou *linear feedback shift register*. Um LFSR consiste em um registrador de deslocamento realimentado com valores gerados por portas do tipo " ou exclusivo ". No exemplo da figura, o LFSR contém quatro estágios, e gera 15 padrões pseudo-aleatórios, excetuando o padrão " 0000 ". Para iniciar a seqüência, deve ser fornecida uma "semente", ou padrão inicial, diferente de " 0000 ". A seqüência de padrões esperada é apresentada na figura.

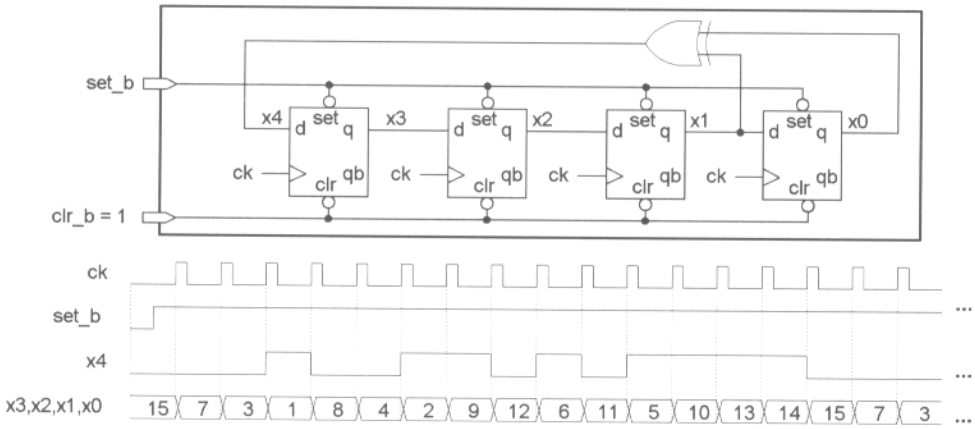
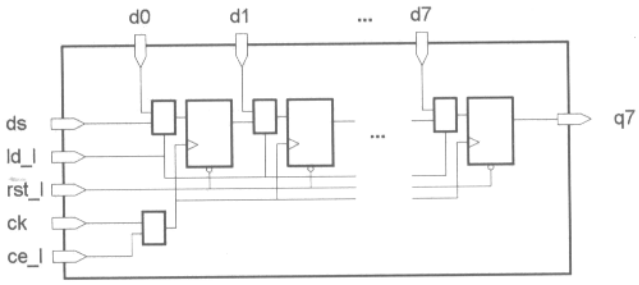


Figura 8.8.1 Registrador de deslocamento com realimentação linear LFSR (Exercício 8.8.8).

8.8.9 Desenvolva o código para um registrador de deslocamento equivalente ao circuito integrado 74166 da família TTL, conforme solicitado no Exercício 6.12.7. Nessa nova proposta, o registrador deve ser descrito na forma de um procedimento. Lembre-se de que uma variável declarada em um subprograma é inicializada a cada chamada do subprograma, e, portanto, não armazena um valor. Para teste do procedimento, insira o subprograma numa região de código concorrente de uma entidade. A Figura 8.8.2 apresenta novamente o diagrama de blocos do registrador.



rst_l	ld_l	ce_l	ck	operação
0	X	X	X	reset assíncrono
1	0	0	↗	carga paralela
1	1	0	↗	desloca
1	X	1	X	desabilitado

Figura 8.8.2 Registrador de deslocamento equivalente ao 74166 (Exercício 8.8.9).

8.8.10 Apresente um código contendo o sobrecarregamento de uma função, conforme ilustrado na Figura 8.8.3. Uma função deve realizar a soma entre dois valores do tipo inteiro, uma outra deve realizar a soma entre três operandos do tipo inteiro, e uma terceira deve realizar a soma entre operandos do tipo " BIT_VECTOR ". Nesta última, o valor da soma é acrescido de um.

8.8.11 Sintetize a descrição proposta para o Exercício 8.8.10, e comente os resultados.

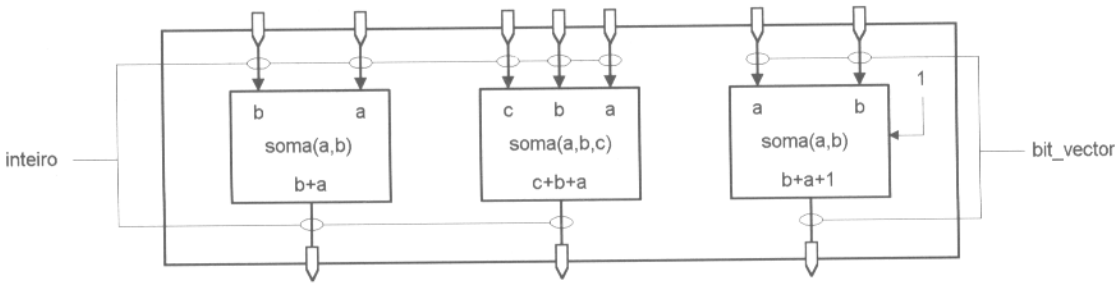


Figura 8.8.3 Descrição com sobrecarregamento da função denominada " soma ".

8.8.12 Proponha um subprograma que realiza a ordenação em um tipo " STRING ". O subprograma deve retornar com os elementos posicionados na forma crescente. O Quadro 8.8.3 ilustra o pseudocódigo de um algoritmo de ordenação simples.

```
-- dado: tipo string com limites de (1 a n)

 De i=2 Ate n Repita
   De j=1 Ate n-1 Repita
     Se dado(j) > dado(j+1)
       troca posicao entre os elementos j e j+1 de dado
     Fim Se
   Fim Repita
 Fim Repita
```

Quadro 8.8.3 Pseudocódigo para ordenação de elementos.

8.8.13 Proponha um subprograma que gera uma forma de onda quadrada com um período de 60 ns. Empregue o modo " INOUT " para o parâmetro formal, e verifique a operação do subprograma em regiões de código concorrente e seqüencial.

CAPÍTULO

9 BIBLIOTECAS E PACOTES

Da mesma forma que componentes e subprogramas, as bibliotecas e os pacotes provêem um auxílio para dividir e organizar um projeto.

As bibliotecas são o local de armazenamento de unidades de projeto compiladas. A linguagem permite ao usuário definir bibliotecas, possibilitando, desse modo, o armazenamento de unidades de projeto afins num mesmo local. As bibliotecas podem ser empregadas, também, como uma área de uso comum que possibilita o emprego de uma mesma descrição por diferentes programadores.

Um pacote permite reunir, entre outros, declarações de constantes, tipos e subprogramas de uso freqüente. Os elementos contidos em um pacote podem ser compartilhados por mais de uma descrição, permitindo a reutilização de um mesmo código em diversos projetos.

9.1 Bibliotecas de projeto

Conforme mencionado, as bibliotecas de projeto, ou *design libraries*, são o local de armazenamento das unidades de projeto compiladas. As bibliotecas são, normalmente, diretórios criados através da ferramenta empregada para compilação e simulação. A referência a uma biblioteca no interior de uma descrição é feita por nome lógico. Esse nome permite à ferramenta identificar o caminho completo do local físico onde a informação está armazenada.

A cláusula " LIBRARY " identifica as bibliotecas que podem ser referenciadas. Uma ou mais bibliotecas podem ser especificadas na lista dos nomes que seguem a palavra reservada (ver Quadro 9.1.1).

A biblioteca denominada " WORK " é a biblioteca padrão para o armazenamento das unidades de projeto, e a biblioteca " STD " armazena os pacotes predefinidos " STANDARD " e " TEXTIO ". Ambas podem ser referenciadas por qualquer unidade de projeto, sem a inclusão da cláusula " LIBRARY STD, WORK " (ver Figura 9.1.1).

```
LIBRARY nome_biblioteca;                -- permite a referencia a biblioteca

LIBRARY biblioteca_1, biblioteca_2;

LIBRARY STD, WORK;                      -- implicito em toda unidade de projeto (exceto pacote Standard)
```

Quadro 9.1.1 Cláusula " LIBRARY ".

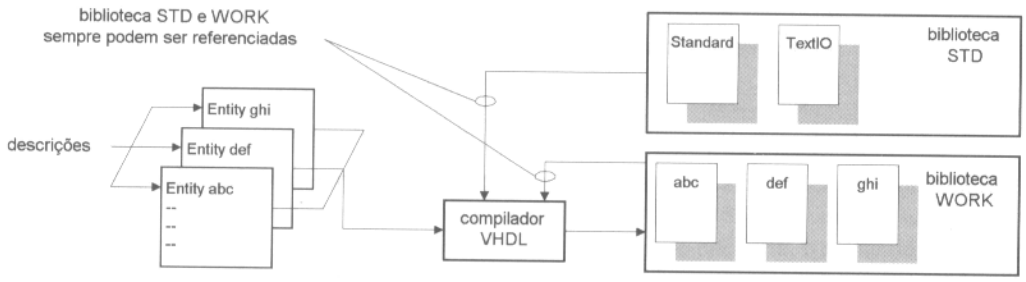


Figura 9.1.1 Representação em blocos das bibliotecas numa operação de compilação.

A Figura 9.1.2 ilustra um exemplo das operações envolvidas num projeto, onde duas descrições, " lb_and3 " e " lb_or3 ", são armazenadas na biblioteca denominada " bibli_1 ". Essas duas entidades são solicitadas por uma terceira entidade, " lb_uso ", na forma de componentes.

Inicialmente deve ser criada a biblioteca " bibli_1 " através de comandos específicos do aplicativo empregado. Em seguida, pode ser executada a compilação das entidades " lb_and3 " e " lb_or3 ", tendo-se o cuidado de definir o armazenamento dos resultados para a biblioteca " bibli_1 ". O local de armazenamento é, também, uma opção específica do aplicativo, definida antes da compilação de uma entidade ou pacote. Uma vez disponíveis os resultados na biblioteca " bibli_1 ", a compilação da entidade " lb_uso " pode ser realizada, desde que ela contenha a cláusula " LIBRARY " identificando o local de armazenamento das entidades " lb_and3 " e " lb_or3 ".

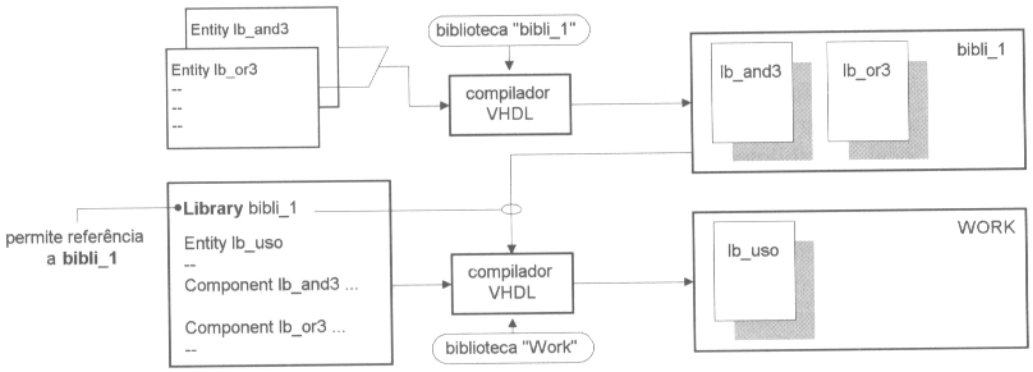


Figura 9.1.2 Operações envolvidas no exemplo proposto nos Quadros 9.1.2 e 9.1.3.

Nos Quadros 9.1.2 e 9.1.3 é apresentado o código do exemplo proposto na Figura 9.1.2. O primeiro quadro contém as entidades " lb_or3 " e " lb_and3 " que realizam operações lógicas " e " e " ou " entre três entradas. O segundo quadro descreve a entidade " lb_uso " que emprega as duas entidades na forma de componentes.

Supondo que o projeto é desenvolvido por um único projetista, a melhor opção seria executar a compilação das três entidades, direcionando o resultado para a biblioteca padrão, biblioteca " WORK ". Se as entidades " lb_and3 " e " lb_or3 " fossem desenvolvidas por times de projetos diferentes, seria conveniente redirecionar o resultado da compilação dessas entidades para uma biblioteca de uso comum, conforme apresentado. O mesmo raciocínio seria válido, no caso de um único projetista envolvido no projeto que desejasse organizar várias entidades numa biblioteca.

```
1  ENTITY lb_or3 IS
2    PORT (i0, i1, i2 :  IN  BIT;
3          s          :  OUT BIT);
4  END lb_or3;
5  ARCHITECTURE teste OF lb_or3 IS
6  BEGIN
7    s <= i0 OR i1 OR i2;
8  END teste;
```

```
1  ENTITY lb_and3 IS
2    PORT (i0, i1, i2 :  IN  BIT;
3          s          :  OUT BIT);
4  END lb_and3;
5  ARCHITECTURE teste OF lb_and3 IS
6  BEGIN
7    s <= i0 AND i1 AND i2;
8  END teste;
```

Quadro 9.1.2 Duas entidades de teste.

```
 1 LIBRARY bibli_1;   -- permite a referencia a biblioteca bibli_1
 2
 3 ENTITY lb_uso IS
 4   PORT (a, b, c :  IN  BIT;
 5         s_e, s_ou  :  OUT BIT);
 6 END lb_uso;
 7
 8 ARCHITECTURE teste OF lb_uso IS
 9   COMPONENT lb_and3  -- resultado da compilacao se encontra em bibli_1
10     PORT (i0, i1, i2 : IN  BIT; s  :  OUT BIT);
11   END COMPONENT;
12
13   COMPONENT lb_or3   -- resultado da compilacao se encontra em bibli_1
14     PORT (i0, i1, i2 : IN  BIT; s  :  OUT BIT);
15   END COMPONENT;
16 BEGIN
17   x1: lb_and3 PORT MAP(a, b, c, s_e);
18   x2: lb_or3  PORT MAP(a, b, c, s_ou);
19 END teste;
```

Quadro 9.1.3 Entidade empregando dois componentes de uma biblioteca definida pelo usuário.

9.2 Pacotes

Os pacotes provêem um local para o armazenamento de constantes, subprogramas, bem como declarações de tipos, sinais globais, e componentes. Um pacote pode ser composto de duas partes: declaração do pacote e corpo do pacote (opcional). Cada parte constitui uma unidade de projeto, ou *design unit* (ver comparação com uma entidade de projeto na Figura 9.2.1).

A declaração do pacote pode conter, por exemplo, a declaração de constantes, tipos, sinais, subprogramas e componentes (ver Quadro 9.2.1). Caso o pacote contenha funções ou procedimentos, a descrição desses subprogramas deve ser apresentada no corpo do pacote.

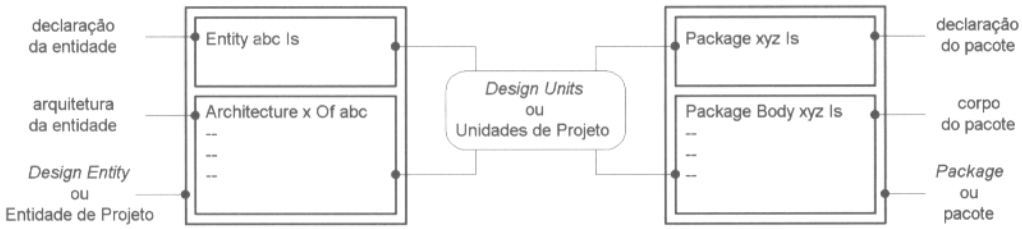


Figura 9.2.1 Exemplo de unidades de projeto.

```
PACKAGE nome_pacote IS
  --
  -- declaracao de:
  --      subprogramas, constantes, componentes, tipos, subtipos, sinais, variaveis compartilhadas
END nome_pacote;
```

```
PACKAGE BODY nome_pacote IS
-- declaracao de:
--      subprogramas, constantes, componentes, tipos, subtipos, sinais, variaveis compartilhadas
--
-- corpo de subprogramas
END nome_pacote;
```

Quadro 9.2.1 Declaração de um pacote e do seu corpo.

Como a declaração e o corpo de um pacote são uma unidade de projeto, eles podem ser compilados, e os itens se tornam visíveis para outras unidades funcionais através da cláusula " USE ". Nessa cláusula (ver parte superior do Quadro 9.2.2) são especificados a biblioteca em que o pacote se encontra, o nome do pacote e o item do pacote selecionado. Caso se deseje tornar todos os itens visíveis, pode-se empregar a palavra reservada " ALL ". Uma outra opção é a identificação direta ao item, conforme ilustrado na parte central e inferior do Quadro 9.2.2.

Nesses casos, os itens do pacote tornam-se visíveis, apresentando-se o caminho completo. A referência à biblioteca na identificação do item pode ser omitida, conforme ilustra a parte inferior do quadro.

```
LIBRARY nome_biblioteca;                    -- itens da biblioteca podem ser referenciados

USE nome_biblioteca.nome_pacote.ALL;        -- todos itens do pacote visiveis
USE nome_biblioteca.nome_pacote.item_abc    -- somente o item item_abc do pacote visivel
```

```
LIBRARY nome_biblioteca;                    -- itens da biblioteca podem ser referenciados

nome_biblioteca.nome_pacote.item_abc        -- identificacao direta ao item item_abc do pacote
```

```
LIBRARY nome_biblioteca;                    -- itens da biblioteca podem ser referenciados
USE nome_biblioteca.nome_pacote;            -- nao e' necessario identificar a biblioteca

nome_pacote.item_abc                        -- identificacao direta ao item item_abc do pacote
```

Quadro 9.2.2 Opções de referência de itens de um pacote.

Um exemplo de um pacote é apresentado no Quadro 9.2.3. Como apenas uma constante é declarada, não é necessário o corpo do pacote. No Quadro 9.2.4 é apresentado um outro pacote sem corpo. Ele contém apenas a declaração de dois componentes, " lb_and3 " e " lb_or3 ". Não é permitida a inclusão da descrição de um componente no corpo de um pacote. No exemplo, os componentes correspondem às entidades descritas no Quadro 9.1.2.

```
1 PACKAGE pack_0 IS          -- declaracao do pacote
2   CONSTANT x1 : INTEGER := 7;
3 END pack_0;
```

Quadro 9.2.3 Exemplo de um pacote contendo unicamente a sua declaração.

```
1 PACKAGE pk_tes1 IS        -- declaracao do pacote
2   COMPONENT lb_and3 PORT (i0, i1, i2 : IN  BIT; s : OUT BIT);
3   END COMPONENT;
4
5   COMPONENT lb_or3 PORT (i0, i1, i2 : IN  BIT; s : OUT BIT);
6   END COMPONENT;
7 END pk_tes1;
```

Quadro 9.2.4 Exemplo de um pacote contendo a declaração de dois componentes.

A Figura 9.2.2 ilustra um exemplo das operações envolvidas na compilação de um pacote, " pk_tes1 ", que contém a declaração de dois componentes. Tanto o corpo do pacote, quanto a descrição das entidades que são solicitadas na forma de componentes, têm a sua compilação direcionada para a biblioteca denominada " bibli_1 ". Supondo a biblioteca criada e as entidades " lb_and3 " e " lb_or3 " compiladas, é necessário compilar a declaração do pacote " pk_tes1 ". No exemplo, o resultado da compilação do pacote é armazenado na biblioteca " bibli_1 ". Uma vez compilado o pacote, é possível compilar a entidade " pk_uso1 ". Ela não necessita da declaração dos componentes, pois essa declaração está contida no pacote " pk_tes1 ".

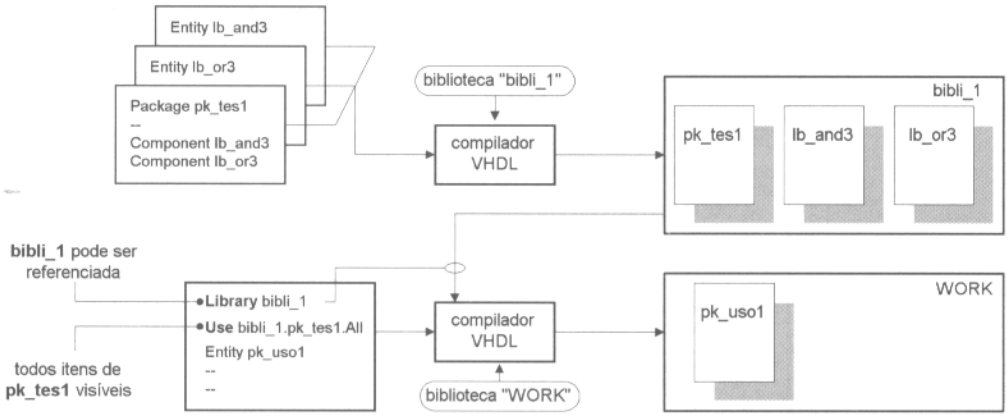


Figura 9.2.2 Exemplo das operações envolvidas nas descrições dos Quadros 9.2.3 e 9.2.4.

O Quadro 9.2.5 contém o código da entidade " pk_uso1 " empregada no exemplo da Figura 9.2.2. A linha 1 permite a referência à biblioteca " bibli_1 ", e a linha 2 torna visível todos os itens do pacote " pk_tes1 " para a entidade. Note que, de acordo com a descrição, o resultado da compilação do pacote " pk_tes1 " deve estar armazenado na biblioteca " bibli_1 ". O emprego do pacote na forma proposta evita a repetição das declarações dos componentes por todas as entidades que os utilizam. Uma comparação entre a entidade " lb_uso " (Quadro 9.1.3) e a entidade " pk_uso1 " (Quadro 9.2.5) evidencia essa facilidade. O pacote " pk_tes1 " é descrito no Quadro 9.2.4, e as entidades " lb_and3 " e " lb_or3 " são descritas no Quadro 9.1.2.

```
 1  LIBRARY bibli_1;            -- permite a referencia a biblioteca bibli_1
 2  USE bibli_1.pk_tes1.ALL;    -- torna todos itens do pacote visiveis
 3
 4  ENTITY pk_uso1 IS
 5    PORT (a, b, c :  IN  BIT;
 6          s_e, s_ou  :  OUT BIT);
 7  END pk_uso1;
 8
 9  ARCHITECTURE teste OF pk_uso1 IS
10  BEGIN
11    x1: lb_and3 PORT MAP(a, b, c, s_e);
12    x2: lb_or3  PORT MAP(a, b, c, s_ou);
13  END teste;
```

Quadro 9.2.5 Descrição empregando o pacote do Quadro 9.4.2.

O Quadro 9.2.6 propõe duas opções para tornar visível um item de um pacote, através da identificação do caminho completo. Em ambas as opções, a cláusula " LIBRARY " na linha 1 permite a referência à biblioteca " bibli_1 ", onde está armazenado o resultado da compilação do pacote " pk_tes1 ", e das entidades " lb_and3 " e " lb_or3 ". Na primeira descrição, a cláusula " USE " não é empregada, necessitando da identificação da biblioteca para tornar o item visível (ver linhas 10 e 11). Na segunda, a cláusula " USE " na linha 2 permite omitir o nome da biblioteca no item referenciado (ver linhas 11 e 12).

```
 1  LIBRARY bibli_1;        -- permite a referencia a biblioteca bibli_1
 2
 3  ENTITY pk_uso2 IS
 4    PORT (a, b, c :  IN  BIT;
 5          s_e, s_ou  :  OUT BIT);
 6  END pk_uso2;
 7
 8  ARCHITECTURE teste OF pk_uso2 IS
 9  BEGIN
10    x1: bibli_1.pk_tes1.lb_and3 PORT MAP(a, b, c, s_e);
11    x2: bibli_1.pk_tes1.lb_or3  PORT MAP(a, b, c, s_ou);
12  END teste;
```

```
 1  LIBRARY bibli_1;          -- permite a referencia a biblioteca bibli_1
 2  USE bibli_1.pk_tes1;      -- elimina a necessidade de identificar o nome da biblioteca
 3
 4  ENTITY pk_uso3 IS
 5    PORT (a, b, c :  IN  BIT;
 6          s_e, s_ou  :  OUT BIT);
 7  END pk_uso3;
 8
 9  ARCHITECTURE teste OF pk_uso3 IS
10  BEGIN
11    x1: pk_tes1.lb_and3 PORT MAP(a, b, c, s_e);  -- nome da biblioteca omitido
12    x2: pk_tes1.lb_or3  PORT MAP(a, b, c, s_ou); -- nome da biblioteca omitido
13  END teste;
```

Quadro 9.2.6 Descrição empregando o pacote do Quadro 9.4.2 - sem a cláusula " USE ".

A Figura 9.2.3 ilustra um outro conjunto de operações, no caso de um pacote contendo duas partes: declaração e corpo. Nesse caso, o comportamento dos componentes empregados nos exemplos anteriores, " lb_and3 " e " lb_or3 ", é descrito na forma de duas funções. Supondo a biblioteca " bibli_1 " criada, deve-se compilar tanto a declaração quanto o corpo do pacote. Uma vez compilado o pacote, é possível compilar a entidade " pk_uso ".

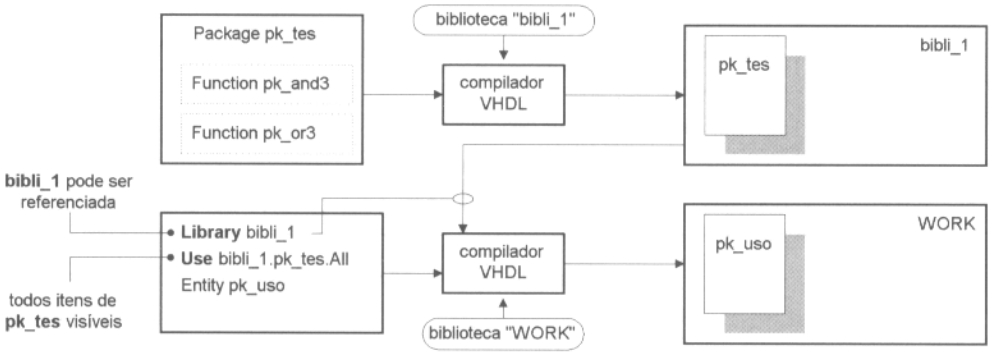


Figura 9.2.3 Exemplo das operações envolvidas nas descrições dos Quadros 9.2.4 e 9.2.5.

O Quadro 9.2.7 apresenta a declaração e o corpo do pacote " pk_tes ", e o Quadro 9.2.8 contém a entidade " pk_uso " que emprega o pacote. A linha 1 do código do Quadro 9.2.8 permite a referência à biblioteca " bibli_1 " que contém o pacote compilado, e a linha 2 torna todos os itens visíveis para a descrição; no caso, as funções " pk_and3 " e " pk_or3 ". Se o resultado da compilação do pacote estivesse armazenado na biblioteca de trabalho " work ", a linha 1 do código do Quadro 9.2.8 poderia ser removida, e a linha 2 deveria ser alterada para " USE work.pk_tes.ALL ".

```
1  PACKAGE pk_tes IS        -- declaracao do pacote
2    FUNCTION pk_and3 (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT;
3    FUNCTION pk_or3  (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT;
4  END pk_tes;
5
6  PACKAGE BODY pk_tes IS  -- corpo do pacote
7    FUNCTION pk_and3 (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT IS
8    BEGIN
9      RETURN (i0 AND i1 AND i2);
10   END pk_and3;
11
12   FUNCTION pk_or3  (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT IS
13   BEGIN
14     RETURN (i0 OR i1 OR i2);
15   END pk_or3;
16 END pk_tes;
```

Quadro 9.2.7 Exemplo de um pacote contendo a declaração e corpo.

```
1  LIBRARY bibli_1;         -- permite a referencia a biblioteca bibli_1
2  USE bibli_1.pk_tes.ALL; -- torna todos itens do pacote visiveis
3
4  ENTITY pk_uso IS
5    PORT (a, b, c    :  IN  BIT;
6          s_e, s_ou :  OUT BIT);
7  END pk_uso;
8
9  ARCHITECTURE teste OF pk_uso IS
10 BEGIN
11   s_e  <= pk_and3(a, b, c);
12   s_ou <= pk_or3(a, b, c);
13 END teste;
```

Quadro 9.2.8 Exemplo de utilização de um pacote.

Ferramentas de síntese

O comportamento das ferramentas de síntese com relação a pacotes e bibliotecas não é padronizado. A Tabela 9.2.1 relaciona as bibliotecas e pacotes padronizados. Com exceção do pacote " TEXTIO ", todos os pacotes relacionados na Tabela 9.2.1 são, normalmente, suportados pelas ferramentas de síntese. Bibliotecas e pacotes definidos pelo usuário costumam, também, ser suportados, observando restrições particulares de cada ferramenta.

Tabela 9.2.1 Bibliotecas e respectivos pacotes armazenados.

Biblioteca	Pacotes na biblioteca
work [1]	-
std [1]	standard [2] textio [3]
ieee [3]	std_logic_1164 [3] std_logic_unsigned [3] std_logic_signed [3] std_logic_arith [3]
[1] Não necessitam da cláusula " Library ".	
[2] Não necessita da cláusula " Use ".	
[3] Biblioteca e pacotes abordados nos Capítulos 10, 11 e 13.	

9.3 Ordem de análise

Usualmente, um arquivo contém uma entidade composta da declaração e arquitetura, ou um pacote composto da declaração e corpo. Essa organização não é obrigatória. Um arquivo pode conter, por exemplo, a descrição completa de mais de uma entidade ou somente a declaração de um pacote. Conforme será apresentado a seguir,

arquivos mantendo em conjunto a declaração e a arquitetura de uma entidade, ou a declaração e o corpo de um pacote, não são a maneira mais vantajosa para a condução de um projeto mais complexo.

A compilação da declaração de uma entidade pode ser realizada separadamente da compilação da sua arquitetura. O mesmo se aplica com relação a declaração e corpo de um pacote. A seqüência das operações de compilação deve observar a dependência entre as unidades compiladas:

- a declaração de um pacote deve ser compilada antes da unidade de projeto que o referencia;
- a declaração de uma entidade deve ser compilada antes da unidade de projeto que o referencia;
- o corpo de um pacote e a arquitetura de uma entidade podem ser compilados após a compilação de suas respectivas declarações. Essas declarações somente são necessárias no momento da simulação.

A Figura 9.3.1 ilustra a ordem de compilação quando ocorre a dependência entre unidades de projeto. No exemplo, a arquitetura " k " da entidade " jkl " solicita o componente " ghi ". A entidade " ghi ", por sua vez, referencia o pacote " def ", que referencia o pacote " abc ". Dadas as dependências, a declaração do pacote " abc " deve ser compilada inicialmente, seguida das compilações da declaração do pacote " def ", da declaração da entidade " ghi ", e da declaração da entidade " jkl ".

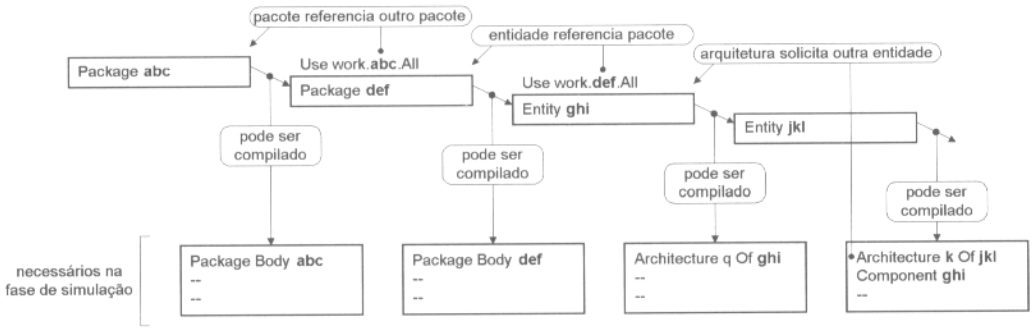


Figura 9.3.1 Exemplo de uma seqüência de compilações.

A Figura 9.3.2 ilustra como uma organização de arquivos separando as unidades de projeto pode ser vantajosa. Caso a declaração da entidade " xyz " seja alterada e recompilada, todas as arquiteturas dessa entidade (caso ela tenha mais de uma arquitetura) e todas as arquiteturas afetadas devem ser recompiladas. Por outro lado, uma alteração na arquitetura da entidade " xyz " necessita de que apenas ela seja recompilada. (Ver Exercício 9.7.4, como exemplo.)

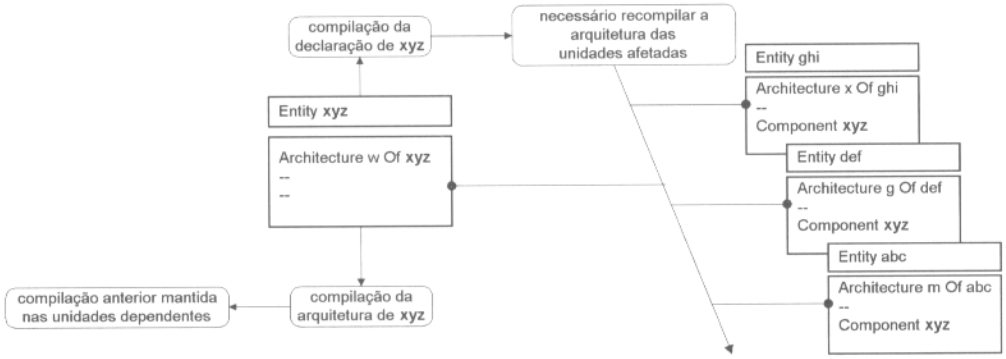


Figura 9.3.2 Opções de recompilação de uma entidade.

Se a declaração de um pacote for alterada e recompilada, todas as unidades de projeto afetadas devem ser recompiladas (ver Figura 9.3.3). Nesse exemplo, a alteração e compilação da declaração do pacote " wkz " resultam em que a declaração e arquitetura das entidades, e declaração e corpo dos pacotes afetados devem ser

submetidos a uma operação de compilação. Se apenas o corpo do pacote for alterado, unicamente ele deve ser recompilado. O Exercício 9.7.5 aborda esse caso.

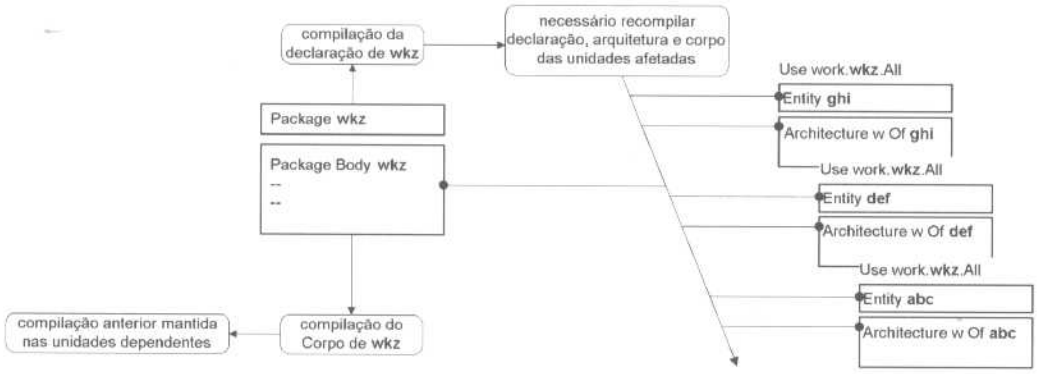


Figura 9.3.3 Opções de recompilação em pacotes.

A alteração da declaração da entidade ou de um pacote indica que uma mudança significativa foi executada. Essa alteração repercute nas unidades de projeto dependentes, levando à necessidade de recompilá-las. A alteração no corpo de um pacote, ou da arquitetura de uma entidade, não modifica a relação entre as declarações das unidades, não sendo necessária, portanto, a recompilação das unidades dependentes. Assim, o armazenamento das unidades de projeto em arquivos diferentes pode aliviar as operações de compilação, quando somente o corpo de um pacote ou a arquitetura de uma entidade forem alterados.

O Quadro 9.3.1 ilustra um arquivo contendo duas entidades. A apresentação nessa forma mostra que um arquivo pode conter várias unidades de projeto, e identifica a seqüência de compilação, considerando que o compilador começa a análise do código a partir do início do arquivo. Neste exemplo, as declarações das entidades são compiladas sem as respectivas arquiteturas. Observe que uma arquitetura somente pode ser compilada após a compilação de sua declaração.

```
1  -- arquivo dep_ver0.vhd: duas entidades
2  ENTITY or3 IS
3    PORT (i0, i1, i2 : IN BIT; s : OUT BIT);
4  END or3;
5
6  ENTITY and3 IS
7    PORT (i0, i1, i2 : IN BIT; s : OUT BIT);
8  END and3;
9
10 ARCHITECTURE teste OF or3 IS
11   BEGIN s <= i0 OR i1 OR i2;
12 END teste;
13
14 ARCHITECTURE teste OF and3 IS
15   BEGIN s <= i0 AND i1 AND i2;
16 END teste;
```

Quadro 9.3.1 Seqüência de compilação: declaração e arquitetura.

O Quadro 9.3.2 contém o código de um pacote, " pk_1 ", e o código de uma entidade, " pk_uso4 ". Como a entidade emprega o pacote " pk_1 ", a declaração do pacote deve ser compilada antes da compilação da declaração da entidade. Note que a compilação do corpo do pacote é a última etapa executada. Isso ilustra que o corpo de um pacote pode até mesmo não existir, caso não seja efetuada a simulação ou síntese da entidade. Naturalmente, essa não é a ordem usual; o corpo de um pacote é normalmente compilado após a compilação da declaração.

```
1  -- arquivo dep_ver1.vhd
2  PACKAGE pk_1 IS     -- declaracao do pacote
3    FUNCTION and3 (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT;
4  END pk_1;
5
6  USE WORK.pk_1.ALL; -- torna todos itens do pacote pk_1 visiveis
7  ENTITY pk_uso4 IS
8    PORT (a, b, c : IN  BIT; s_e : OUT BIT);
9  END pk_uso4;
10
11 ARCHITECTURE teste OF pk_uso4 IS
12 BEGIN
13   s_e  <= and3(a, b, c);
14 END teste;
15
16 PACKAGE BODY pk_1 IS  -- corpo do pacote necessario para simulacao
17   FUNCTION and3 (SIGNAL i0, i1, i2 : IN  BIT) RETURN BIT IS
18   BEGIN RETURN (i0 AND i1 AND i2);
19   END and3;
20 END pk_1;
```

Quadro 9.3.2 Explorando possíveis ordens de compilação: entidade empregando um pacote.

No Quadro 9.3.3 são apresentados um pacote, " pk_2 ", e duas entidades, " pk_uso5 " e " pk_uso6 ". Como ambas as entidades dependem do pacote " pk_2 ", a compilação da declaração dessas entidades deve ser realizada após a compilação da declaração do pacote. Note, ainda, que, em um arquivo contendo mais de uma unidade de projeto, cada cláusula " USE " se aplica à unidade de projeto que a segue. Por exemplo, a remoção da linha 11 causaria um erro de compilação, pois os itens do pacote " pk_2 " não seriam visíveis para " pk_uso6 ".

```
1  -- arquivo dep_ver2.vhd
2  PACKAGE pk_2 IS     -- declaracao do pacote
3    FUNCTION and2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT;
4  END pk_2;
5
6  USE WORK.pk_2.ALL; -- torna itens do pacote pk_2 visiveis para pk_uso5
7  ENTITY pk_uso5 IS
8    PORT (a, b : IN  BIT; s_e5 : OUT BIT);
9  END pk_uso5;
10
11 USE WORK.pk_2.ALL; -- torna itens do pacote pk_2 visiveis para pk_uso6
12 ENTITY pk_uso6 IS
13   PORT (a, b : IN  BIT; s_e6 : OUT BIT);
14 END pk_uso6;
15
16 ARCHITECTURE teste OF pk_uso5 IS
17 BEGIN  s_e5 <= and2(a, b);
18 END teste;
19
20 ARCHITECTURE teste OF pk_uso6 IS
21 BEGIN  s_e6 <= and2(a, b);
22 END teste;
23
24 PACKAGE BODY pk_2 IS  -- corpo do pacote pk_1
25   FUNCTION and2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT IS
26   BEGIN RETURN (i0 AND i1);
27   END and2;
28 END pk_2;
```

Quadro 9.3.3 Explorando possíveis ordens de compilação: entidades empregando um pacote.

O Quadro 9.3.4 contém um arquivo mais longo, que ilustra um caso de dependência entre pacotes. O pacote " pk_3b " emprega a função " and2 " do pacote " pk_3a " (ver linha 23). Devido à dependência, a compilação da declaração do pacote " pk_3a " deve ser realizada primeiro. Somente após essa compilação, a compilação da declaração do pacote " pk_3b " pode ser realizada. A declaração da entidade " pk_uso8 ", por sua vez, somente pode ser compilada após a compilação da declaração do pacote " pk_3b ", pois ela emprega este último pacote. Observe que a ordem na compilação do corpo dos pacotes não é importante. O corpo do pacote " pk_3b " (linhas 21 a 25) pode ser compilado primeiro, apesar de referenciar um elemento do pacote " pk_3a " (ver linhas 27 a 31). Isso porque o corpo do pacote somente será necessário na fase de simulação ou síntese.

```
1  -- arquivo dep_ver3.vhd
2  PACKAGE pk_3a IS     -- declaracao do pacote
3    FUNCTION and2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT;
4  END pk_3a;
5
6  USE WORK.pk_3a.ALL; -- torna todos itens do pacote pk_3a visiveis para pk_3b
7  PACKAGE pk_3b IS     -- declaracao do pacote
8    FUNCTION and4 (SIGNAL i0, i1, i2, i3 : IN  BIT) RETURN BIT;
9  END pk_3b;
10
11 USE WORK.pk_3b.ALL; -- torna itens do pacote pk_3b visiveis para pk_uso8
12 ENTITY pk_uso8 IS
13   PORT (a, b, c, d : IN  BIT; s_e : OUT BIT);
14 END pk_uso8;
15
16 ARCHITECTURE teste OF pk_uso8 IS
17 BEGIN
18   s_e  <= and4(a, b, c, d);
19 END teste;
20
21 PACKAGE BODY pk_3b IS  -- corpo do pacote pk_3b
22   FUNCTION and4 (SIGNAL i0, i1, i2, i3 : IN  BIT) RETURN BIT IS
23   BEGIN RETURN and2(i0, i1) AND and2(i2, i3);
24   END and4;
25 END pk_3b;
26
27 PACKAGE BODY pk_3a IS  -- corpo do pacote pk_3a
28   FUNCTION and2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT IS
29   BEGIN RETURN (i0 AND i1);
30   END and2;
31 END pk_3a;
```

Quadro 9.3.4 Explorando possíveis ordens de compilação: pacote empregando um pacote.

A Figura 9.3.4 ilustra o resultado da síntese, e o Quadro 9.3.5 as mensagens geradas pela ferramenta durante a fase de leitura do código apresentado no Quadro 9.3.4. Conforme se pode observar, a descrição foi corretamente sintetizada. As declarações e a arquitetura foram armazenadas na biblioteca " work " da ferramenta, na mesma seqüência do código. Uma alteração na seqüência apresentada, que não respeite a dependência entre as unidades, acarretaria uma mensagem de erro na fase de compilação.

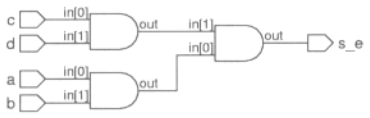


Figura 9.3.4 Circuito sintetizado a partir do código do Quadro 9.3.4.

```
-- Loading package standard into library std
-- Reading vhdl file /vhdl/dep_ver3.vhd into library work
-- Loading package pk_3a into library work
-- Loading package pk_3b into library work
-- Loading entity pk_uso8 into library work
-- Loading architecture teste of pk_uso8 into library work
-- Compiling root entity pk_uso8(teste)
-- Pre Optimizing Design .work.pk_uso8.teste
-- Boundary optimization.
Info: Finished reading design
```

Quadro 9.3.5 Mensagens da ferramenta na operação de leitura do código.

9.4 Constantes deferidas

O valor de uma constante deferida, ou *deferred constant*, é definido no corpo de um pacote. Na declaração do pacote constam apenas o tipo e o nome da constante. O emprego de uma constante deferida é interessante, considerando as operações necessárias na recompilação de unidades de projeto. Durante o processo de elaboração de um código, a alteração do valor de uma constante definida na declaração de um pacote acarreta a necessidade de recompilação de todas as unidades de projeto dependentes. Empregando-se uma constante deferida, como o valor dessa constante está no corpo do pacote, a mudança de seu valor necessita, apenas, da recompilação do corpo do pacote.

O Quadro 9.4.1 contém um exemplo do emprego de uma constante deferida. Na declaração do pacote (linhas 2 a 4), apenas o nome e tipo da constante são definidos. No corpo do pacote, a declaração da constante é repetida, agora, com a definição do seu valor. Deve-se observar que constantes deferidas nem sempre são suportadas pelas ferramentas de síntese.

```
1  -- arquivo dep_ver6.vhd
2  PACKAGE pk_6 IS
3    CONSTANT mascara_glb : BIT_VECTOR; -- declaracao da constante
4  END pk_6;
5
6  PACKAGE BODY pk_6 IS                  -- valor da constante definido no corpo do pacote
7   CONSTANT mascara_glb : BIT_VECTOR(2 DOWNTO 0) := "101";
8  END pk_6;
9
10 USE WORK.pk_6.ALL;
11 ENTITY pk_uso11 IS
12   PORT (x       : IN BIT_VECTOR(3 DOWNTO 0);
13         gl2,gl1,gl0,lc2,lc1,lc0 : OUT BIT);
14 END pk_uso11;
15
16 ARCHITECTURE teste OF pk_uso11 IS
17   CONSTANT mascara_lcl : BIT_VECTOR(2 DOWNTO 0) := "101";
18 BEGIN
19   gl2 <= mascara_glb(2) XOR x(2);
20   gl1 <= mascara_glb(1) XOR x(1);
21   gl0 <= mascara_glb(0) XOR x(0);
22   lc2 <= mascara_lcl(2) XOR x(2);
23   lc1 <= mascara_lcl(1) XOR x(1);
24   lc0 <= mascara_lcl(0) XOR x(0);
25 END teste;
```

Quadro 9.4.1 Exemplo de uma constante deferida — *deferred constant*.

A Figura 9.4.1 ilustra o resultado da síntese da entidade " pk_uso11 ". Conforme se pode observar, a constante deferida " mascara_glb " foi corretamente empregada pela ferramenta de síntese. As portas de saída " gl2 ", " gl1 " e " gl0 " estão interligadas às portas de saída " lc2 ", " lc1 " e " lc0 ", respectivamente. Note que a ferramenta de síntese minimizou o circuito; nas operações de "ou-exclusivo" com o valor " 1 " foi inserido um inversor; e nas operações com o valor " 0 " as portas de entrada e saída foram interligadas.

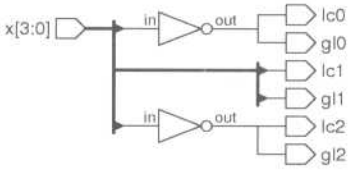


Figura 9.4.1 Circuito sintetizado corretamente a partir do código do Quadro 9.4.1.

9.5 Sinais globais

A região de declaração de um pacote pode conter a declaração de sinais globais. Esses sinais provêem um mecanismo de comunicação entre entidades solicitadas na forma de componentes. Entretanto, sinais globais não são ainda suportados pelas ferramentas de síntese, limitando a sua aplicação somente em simulações.

O Quadro 9.5.1 ilustra a utilização de um sinal global. Duas entidades, " pk_uso9 " e " pk_uso10 ", se comunicam através do sinal global " glb " declarado no pacote " pk_4 " (ver linha 3). O dado na porta " a " de " pk_uso9 " é transferido para o sinal " glb " (ver linha 10), e o dado presente em " glb " é transferido para a interface de saída " b " de " pk_10 " (ver linha 17). A entidade " pk_uso11 " solicita as outras duas entidades na forma de componentes. O dado da porta " x " é transferido para o componente " x1 ". Devido à comunicação entre os componentes, via sinal global " glb ", o valor de " x " é transferido para a porta " y ".

```
 1  -- arquivo dep_ver4.vhd
 2  PACKAGE pk_4 IS
 3    SIGNAL glb : BIT; -- sinal global definido
 4  END pk_4;
 5
 6  USE WORK.pk_4.ALL;  -- dado tranferido para o sinal global glb
 7  ENTITY pk_uso9  IS PORT (a : IN  BIT);
 8  END pk_uso9;
 9  ARCHITECTURE teste OF pk_uso9 IS
10    BEGIN  glb <= a;
11  END teste;
12
13  USE WORK.pk_4.ALL;  -- dado lido do sinal global glb
14  ENTITY pk_uso10 IS PORT (b : OUT BIT);
15  END pk_uso10;
16  ARCHITECTURE teste OF pk_uso10 IS
17    BEGIN  b <= glb;
18  END teste;
19
20  -- entidade pk_uso11: componentes se comunicam via sinal global glb
21  ENTITY pk_uso11 IS PORT (x : IN BIT; y : OUT BIT);
22  END pk_uso11;
23  ARCHITECTURE teste OF pk_uso11 IS
24    COMPONENT pk_uso9  PORT (a : IN  BIT); END COMPONENT;
25    COMPONENT pk_uso10 PORT (b : OUT BIT); END COMPONENT;
26  BEGIN
27    x1: pk_uso9  PORT MAP(x);
28    x2: pk_uso10 PORT MAP(y);
29  END teste;
```

Quadro 9.5.1 Exemplo de um sinal global.

O Quadro 9.5.2 apresenta as mensagens geradas pela ferramenta na leitura do código do Quadro 9.5.1. O sinal global " glb ", declarado na linha 3, foi detectado e acarretou uma mensagem de alerta, relatando que sinais globais não são suportados. Isso resultou em um erro na linha 10, pois a declaração do sinal " glb " foi ignorada pelo compilador da ferramenta.

```
-- Loading package standard into library std
-- Reading vhdl file /vhdl/dep_ver4.vhd into library work
-- Loading package pk_4 into library work
"/vhdl/dep_ver4.vhd",line 3: Warning, signal glb : global signals are not yet supported.
-- Loading entity pk_uso9 into library work
"/vhdl/dep_ver4.vhd",line 10: Error, identifier glb is not declared.
Error in file /vhdl/dep_ver4.vhd.
-- Error found in VHDL source
```

Quadro 9.5.2 Mensagens da ferramenta na operação de leitura do código.

9.6 Principais pontos abordados

- Unidades de projeto (declaração e arquitetura de entidades, declaração e corpo de pacotes) compiladas são armazenadas em bibliotecas de projeto ou *design libraries*.
- As unidades de projeto contidas numa biblioteca podem ser referenciadas por uma descrição através da cláusula " LIBRARY ", seguida do nome lógico da biblioteca.
- As bibliotecas " WORK " e " STD " podem ser referenciadas por todas as descrições.
- Os pacotes, ou *packages*, são empregados como uma área comum para o armazenamento de constantes, declarações de tipos, subprogramas e declaração de componentes.
- Um pacote consiste em duas partes: a declaração do pacote e o corpo do pacote (opcional). A primeira parte pode conter a declaração de constantes, tipos, sinais globais, funções, procedimentos e componentes. No corpo do pacote são inseridos, também, descrições de subprogramas e valores de constantes deferidas.
- Os itens contidos em um pacote se tornam visíveis para outras unidades de projeto através da cláusula " USE ". Dois formatos são empregados: " USE nome_biblioteca.nome_pacote.ALL " (todos os itens do pacote são visíveis) e " USE nome_biblioteca.nome_pacote.item_abc " (somente " item_abc " do pacote é visível).
- A seqüência das operações de compilação deve observar a dependência entre as unidades compiladas. A declaração de uma entidade deve ser compilada antes da sua arquitetura, e a declaração de um pacote deve ser compilada antes da compilação do seu corpo.

- Constantes deferidas, ou *deferred constants*, têm o seu valor definido no corpo de um pacote. Na declaração do pacote, apenas o tipo e o nome da constante são incluídos.
- Sinais globais podem ser definidos em pacotes como um meio de comunicação entre entidades.

9.7 Exercícios

9.7.1 Desenvolva um pacote contendo subprogramas descrevendo as operações realizadas pelos blocos apresentados na Figura 9.7.1. O bloco " x " contém um somador completo e um circuito de seleção controlado pelo sinal " qi ". Na condição " qi=1 " o resultado da soma é transferido para a saída " ri+1 ", na condição " qi=0 " é transferido o sinal " ri ". O bloco " y " contém apenas o circuito somador. Todos os sinais são do tipo " bit ". O local de armazenamento do pacote compilado deve ser numa nova biblioteca criada especialmente para esse pacote, denominada " dv_celulas ".

Figura 9.7.1 Células empregadas nos Exercícios 9.7.1, 9.7.2 e 9.7.3.

9.7.2 O algoritmo "divisão com restauração", para dois números na base dez, pode ser compreendido pela seqüência descrita na Figura 9.7.2. O processo consiste em subtrair o resto parcial até que o resultado atinja um valor negativo. Quando o valor negativo for encontrado, significa que a operação de subtração foi realizada uma vez além do necessário, e o valor parcial anterior deve ser restaurado.

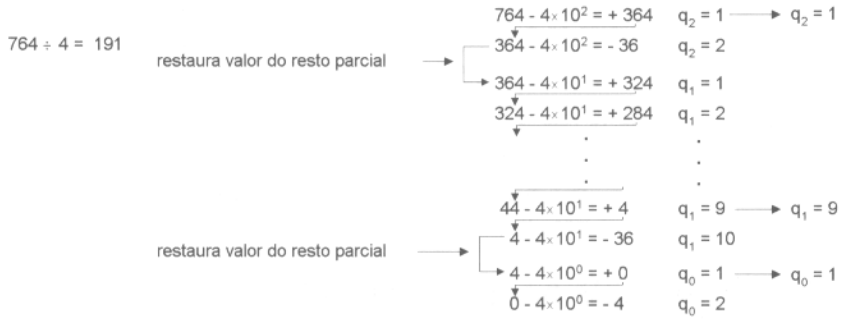


Figura 9.7.2 Ilustração do algoritmo de divisão com restauração — base 10.

Na base dois, o dígito do cociente pode assumir o valor zero ou um. Caso a operação de subtração resulte num valor maior ou igual a zero, o dígito do cociente é igual a " 1 ", e o resto parcial corresponde ao resultado da operação de subtração. Caso a subtração resulte num valor negativo, o dígito do cociente é igual a " 0 ", e o resto parcial anterior é restaurado transformando-se no novo resto parcial (ver exemplo da Figura 9.7.3).

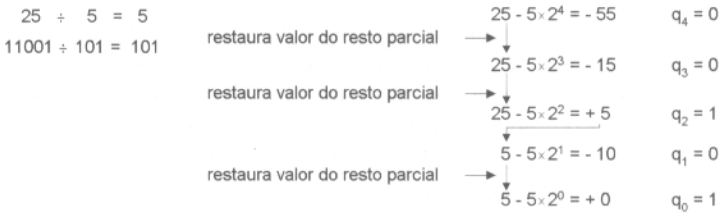


Figura 9.7.3 Algoritmo de divisão com restauração — base 2.

A divisão com restauração pode ser implementada por células que contenham um subtrator completo e um circuito de seleção, denominadas "subtrator controlado". A saída da célula é interligada ao circuito de seleção, o qual transfere o resultado parcial da subtração ou o dígito do minuendo, conforme um sinal de controle externo. Esse controle corresponde ao último bit da cadeia de subtratores, e sinaliza se o resultado da operação de subtração foi negativo ou não.

A Figura 9.7.4 ilustra uma implementação de um divisor por restauração onde a entrada " a " é o dividendo, e a entrada " b " é o divisor, ambas com 4 bits. A operação de subtração na célula do subtrator controlado é implementada por um circuito somador completo, conforme descrito no Exercício 9.7.1. Desse modo, para realização da subtração é necessário complementarmos os bits correspondentes ao divisor e impormos um nível alto na entrada " ci " da primeira célula.

Apresente a descrição de um divisor com " a " contendo 5 bits e " b " contendo 4 bits. As células devem estar armazenadas no pacote desenvolvido no Exercício 9.7.1.

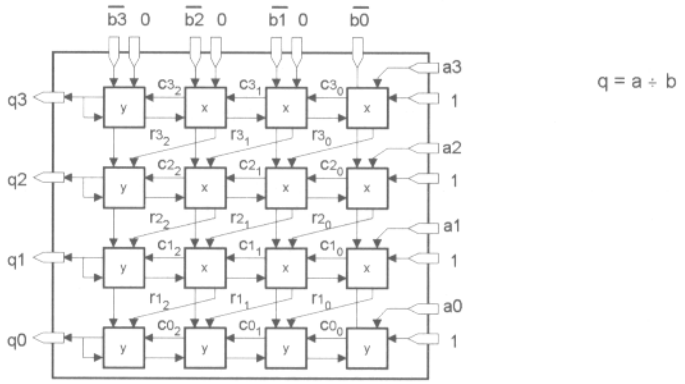


Figura 9.7.4 Divisor do tipo com restauração na forma de uma rede de células.

9.7.3 Considerando o algoritmo de divisão apresentado no Exercício 9.7.2, desenvolva uma descrição para o divisor proposto na Figura 9.7.5. Nessa nova implementação, apenas uma linha de células do tipo subtrator controlado é implementada. A divisão é executada seqüencialmente com o auxílio de dois registradores de deslocamento. A cada ciclo de relógio, um bit do cociente é armazenado, sinal " ds ", e um novo bit do dividendo é inserido na primeira célula da linha, sinal " qp4 ". O número de ciclos de relógio necessário para o processo de divisão é função do número de bits do dividendo, entrada " a ".

Insira na implementação uma unidade de controle para o divisor. Essa unidade, detectando o sinal " ld=1 ", transfere o valor da entrada " a " para o registrador de deslocamento " qp " e inicia o processo de divisão. Uma vez atingido o número de ciclos de relógio para o término da operação, o valor presente no registrador " qs " é transferido para o registrador de saída, mantendo o valor estável. Proponha um novo pacote contendo as operações dos registradores na forma de subprogramas. Tenha cuidado com a descrição dos registradores em subprogramas; lembre-se de que variáveis declaradas em funções ou procedimentos são inicializadas a cada chamada do subprograma.

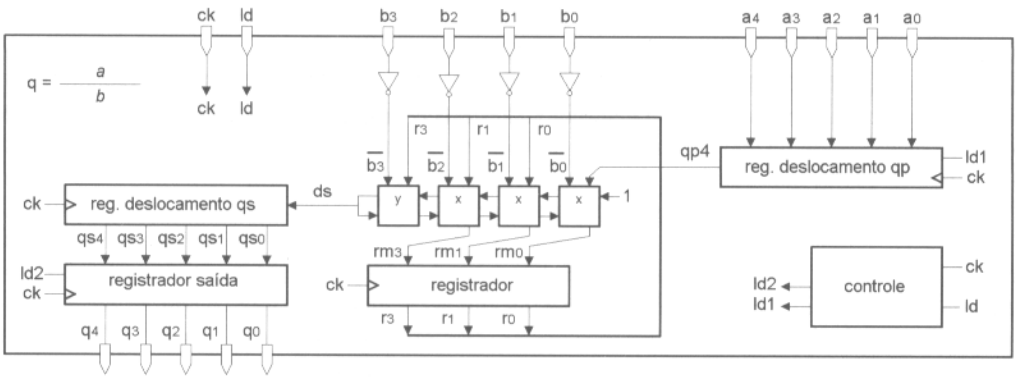


Figura 9.7.5 Divisor do tipo com restauração seqüencial.

9.7.4 O objetivo do exercício é verificar a ordem na análise da compilação e os reflexos que a alteração de uma unidade pode acarretar nas unidades que a referenciam. Considerando o Quadro 9.7.1, altere a declaração da entidade " and2 " mudando, por exemplo, o nome das portas. Recompile somente a declaração e tente simular a entidade " and3 ". Com base na mensagem de erro, proceda com as alterações necessárias para que a simulação possa ser executada com sucesso.

```
1  -- arquivo dep_err0.vhd: duas entidades
2  -- VHDL-1993
3  ENTITY and2 IS
4    PORT (i0, i1: IN BIT; s : OUT BIT);
5  END and2;
6
7  ENTITY and3 IS
8    PORT (i0, i1, i2 : IN BIT; s : OUT BIT);
9  END and3;
10
11 ARCHITECTURE arq_e2 OF and2 IS
12   BEGIN s <= i0 AND i1;
13 END arq_e2;
14
15 ARCHITECTURE arq_e3 OF and3 IS
16   SIGNAL int : BIT;
17 BEGIN
18   x1: ENTITY WORK.and2(arq_e2) PORT MAP(i0,  i1, int);
19   x2: ENTITY WORK.and2(arq_e2) PORT MAP(int, i2, s);
20 END arq_e3;
```

Quadro 9.7.1 Teste da dependência entre unidades (Exercício 9.7.4).

9.7.5 O exercício é semelhante ao proposto anteriormente (ver Quadro 9.7.2). Nesse caso, altere a declaração do pacote. Recompile somente essa declaração, e tente simular a entidade " or_multiplo ". Com base na mensagem de erro, altere o código para que a simulação possa ser executada com sucesso.

```
1  -- arquivo dep_err3.vhd
2  PACKAGE pk_sup IS    -- declaracao do pacote
3    FUNCTION or2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT;
4  END pk_sup;
5
6  USE WORK.pk_sup.ALL; -- torna todos itens do pacote pk_sup visiveis para pk_inf
7  PACKAGE pk_inf IS    -- declaracao do pacote
8    FUNCTION or4 (SIGNAL i0, i1, i2, i3 : IN  BIT) RETURN BIT;
9  END pk_inf;
10
11 USE WORK.pk_inf.ALL; -- torna itens do pacote pk_inf visiveis para or_multiplo
12 ENTITY or_multiplo IS
13   PORT (a, b, c, d : IN  BIT; s_e : OUT BIT);
14 END or_multiplo;
15
16 ARCHITECTURE teste OF or_multiplo IS
17 BEGIN
18   s_e  <= or4(a, b, c, d);
19 END teste;
20
21 PACKAGE BODY pk_inf IS  -- corpo do pacote pk_inf
22   FUNCTION or4 (SIGNAL i0, i1, i2, i3 : IN  BIT) RETURN BIT IS
23   BEGIN RETURN or2(i0, i1) OR or2(i2, i3);
24   END or4;
25 END pk_inf;
26
27 PACKAGE BODY pk_sup IS  -- corpo do pacote pk_sup
28   FUNCTION or2 (SIGNAL i0, i1 : IN  BIT) RETURN BIT IS
29   BEGIN RETURN (i0 OR i1);
30   END or2;
31 END pk_sup;
```

Quadro 9.7.2 Teste da dependência entre unidades (Exercício 9.7.5).

CAPÍTULO

10 PADRÃO IEEE 1164

O tipo " bit " definido no pacote padrão nem sempre é adequado para descrever o comportamento de uma interligação entre portas lógicas, e situações encontradas numa simulação. No teste de um circuito, por exemplo, o simulador assume o valor inicial " 0 " para um objeto do tipo " bit " sem inicialização. Essa condição pode encobrir um comportamento não-previsto pela descrição.

Muitas vezes, em um circuito real, é necessário incluir um barramento de comunicação com várias saídas interligadas a um mesmo nó. Nesse caso, os circuitos que acionam essas linhas possuem uma condição de operação que permite levar a saída da porta para um estado de alta impedância, e apenas uma porta controla o nó num dado instante. Essa condição de alta impedância não é prevista no tipo " bit ".

Um outro caso ocorre nas portas com saída tipo coletor, ou dreno, aberto. Nesses circuitos, o nível lógico alto é imposto por resistor e qualquer saída pode levar o nó ao nível lógico baixo. O nível lógico alto é caracterizado por uma capacidade de fornecer corrente fraca em comparação com um nível lógico baixo.

Em barramentos de comunicação, ou linhas do tipo "dreno aberto" acionadas por mais de uma saída, ocorre um outro problema além da ausência de uma condição que defina um estado de alta impedância. A linguagem VHDL não permite que um sinal seja acionado por mais de um controlador.

Assim, novos tipos foram propostos para descrever mais precisamente as condições encontradas em circuitos reais e problemas associados a simulação.

10.1 Tipos definidos

O padrão IEEE 1164 foi desenvolvido para substituir tipos não-padronizados propostos por usuários e fabricantes de aplicativos para a modelagem das condições encontradas em um circuito digital. O padrão corresponde ao pacote " std_logic_1164 " que contém, entre outros itens, a definição de tipos caracterizados por um *estado lógico*. Um estado lógico é determinado por duas condições: *nível lógico* e *força*. Assim, com o conceito de força associado, é possível estabelecer um critério de decisão quando controladores acionam um mesmo sinal com níveis lógicos diferentes.

No pacote " std_logic_1164 " é definido o tipo enumerado " std_ulogic " que contém nove valores (ver Tabela 10.1.1). Cada valor possui um grau de força e um nível lógico. " U " é o valor com grau de força mais elevado, e " - " é o valor mais fraco. Os valores " 0 " e " 1 " possuem o mesmo grau de força, assim como " L " e " H ". Com relação ao nível lógico, " 1 " é igual a " H ", e " 0 " é igual a " L ".

Na linguagem VHDL não é possível que um sinal seja acionado por mais de um controlador, ou *driver*, sem uma função que determine o valor resultante. Essa função, denominada função de resolução, ou *resolution function*, estabelece o valor final de um sinal considerando as condições impostas pelos diversos controladores que o acionam. O pacote " std_logic_1164 " define tipos contendo uma função de resolução para esses casos.

O tipo " std_ulogic " não tem uma função de resolução incorporada. Isso implica que somente um controlador pode acionar um sinal declarado como " std_ulogic ". O tipo " std_logic " é um subtipo do tipo " std_ulogic " contendo o mesmo conjunto de valores. A diferença entre esses dois tipos reside no fato de que o tipo " std_logic " é declarado com uma função de resolução, podendo, desse modo, ser empregado em descrições de barramentos de sinais ou outras condições que necessitem da conexão de mais de uma porta a um mesmo ponto do circuito. Como o tipo " std_logic " é um subtipo de " std_ulogic ", é possível a troca de valores entre sinais declarados como tipo " std_logic " ou " std_ulogic ".

Tabela 10.1.1 Valores definidos para o tipo " std_ulogic ".

valor		estado lógico correspondente			
U		uninitialized		não-inicializado	
X		forcing unknown		impondo desconhecido	
0	1	forcing 0	forcing 1	impondo 0	impondo 1
W		weak unknown		desconhecido fraco	
L	H	weak 0	weak 1	0 fraco	1 fraco
Z		high impedance		alta impedância	
-		do not care		não importa	

Na Tabela 10.1.2, é descrito o valor resultante para um sinal acionado por dois controladores do tipo " std_logic ". Conforme se pode observar, o valor " U " possui o nível mais elevado de força; o conflito desse valor com qualquer outro resulta sempre no valor " U ". Para valores iguais, o valor resultante não é alterado. No caso dos valores " X ", " 0 " ou " 1 ", um conflito entre esses valores resulta no valor " X ", correspondendo a um nível lógico desconhecido com força elevada. De modo semelhante, um conflito entre os valores " W ", " L " ou " H " resulta no valor " W ", com nível lógico desconhecido e força fraca.

Tabela 10.1.2 Função de resolução - tipos " std_logic " - 2 controladores acionando o mesmo nó.

U	U	U	U	U	U	U	U
X	X	X	X	X	X	X	U
0 / 1	X	0 / 1	0 / 1	0 / 1	*A	X	U
W	X	W	W	W	0 / 1	X	U
L / H	X	L / H	*B	W	0 / 1	X	U
Z	X	Z	L / H	W	0 / 1	X	U
-	X	X	X	X	X	X	U
	-	**Z**	**L / H**	**W**	**0 / 1**	**X**	**U**

*A Controladores com níveis lógicos iguais resultam no mesmo valor; controladores com níveis diferentes resultam no valor " X ".

*B Controladores com níveis lógicos iguais resultam no mesmo valor; controladores com níveis diferentes resultam no valor " W ".

A Figura 10.1.1 ilustra alguns casos possíveis de ocorrerem com os tipos definidos no pacote " std_logic_1164 ". Os valores " 0 " e " 1 " podem ser interpretados como a saída de uma porta lógica, ou um sinal ligado à alimentação. O valor " H " pode ser interpretado como uma saída do tipo coletor aberto (*open collector*) ou dreno aberto (*open drain*), onde o nível lógico alto é imposto por um resistor. Nos valores " X " e " W ", o nível lógico não é definido. Um sinal assumindo esse valor pode ser conseqüência de um conflito ocorrido pela imposição de níveis lógicos diferentes com mesma força. O valor " U " corresponde à condição de um nó ao qual não foi aplicado nenhum valor. Por último, o valor " - " pode ser empregado pelo aplicativo de síntese do circuito com objetivo de reduzir o circuito necessário para implementação. Ele não está associado a um nível lógico.

No pacote IEEE 1164, é possível, também, empregar vetores contendo tipos " std_ulogic " e " std_logic ". Dois tipos são definidos: o tipo " std_ulogic_vector " e o tipo " std_logic_vector ". O primeiro é um vetor contendo tipos " std_ulogic ", ou seja, sem uma função de resolução, e o segundo é um vetor contendo tipos " std_logic ". A utilização desses vetores é feita de maneira semelhante aos vetores " bit_vector " compostos de tipos " bit ".

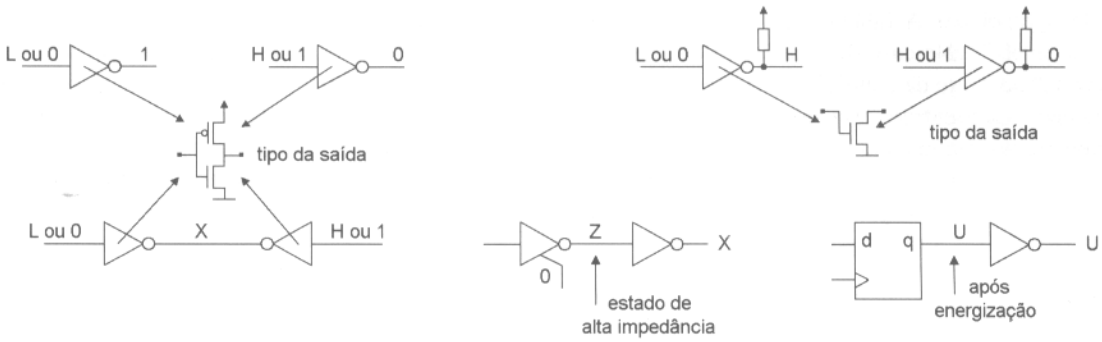


Figura 10.1.1 Exemplos de valores definidos no pacote " std_logic_1164 ".

Note que " std_logic_vector " não é um subtipo de " std_ulogic_vetor "; desse modo, não é possível a troca de valores entre objetos declarados com esses tipos.

10.2 Funções e subtipos definidos no pacote " std_logic_1164 "

Os subtipos definidos no pacote " std_logic_1164 " são apresentados na Tabela 10.2.1. Conforme visto, o tipo " std_logic " é um subtipo de " std_ulogic " definido com uma função de resolução. Outros subtipos, contendo uma parcela dos valores são, também, definidos. Eles podem ser empregados quando não é necessária a consideração de todos os valores na descrição.

Tabela 10.2.1 Subtipos.

`SUBTYPE std_logic IS resolved std_ulogic;`
`SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')`
`SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')`
`SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')`
`SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')`

Na Tabela 10.2.2 são relacionados os operadores lógicos suportados pelo padrão. Note que os operadores são funções que sobrecarregam as funções predefinidas no pacote padrão. Os parâmetros formais das funções são declarados como tipo " std_ulogic ", e o valor de retorno é declarado como tipo " UX01 ". Tendo em vista que os tipos " std_logic ", " X01 ", " X01Z ", " UX01 " e " UX01Z " são subtipos de " std_ulogic ", as funções podem ser empregadas para qualquer subtipo de " std_ulogic ". O valor de retorno da função sendo tipo " UX01 ", modela uma porta na qual os níveis lógicos na saída são impostos por elementos ativos, pois apenas os valores com maior força são resultantes na saída: " 1 ", " 0 ", " U " e " X ".

Tabela 10.2.2 Operadores lógicos para tipos " std_ulogic ".

`FUNCTION "and" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "or" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "nor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "xor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;`
`FUNCTION "not" ( l : std_ulogic ) RETURN UX01;`
Obs.: A função " xnor " não é suportada pela versão VHDL-1987.

A Tabela 10.2.3 apresenta o valor de retorno para todas as condições de entrada possíveis da função " and ". No caso de parâmetros de entrada com valores " 0 " ou " L ", o valor de retorno será " 0 ", independentemente da condição do outro parâmetro de entrada. O retorno do valor " 1 " ocorre quando ambos os parâmetros de entrada têm valores correspondentes a um nível lógico alto, valores " 1 " ou " H ". Nas condições restantes, excetuando o caso de um parâmetro de entrada com o valor " U ", o valor de retorno será " X ". O estado " U " em um dos parâmetros de entrada resulta no valor de retorno " U ", caso o valor do outro parâmetro de entrada não corresponda a um nível lógico baixo, valores " 0 " ou " L ". Tabelas semelhantes são descritas para as funções " or " e " xor ".

Tabela 10.2.3 Tabela verdade para a função " and ".

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

As funções que tratam dos operadores lógicos com tipos " std_logic_vector " e " std_ulogic_vector " realizam a função lógica equivalente entre cada elemento de mesma posição do vetor. Por exemplo, no caso da função " and ", é efetuada a operação lógica " e " descrita na Tabela 10.2.3 para cada elemento do vetor. A Tabela 10.2.4 relaciona essas funções. Note que são definidas funções distintas para cada tipo, pois o tipo " std_logic_vector " não é um subtipo de " std_ulogic_vector ". Assim, ambos os parâmetros de entrada das funções que executam as operações lógicas para os tipos " std_ulogic_vector " e " std_logic_vector " devem ser do mesmo tipo.

Tabela 10.2.4 Operadores lógicos para tipos " std_logic_vector " e " std_ulogic_vector ".

FUNCTION "and" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "and" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "nand" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "nand" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "or" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "or" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "nor" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "nor" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "xor" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "xor" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "xnor" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "xnor" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "not" (l : std_logic_vector) RETURN std_logic_vector;
FUNCTION "not" (l : std_ulogic_vector) RETURN std_ulogic_vector;
Obs.: A função " xnor " não é suportada pela versão VHDL-1987.

As funções de conversão para tipos " bit ", " bit_vector ", " std_logic ", " std_ulogic ", " std_logic_vector " e " std_ulogic_vector " estão relacionadas na Tabela 10.2.5. As funções " To_bit " e " To_bit_vector " detectam o nível lógico dos objetos de entrada. Valores " 1 " ou " H " são convertidos para o valor " 1 " do tipo " bit ", e valores " 0 " ou " L " são convertidos para o valor " 0 ". Qualquer outro valor retorna com o valor de " xmap " (ver exemplos na tabela). As conversões para tipos " std_logic ", " std_ulogic ", " std_logic_vector " e " std_ulogic_vector " mudam o tipo, mantendo o valor.

Tabela 10.2.5 Conversão para " bit ", " std_logic ", " std_ulogic " e vetores contendo esses tipos.

FUNCTION To_bit (s : std_ulogic; xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector (s : std_logic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_bitvector (s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_StdULogic (b : BIT) RETURN std_ulogic;
FUNCTION To_StdLogicVector (b : BIT_VECTOR) RETURN std_logic_vector;
FUNCTION To_StdLogicVector (s : std_ulogic_vector) RETURN std_logic_vector;
FUNCTION To_StdULogicVector (b : BIT_VECTOR) RETURN std_ulogic_vector;
FUNCTION To_StdULogicVector (s : std_logic_vector) RETURN std_ulogic_vector;
s <= TO_BITVECTOR (a); -- retorna '0' se a= 'U', 'X', 'Z', 'W' ou '-' s_1 <= TO_BITVECTOR (a,'1'); -- retorna '1' se a= 'U', 'X', 'Z', 'W' ou '-'

As funções de conversão para os tipos " X01 ", " X01Z " e " UX01 " estão relacionadas na Tabela 10.2.6. No caso dos tipos " std_ulogic ", " std_logic ", " std_ulogic_vector " e " std_logic_vector ", cada função estabelece uma correspondência entre os 9 valores definidos no padrão e o subconjunto de valores escolhido. A Tabela 10.2.7 descreve a equivalência definida para cada subconjunto nesses casos. Para os tipos " bit " e " bit_vector ", a correspondência é direta: o valor " 0 " tipo bit retorna " 0 ", e o valor " 1 " tipo bit retorna " 1 ".

Tabela 10.2.6 Conversão para os tipos " X01 ", " X01Z " e " UX01Z ".

FUNCTION To_X01 (s : std_logic_vector) RETURN std_logic_vector;
FUNCTION To_X01 (s : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION To_X01 (s : std_ulogic) RETURN X01;
FUNCTION To_X01 (b : BIT_VECTOR) RETURN std_logic_vector;
FUNCTION To_X01 (b : BIT_VECTOR) RETURN std_ulogic_vector;
FUNCTION To_X01 (b : BIT) RETURN X01;
FUNCTION To_X01Z (s : std_logic_vector) RETURN std_logic_vector;
FUNCTION To_X01Z (s : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION To_X01Z (s : std_ulogic) RETURN X01Z;
FUNCTION To_X01Z (b : BIT_VECTOR) RETURN std_logic_vector;
FUNCTION To_X01Z (b : BIT_VECTOR) RETURN std_ulogic_vector;
FUNCTION To_X01Z (b : BIT) RETURN X01Z;
FUNCTION To_UX01 (s : std_logic_vector) RETURN std_logic_vector;
FUNCTION To_UX01 (s : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION To_UX01 (s : std_ulogic) RETURN UX01;
FUNCTION To_UX01 (b : BIT_VECTOR) RETURN std_logic_vector;
FUNCTION To_UX01 (b : BIT_VECTOR) RETURN std_ulogic_vector;
FUNCTION To_UX01 (b : BIT) RETURN UX01;

Tabela 10.2.7 Conversão de tipos: correspondência de valores.

valor de entrada	valor convertido		
	função: To_X01	função: To_X01Z	função: To_UX01
U	X	X	U
X	X	X	X
0	0	0	0
1	1	1	1
Z	X	Z	X
W	X	X	X
L	0	0	0
H	1	1	1
-	X	X	X
tipo do argumento de entrada das funções:	" std_ulogic ", " std_ulogic_vector ", " std_logic " e " std_logic_vector ".		

Duas funções para a detecção de bordas de subida ou descida são disponíveis no pacote. Essas funções estão relacionadas na Tabela 10.2.8 e podem, por exemplo, ser empregadas na descrição de elementos do circuito sensíveis a bordas de um sinal de relógio.

Tabela 10.2.8 Detecção de bordas de subida e descida.

```
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
```

As funções relacionadas na Tabela 10.2.9 detectam a presença de valores sem uma correspondência com nível lógico alto ou baixo. Elas retornam com o valor verdadeiro, caso o argumento de entrada contenha um dos seguintes valores: " U ", " X ", " Z ", " W " ou " - ".

Tabela 10.2.9 Detecção de valores sem correspondência com um nível lógico alto ou baixo.

```
FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;
```

10.3 Descrições empregando o pacote " std_logic_1164 "

Na maioria das ferramentas, o pacote " std_logic_1164 " é previamente compilado e armazenado na biblioteca " IEEE ". Assim, para tornar a biblioteca disponível e os itens do pacote visíveis para uma descrição, é necessária a inserção das cláusulas " LIBRARY IEEE " e " USE ieee.std_logic_1164.ALL ".

Funções lógicas

No Quadro 10.3.1 é apresentado um exemplo onde quatro operações lógicas são realizadas entre sinais declarados como tipo " std_logic_vector ", e o resultado da simulação é apresentado na Figura 10.3.1. Conforme se pode observar, a saída " s " corresponde a uma ligação física com a entrada " a ", porque os valores são transmi-

tidos sem alteração para saída. No caso das saídas restantes, o valor de retorno é o subtipo " UX01 ", resultado de acordo com as declarações das funções apresentadas na Tabela 10.2.2.

Desse modo, as operações lógicas descrevem portas com capacidade de drenar e fornecer corrente, pois os valores " L " e " H " são convertidos para valores com um nível de força mais levado, " 0 " e " 1 ". De maneira semelhante, os valores com nível lógico indefinido, valores " Z " e " W ", são convertidos para valores com nível lógico indefinido, mas com um nível de força mais elevado, estado " X ".

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164b IS
 5    PORT( a,b                    : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
 6          s, s_not, s_and, s_or : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
 7    END std1164b;
 8
 9  ARCHITECTURE exemplo OF std1164b IS
10  BEGIN
11    s     <= a;
12    s_not <=   NOT a;
13    s_and <= a AND b;
14    s_or  <= a OR  b;
15  END exemplo;
```

Quadro 10.3.1 Primeiro exemplo empregando o pacote " std_logic_1164 ".

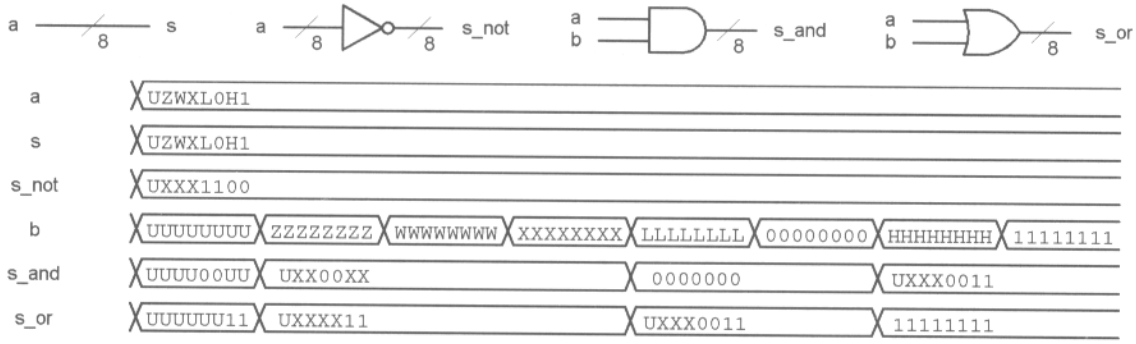


Figura 10.3.1 Simulação da entidade apresentada no Quadro 10.3.1.

Operação da função de resolução em caso de conflito

No Quadro 10.3.2 é apresentada uma descrição contendo sinais acionados por dois controladores distintos. Cada sinal é um vetor de oito bits. Esse exemplo não tem uma aplicação prática; ele serve apenas para verificar a função de resolução de tipos " std_logic ". A simulação da descrição é apresentada na Figura 10.3.2. Os valores na saída " s " seguem a função de resolução contida na Tabela 10.1.2. No caso da saída " s_not ", deve-se lembrar que o valor de retorno da função " not " é o subtipo " UX01 ". Assim, cada controlador pode aplicar unicamente os valores " U ", " X ", " 0 " e " 1 " ao sinal " s_not ", e o valor resultante é definido, então, com base na função de resolução apresentada na Tabela 10.1.2.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164c IS
 5    PORT( a, b     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
 6          s, s_not : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
 7    END std1164c;
 8
 9  ARCHITECTURE exemplo OF std1164c IS
10  BEGIN
11    s     <= a;       s     <= b;
12    s_not <= NOT a;   s_not <= NOT b;
13  END exemplo;
```

Quadro 10.3.2 Segundo exemplo empregando o pacote " std_logic_1164 ".

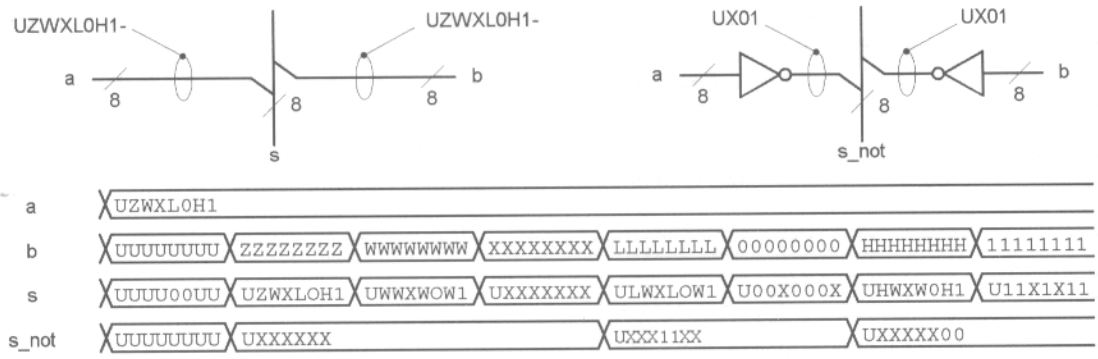


Figura 10.3.2 Simulação da entidade apresentada no Quadro 10.3.2.

Operações com o valor não importa " - "

O valor " - ", conforme proposto pelo padrão, corresponde à condição não importa. O tratamento dado a esse valor pelo simulador, contudo, não incorpora o significado dessa condição. A comparação de um valor com " - " deveria sempre retornar o valor verdadeiro, tendo em vista que o elemento " - " pode ser associado a qualquer valor. O simulador, entretanto, trata o valor " - " do mesmo modo que a outros valores, e somente considera verdadeira a condição de comparação, caso os dois elementos sejam idênticos. O significado do valor " - " é considerado na função " std_match ", presente no pacote " numeric_std " (ver Capítulo 11).

No exemplo apresentado no Quadro 10.3.3, o sinal " s1 " deveria receber sempre o valor da porta " v ", caso o significado do valor "não importa" fosse considerado. Como a condição verdadeira somente ocorre se os dois valores forem idênticos, " s1 " recebe sempre o valor da porta " f ". As declarações apresentadas nas linhas 14 e 15 do quadro complementam esse raciocínio.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164j IS
 5    PORT (f, v      : IN  STD_LOGIC;
 6          s1,s2,s3  : OUT STD_LOGIC);
 7  END std1164j;
 8
 9  ARCHITECTURE teste OF std1164j IS
10    CONSTANT nao_importa : STD_LOGIC := '-';
11    CONSTANT valor_1     : STD_LOGIC := '1';
12  BEGIN
13   s1 <= v WHEN valor_1     = '-' ELSE f;   -- equivalente a:  s1 <= f;
14   s2 <= v WHEN valor_1     = 'X' ELSE f;   -- equivalente a:  s2 <= f;
15   s3 <= v WHEN nao_importa = '-' ELSE f;   -- equivalente a:  s3 <= v;
16  END teste;
```

Quadro 10.3.3 Significado do valor " - ".

10.4 Síntese empregando tipos do pacote " std_logic_1164 "

No processo de síntese de um tipo enumerado, deve-se estabelecer um código para cada elemento do tipo. Assim, para um total de v valores definidos no tipo, é necessário um número de bits n, dado pela relação $n = \lceil \log_2 v \rceil$. O operador $\lceil \alpha \rceil$ determina o menor inteiro, maior ou igual ao número real α.

O tipo enumerado " std_ulogic " definido no pacote " std_logic_1164 " possui nove valores, o que levaria à necessidade de quatro bits para sua codificação. Isso, naturalmente, não corresponde ao objetivo do padrão em modelar o comportamento de uma única interligação digital. Diante desse fato, as ferramentas de síntese estabelecem uma correspondência entre os valores e o comportamento desejado para a interligação.

O valor " - " não possui uma correspondência com um nível lógico válido; entretanto, ele pode ser empregado pelo sintetizador para minimização lógica. Como o valor " - " é identificado com a condição "não importa", o sintetizador pode atribuir o nível lógico que resulte no circuito mais econômico na tecnologia-alvo da síntese.

Os valores " U ", " X " e " W " não correspondem, também, a nenhum nível lógico válido. Esses valores auxiliam na simulação, permitindo detectar conflitos em sinais acionados por mais de um controlador, e definir a condição "não-inicializado" para pontos do circuito que não receberam nenhuma atribuição de valor após o início da simulação. A ferramenta de síntese pode adotar a condição "não importa" para esses valores, tendo em vista que esses três valores não têm uma correspondência com um nível lógico válido. Deve-se observar, entretanto, que descrições contendo atribuição de valores " U ", " X " ou " W ", bem como comparações com um desses três valores podem ser consideradas duvidosas do ponto de vista de síntese. Um circuito digital consegue interpretar apenas níveis lógicos válidos; para qualquer outra condição, não é possível prever o comportamento do circuito. Assim, descrições contendo essas operações estariam tentando modelar um comportamento impossível de ser sintetizado. No caso de descrições construídas com a finalidade de gerar os estímulos de teste para uma outra descrição, operações com esses valores, bem como outros valores sem uma correspondência com um nível lógico, são perfeitamente razoáveis.

O valor " Z " não possui, também, um nível lógico válido. A atribuição do valor " Z " a um objeto é o modo de indicar à ferramenta de síntese a necessidade da inclusão de um circuito lógico com uma saída capaz de ficar em um estado de alta impedância.

Por último, têm-se os valores " 0 ", " 1 ", " L " e " H ", os únicos com correspondência a níveis lógicos válidos. Com relação à comparação de valores, a ferramenta de síntese pode estabelecer uma equivalência entre os valores " L " e " H " com os valores " 0 " e " 1 ", tendo em vista que uma porta lógica não distingue a força do valor imposto; apenas o nível lógico deste. Com relação à atribuição de valores, um sinal recebendo o valor " H " de um controlador deveria indicar ao sintetizador a necessidade de uma porta com saída do tipo "coletor aberto" ou "dreno aberto", para satisfazer a condição de um nível lógico alto fraco. Normalmente, essa inferência não ocorre nas ferramentas de síntese, e o circuito gerado contém uma saída convencional. Isso, de certo modo, corresponde à modelagem dada às operações lógicas contidas no padrão, pois, conforme visto, essas funções retornam um tipo " UX01 ", o que corresponde a uma porta com capacidade de impor níveis lógicos com força.

A seguir, serão apresentados alguns exemplos de descrições e o respectivo circuito sintetizado. O sintetizador empregado estabelece a correspondência com os valores encontrados no padrão, conforme a Tabela 10.4.1. O valor " X " nesse sintetizador é tratado de modo semelhante ao valor " - ", isto é, dependendo do contexto, a ferramenta escolhe o valor " 0 " ou " 1 ".

Tabela 10.4.1 Codificação de valores adotada pela ferramenta de síntese empregada.

valores " std_ulogic "	U	X	0	1	Z	W	L	H	-
valores codificados	X	X	0	1	Z	X	0	1	X

Saídas com estado em alta impedância

O Quadro 10.4.1 apresenta o código típico para a criação de uma porta de saída com estado de alta impedância, e a simulação da entidade é ilustrada pela Figura 10.4.1. A estrutura criada para a saída " s " corresponde a uma chave conectada ou não com a entrada " a ". Na condição " hab=1 " os valores são transmitidos sem alteração para o sinal; na condição " hab=0 ", é transferido o valor " ZZZZZZZZ ". No caso do sinal " s_not ", como a função " not " retorna o subtipo " UX01 ", apenas os valores " U ", " X ", " 0 " e " 1 " devem estar presentes na condição " hab=1 ".

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY std1164d IS
5    PORT( hab      : IN  BIT;
6          a        : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
7          s, s_not : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
8    END std1164d;
9
10 ARCHITECTURE exemplo OF std1164d IS
11 BEGIN
12   s     <=     a WHEN hab = '1' ELSE "ZZZZZZZZ";
13   s_not <= NOT a WHEN hab = '1' ELSE "ZZZZZZZZ";
14 END exemplo;
```

Quadro 10.4.1 Descrição de saídas com estado de alta impedância na região de código concorrente.

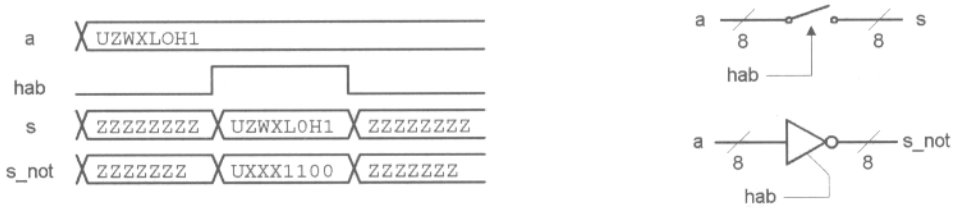


Figura 10.4.1 Simulação da entidade apresentada no Quadro 10.4.1.

O Quadro 10.4.2 apresenta uma outra descrição para saídas com estado em alta impedância empregando um processo. Note que tanto o sinal da entrada " a ", como o sinal que habilita as portas, " hab ", devem estar na lista de sensibilidade do processo. O resultado da simulação não sofre alterações; ele segue a carta de tempos apresentada na Figura 10.4.1.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY std1164e IS
 5   PORT( hab      : IN  BIT;
 6         a        : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
 7         s, s_not : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
 8   END std1164e;
 9
10 ARCHITECTURE exemplo OF std1164e IS
11 BEGIN
12   abc: PROCESS (a, hab)
13     BEGIN
14     IF (hab = '1') THEN
15       s     <=  a;
16       s_not <= NOT a;
17     ELSE
18       s     <= (OTHERS => 'Z');
19       s_not <= "ZZZZZZZZ";
20     END IF;
21   END PROCESS abc;
22 END exemplo;
```

Quadro 10.4.2 Descrição de saídas com estado de alta impedância empregando um processo.

Portas com estado de alta impedância na saída podem ser criadas definindo-se um bloco contendo a palavra reservada " GUARDED ", e uma expressão de guarda. Conforme visto, as declarações de transferência de sinal contendo a palavra " GUARDED " somente acionam o sinal, quando a condição de guarda for verdadeira. O Quadro 10.4.3 apresenta um novo código para o circuito descrito anteriormente empregando um bloco com guarda.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY std1164f IS
 5   PORT( hab      : IN  BIT;
 6         a        : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
 7         s, s_not : OUT STD_LOGIC_VECTOR (7 DOWNTO 0) BUS);
 8   END std1164f;
 9
10 ARCHITECTURE exemplo OF std1164f IS
11 BEGIN
12   abc: BLOCK (hab = '1')
13   BEGIN
14      s     <= GUARDED a;
15      s_not <= GUARDED NOT a;
16   END BLOCK;
17 END exemplo;
```

Quadro 10.4.3 Porta de saída com estado de alta impedância empregando bloco com guarda.

A expressão de guarda é inserida após a declaração do bloco; no caso, " hab=1 ". O controlador dos sinais " s " e " s_not " somente é habilitado com " hab=1 "; caso contrário, os sinais " s " e " s_not " não sofrem nenhuma alteração. Observe que essa condição leva à inferência de um elemento de memória, um *latch*, e não uma condição de alta impedância. Devido a esse motivo, é necessário que os sinais " s " e " s_not " sejam declarados como " BUS ". Esse modelo para descrição de portas com estado de alta impedância não é aceito por todas as ferramentas de síntese.

O resultado da síntese das descrições propostas nos Quadros 10.4.1, 10.4.2 e 10.4.3 é o mesmo. A Figura 10.4.2 ilustra o circuito gerado. Conforme se pode notar, ambas as saídas empregam portas com saída em alta impedância, possibilitando impor unicamente as condições correspondentes aos valores " Z ", " 0 " e " 1 ". Numa tecnologia empregando transistores MOS, é possível a implementação de circuitos que operam de maneira equivalente ao proposto para a saída " s " na Figura 10.4.2. Esses circuitos são conhecidos como portas de passagem, ou *transmission gates*. Os circuitos realizam uma função similar a uma chave, transmitindo de modo bidirecional uma informação. A ferramenta de síntese desconsidera essa possibilidade, pois no nível RTL a saída " s " é acionada por uma porta lógica.

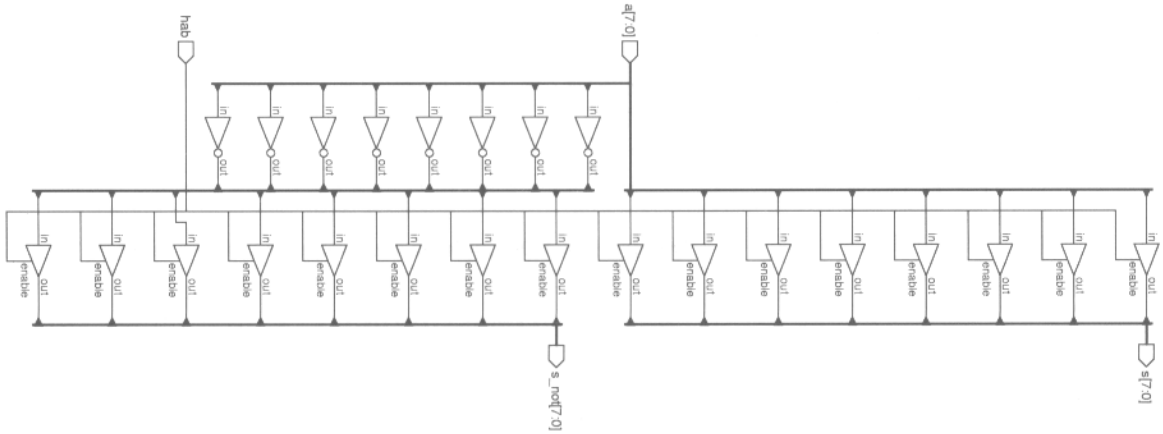


Figura 10.4.2 Circuito sintetizado — descrições dos Quadros 10.4.1, 10.4.2 e 10.4.3.

Resultado da codificação de valores no circuito gerado

Conforme visto na Tabela 10.4.1, a ferramenta de síntese estabelece uma codificação entre os nove valores definidos no padrão IEEE 1164 e os níveis lógicos alto e baixo. O Quadro 10.4.4 apresenta uma descrição de teste para verificação dessa codificação. Basicamente são analisados o comportamento da ferramenta em um código empregando níveis lógicos válidos como " L " e " 0 ", a necessidade de se especificar todos os valores em declarações com tomada de decisão, e a operação da cláusula " OTHERS ".

Na construção " WITH SELECT " (linhas 11 a 14), o sinal " x " recebe o valor " H " se " a=0 "; " L ", se " a=1 "; e " Z ", para os seis valores restantes. O código descreve, portanto, a inversão do nível lógico para as condições " a=0 " e " a=1 ", e um estado de alta impedância para as outras condições.

Observando o circuito sintetizado na Figura 10.4.3, verifica-se que apenas a operação de inversão foi implementada; não existe um circuito para levar a saída " x " para o estado de alta impedância. No processo de síntese, a condição " WHEN OTHERS " foi ignorada pela ferramenta, pois dois níveis lógicos válidos foram definidos para a entrada " a " (linhas 12 e 13). Sendo assim, todas as outras condições foram consideradas como "não importa".

O mesmo raciocínio se aplica à segunda construção " WITH SELECT " entre as linhas 15 e 18. Novamente, a ferramenta emprega a correspondência entre os valores " L " e " 0 ", e entre os valores " H " e " 1 ". Assim, os dois níveis lógicos válidos possíveis para o sinal " a " apresentados nas linhas 16 e 17 definem o circuito gerado. No Quadro 10.4.5 são apresentadas duas mensagens de informação geradas pela ferramenta de síntese, alertando que as condições englobadas pela cláusula " OTHERS " não são usadas.

Com relação ao tipo das saídas " x " e " y " na Figura 10.4.3, observa-se que elas são do mesmo tipo. Conforme visto, as ferramentas de síntese não fazem distinção com relação à atribuição de valores do tipo " H " ou " 1 " e do tipo " L " ou " 0 ".

A terceira construção " WITH SELECT " (linhas 19 a 21) descreve um circuito que impõe o valor " 1 " na saída " z " com a condição " a=1 ", ou deixa essa saída em estado de alta impedância nas condições restantes.

No circuito sintetizado, " a=1 " leva a saída " z " ao nível de tensão correspondente ao nível lógico alto, e " a=0 " desabilita a porta que aciona essa saída. Nesse caso, a condição " a=0 " foi englobada com todas as condições restantes.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY std1164i IS
 5   PORT (a        : IN  STD_LOGIC;
 6         x,y,z    : OUT STD_LOGIC);
 7 END std1164i;
 8
 9 ARCHITECTURE teste OF std1164i IS
10 BEGIN
11   WITH a SELECT
12     x <= 'H' WHEN '0',
13          'L' WHEN '1',
14          'Z' WHEN OTHERS;
15   WITH a SELECT
16     y <= '1' WHEN 'L',
17          '0' WHEN 'H',
18          'Z' WHEN OTHERS;
19   WITH a SELECT
20     z <= '1' WHEN '1',
21          'Z' WHEN OTHERS;
22 END teste;
```

Quadro 10.4.4 Descrição de um circuito de teste para síntese.

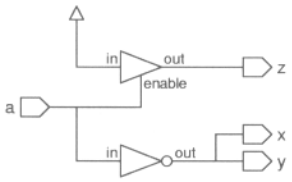


Figura 10.4.3 Circuito sintetizado — Quadro 10.4.4 — nível RTL.

```
-- Compiling root entity std1164i(teste)
"/vhdl/std1164i.vhd",line 11: Info, others clause is never selected for synthesis.
"/vhdl/std1164i.vhd",line 15: Info, others clause is never selected for synthesis.
--
```

Quadro 10.4.5 Mensagem da ferramenta de síntese — Quadro 10.4.1.

Síntese com o valor "não importa" - comparação

Conforme visto, a comparação de um valor qualquer com o valor " - " não retorna sempre um valor verdadeiro. Esse comportamento é igualmente observado nas ferramentas de síntese, conforme ilustra a descrição do Quadro 10.4.6. Para síntese, deve-se, também, empregar a função " std_match ", contida no pacote " numeric_std ", caso se deseje observar o significado do valor " - ".

Na linha 11, considerando o significado do valor " - ", " s1 " deveria ir para o nível alto na condição " t(2), t(0)=0,1 ". Do mesmo modo, na linha 12, o valor " 0 " deveria ser transferido para " s2 " em qualquer condição. O resultado da síntese para os sinais " s1 " e " s2 ", entretanto, segue o comportamento descrito pelo simulador (ver Figura 10.4.4). A comparação entre dois valores é verdadeira unicamente quando os operandos forem idênticos. A ferramenta de síntese segue essa conduta e considera a condição " t(1)='-' " sempre falsa, porque no circuito real, " t(1) " somente pode receber os níveis lógicos alto ou baixo. Assim, as entradas " t " não são utilizadas, e as saídas " s1 " e " s2 " estão permanentemente em estado de alta impedância.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164h IS
 5    PORT (t      : IN  STD_LOGIC_VECTOR (2 DOWNTO 0);
 6          s1,s2  : OUT STD_LOGIC);
 7  END std1164h;
 8
 9  ARCHITECTURE teste OF std1164h IS
10  BEGIN
11    s1 <= '1' WHEN t=  "0-1" ELSE 'Z';
12    s2 <= '0' WHEN t(1)= '-' ELSE 'Z';
13  END teste;
```

Quadro 10.4.6 Síntese com o valor " - " — comparação.

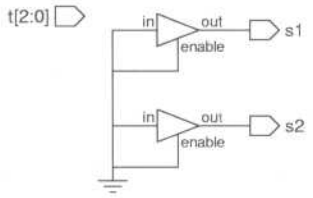


Figura 10.4.4 Resultado da síntese — descrição do Quadro 10.4.6.

Síntese com o valor "não importa" - minimização de tabelas

No Quadro 10.4.7 são apresentados outros dois casos empregando a construção " WITH SELECT ". No primeiro caso (linhas 11 a 14), três condições são cobertas pelo valor " - ". A melhor opção de minimização resultaria em " s3=t(2) " (ver mapa de Kanaugh, na Figura 10.4.5). Entretanto, conforme mostra o esquema do circuito sintetizado, a ferramenta de síntese optou pelo valor " 0 " nas condições "não importa", resultando em " s3=t(2).t(1) ". No segundo caso, a melhor opção de minimização, " s4=1 ", foi adotada pela ferramenta.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164y IS
 5    PORT (t      : IN  STD_LOGIC_VECTOR (2 DOWNTO 0);
 6          s3,s4  : OUT STD_LOGIC);
 7  END std1164y;
 8
 9  ARCHITECTURE teste OF std1164y IS          --        t1,t0
10  BEGIN                                      --      00 01 11 10
11    WITH t SELECT                            -- t2   ------------
12      s3 <= '1' WHEN "111" | "110",          --  0  | 0  0  -  0 |
13            '0' WHEN "000" | "001" | "010",  --  1  | -  -  1  1 |
14            '-' WHEN OTHERS;                 --       ------------
15    WITH t SELECT                            --       ------------
16      s4 <= '1' WHEN "111" | "110",          --  0  | -  -  -  - |
17            '-' WHEN OTHERS;                 --  1  | -  -  1  1 |
18  END teste;                                 --       ------------
```

Quadro 10.4.7 Síntese com o valor " - " — minimização de tabelas.

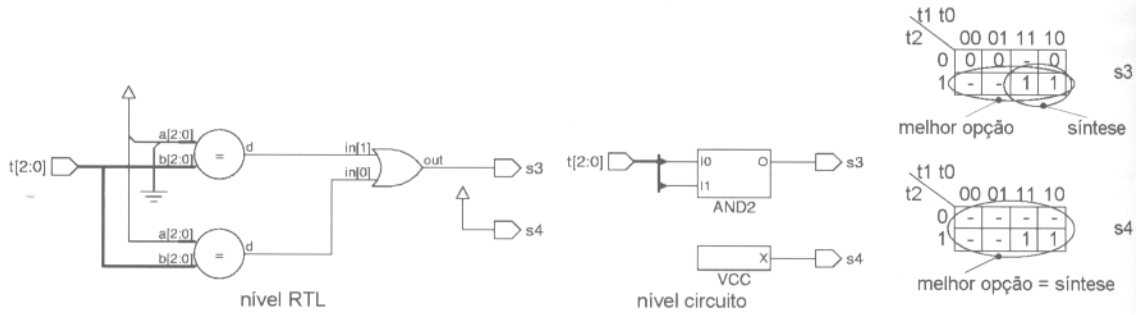


Figura 10.4.5 Resultado da síntese — descrição do Quadro 10.4.7.

Detecção de bordas de subida ou descida

Do ponto de vista de simulação, a identificação de bordas de subida ou descida em sinais empregando tipos " std_logic " deve considerar a existência de outros valores, além de " 0 " e " 1 ". Para uma transição válida, os valores assumidos pelo objeto antes e depois do evento devem ter uma correspondência com níveis lógicos válidos. Uma mudança de valor entre " Z " e " 0 ", por exemplo, não pode ser caracterizada como uma borda de descida.

No caso de uma borda de subida, o teste deve verificar a ocorrência de um evento e testar se o valor atual corresponde a um nível lógico alto, valores " H " ou " 1 ". Como o nível lógico anterior ao evento pode ser um valor diferente de " L " ou " 0 ", é necessário, ainda, verificar se o valor anterior atende essa condição.

Assim, a detecção de uma borda de subida deve executar os testes apresentados na linha 13 do Quadro 10.4.8. Nesses casos, é conveniente empregar as funções disponíveis no pacote " std_logic_1164 " para detecção de bordas de subida ou descida: funções " rising_edge " e " falling_edge ". Na linha 17 foi empregada a função " rising_edge " em substituição ao teste da linha 13.

Com relação à síntese, apenas níveis lógicos válidos são considerados. Assim, a ocorrência de um evento seguido de um nível lógico válido é suficiente para que a ferramenta interprete como uma borda de subida ou descida. Desse modo, o teste executado na linha 20 do código é suficiente para um circuito visando à síntese. Entretanto, para manter uma compatibilidade com a simulação é conveniente empregar as funções " rising_edge " e " falling_edge " que são, também, aceitas pelas ferramentas de síntese. O teste apresentado na linha 13 não é apropriado para síntese, pois o atributo " LAST_VALUE " não é possível de ser sintetizado, e o comportamento da ferramenta nesses casos nem sempre é definido.

Conforme se pode observar na Figura 10.4.6, os três *flip flops* descritos no Quadro 10.4.8 foram inferidos corretamente, apesar de a descrição conter o atributo " LAST_VALUE ".

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3
 4  ENTITY std1164g IS
 5    PORT (ck, da, db, dc : IN  STD_LOGIC;
 6          qa, qb, qc     : OUT STD_LOGIC);
 7  END std1164g;
 8
 9  ARCHITECTURE teste OF std1164g IS
10  BEGIN
11    abc: PROCESS (ck)
12    BEGIN
13      IF ck'EVENT AND (ck='1' OR ck='H')AND(ck'LAST_VALUE='0' OR ck'LAST_VALUE='L')
14        THEN qa <= da;
15      END IF;
16
17      IF rising_edge(ck) THEN qb <= db;
18      END IF;
19
20      IF ck'EVENT AND ck='1' THEN qc <= dc;
21      END IF;
22     END PROCESS abc;
23  END teste;
```

Quadro 10.4.8 Detectando borda de subida num sinal tipo " std_logic ".

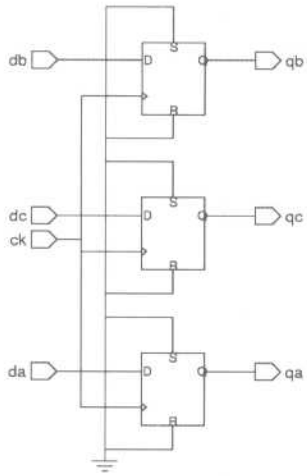


Figura 10.4.6 *Flip flops* inferidos a partir da descrição do Quadro 10.4.8.

Comando " Null ": construção " Case When " e valores " Std_Logic "

Na construção " CASE WHEN ", todos os valores que a expressão de escolha pode assumir devem ser cobertos. Caso a expressão de escolha empregue tipos " Std_Logic ", inúmeras combinações de valores sem correspondência com níveis lógicos válidos são possíveis. Para descrições visando à síntese, essas combinações não têm significado, e são ignoradas pelas ferramentas de síntese. Entretanto, as ferramentas de síntese abortam a operação se um dos valores da expressão de escolha for omitido, devido ao erro de sintaxe.

Uma opção para contornar esse problema é o emprego da palavra reservada " OTHERS " para englobar as condições contendo valores sem correspondência com um nível lógico válido. Na condição " OTHERS ", é executado o comando " NULL " que não realiza nenhuma operação. O Quadro 10.4.9 contém duas descrições semelhantes empregando tipos " BIT " e " Std_Logic ". No comando " CASE " da entidade " null_1y ", a expressão de escolha " sel " tem todos os seus valores cobertos em quatro opções. No caso da entidade " null_1x ", é necessária a inclusão da opção " WHEN OTHERS " para agrupar os valores não-cobertos da expressão de escolha.

```
 1 ENTITY null_1y IS
 2   PORT (y   :  IN  bit_vector(2 DOWNTO 0);
 3         sel :  IN  bit_vector(1 DOWNTO 0);
 4         s   :  OUT bit);
 5 END null_1y;
 6
 7 ARCHITECTURE teste OF null_1y IS
 8 BEGIN
 9   abc: PROCESS (y, sel)
10   BEGIN
11     CASE sel IS
12       WHEN "00"   => s <= y(0) AND y(1);
13       WHEN "10"   => s <= y(2) XOR y(0);
14       WHEN "01"   => s <= y(0)  OR y(1);
15       WHEN "11"   => s <= y(2) NOR y(0);
16     END CASE;
17   END PROCESS abc;
18 END teste;
```

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY null_1x IS
 5   PORT (x   : IN  std_logic_vector(2 DOWNTO 0);
 6         sel : IN  std_logic_vector(1 DOWNTO 0);
 7         s   : OUT std_logic);
 8 END null_1x;
 9
10 ARCHITECTURE teste OF null_1x IS
11 BEGIN
12   abc: PROCESS (x, sel)
13   BEGIN
14     CASE sel IS
15       WHEN "00"   => s <= x(0) AND x(1);
16       WHEN "10"   => s <= x(2) XOR x(0);
17       WHEN "01"   => s <= x(0)  OR x(1);
18       WHEN "11"   => s <= x(2) NOR x(0);
19       WHEN OTHERS => NULL;
20     END CASE;
21   END PROCESS abc;
22 END teste;
```

Quadro 10.4.9 Comando " NULL ", construção " CASE WHEN " e valores " Std_Logic ".

O resultado da síntese para as duas descrições é ilustrado na Figura 10.4.7. Os circuitos são idênticos, indicando que os códigos foram interpretados corretamente pela ferramenta: os esquemas contêm apenas lógica combinacional. Observe que, no caso da entidade " null_1x ", a ferramenta de síntese informa que a cláusula " OTHERS " da

construção " CASE WHEN " não foi empregada no processo de síntese (ver Quadro 10.4.10). A remoção dessa cláusula, entretanto, impede a síntese, devido ao erro de sintaxe, conforme observado anteriormente.

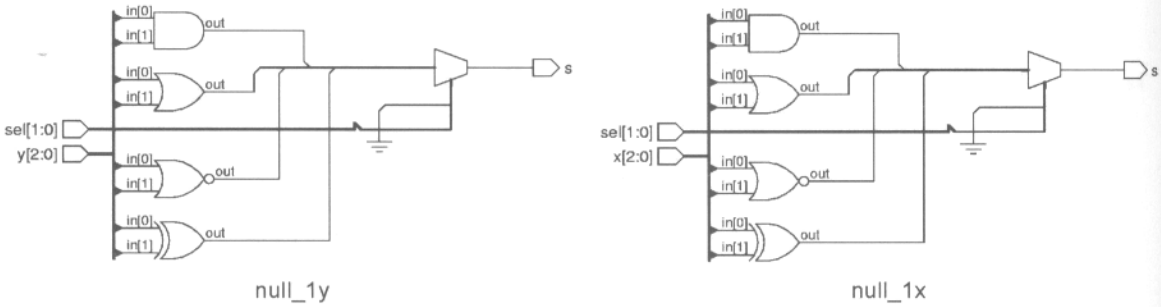


Figura 10.4.7 Resultado da síntese, descrições do Quadro 10.4.9.

```
-- Compiling root entity null_1x(teste)
"\vhdl\null_1x.vhd",line 14: Info, others clause is never selected for synthesis.
 -
```

Quadro 10.4.10 Mensagem gerada pela ferramenta de síntese.

10.5 Principais pontos abordados

- O tipo " std_ulogic " definido no pacote " std_logic_1164 ":
  - pode assumir os valores " U ", " X ", " 0 ", " 1 ", " Z ", " W ", " L ", " H " e " - " ;
  - não tem uma função de resolução incorporada;
  - somente um controlador pode acionar um sinal " std_ulogic ".
- O tipo " std_logic ":
  - é um subtipo " std_ulogic " que contém o mesmo conjunto de valores;
  - é declarado com uma função de resolução;
  - mais de um controlador pode acionar um sinal " std_logic ".
- No pacote " std_logic_1164 ", são definidas funções que realizam operações lógicas sobrecarregando as funções lógicas definidas no pacote padrão. Funções de conversão de tipos são, também, definidas.
- As ferramentas de síntese empregam os valores sem um nível lógico válido do modo mais conveniente para a elaboração do circuito.
- A ferramenta de síntese infere a necessidade de portas com saídas de estado de alta impedância pela atribuição do valor " Z ".
- A função " std_match " é mais adequada, caso se deseje considerar o significado correto do valor " - " em uma comparação. Ela está contida no pacote " numeric_std ", apresentado no Capítulo 11.
- As funções " rising_edge " e " falling_edge " devem ser empregadas para a detecção de bordas de subida e descida. Elas permitem manter uma compatibilidade com a simulação e síntese.

10.6 Exercícios

10.6.1 Na entidade " std1164c ", apresentada no Quadro 10.3.2, altere o tipo dos sinais " s " e " s_not " de " std_logic_vector " para " std_ulogic_vector ". Compile a nova descrição e comente o resultado obtido.

10.6.2 Remova a palavra reservada " BUS " na linha 7 da entidade " std1164f " apresentada no Quadro 10.4.3. Compile e simule a entidade conforme os estímulos propostos na Figura 10.4.1. Explique a nova operação do circuito.

10.6.3 Apresente uma descrição para o circuito mostrado na Figura 10.6.1 empregando a construção " WHEN ELSE ".

10.6.4 Apresente uma descrição para o circuito mostrado na Figura 10.6.1 empregando a construção " IF ELSE ".

10.6.5 Apresente uma descrição para o circuito mostrado na Figura 10.6.1 empregando a construção " BLOCK GUARDED ".

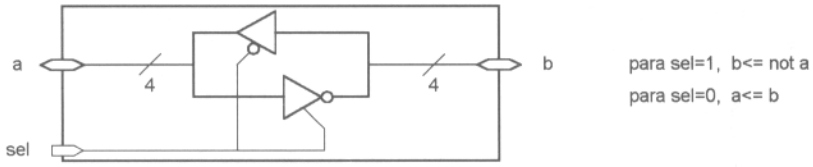


Figura 10.6.1 Circuito dos Exercícios 10.6.3, 10.6.4 e 10.6.5.

10.6.6 Sintetize as descrições propostas para os Exercícios 10.6.3, 10.6.4 e 10.6.5 e verifique se o código foi interpretado corretamente pela ferramenta de síntese.

10.6.7 O código do Quadro 10.6.1 descreve uma máquina de estados para o controle de velocidade. Conforme o valor do sinal " acelera_b ", o valor do sinal " velocidade " é incrementado ou decrementado a cada borda de subida de " ck ". Altere o tipo dos sinais " ck " e " rst " para " std_logic ", e o tipo do sinal " velocidade " para " std_logic_vector ". Considere as implicações na construção " CASE WHEN ".

```
 1  ENTITY null_al0 IS
 2    PORT( ck, rst     : IN  BIT;
 3          acelera_b   : IN  BOOLEAN;
 4          velocidade  : BUFFER BIT_VECTOR (1 DOWNTO 0));
 5    END null_al0;
 6
 7  ARCHITECTURE exemplo OF null_al0 IS
 8  BEGIN
 9    abc: PROCESS
10    BEGIN
11      WAIT UNTIL (ck'EVENT AND ck='1');
12      IF rst = '1' THEN velocidade <= "00";  -- parado
13      ELSE
14        CASE velocidade IS
15          WHEN "00" =>
16            IF acelera_b THEN velocidade <= "01"; ELSE velocidade <= "00"; END IF;
17          WHEN "01" =>
18            IF acelera_b THEN velocidade <= "10"; ELSE velocidade <= "00"; END IF;
19          WHEN "10" =>
20            IF acelera_b THEN velocidade <= "11"; ELSE velocidade <= "01"; END IF;
21          WHEN "11" =>
22            IF acelera_b THEN velocidade <= "11"; ELSE velocidade <= "10"; END IF;
23        END CASE;
24      END IF;
25    END PROCESS abc;
26  END exemplo;
```

Quadro 10.6.1 Descrição de uma máquina de estados — Exercício 10.6.7.

10.6.8 Sintetize a descrição do Quadro 10.6.1 e a descrição proposta para o Exercício 10.6.7. Observe que o processo não tem uma lista de sensibilidade, e o sinal de relógio que controla a máquina é definido no comando " WAIT ". Compare e comente os resultados.

10.6.9 A Figura 10.6.2 contém o diagrama de blocos de uma interface de comunicação serial, e a Figura 10.6.3 ilustra a operação da interface. A comunicação dos dados é feita por uma linha de dados bidirecional denominada " sda ", *serial data*, e o sinal de relógio para o controle da transmissão é feito pela linha " scl ", *serial clock*. A direção da transmissão é definida pela linha " rec_env ", na condição " rec_env=1 " os dados são recebidos pela interface. A cada borda de subida do sinal " scl " um dado é transmitido através da linha " sda ". A leitura ou escrita de dados para o interior da interface na forma paralela é feita pela linha denominada " dado ". O controle das operações com a linha " dado " é feito pelos terminais " ce " e " rd_wr ". O primeiro habilita a interface na condição " ce=0 "; e o segundo define a operação: " rd_wr=0 " escrita, " rd_wr=1 " leitura.

Apresente uma descrição para a interface proposta. No diagrama da Figura 10.6.2 o registrador " ds " armazena os dados a serem transferidos, e o seu controle é executado pelos sinais " ce " e " rd_wr ". O registrador " di " é um registrador de deslocamento controlado por " scl ". A entrada " reset " pode ser empregada para inicializar o controle interno da máquina.

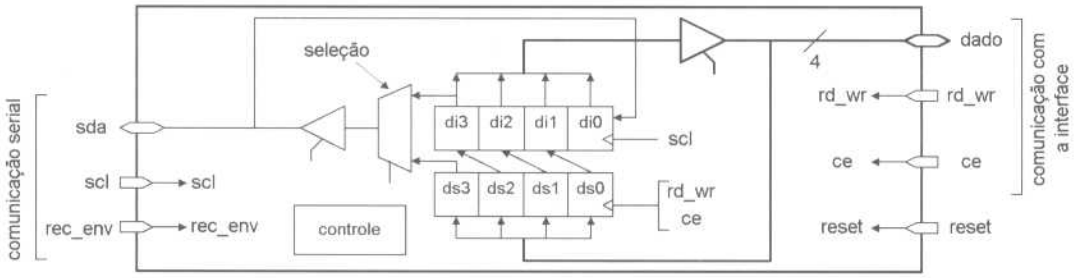


Figura 10.6.2 Diagrama de blocos da interface de comunicação serial.

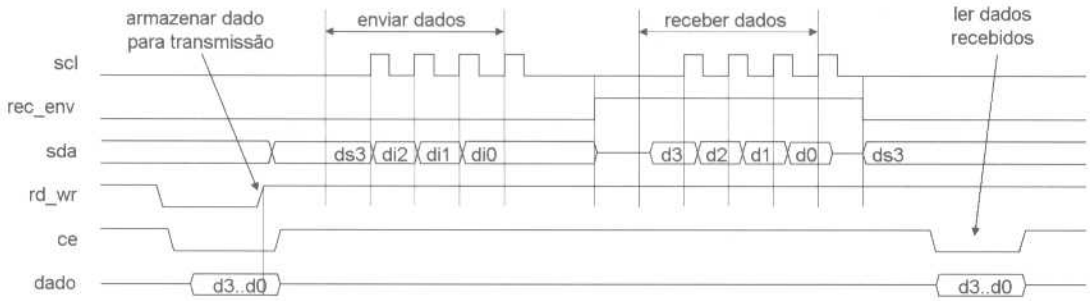


Figura 10.6.3 Esquema de operação e controle da interface.

10.6.10 A linha " rec_env " da interface proposta no exercício anterior pode ser eliminada inserindo-se um mecanismo de controle de início e fim de transmissão, e um bit de dados definindo o sentido da comunicação dos dados. A Figura 10.6.4 ilustra o esquema de controle proposto baseado no padrão I²C de comunicação. Na carta de tempos, a linha " sda " foi dividida em " sda entrada " e " sda saida " para identificar o sentido da comunicação num dado instante. Isto é, a informação está sendo recebida ou transmitida pela interface.

O primeiro dado recebido, ciclo 1, define a operação da interface. Caso esse bit tenha um nível lógico baixo, a interface deve enviar dados; na outra condição, a interface deve receber dados. O início de uma operação é definido por uma transição de um nível lógico alto para um nível lógico baixo na linha " sda " com " scl=1 " (ver ciclo 0). O fim de uma operação é identificado por uma transição de um nível lógico baixo para um nível lógico alto em " sda " com " scl=1 ", conforme mostrado no ciclo 7. Esse mecanismo de detecção de início e fim impõe a seguinte condição na linha " sda ": os dados presentes nessa linha devem permanecer estáveis, se " scl=1 ". Se essa condição não for respeitada, poderá ser identificada uma falsa condição de início ou fim, devido a uma transição na linha " sda ". Devido a esse fato, no esquema de operação proposto os dados são transmitidos a cada borda de descida de " scl ", assegurando, assim, a manutenção da informação na linha " sda " com " scl=1 ".

Proponha uma descrição que receba e transmita dados segundo o esquema apresentado na Figura 10.6.4. Alguns cuidados devem ser tomados para que a descrição seja possível de ser sintetizada. Um elemento de memória não pode ser sensível tanto à borda de descida como a uma borda de subida; assim, a recepção dos dados e a transmissão dos dados devem empregar registradores distintos (ver Figura 10.6.5). O mecanismo de identificação do início de operação e fim de operação sofre do mesmo problema. Um é ativado na borda de descida, e o outro na borda de subida.

Um outro ponto que deve ser levado em consideração é quanto à linha " sda ". Na proposta do padrão I²C, essa linha é do tipo dreno aberto para manter um nível lógico definido nos intervalos que nenhum periférico está controlando " sda " (ver Figuras 10.6.4 e 10.6.5). Note que, no esquema proposto, a linha " sda " é acionada por uma saída com estado de alta impedância. Assim, logo após o ciclo 1, numa operação para enviar dados, a interface " A " libera a linha " sda " para que a interface " B " assuma o controle. Nesse intervalo de tempo que o barramento fica em estado de alta impedância, uma interferência pode causar uma perturbação na linha " sda " levando à detecção de uma falsa condição de início ou fim. Desse modo, é necessário que a implementação do circuito contenha uma resistência para impor o nível lógico alto quando a linha " sda " se encontra em estado de alta impedância. Como uma saída do tipo dreno aberto não é normalmente inferida pelas ferramentas de síntese, a descrição deve empregar uma saída com estado da alta impedância.

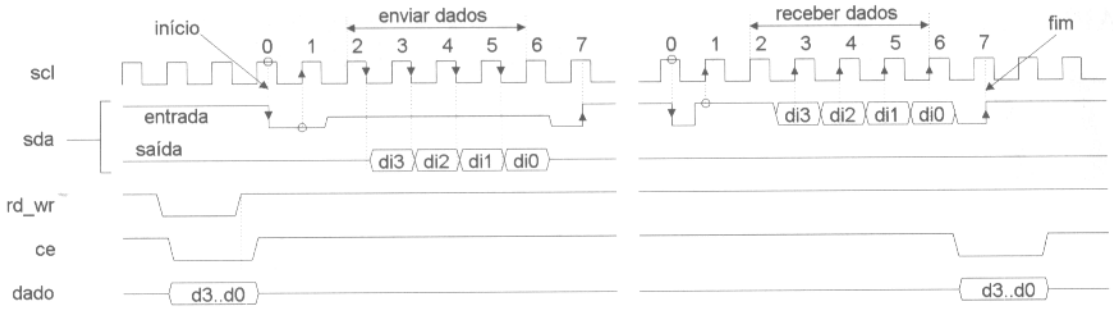


Figura 10.6.4 Esquema de operação e controle da interface II.

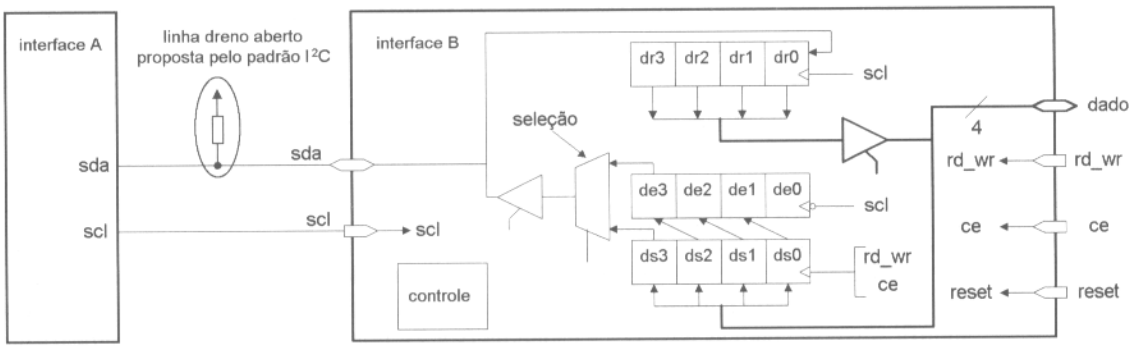


Figura 10.6.5 Diagrama de blocos de uma segunda proposta de comunicação serial.

CAPÍTULO

11 PADRÃO IEEE 1076.3

O padrão IEEE 1076.3 define tipos numéricos na forma de vetores compostos por elementos do tipo " bit " e " std_logic ". No padrão são definidas, também, funções aritméticas, lógicas e de conversão para esses tipos numéricos, visando à síntese da descrição. Dessa forma, operações aritméticas realizadas entre tipos " integer ", como soma e subtração, podem ser, também, executadas entre vetores compostos por elementos do tipo " bit " e " std_logic ".

11.1 Tipos definidos

No padrão são definidos dois tipos numéricos na forma de vetores denominados " unsigned " e " signed ". Esses tipos representam números compostos por elementos do tipo " bit " ou " std_logic ", onde os elementos mais significativos ocupam as posições à esquerda (ver Figura 11.1.1). No caso do vetor " signed ", os números são representados no formato complemento de dois, e o elemento mais à esquerda representa o sinal.

O padrão é composto de dois pacotes denominados " numeric_bit " e " numeric_std ". O pacote " numeric_bit " emprega tipos " bit " e " bit_vector ", e o pacote " numeric_std " opera com tipos " std_logic " e " std_logic_vector ". Deve-se observar que os dois pacotes não podem ser empregados conjuntamente, porque ambos denominam os tipos numéricos como " unsigned " e " signed ". Se empregado o pacote " numeric_std ", os tipos " unsigned " e " signed " são compostos por elementos do tipo " std_logic "; caso opte-se pelo pacote " numeric_bit ", os tipos " unsigned " e " signed " contêm elementos do tipo " bit ".

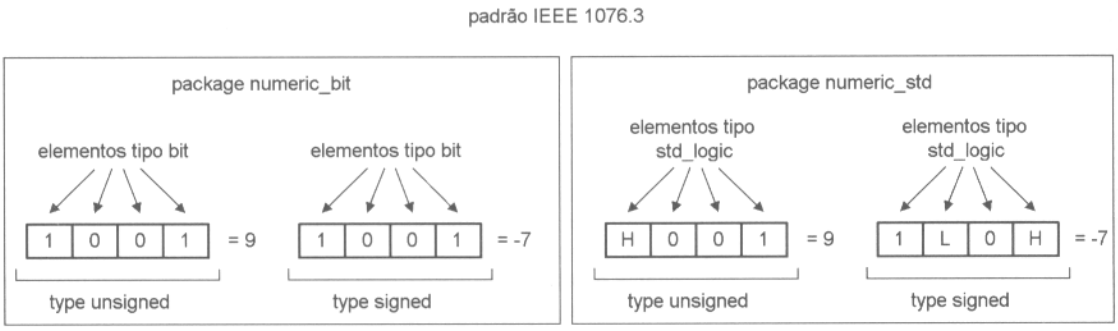


Figura 11.1.1 Exemplo de vetores definidos nos pacotes " numeric_bit " e " numeric_std ".

11.2 Operações definidas

Assim como no padrão IEEE 1164, várias funções definidas nos pacotes " numeric_bit " e " numeric_std " possuem a mesma designação de operações predefinidas, como, por exemplo, " + ", " - ", " and ", " not " etc. As funções sobrecarregam os operadores do pacote padrão, e o tipo dos parâmetros determina qual função é empregada.

As funções disponíveis no padrão são apresentadas nas tabelas a seguir. A Tabela 11.2.1 contém a lista de funções para operadores aritméticos com um único parâmetro, e as Tabelas 11.2.2, 11.2.3, 11.2.4 e 11.2.5, a lista

de funções para operadores aritméticos com dois parâmetros. As Tabelas 11.2.6 e 11.2.7 contêm as funções de operadores lógicos e operadores relacionais.

Tabela 11.2.1 Funções para operações aritméticas com um parâmetro.

função	parâmetro	retorna
abs -	signed	signed

Tabela 11.2.2 Funções para operações aritméticas " + " e " - ".

função	parâmetro L	parâmetro R	retorna
+ -	unsigned	unsigned	unsigned
	signed	signed	signed
	unsigned	natural	unsigned
	natural	unsigned	unsigned
	integer	signed	signed
	signed	integer	signed

Tabela 11.2.3 Funções para operação aritmética " * ".

função	parâmetro L	parâmetro R	retorna
*	unsigned	unsigned	unsigned
	signed	signed	signed
	unsigned	natural	unsigned
	natural	unsigned	unsigned
	signed	integer	signed
	integer	signed	signed

Tabela 11.2.4 Funções para operação aritmética " / ".

função	parâmetro L	parâmetro R	retorna
/	unsigned	unsigned	unsigned
	signed	signed	signed
	unsigned	natural	unsigned
	natural	unsigned	unsigned
	signed	integer	signed
	integer	signed	signed

Tabela 11.2.5 Funções para operações aritméticas " rem " e " mod ".

função	parâmetro L	parâmetro R	retorna
rem mod	unsigned	unsigned	unsigned
	signed	signed	signed
	unsigned	natural	unsigned
	natural	unsigned	unsigned
	signed	integer	signed
	integer	signed	signed

Tabela 11.2.6 Funções para operações lógicas.

função	parâmetro L	parâmetro R	retorna
not and or xor nand nor xnor	unsigned	unsigned	unsigned
	signed	signed	signed
Nota a: A função " not " possui apenas um parâmetro. Nota b: A função " xnor " não é suportada na versão VHDL-1987.			

Tabela 11.2.7 Funções para operações relacionais.

função	parâmetro L	parâmetro R	retorna
> < <= >= = /=	unsigned	unsigned	boolean
	signed	signed	boolean
	unsigned	natural	boolean
	natural	unsigned	boolean
	integer	signed	boolean
	signed	integer	boolean

A Tabela 11.2.8 relaciona as funções de deslocamento e rotação disponíveis. Nas funções " shift_right ", " shift_left ", " rotate_right " e " rotate_left ", o deslocamento deve ser maior ou igual a zero. No caso das funções " srl " ou *shift right logical*, " sll " ou *shift left logical*, " ror " ou *rotate right logical* e " rol " ou *rotate left logical*, o valor do deslocamento pode ser menor do que zero. Conforme se pode observar nas Figuras 11.2.1 e 11.2.2, as operações de deslocamento e rotação das funções são muito semelhantes, exceção feita à função " shift_right ", que executa um deslocamento preservando o sinal, o elemento mais à esquerda do vetor.

As funções " srl ", " sll ", " ror " e " rol " sobrecarregam os operadores de mesmo nome do pacote padrão. Desse modo, elas podem ser invocadas empregando a notação de operador ou a notação de função. Exemplo: " sc <= x srl 1; " ou " sd <= "srl" (x,1); ". Observe que, no segundo caso, o nome da função deve estar entre aspas.

Tabela 11.2.8 Funções de deslocamento e rotação.

função	valor	deslocamento	retorna
shift_right rotate_right	unsigned	natural	unsigned
shift_left rotate_left	signed	natural	signed
srl sll	unsigned	integer	unsigned
ror rol	signed	integer	signed
Exemplos: sa <= shift_left (x, 3); sb <= rotate_left ("0010", n);		sc <= x srl 1; sd <= "srl" (x,1);	
Nota 1: As funções " sll ", " srl ", " rol " e " ror " não são suportadas na versão VHDL-1987.			
Nota 2: Para " sll ", " srl ", " rol " e " ror " é possível o emprego das notações de operador e função.			

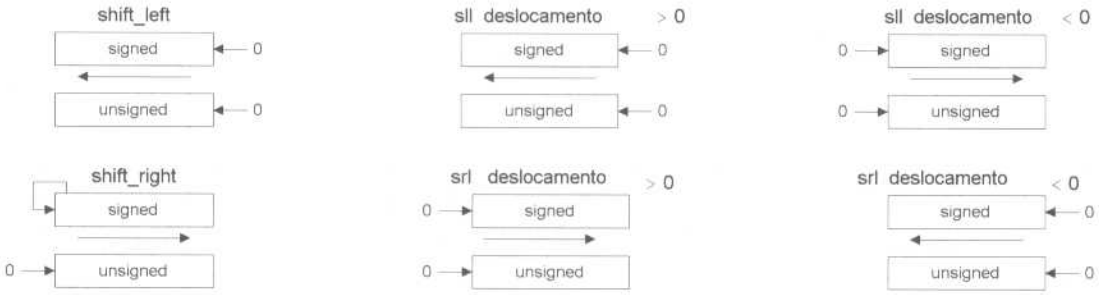


Figura 11.2.1 Operações de deslocamento executadas.

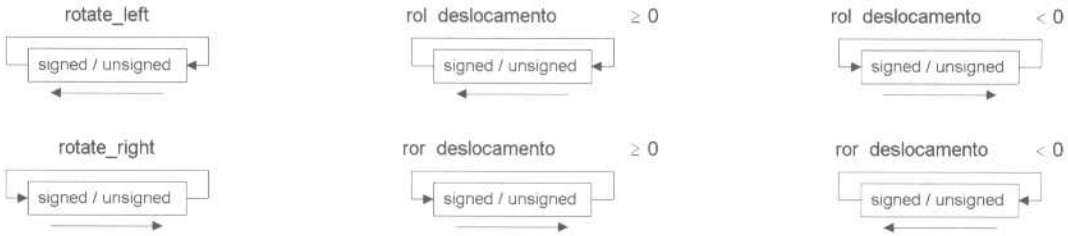


Figura 11.2.2 Operações de rotação executadas.

Operações de escalonamento no tamanho de vetores para um número maior ou menor de elementos podem ser executadas através das funções " resize " (ver Tabela 11.2.9). No caso de um parâmetro do tipo " signed ", em uma operação de escalonamento para um tamanho maior, o valor do elemento mais à esquerda do parâmetro é repetido nas novas posições à esquerda criadas. Desse modo, um número negativo representado em complemento de dois não é alterado (ver Figura 11.2.3). O escalonamento para um tamanho menor desloca o valor do sinal para a posição mais à esquerda no novo vetor.

Um valor do tipo " unsigned ", em uma operação de escalonamento para um tamanho maior, retorna com as novas posições criadas preenchidas com o valor " 0 ". No caso de uma redução, os elementos mais à esquerda são descartados.

Tabela 11.2.9 Exemplo de funções para escalonamento no tamanho de vetores.

função	parâmetro	tamanho	retorna
resize	unsigned	natural	unsigned
	signed	natural	signed
Exemplos: x_9bits <= resize(y, 9); -- x_9bits : Signed(8 Downto 0) x_5bits <= resize(y, 5); -- x_5bits : Signed(4 Downto 0)			

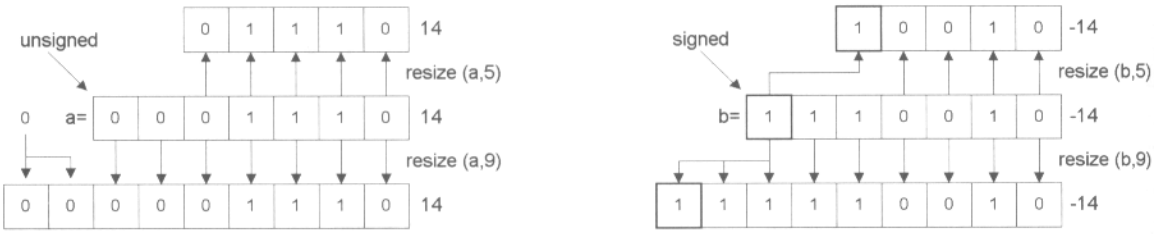


Figura 11.2.3 Operações com a função " resize ".

Conversões para tipos " natural ", " integer ", " unsigned " e " signed " podem ser executadas através das funções " to_integer ", " to_unsigned " e " to_signed ". No caso das conversões para tipos " unsigned " e " signed ", é necessário definir na função o tamanho do vetor criado. As Tabelas 11.2.10 e 11.2.11 relacionam essas funções.

Tabela 11.2.10 Funções de conversão para os tipos " natural " e " integer ".

função	parâmetro	retorna
to_integer	unsigned	natural
	signed	integer
Exemplo: x_integer <= To_integer (y_signed);		

Tabela 11.2.11 Funções de conversão para os tipos " unsigned " e " signed ".

função	parâmetro	tamanho	retorna
to_unsigned	natural	natural	unsigned
to_signed	integer	natural	signed
Exemplos: x_unsigned <= To_unsigned (y_natural, n); -- x_unsigned composto de n elementos x_signed <= To_signed (y_integer, 4); -- x_signed composto de 4 elementos			

Conforme visto, o teste de igualdade contendo valores definidos no padrão 1164 não leva em consideração o significado dos valores. No padrão 1076.3 foram incluídas funções que verificam as equivalências entre valores considerando o significado dos valores. A Tabela 11.2.12 relaciona essas funções, e a Tabela 11.2.13 apresenta o valor de retorno para cada caso. Como se pode observar, a comparação de um valor com o valor não importa, " - ", retorna sempre como verdadeiro. O mesmo ocorre entre valores com o mesmo nível lógico; " L " é equiva-

lente ao valor " 0 ", e " H " é equivalente ao valor " 1 ". Note que o tipo " std_logic " não foi incluído na relação da Tabela 11.2.12 porque ele é um subtipo de " std_ulogic ".

Tabela 11.2.12 Funções para equivalência entre objetos — tipos definidos no padrão 1164.

função	parâmetro L	parâmetro R	retorna
std_match	std_ulogic	std_ulogic	boolean
	unsigned	unsigned	boolean
	signed	signed	boolean
	std_logic_vector	std_logic_vector	boolean
	std_ulogic_vector	std_ulogic_vector	boolean

Tabela 11.2.13. Equivalência dos valores definidos no padrão 1164 para as funções " std_match ".

	U	X	0	1	Z	W	L	H	-
U	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**
X	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**
0	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	FALSE	**TRUE**
1	FALSE	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	**TRUE**
Z	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**
W	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**
L	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	FALSE	**TRUE**
H	FALSE	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	**TRUE**
-	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**

As funções " to_01 " convertem valores " H " ou " L " para os valores " 1 " ou " 0 " (ver Tabela 11.2.14). A presença dos valores " U ", " X ", " Z ", " W " ou " - " gera uma mensagem de alerta, e o valor de retorno de todos os elementos do vetor é dado pelo parâmetro opcional. O parâmetro opcional é um tipo " std_logic " com valor igual a " 0 ". Assim, caso seja inserido esse parâmetro, é necessário que os itens do pacote " std_logic_1164 " sejam visíveis à entidade.

Tabela 11.2.14 Funções de conversão para valores " 1 " ou " 0 ".

função	parâmetro	parâmetro opcional	retorna
to_01	unsigned	std_logic := '0'	unsigned
	signed	std_logic := '0'	signed
Exemplos:	x_uns_01 <= to_01(x_uns, '1'); -- retorna 1...1 caso um dos elementos diferentes de 1,0,H ou L		
	x_sig_01 <= to_01(x_sig); -- retorna 0...0 caso um dos elementos diferentes de 1,0,H ou L		

11.3 Exemplos de utilização do padrão IEEE 1076.3

Da mesma forma que o pacote " std_logic_1164 ", os pacotes " numeric_std " e " numeric_bit " são previamente compilados e armazenados na biblioteca " IEEE ". Assim, para tornar visível um desses pacotes para uma entidade, é necessária a inserção da cláusula " LIBRARY ieee " para que a biblioteca possa ser referenciada. Além disso, deve-se incluir a cláusula que define o pacote utilizado. No caso do emprego de vetores compostos de tipos

" std_logic ", deve-se tornar visível o pacote " numeric_std " através da cláusula " USE ieee.numeric_std.ALL ". Caso a descrição empregue vetores compostos por tipos " bit ", deve-se tornar visível o pacote " numeric_bit " através da cláusula " USE ieee.numeric_bit.ALL ".

O pacote " numeric_std " contém uma chamada ao pacote " std_logic_1164 " para empregar os tipos " std_logic ". Note, entretanto, que a visibilidade do pacote " std_logic_1164 " não é estendida à entidade que segue a cláusula " USE ieee.numeric_std.ALL ". Assim, caso se deseje empregar elementos do pacote " std_logic_1164 ", deve ser incluída, também, a cláusula " USE ieee.std_logic_1164.ALL " antes da declaração da entidade.

Funções aritméticas

O Quadro 11.3.1 apresenta uma descrição para o teste das funções " abs " e " - ". A Figura 11.3.1 ilustra o resultado da simulação da entidade, apresentando os dados no formato binário e decimal. Note que no caso do valor " -4 " ocorre um problema, pois não é possível representar o valor " +4 " em complemento de dois com apenas 3 bits.

```
 1  LIBRARY ieee;
 2  USE ieee.numeric_std.ALL;
 3
 4  ENTITY std1076a IS
 5    PORT (a              : IN  SIGNED(2 DOWNTO 0);
 6          s_abs, s_minus : OUT SIGNED(2 DOWNTO 0));
 7  END std1076a;
 8
 9  ARCHITECTURE teste OF std1076a IS
10  BEGIN
11    s_abs   <= abs(a);
12    s_minus <= -a;
13  END teste;
```

Quadro 11.3.1 Primeiro exemplo empregando o padrão IEEE 1076.3.

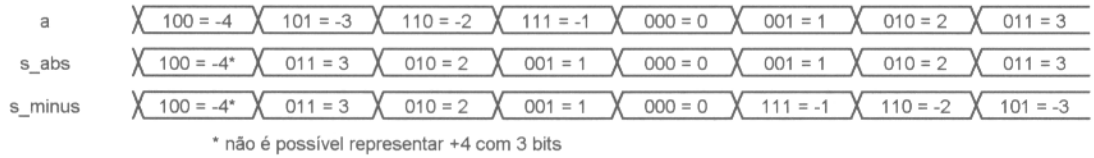


Figura 11.3.1 Simulação da entidade apresentada no Quadro 11.3.1.

Funções lógicas

As funções lógicas têm o mesmo comportamento de suas correspondentes do pacote " std_logic_1164 ". No exemplo do Quadro 11.3.2, é verificado o desempenho de algumas funções lógicas quando excitadas com diferentes valores. O resultado é apresentado na Figura 11.3.2.

```
 1  LIBRARY ieee;
 2  USE ieee.numeric_std.ALL;
 3
 4  ENTITY std1076b IS
 5    PORT( a,b                 : IN  SIGNED (7 DOWNTO 0);
 6          s_not, s_and, s_or  : OUT SIGNED (7 DOWNTO 0));
 7    END std1076b;
 8
 9  ARCHITECTURE exemplo OF std1076b IS
10  BEGIN
11    s_not <=   NOT a;
12    s_and <= a AND b;
13    s_or  <= a OR  b;
14  END exemplo;
```

Quadro 11.3.2 Operações lógicas do padrão IEEE 1076.3.

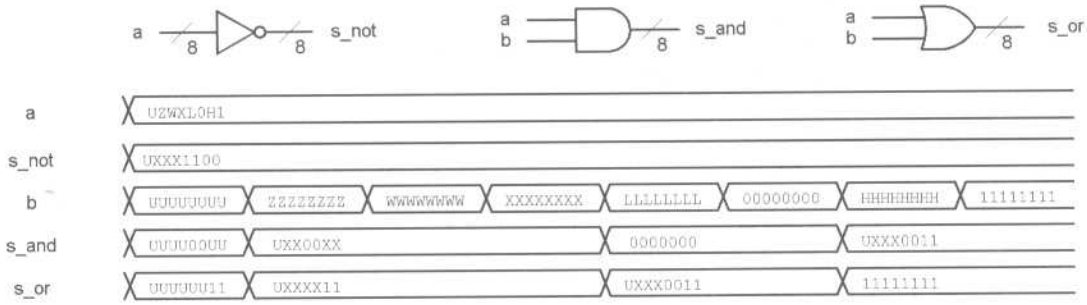


Figura 11.3.2 Simulação da entidade apresentada no Quadro 11.3.2.

Saídas com estado em alta impedância

Os tipos " unsigned " ou " signed " do pacote " numeric_std " são compostos por elementos do tipo " std_logic ", e possuem, portanto, uma função de resolução. Assim, é possível que mais de um controlador acione um sinal, conforme ilustra a descrição do Quadro 11.3.3. Nesse exemplo, o sinal " saida " pode receber o valor das portas " xa " ou " xb ", bem como ficar em um estado de alta impedância. Note que nessa descrição é necessária a inclusão da cláusula " USE ieee.std_logic_1164.ALL " devido ao emprego do tipo " std_logic " (ver linhas 3 e 7). A Figura 11.3.3 apresenta o resultado da simulação da descrição proposta.

```
 1  LIBRARY ieee;
 2  USE ieee.numeric_std.ALL;
 3  USE ieee.std_logic_1164.ALL;        -- necessario devido ao emprego do tipo std_logic
 4
 5  ENTITY std1076c IS
 6    PORT( xa, xb        : IN  SIGNED(3 DOWNTO 0);
 7          hab_a, hab_b  : STD_LOGIC;
 8          saida         : OUT SIGNED(3 DOWNTO 0));
 9    END std1076c;
10
11  ARCHITECTURE exemplo OF std1076c IS
12  BEGIN
13    saida <= xa WHEN hab_a = '1' ELSE "ZZZZ";
14    saida <= xb WHEN hab_b = '1' ELSE "ZZZZ";
15  END exemplo;
```

Quadro 11.3.3 Saídas com estado de alta impedância.

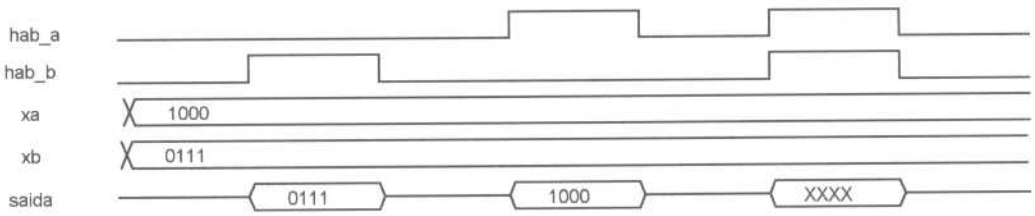


Figura 11.3.3 Simulação da entidade descrita no Quadro 11.3.3.

Operações relacionais com tipos " unsigned " e " signed "

Os resultados das operações relacionais entre tipos " unsigned " e " signed " diferem dos resultados obtidos entre tipos " bit_vector ". Em um tipo " bit_vector ", a comparação é realizada da esquerda para a direita, com os vetores alinhados à esquerda.

Nos tipos " unsigned " e " signed ", as operações consideram os valores como uma representação binária de números inteiros. A Figura 11.3.4 contém diversos exemplos que ilustram os resultados de operações relacionais entre objetos do tipo " bit_vector ", " unsigned " e " signed ".

tipos " bit_vector "	tipos " unsigned "	tipos " signed "
a_bit = 001 b_bit = 00001 a_bit > b_bit	a_uns = 001 =1 b_uns = 00001 =1 a_uns = b_uns	a_sgn = 001 = 1 b_sgn = 00001 = 1 a_sgn = b_sgn
a_bit = 100 b_bit = 01000 a_bit > b_bit	a_uns = 100 =4 b_uns = 01000 =8 a_uns < b_uns	a_sgn = 100 = -4 b_sgn = 01000 = 8 a_sgn < b_sgn
a_bit = 010 b_bit = 10000 a_bit < b_bit	a_uns = 010 =2 b_uns = 10000 =16 a_uns < b_uns	a_sgn = 010 = 2 b_sgn = 10000 = -16 a_sgn > b_sgn
a_bit = 100 b_bit = 00100 a_bit > b_bit	a_uns = 100 =4 b_uns = 00100 =4 a_uns = b_uns	a_sgn = 100 = -4 b_sgn = 00100 = 4 a_sgn < b_sgn
a_bit = 000 b_bit = 00000 a_bit < b_bit	a_uns = 000 =0 b_uns = 00000 =0 a_uns = b_uns	a_sgn = 000 = 0 b_sgn = 00000 = 0 a_sgn = b_sgn

Figura 11.3.4 Operações relacionais com tipos " bit_vector ", " unsigned " e " signed ".

11.4 Exemplo de síntese com o padrão IEEE 1076.3

No padrão IEEE 1076.3, são definidas várias funções que permitem realizar operações aritméticas. Assim como nos tipos inteiros, a síntese dessas operações nem sempre é possível de ser realizada diretamente pela ferramenta, devido à complexidade do circuito a ser gerado. Além disso, uma mesma operação aritmética pode ser implementada por diferentes técnicas. A escolha da melhor técnica deve levar em conta uma relação, geralmente conflitante, entre o custo de implementação e o desempenho final do circuito. Essas considerações fogem do escopo da linguagem.

Operações de soma e subtração são normalmente suportadas e empregam somadores do tipo "vai um serial", ou *ripple carry*, devido ao baixo custo de implementação. Uma operação de multiplicação tem um custo de implementação mais elevado e pode ser suportada por algumas ferramentas de síntese. No caso da necessidade de mais de uma operação de multiplicação num mesmo circuito, deve-se procurar um compartilhamento desse recurso no circuito, devido ao custo de implementação. A operação de divisão não é normalmente implementada; entretanto, algumas ferramentas de síntese suportam divisões por potências de dois. Nesse caso, a operação se restringe ao deslocamento do valor. As operações " mod " e " rem " não são suportadas.

Muitas vezes, a ferramenta de síntese pode lançar mão de módulos predefinidos para uma determinada tecnologia. Assim, uma vez inferida a operação a ser executada, a ferramenta emprega o módulo mais eficiente para implementar a operação.

Um outro ponto que convém ser relembrado é quanto à definição dos limites em tipos " integer " e " natural ". Uma operação aritmética com um tipo " integer " ou " natural " sem os limites definidos leva à geração de módulos com 32 bits. Assim, é importante restringir os limites de um objeto, de modo a evitar a geração de módulos com tamanhos além do necessário.

Valor "não importa" e a função " std_match "

O significado dos valores definidos no pacote " std_logic_1164 " pode ser considerado com a função " std_match ". No Quadro 11.4.1 é apresentada uma descrição para verificação dessa função em uma operação de síntese. O resultado pode ser observado na Figura 11.4.1.

Considerando o significado do valor " - ", as saídas " s1 " e " sm1 " devem ser levadas para o nível lógico alto nas condições " t(2),t(0)=0,1 " e " tm(2),tm(0)=0,1 ", e levadas para um estado de alta impedância nas condições restantes. De modo semelhante, as saídas " s2 " e " sm2 " devem, sempre, estar em nível lógico baixo.

Conforme visto no Capítulo 10, o resultado da síntese para os sinais " s1 " e " s2 " segue o comportamento do simulador. Isto é, a condição de comparação entre dois valores somente é considerada verdadeira, quando os valores forem idênticos (ver Figura 11.4.1). A ferramenta de síntese aplica essa conduta e considera a condição " t(1)='-' " sempre falsa. Assim, no circuito sintetizado têm-se as saídas " s1 " e " s2 " sempre em estado de alta impedância, e a entrada " t " não é considerada. No caso das saídas " sm1 " e " sm2 " que empregam a função " std_match ", o resultado segue o significado do valor " - ". A saída " sm1 " é controlada pela entrada " tm ", e a saída " sm2 " está permanentemente em nível lógico baixo.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY std1076h IS
 6    PORT (t,tm          : IN  STD_LOGIC_VECTOR (2 DOWNTO 0);
 7          s1,s2,sm1,sm2 : OUT STD_LOGIC);
 8  END std1076h;
 9
10  ARCHITECTURE teste OF std1076h IS
11  BEGIN
12    sm1 <= '1' WHEN std_match(tm, "0-1") ELSE 'Z';
13    sm2 <= '0' WHEN std_match(tm(1),'-') ELSE 'Z';
14
15    s1 <= '1' WHEN (t = "0-1") ELSE 'Z';
16    s2 <= '0' WHEN (t(1)= '-') ELSE 'Z';
17  END teste;
```

Quadro 11.4.1 Síntese com o valor "não importa".

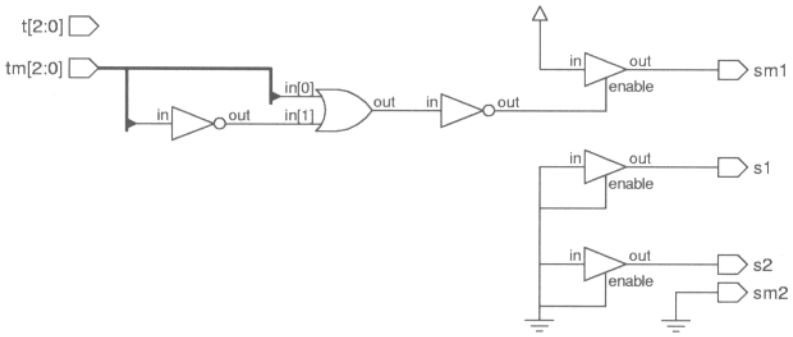


Figura 11.4.1 Circuito sintetizado a partir da descrição do Quadro 11.4.1.

Transbordo em operações de soma com tipos " unsigned " e " signed "

A ocorrência de transbordo, ou *overflow*, na operação de soma entre tipos " unsigned " e " signed " necessita de mecanismos externos para ser detectada. Uma operação de soma retorna com um vetor com dimensão igual ao parâmetro de maior tamanho. Nenhuma informação que indique o transbordo é devolvida pela função.

No caso de dois tipos " unsigned ", um artifício simples para detectar um transbordo é executar a operação de soma concatenando um bit extra de valor " 0 " com vetor do parâmetro de maior dimensão. Na Figura 11.4.2 é ilustrado o esquema no caso de parâmetros com três e quatro bits. A operação para ser realizada necessita de um somador com um bit extra (" u_tras " no exemplo) que sinaliza a ocorrência de transbordo.

O Quadro 11.4.2 contém a descrição do referido exemplo, e a Figura 11.4.3, o circuito sintetizado. Conforme se pode observar, foi empregado um somador de quatro bits, e o sinal do transbordo está ligado à saída " cout " da primitiva empregada pelo sintetizador para a operação de soma.

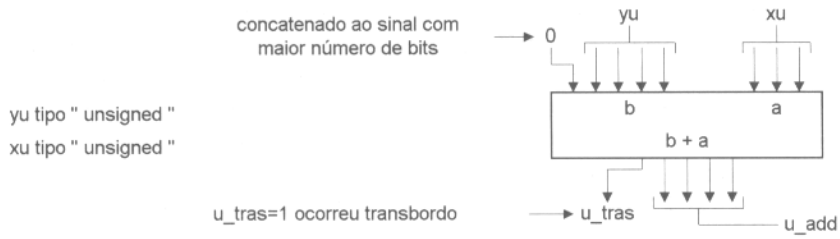


Figura 11.4.2 Detectando o transbordo numa operação de soma com tipos " unsigned ".

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY std1076o IS
 6    PORT (xu     : IN  UNSIGNED(2 DOWNTO 0);
 7          yu     : IN  UNSIGNED(3 DOWNTO 0);
 8          u_add  : OUT UNSIGNED(3 DOWNTO 0); -- u_add=xu+yu
 9          u_tras : OUT STD_LOGIC);           -- u_tras_u =1 caso ocorra transbordo em xu+yu
10  END std1076o;
11
12  ARCHITECTURE teste OF std1076o IS
13    SIGNAL u_adi : UNSIGNED(4 DOWNTO 0);  -- sinal interno contendo resultado
14  BEGIN
15    u_adi  <= xu + ('0'& yu);
16    u_tras <= u_adi(4);
17    u_add  <= u_adi(3 DOWNTO 0);
18  END teste;
```

Quadro 11.4.2 Detectando o transbordo numa operação de soma com tipos " unsigned ".

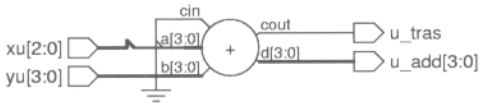


Figura 11.4.3 Circuito sintetizado com detecção de transbordo com tipos " unsigned ".

No caso de dois tipos " signed ", as operações executadas entre os dois números podem ser de soma ou subtração. Para dois vetores de mesmo tamanho, a operação entre dois números com sinais diferentes nunca leva a um transbordo. No caso de dois números de mesmo sinal o transbordo pode ocorrer. Conforme ilustrado na Figura 11.4.4, o transbordo pode ser detectado verificando-se a ocorrência de uma mudança no sinal do resultado. Por exemplo, dois números positivos somados devem resultar num número positivo; no caso de um transbordo, o bit mais à esquerda vai para o valor " 1 ", indicando assim um número negativo. O mesmo raciocínio pode ser aplicado no caso de dois números negativos.

A descrição para o caso da soma de dois valores com quatro bits é apresentada no Quadro 11.4.3. A construção " WITH SELECT " implementa um decodificador que detecta as duas condições de transbordo. A Figura 11.4.5 ilustra o circuito sintetizado, nível RTL. O sinal " s_trans " emprega dois comparadores de três bits para verificar a ocorrência das duas condições de transbordo. O bloco de soma corresponde a um somador de três bits, e o sinal de transbordo do bloco não é utilizado.

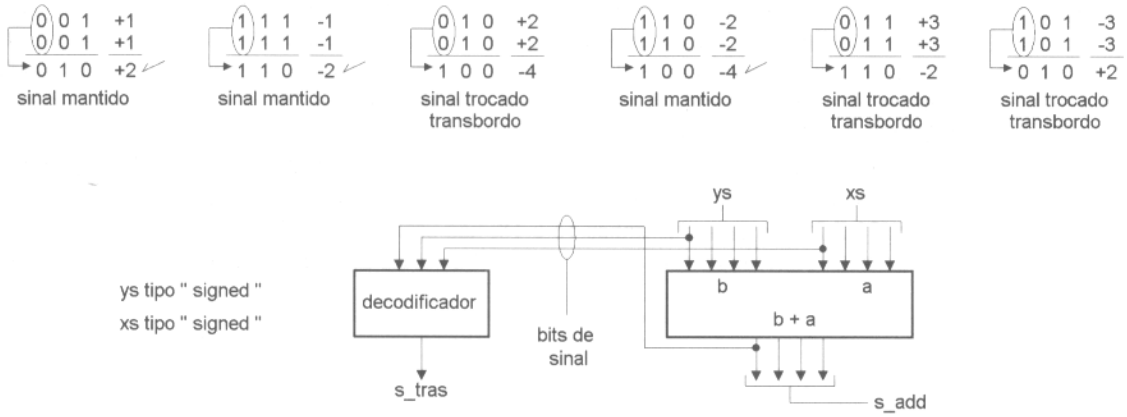


Figura 11.4.4 Exemplo de operações em complemento de dois com três bits.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3 USE ieee.numeric_std.ALL;
 4
 5 ENTITY std1076g IS
 6   PORT (xs     : IN     SIGNED (3 DOWNTO 0);
 7         ys     : IN     SIGNED (3 DOWNTO 0);
 8         s_add  : BUFFER SIGNED (3 DOWNTO 0);
 9         s_tras : OUT    STD_LOGIC);
10 END std1076g;
11
12 ARCHITECTURE teste OF std1076g IS
13   SIGNAL teste : STD_LOGIC_VECTOR(2 DOWNTO 0);
14 BEGIN
15   s_add <= xs + ys;
16
17   teste <= xs(3) & ys(3) & s_add(3);
18   WITH teste SELECT                        -- decodificador
19     s_tras <= '1' WHEN "001" | "110",      -- condicoes de transbordo
20               '0' WHEN OTHERS;             -- resultado correto
21 END teste;
```

Quadro 11.4.3 Detectando o transbordo numa operação de soma com tipos " signed ".

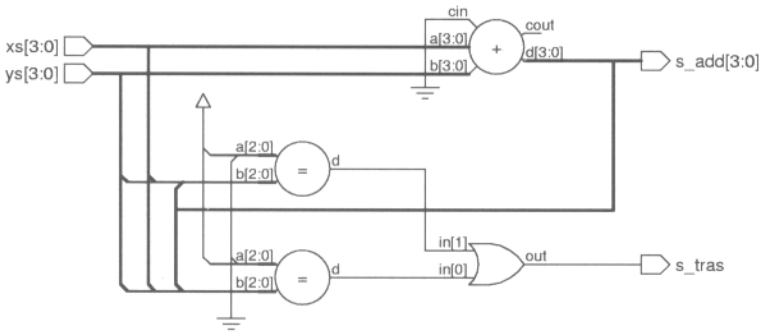


Figura 11.4.5 Circuito sintetizado com detecção de transbordo com tipos " signed ".

Circuitos síncronos

Circuitos contadores são extremamente simples e convenientes de serem descritos com os pacotes " numeric_bit " e " numeric_std ". No Quadro 11.4.4 é apresentado um contador decimal com inicialização assíncrona. As entradas " ck " e " rst " são do tipo " std_logic ", e a saída é do tipo " std_logic_vector ".

O contador é implementado através do sinal " q_uns ", tipo " unsigned ". O valor da contagem, entretanto, não pode ser transferido diretamente para a saída " q " porque ele é do tipo " std_logic_vetor ". Como um tipo " unsigned " é um vetor contendo valores do tipo " std_logic ", a conversão de tipo pode ser executada se os elementos forem transferidos um a um para o novo vetor. Essa transferência é executada através da construção " FOR GENERATE " (linhas 23 a 25).

Observe que a linha 17 contém a comparação " q_uns /= 10 " e a expressão " q_uns <= q_uns +1 ". Isso é possível, porque as funções " /= " e " + " permitem que um dos parâmetros seja do tipo " integer " (ver Tabelas 11.2.2 e 11.2.7). No caso das linhas 15 e 18, não seria possível substituir a expressão " q_uns <= "0000" " por " q_uns <= 0 ", devido ao fato de que o valor " 0 " é do tipo inteiro, e o sinal " q_uns " é tipo " unsigned ".

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY std1076w IS
 6    PORT (ck, rst  : IN  STD_LOGIC;
 7          q        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
 8  END std1076w;
 9
10  ARCHITECTURE teste OF std1076w IS
11    SIGNAL q_uns : UNSIGNED(3 DOWNTO 0);
12  BEGIN
13    abc: PROCESS (ck, rst)
14    BEGIN
15      IF rst = 1 THEN q_uns <= "0000";
16      ELSIF rising_edge(ck) THEN
17        IF q_uns /= 10 THEN q_uns <= q_uns +1; -- soma unsigned + integer possivel
18        ELSE                q_uns <= "0000";
19        END IF;
20      END IF;
21    END PROCESS abc;
22
23    converte: FOR i IN 0 TO 3 GENERATE
24      q(i) <= q_uns(i);
25    END GENERATE converte;
26  END teste;
```

Quadro 11.4.4 Contador decimal com saída do tipo " std_logic_vector ".

A Figura 11.4.6 ilustra o resultado da síntese do contador descrito no Quadro 11.4.4 nível RTL e nível porta lógica. Conforme se pode notar, no nível RTL, o contador e a pequena rotina de conversão de tipo foram corretamente interpretados pela ferramenta de síntese. No esquema nível porta lógica, o bloco " lpm_conter_4_300_0 " é um módulo contador disponível para a tecnologia escolhida para a implementação. A ferramenta de posicionamento deve empregar elementos internos para a geração mais eficiente do contador. O bloco " F4_LUT " é uma das formas de implementar lógica combinacional nessa tecnologia. O bloco é uma tabela na qual pode ser programada qualquer função combinacional de três variáveis.

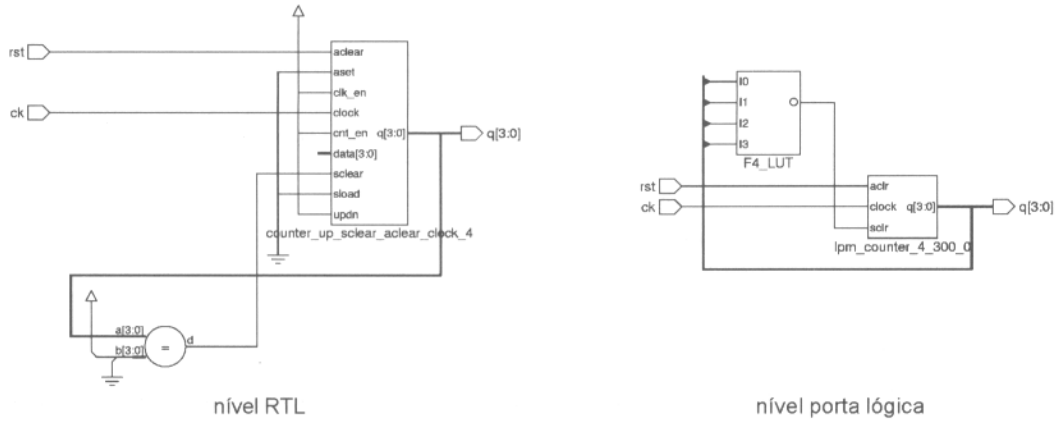


Figura 11.4.6 Contador sintetizado, nível RTL e porta lógica.

O Quadro 11.4.5 contém a descrição de dois registradores de deslocamento com carga síncrona. O objetivo desse código é ilustrar a operação de uma das funções de deslocamento apresentadas na Tabela 11.2.8. As entradas " ck ", " ld ", " d1 " e " d2 " correspondem aos sinais de relógio, carga do bloco e valor de carga. Os valores armazenados em " q1 " e " q2 " sofrem o mesmo deslocamento, tendo em vista que a função " shift_left " na linha 21 executa a mesma operação do código da linha 20.

O circuito sintetizado é apresentado na Figura 11.4.7 e, conforme se pode observar, as duas formas empregadas para a descrição do registrador de deslocamento geraram o mesmo circuito.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY std_sft IS
 6    PORT (ck, ld    :  IN     STD_LOGIC;
 7          d1, d2    :  IN     UNSIGNED (4 DOWNTO 0);
 8          q1, q2    :  BUFFER UNSIGNED (4 DOWNTO 0));
 9  END std_sft;
10
11  ARCHITECTURE teste OF std_sft IS
12  BEGIN
13    abc: PROCESS (ck)
14    BEGIN
15      IF rising_edge(ck) THEN
16        IF ld = '1' THEN
17          q1 <= d1;
18          q2 <= d2;
19        ELSE
20          q1(q1'LENGTH-1 DOWNTO 0) <= q1(q1'LENGTH-2 DOWNTO 0) & '0';
21          q2 <= shift_left(q2, 1);
22        END IF;
23      END IF;
24    END PROCESS abc;
25  END teste;
```

Quadro 11.4.5 Registradores de deslocamento com carga síncrona.

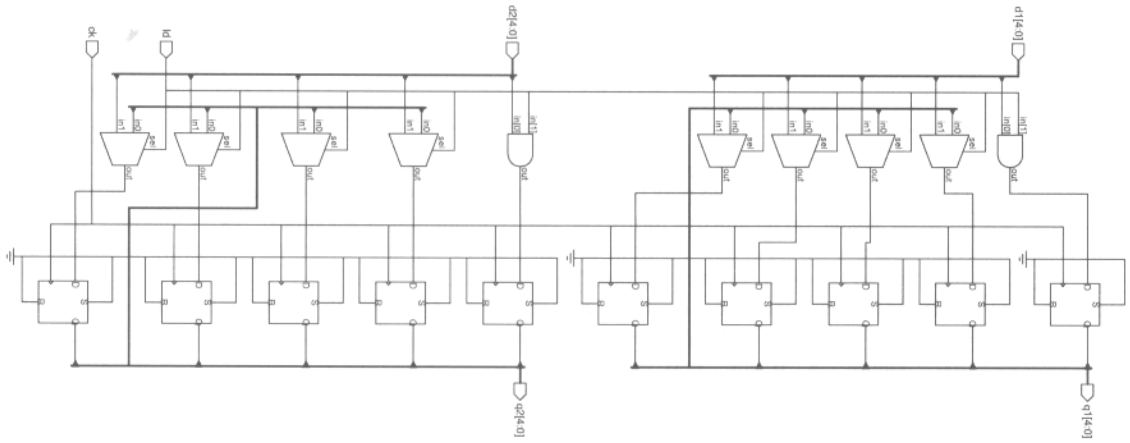


Figura 11.4.7 Circuito sintetizado, nível RTL, da descrição do Quadro 11.4.5.

11.5 Principais pontos abordados

- O padrão IEEE 1076.3 define tipos numéricos na forma de vetores compostos de elementos do tipo " bit " e " std_logic ".
- Funções aritméticas, lógicas e de conversão para os tipos numéricos são fornecidas.
- Dois tipos numéricos são definidos, " unsigned " e " signed ", para a representação de um valor numérico na forma de um vetor.

- O tipo " unsigned " representa um número sem sinal, e tipo " signed " representa um número com sinal no formato complemento de dois.
- O padrão define os pacotes denominados " numeric_bit " e " numeric_std ".
- No pacote " numeric_bit " são empregados os tipos " bit " e " bit_vector ", e no pacote " numeric_std " os tipos " std_logic " e " std_logic_vector ".
- Somente um pacote pode ser empregado por vez, porque ambos denominam os tipos numéricos do mesmo modo: " unsigned " e " signed ".
- A função " std_match " permite comparar valores do tipo " std_logic " considerando o significado desses valores.

11.6 Exercícios

11.6.1 Proponha uma descrição para verificar a operação das funções " shift_right ", " shift_left ", " rotate_right ", " rotate_left ", " srl ", " sll ", " ror " e " rol ".

11.6.2 Crie uma descrição de teste das funções " resize ", e comprove as operações de aumento e redução do tamanho em vetores do tipo " signed " e " unsigned ".

11.6.3 Verifique as operações das funções de conversão para tipos " natural ", " integer ", " unsigned " e " signed ".

11.6.4 Considerando o código do Quadro 11.6.1, comente e compare o resultado que deve ser esperado para os sinais da linha 6 e linha 7.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY std1076k IS
 6    PORT (H_igual_ni,     H_igual_1,     H_igual_H     : OUT BOOLEAN;
 7          H_std_match_ni, H_std_match_1, H_std_match_H : OUT BOOLEAN);
 8  END std1076k;
 9
10  ARCHITECTURE teste OF std1076k IS
11     CONSTANT valor_H     : STD_LOGIC := H;
12  BEGIN
13   H_igual_ni <= TRUE WHEN valor_H = - ELSE FALSE;
14   H_igual_1  <= TRUE WHEN valor_H = 1 ELSE FALSE;
15   H_igual_H  <= TRUE WHEN valor_H = H ELSE FALSE;
16
17   H_std_match_ni <= TRUE WHEN std_match(valor_H, -) ELSE FALSE;
18   H_std_match_1  <= TRUE WHEN std_match(valor_H, 1) ELSE FALSE;
19   H_std_match_H  <= TRUE WHEN std_match(valor_H, H) ELSE FALSE;
20   END teste;
```

Quadro 11.6.1 Exercício 11.6.4.

11.6.5 Apresente um código para um procedimento que implemente um contador com inicialização assíncrona e sensível a borda de subida. Os parâmetros formais do procedimento são os seguintes: " ck " (relógio), " rst " (inicialização), " max_cnt " (constante que define o valor máximo de contagem), e " q_uns " (valor da contagem). O número de bits do contador é determinado pelo tamanho do vetor " q_uns ".

Insira o procedimento dentro de um pacote, e faça o teste a partir de uma outra entidade que realiza uma chamada ao contador (ver Figura 11.6.1). Empregue os seguintes tipos nos parâmetros formais: " ck " e " rst ", tipo " std_logic ", " max_cnt " tipo " natural ", e " q_uns " tipo " unsigned ".

Sintetize o código para verificar se a ferramenta aceita a descrição proposta.

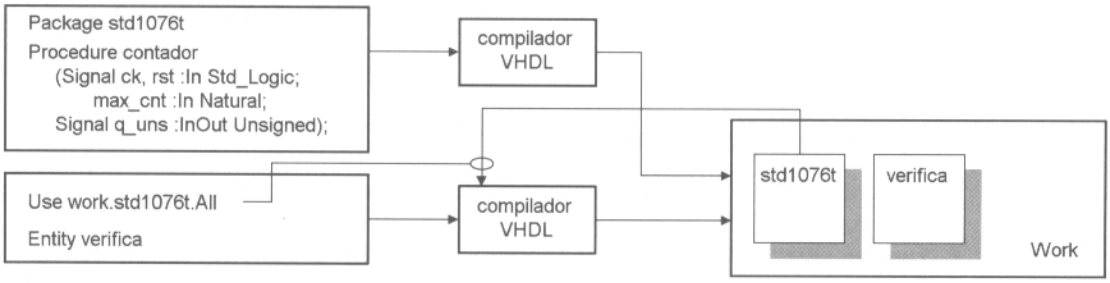


Figura 11.6.1 Exercício 11.6.5.

11.6.6 Considere uma unidade lógica aritmética com oito funções, conforme ilustrado na Figura 11.6.2. A operação executada entre os dados " rs " e " ss " é definida pelo sinal " opr ", a saída " ov " indica um transbordo numa operação de soma ou subtração, a saída " zr " indica que o resultado da operação é nulo, e entrada " ci " corresponde a um dado do tipo "vem um".

Apresente uma descrição para esse circuito, empregando para os sinais " rs " e " ss " tipo " signed " com elementos do tipo " bit ", e para os sinais restantes, tipo " bit ". O número de bits dos sinais " rs ", " ss " e " ts " deve ser definido através de uma cláusula " GENERIC " no início do código.

opr	operação	ov	zr
000	ts ← rs + ss	↑↓	↑↓
001	ts ← rs + ss + ci	↑↓	↑↓
010	ts ← rs - ss	↑↓	↑↓
011	ts ← rs - ss - ci	↑↓	↑↓
100	ts ← rs E ss	?	?
101	ts ← rs OU ss	?	?
110	ts ← rs OU-EXCLUSIVO ss	?	?
111	ts ← NAO rs	?	?

? = indeterminado

↑↓ = afetado pela operação

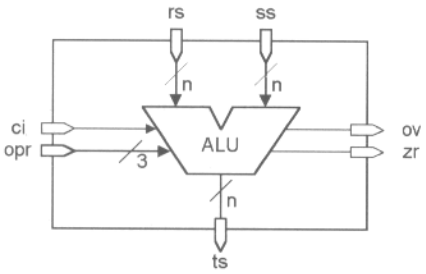


Figura 11.6.2 Exercício 11.6.6.

11.6.7 O código proposto no Quadro 11.6.2 realiza a operação " $a \times 2^n$ " com sinais do tipo inteiro. Altere o código de modo a empregar tipos " unsigned ", observando o número de bits necessário para a gama de valores que cada sinal pode assumir. Como a função " x^y " não é definida nos pacotes do padrão 1076.3, é necessário substituí-la por uma outra operação. Sugestão: Analise como a operação " 2^n " pode ser realizada através de um deslocamento (ver Figura 11.6.2).

Analise, também, o número de bits necessário para realizar as operações.

```
1  ENTITY ex_a1 IS
2    PORT(n : IN  INTEGER RANGE 0 TO 3;
3         a : IN  INTEGER RANGE 0 TO 31;
4         s : OUT INTEGER RANGE 0 TO 256);
5    END ex_a1;
6
7  ARCHITECTURE exemplo OF ex_a1 IS
8  BEGIN
9    s <= a * 2**n;
10 END exemplo;
```

Quadro 11.6.2 Código do Exercício 11.6.7.

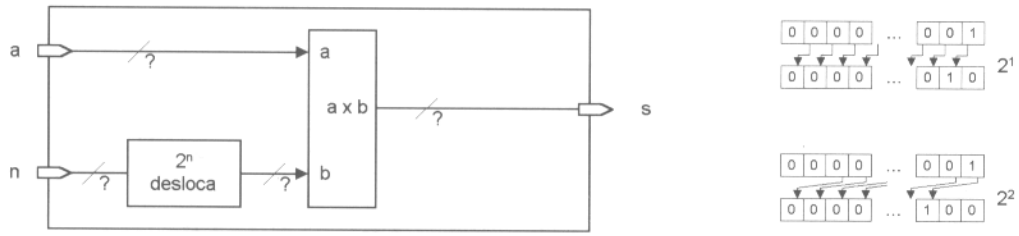


Figura 11.6.3 Sugestão para a alteração do código do Exercício 11.6.7.

11.6.8 Uma análise das operações executadas na descrição do Quadro 11.6.2 permite verificar que é possível substituir as operações executadas em " $a \times 2^n$ " por deslocamentos. Como " n " pode assumir os valores " 0 ", " 1 ", " 2 " e " 3 ", a operação " $\times 2^n$ ", na realidade, se resume em multiplicar " a " por " 1 ", " 2 ", " 4 " e " 8 ", respectivamente (ver Figura 11.6.4). Proponha uma descrição levando em consideração esse fato.

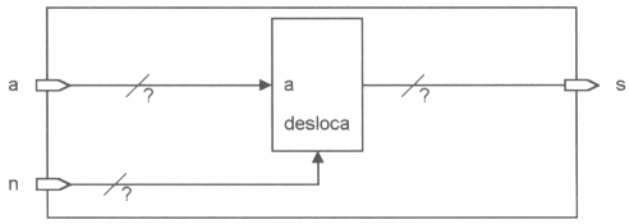


Figura 11.6.4 Nova alteração do código do Exercício 11.6.7.

11.6.9 Sintetize o código do Quadro 11.6.2 e os códigos propostos para a solução dos Exercícios 11.6.7 e 11.6.8. Analise como a ferramenta de síntese tratou a operação " 2^n " e a operação de multiplicação, para os três casos. Verifique o tamanho do circuito resultante, comparando o número de portas ou elementos equivalentes empregados.

11.6.10 Proponha um código para realizar a operação "$\sum_{i=0}^{7} a_i \times 2^{n_i}$". A Figura 11.6.5 contém o diagrama de blocos de uma possível implementação, e a Figura 11.6.6 a respectiva carta de tempos. A cada ciclo de relógio, o valor resultante da operação " $a \times 2^n$ " é acumulado no registrador " reg_sub ". Essa operação é realizada por 8 ciclos. No oitavo ciclo, o resultado da saída do somador é transferido para o registrador " reg_total ", e o conteúdo de " reg_sub " é levado a zero. A entrada " rst " é empregada para inicializar a operação.

Nessa descrição empregue tipos " INTEGER " para facilitar a análise, e determine a gama de valores que cada objeto pode assumir. Isso é necessário para economizar recursos na implementação da descrição.

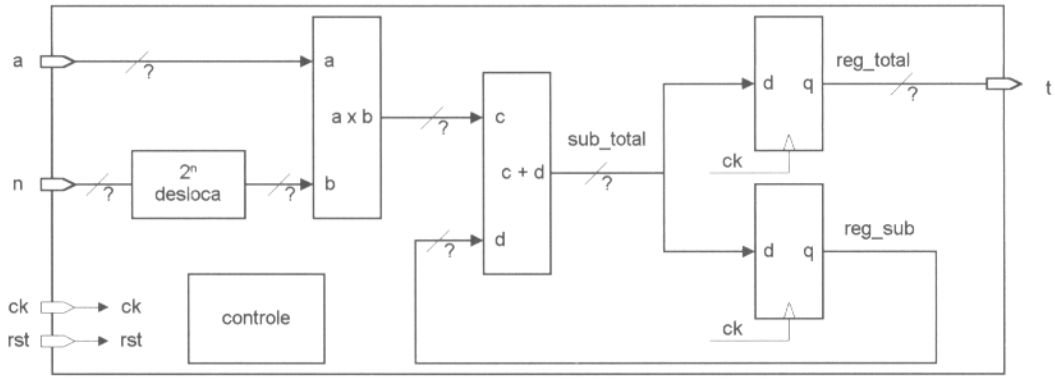


Figura 11.6.5 Diagrama de blocos para solução dos Exercícios 11.6.10 e 11.6.11.

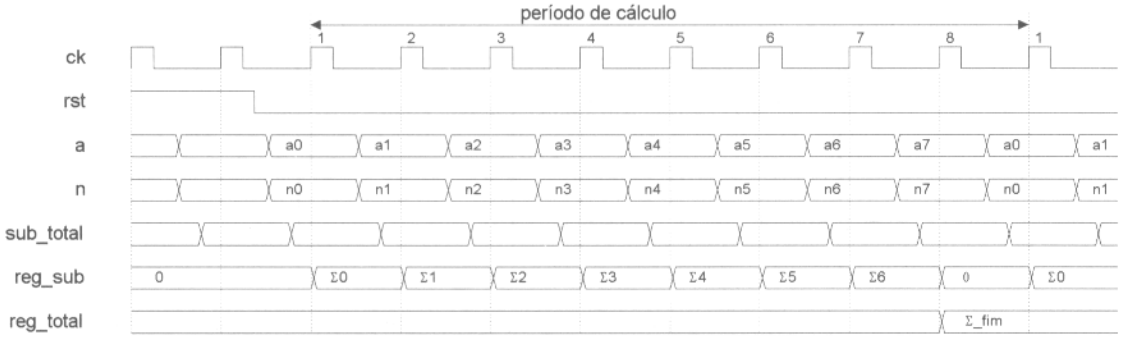


Figura 11.6.6 Carta de tempos para os Exercícios 11.6.10 e 11.6.11.

11.6.11 Altere o código proposto para o Exercício 11.6.10 de modo a empregar tipos " unsigned ". Determine o número de bits necessário para a gama de valores que cada sinal pode assumir, e complete o diagrama de blocos da Figura 11.6.5 com os valores estabelecidos.

11.6.12 O código do Quadro 11.6.5 não está correto. Para a correção, atente para o tipo do parâmetro opcional da função " to_01 ".

```
 1  LIBRARY ieee;
 2  USE ieee.numeric_std.ALL;
 3
 4  ENTITY ts_to_01 IS
 5    PORT (a  : IN  SIGNED(1 DOWNTO 0);
 6          b  : IN  UNSIGNED(1 DOWNTO 0);
 7          sa : OUT SIGNED(1 DOWNTO 0);
 8          sb : OUT UNSIGNED(1 DOWNTO 0));
 9  END ts_to_01;
10
11  ARCHITECTURE errada OF ts_to_01 IS
12    CONSTANT zero : STD_LOGIC := '0';
13  BEGIN
14    sa   <= to_01(a, '1');
15    sb   <= to_01(b, zero);
16  END errada;
```

Quadro 11.6.3 Código do Exercício 11.6.12.

CAPÍTULO

12 DECLARAÇÃO DE TIPOS E PSEUDÔNIMOS

Na linguagem VHDL, todo objeto é declarado segundo um tipo. No Capítulo 2 foram apresentados alguns tipos predefinidos no pacote padrão, e exemplos para a criação de tipos enumerados. Neste capítulo são abordadas as opções para criação de novos tipos, com mais detalhes. A declaração de um novo tipo tem várias aplicações em síntese, como a definição dos estados de máquinas seqüenciais e memórias. A declaração de pseudônimos, um nome alternativo para um item, é também apresentada.

12.1 Tipos

Um tipo representa o conjunto de valores que um objeto pode armazenar e o conjunto de operações que podem ser realizadas com esse objeto. O pacote padrão da linguagem, por exemplo, contém uma relação de tipos predefinidos e as operações associadas a cada tipo. Um objeto do tipo " BOOLEAN " pode assumir os valores " FALSE " e " TRUE ", e são definidas as operações lógicas, como " AND " e " OR ", entre objetos declarados segundo esse tipo. Uma operação de multiplicação, por sua vez, é especificada no pacote padrão para um objeto do tipo numérico; ela não é definida para um objeto do tipo " BOOLEAN ".

A Figura 12.1.1 ilustra as quatro classes de tipos da linguagem: " SCALAR ", " COMPOSITE ", " ACCESS " e " FILE ". Os tipos escalares são ordenados e os valores não contêm outros elementos. Nos tipos compostos os valores contêm outros elementos: no caso " ARRAY ", os elementos são do mesmo tipo, e no caso " RECORD " os elementos podem ser de tipos diferentes. O tipo " ACCESS " é empregado para valores alocados dinamicamente. O tipo " FILE " é empregado para representação de arquivos: um objeto da classe " FILE " contém valores armazenados no sistema de arquivos do ambiente de simulação e compilação. As operações com arquivos são apresentadas no Capítulo 13.

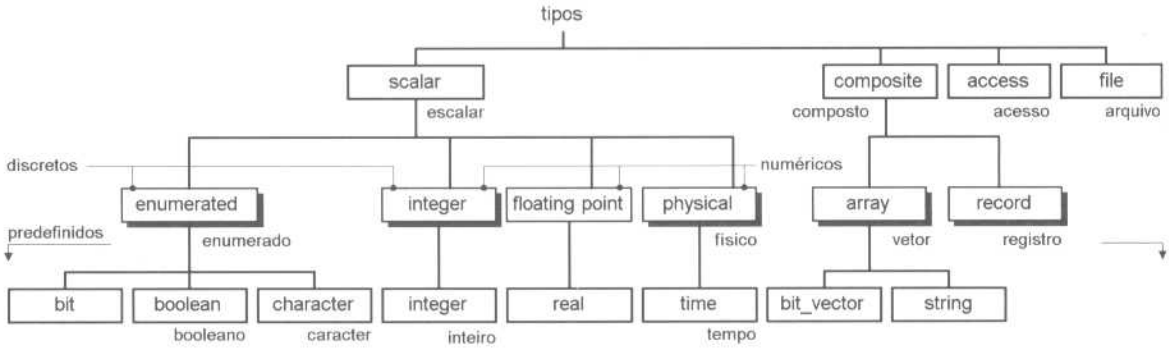


Figura 12.1.1 Declaração de novos tipos abordados no capítulo.

12.2 Definição de tipos enumerados

Os tipos enumerados pertencem à classe dos tipos escalares, sendo predefinidos no pacote padrão os tipos " BIT ", " BOOLEAN " e " CHARACTER ". A declaração de um novo tipo enumerado segue o exemplo apresentado no

Quadro 12.2.1. Os valores que o objeto pode assumir são apresentados ordenadamente na especificação. O primeiro elemento declarado no tipo corresponde ao menor valor que um objeto desse tipo pode assumir, e o último elemento, o maior valor.

Uma mesma designação pode ser empregada na definição de tipos enumerados distintos. Por exemplo, os tipos " BIT " e " STD_LOGIC " possuem os valores designados como " 0 " e " 1 ". Nesse caso, os designadores sofrem um sobrecarregamento.

Conforme visto no Capítulo 6, a declaração de tipos enumerados é muito útil para definição de estados em máquinas de estado finito. Os valores que o tipo pode assumir ficam associados a condições da máquina, facilitando a interpretação do código e operação da máquina. As ferramentas de síntese suportam a declaração de tipos enumerados, e permitem estabelecer diferentes modos de codificação dos valores definidos para o tipo.

```
TYPE temperatura IS (baixa, media, alta);
TYPE cores IS  ('R', 'G', 'B');
```

Quadro 12.2.1 Declaração de tipo enumerado.

12.3 Definição de tipos físicos

Conforme visto, os tipos físicos pertencem, também, à classe dos escalares, e podem ser empregados para representação de grandezas físicas, como tensão, corrente, velocidade, etc. Nesse tipo, é definida uma unidade básica através da palavra reservada " UNITS " e unidades sucessivas múltiplas desta. Os limites de valores que o tipo pode assumir são dependentes da implementação, sendo assegurada a faixa entre −2147483647 a +2147483647. A declaração de tipos físicos não é suportada pelas ferramentas de síntese.

O tipo " TIME ", declarado no pacote padrão para a grandeza tempo, tem como unidade básica o valor " fs ". No Quadro 12.3.1 é ilustrado o exemplo da declaração de um tipo físico denominado " resistencia ", tendo como unidade básica o valor " ohm ".

```
TYPE resistencia IS RANGE 0 TO 1E7
  UNITS ohm;
        Kohm = 1000 ohm;
        Mohm = 1000 ohm;
  END UNITS;
```

Quadro 12.3.1 Declaração de tipo físico.

12.4 Definição de subtipos

A declaração de um subtipo estabelece um novo limite no campo definido. Os tipos não são diferentes; apenas o limite é redefinido. Assim, a transferência de valores entre objetos de um tipo e subtipo é permitida. A declaração de um subtipo é feita através da palavra reservada " SUBTYPE ", seguida do nome do subtipo, do nome do tipo ao qual ele pertence e dos novos limites estabelecidos.

No Quadro 12.4.1 são apresentados os subtipos escalares " natural " e " positive " declarados no pacote padrão.

```
TYPE    integer  IS RANGE -2147483648 TO 2147483647;
SUBTYPE natural  IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS integer RANGE 1 TO integer'HIGH;
```

Quadro 12.4.1 Subtipos declarados no pacote padrão.

No Quadro 12.4.2 é apresentado um código para teste de transferência de valores entre objetos de um tipo e um subtipo. A operação na linha 18 é válida para qualquer valor transferido para " b_type ". Na linha

19, caso " b_sbtype " receba um valor diferente de " vermelho ", " verde ", ou " azul ", será gerada uma mensagem de erro no instante da simulação, pois o valor não pertence aos limites definidos para o subtipo " cores_primarias ".

```
1   -- arquivo array_t3.vhd
2   PACKAGE tipos IS
3     TYPE cores IS (vermelho, verde, azul, marrom, laranja, amarelo, violeta, branco);
4     SUBTYPE cores_primarias IS cores RANGE vermelho TO azul;
5   END tipos;
6
7   USE WORK.tipos.ALL;
8
9   ENTITY array_t3 IS
10    PORT (a_type   : IN  cores;
11          b_type   : OUT cores;
12          b_sbtype : OUT cores_primarias);
13  END array_t3;
14
15  ARCHITECTURE teste OF array_t3 IS
16
17  BEGIN
18    b_type   <= a_type;  -- valida para qualquer valor
19    b_sbtype <= a_type;  -- valida para os valores: vermelho, verde e azul
20  END teste;
```

Quadro 12.4.2 Código empregando subtipo.

12.5 Definição de tipos vetor

Um objeto do tipo composto da classe vetor agrupa elementos de um mesmo tipo. Em um vetor unidimensional, a identificação de um elemento é feita por meio de um índice que determina a posição do elemento. No caso de um vetor multidimensional, um conjunto de índices identifica o elemento. Em geral, vetores multidimensionais não são suportados pelas ferramentas de síntese.

Declaração de um vetor

A declaração de um vetor contém:

- o nome do vetor (após a palavra reservada " TYPE ");
- os limites do índice (após a palavra reservada " ARRAY ");
- o tipo (após a palavra reservada " OF ").

Os limites do índice especificam o número de elementos do vetor, e o tipo determina o tipo dos elementos que compõem o vetor (ver exemplos no Quadro 12.5.1).

Os limites do índice são estabelecidos por tipos discretos (tipos enumerados ou tipo inteiro). A palavra reservada " RANGE " pode ser empregada para restringir as dimensões do vetor (ver linhas 6 e 11 do Quadro 12.5.1).

As declarações de vetores propostas no Quadro 12.5.1 resultam em cinco tipos distintos, de características idênticas. Todos possuem as mesmas dimensões, empregam o mesmo tipo para o valor do índice e contêm elementos do tipo " BIT ".

A Figura 12.5.1 ilustra um outro exemplo de declaração de tipos vetor contendo elementos do tipo " CHARACTER ". Observe o emprego de agregados para a definição de valores na constante " az " e no sinal " ay ".

O Quadro 12.5.2 contém exemplos de declaração de vetores unidimensionais com limites definidos por um tipo enumerado. O tipo " cores ", declarado na linha 5, contém três valores: " vermelho ", " verde " e " azul ". Os tipos " vetor_dx " e " vetor_dy " são vetores compostos de elementos do tipo " CHARACTER " cujo índice é o tipo " cores ". A Figura 12.5.2 ilustra o conteúdo dos objetos declarados no código.

```
1  ENTITY teste_b IS
2  END    teste_b;
3
4  ARCHITECTURE teste OF teste_b IS
5    TYPE vetor_bx IS ARRAY (0 TO 5) OF BIT;                -- limites: definidos por um tipo INTEGER
6    TYPE vetor_by IS ARRAY (INTEGER RANGE 0 TO 5) OF BIT;
7
8    SUBTYPE s_int IS INTEGER RANGE 0 TO 5;
9    TYPE vetor_bz IS ARRAY (s_int) OF BIT;                 -- limites: definidos por um tipo
10   TYPE vetor_bk IS ARRAY (0 TO s_int'HIGH) OF BIT;
11   TYPE vetor_bw IS ARRAY (s_int RANGE 0 TO s_int'HIGH) OF BIT;
12
13   SIGNAL bx : vetor_bx;  -- objetos declarados segundo os vetores definidos
14   SIGNAL by : vetor_by;
15   SIGNAL bz : vetor_bz;
16   SIGNAL bk : vetor_bk;
17   SIGNAL bw : vetor_bw;
18 BEGIN
19   bx  <= "111101";  by  <= "111011";
20
21   bz  <= "110111";  bk  <= "101111";   bw  <= "011111";
22 END teste;
```

Quadro 12.5.1 Declaração de vetores unidimensionais contendo elementos tipo " BIT ".

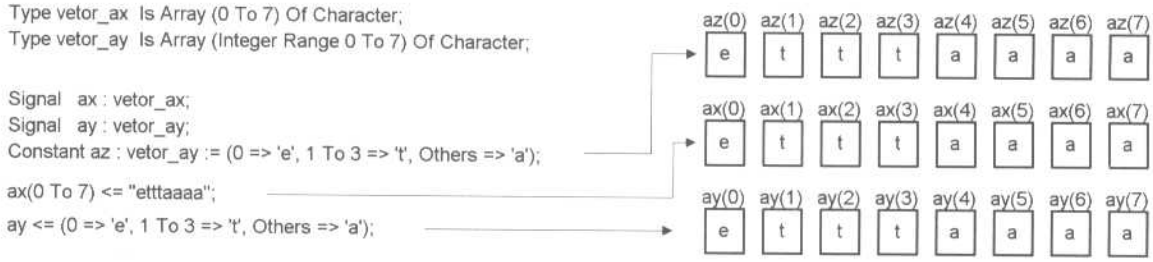


Figura 12.5.1 Ilustração de vetores unidimensionais contendo elementos tipo " CHARACTER ".

```
1  ENTITY teste_d2 IS
2  END teste_d2;
3
4  ARCHITECTURE teste OF teste_d2 IS
5    TYPE cores IS  (vermelho, verde, azul);  -- definicao de um tipo enumerado
6
7    -- vetores com limite definido por tipo enumerado:
8    TYPE vetor_dx  IS ARRAY (cores) OF CHARACTER;
9    TYPE vetor_dy  IS ARRAY (cores RANGE vermelho TO azul) OF CHARACTER;
10   TYPE vetor_dz  IS ARRAY (vermelho TO azul) OF CHARACTER;
11
12   SIGNAL   dx: vetor_dx;
13   SIGNAL   dy: vetor_dy;
14   SIGNAL   dz: vetor_dz;
15   CONSTANT cz: vetor_dz := ('R','G','B');
16
17 BEGIN
18   dx(vermelho) <= 'R';  dx(verde) <= 'G'; dx(azul) <= 'B';
19   dy(vermelho TO azul) <= ('R','G','B');
20   dz <= cz;
21 END teste;
```

Quadro 12.5.2 Declaração de um vetor unidimensional — índice: tipo declarado no código.

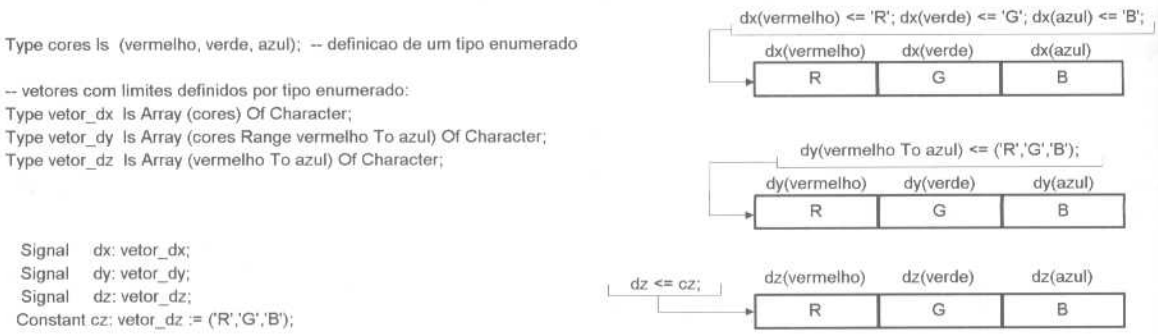


Figura 12.5.2 Ilustração dos vetores criados a partir da descrição do Quadro 12.5.2.

Definição de um vetor com limites em aberto

Na declaração de um vetor com os limites em aberto, *unconstrained array*, o número de elementos não é especificado na declaração do tipo. Os limites são definidos posteriormente:

- na declaração de um objeto que emprega o tipo com os limites em aberto;
- na declaração de um subtipo (que tem como base o tipo com os limites em aberto);
- ou em um subprograma.

Neste último caso, os limites são definidos a partir do parâmetro real passado ao subprograma.

A declaração do tipo é semelhante à declaração de um vetor com os limites definidos. Após a palavra reservada " RANGE ", é introduzido o delimitador " <> ", denominado *box*, para indicar o limite em aberto.

No Quadro 12.5.3 são apresentados os vetores " BIT_VECTOR " e " STRING ", predefinidos no pacote padrão, e a declaração de um tipo equivalente ao tipo " BIT_VECTOR ". Observe como os limites são estabelecidos na declaração do sinal " x1 ". Eles são determinados da mesma forma empregada na definição dos limites de um tipo " BIT_VECTOR ".

```
TYPE bit_vector IS ARRAY (NATURAL  RANGE <>) OF bit;          -- declarado no pacote std
TYPE string     IS ARRAY (POSITIVE RANGE <>) OF character;    -- declarado no pacote std

TYPE   vetor_bit IS ARRAY (NATURAL  RANGE <>) OF BIT;       -- tipo equivalente ao bit_vector
SIGNAL x1 : vetor_bit(3 DOWNTO 0);                          -- objeto que emprega o tipo
```

Quadro 12.5.3 Vetores declarados no pacote "std " e a declaração de um vetor.

No Quadro 12.5.4 são declarados um tipo e um subtipo no pacote " tipos " (linhas 2 a 5). Ambos são vetores compostos de elementos do tipo " CHARACTER ". O número de elementos do tipo " sem_limite " não é especificado. O subtipo " com_limite ", por sua vez, contém quatro elementos na faixa de " 0 " a " 3 ".

A declaração do sinal " saida_1 " na linha 10 não necessita da definição dos limites, pois essa informação está contida na declaração do subtipo a que ele pertence, " com_limite ". No caso do sinal " saida_2 ", na linha 11, é necessária a definição dos limites, dado que o tipo não especifica as dimensões do vetor.

O Quadro 12.5.5 contém uma função que reverte a posição de elementos de um vetor (ver linhas 15 a 22). Na função denominada " reverte ", o parâmetro formal é um vetor com os limites em aberto (ver linha 15). O parâmetro real passado à função define as dimensões do vetor. Note que esse tópico foi apresentado no Capítulo 8, na discussão de subprogramas com argumentos sem especificação de limites.

```
1  -- aquivo teste2a.vhd
2  PACKAGE tipos IS
3   TYPE    sem_limite IS ARRAY(NATURAL RANGE <>) OF CHARACTER; -- declarando o tipo
4   SUBTYPE com_limite IS sem_limite(0 TO 3);                   -- sub-tipo estabelece os limites
5  END tipos;
6
7  USE WORK.tipos.ALL;
8
9  ENTITY teste2a IS
10   PORT(saida_1 : OUT com_limite;              -- limites definidos pelo sub_tipo
11        saida_2 : OUT sem_limite(4 DOWNTO 0)); -- objeto define os limites do tipo
12 END teste2a;
13
14 ARCHITECTURE teste OF teste2a IS
15 BEGIN
16   saida_1 <= ('a','b','c','d');      -- transf. valores para o objeto do tipo com_limite
17   saida_2 <= ('a','b','c','d','f'); -- transf. valores para o objeto do sub-tipo sem_limite
18 END teste;
```

Quadro 12.5.4 Vetor com os limites em aberto — limites definidos por subtipo e objeto.

```
1  -- aquivo teste2b.vhd
2  PACKAGE tipos IS
3   TYPE    sem_limite IS ARRAY(NATURAL RANGE <>) OF CHARACTER; -- limites nao definidos
4  END tipos;
5
6  USE WORK.tipos.ALL;
7
8  ENTITY teste2b IS
9    PORT(entrada : IN  sem_limite(4 DOWNTO 0); -- ou (0 TO 4) objeto define os limites do tipo
10        saida   : OUT sem_limite(4 DOWNTO 0); -- ou (0 TO 4) objeto define os limites do tipo
11 END teste2b;
12
13 ARCHITECTURE teste OF teste2b IS
14
15 FUNCTION reverte (vetor: sem_limite) RETURN sem_limite IS  -- parametro real define os limites
16   VARIABLE vetor_reverso : sem_limite(vetor'RANGE);
17 BEGIN
18   FOR i IN vetor'RANGE LOOP
19      vetor_reverso(vetor'LENGTH-1-i):= vetor(i);
20   END LOOP;
21   RETURN vetor_reverso;
22 END reverte;
23
24 BEGIN
25   saida <= reverte(entrada);  -- parametro real "entrada" com limites definidos
26 END teste;
```

Quadro 12.5.5 Vetor com os limites em aberto — limites definidos por subprograma.

Vetor composto de elementos do tipo vetor

A Figura 12.5.3 apresenta alguns exemplos de vetores compostos com elementos do tipo vetor, e declarações de constantes empregando os vetores. Esse tipo de declaração é suportado pela maioria das ferramentas de síntese, e permite a criação de objetos com uma organização equivalente a um arranjo. Objetos criados dessa forma são muito úteis na descrição de memórias.

No exemplo, o vetor " vetor_2d " é composto de oito elementos do tipo " bit_vector ", contendo cada elemento quatro bits. O vetor " vetor_3d " é composto de três elementos do tipo " vetor_2d ". Esse arranjo cria uma estrutura equivalente a um arranjo de três dimensões. Observe, também, as diversas opções para a definição de valores das constantes empregando parte de vetores e agregados.

O Quadro 12.5.6 contém exemplos de transferência de valores entre os vetores compostos de elementos do tipo vetor. A seleção de um elemento do vetor é feita através de índices separados por parênteses. Por exemplo, um único elemento de " vetor_2d " pode ser selecionado na forma " objeto(indice_v2)(indice_v1) ", onde " indice_v1 " corresponde ao índice do vetor primeiramente declarado. De modo semelhante, um elemento de " vetor_3d " pode ser identificado na forma " objeto(indice_v3)(indice_v2)(indice_v1) ".

```
Type vetor_2d  Is Array (0 To 7) Of Bit_Vector(3 Downto 0);
Type vetor_3d  Is Array (0 T0 2) Of vetor_2d;

Constant va: vetor_2d := (('1','0','0','0'), Others => ('1','0','1','0'));
Constant vb: vetor_2d := (("1000"),Others => ("1010"));
Constant vc: vetor_2d := (0=>('1','1','0','0'), Others => ('1','0','0','1'));
Constant vd: vetor_2d := (("1100"), Others => ('1','0','0','1'));
Constant ve: vetor_2d := (0 To 2 =>('0','1','1','0'), Others => ('1','0','1','1'));
Constant vf: vetor_2d := (Others => ('1','0','1','1'));
Constant vg: vetor_2d := (Others => (Others => ('1'))); -- preenche com mesmo valor
Constant vh: vetor_3d := (Others => (Others => (Others => ('1'))));
Constant vi: vetor_3d := ((Others => ('0','0','0','0')),(Others => ('0','0','0','1')),(Others => ('0','0','1','0')));
Constant vj: vetor_3d := (("0000", Others =>"0000"),("0001", Others =>"0001"),(Others =>"0010"));
```

ve(7)	1	0	1	1
ve(6)	1	0	1	1
ve(5)	1	0	1	1
ve(4)	1	0	1	1
ve(3)	1	0	1	1
ve(2)	0	1	1	0
ve(1)	0	1	1	0
ve(0)	0	1	1	0

Figura 12.5.3 Declaração de vetores compostos de elementos do tipo vetor.

A declaração de vetores nessa forma permite selecionar, de maneira eficiente, conjuntos de elementos, por meio de agregados e parte de vetores (ver exemplos no Quadro 12.5.6 e nas Figuras 12.5.4 e 12.5.5). A declaração de tipos multidimensionais, que será apresentada a seguir, é menos flexível nesse aspecto.

```
 1  ENTITY teste_e1 IS
 2  END teste_e1;
 3
 4  ARCHITECTURE teste OF teste_e1 IS
 5    TYPE vetor_2d  IS ARRAY (0 TO 7) OF BIT_VECTOR(3 DOWNTO 0);
 6    TYPE vetor_3d  IS ARRAY (0 TO 2) OF vetor_2d;
 7
 8    SIGNAL   s_2d, t_2d: vetor_2d;
 9    CONSTANT c_2d: vetor_2d := (0 TO 2 =>('0','0','0','0'), OTHERS => ('1','0','1','1'));
10    SIGNAL   s_3d, t_3d: vetor_3d;
11
12  BEGIN
13    s_2d(7)(2) <= c_2d(7)(1);                    -- 1 elemento
14    s_2d(3) <= "1000";                           -- 1 indice
15    s_2d(4 TO 6) <= ("1010", OTHERS => "1011"); -- faixa de indices
16    s_2d(0 TO 2) <= c_2d(3 TO 5);                -- faixa de indices
17    t_2d <= c_2d;                                -- vetor completo
18
19    s_3d(2)(7)(3) <= c_2d(7)(1);                    -- 1 elemento
20    s_3d(0)(1)(2 DOWNTO 0) <= c_2d(3)(3 DOWNTO 1); -- faixa
21    s_3d(1)(2 TO 3) <= c_2d(5 TO 6);                -- faixa
22    t_3d(2) <= c_2d;                                -- faixa
23    t_3d(0 TO 1) <= c_2d & c_2d;                    -- faixa
24  END teste;
```

Quadro 12.5.6 Declaração de um vetor unidimensional — elementos do tipo " bit_vector ".

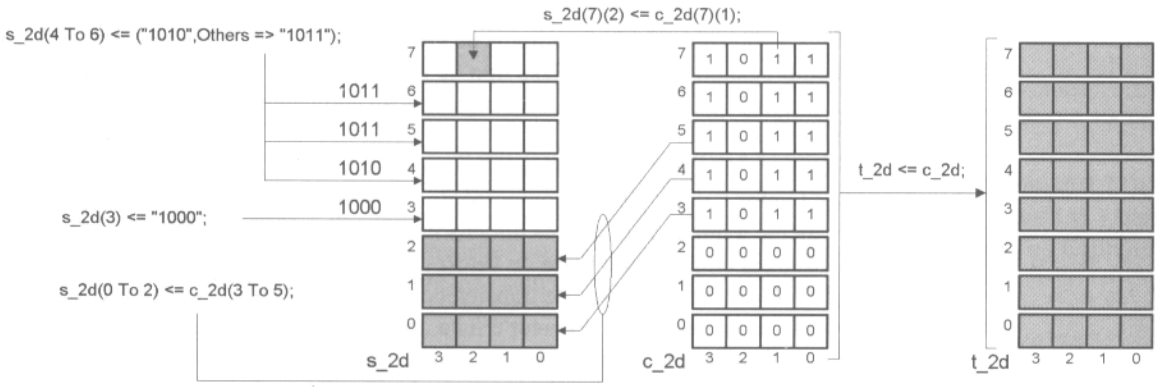


Figura 12.5.4 Operações de transferência com os sinais do tipo " vetor_2d " do Quadro 12.5.6.

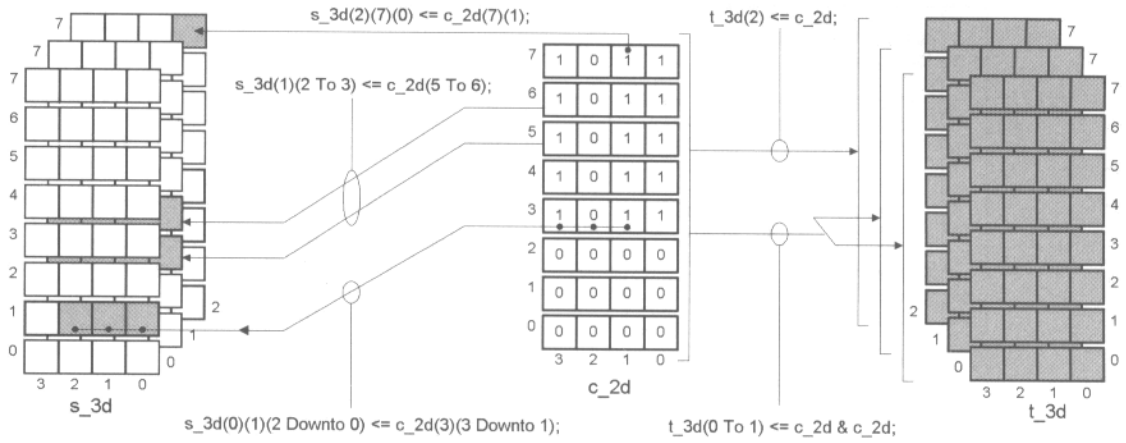


Figura 12.5.5 Operações de transferência com os sinais do tipo " vetor_3d " do Quadro 12.5.6.

Vetor multidimensional

A Figura 12.5.6 contém exemplos de vetores multidimensionais e declarações de constantes empregando os vetores. Quanto à síntese, vetores multidimensionais não são, normalmente, suportados pelas ferramentas. No exemplo, o vetor " vetor_2m " é um arranjo de duas dimensões de elementos do tipo " bit ". O vetor " vetor_3m " é um arranjo de três dimensões, composto também de elementos do tipo " bit ".

```
Type vetor_2m  Is Array          (0 To 7, 3 Downto 0) Of Bit;
Type vetor_3m  Is Array (0 To 2, 0 To 7, 3 Downto 0) Of Bit;

Constant ma: vetor_2m := (('1','0','0','0'), Others => ('1','0','1','0'));
Constant mb: vetor_2m := (("1000"),Others => ("1010"));
Constant mc: vetor_2m := (0=>('1','1','0','0'), Others => ('1','0','0','1'));
Constant md: vetor_2m := (("1100"), Others => ('1','0','0','1'));
Constant me: vetor_2m := (0 To 2 =>('0','1','1','0'), Others => ('1','0','1','1'));
Constant mf: vetor_2m := (Others => ('1','0','1','1'));
Constant mg: vetor_2m := (Others => (Others => ('1'))); -- preenche com mesmo valor
Constant mh: vetor_3m := (Others => (Others => (Others => ('1'))));
Constant mi: vetor_3m := ((Others => ('0','0','0','0')),(Others => ('0','0','0','1')),(Others => ('0','0','1','0')));
Constant mj: vetor_3m := (("0000", Others =>"0000"),("0001", Others =>"0001"),(Others =>"0010"));
```

me(a,b)

a \ b	3	2	1	0
7	1	0	1	1
6	1	0	1	1
5	1	0	1	1
4	1	0	1	1
3	1	0	1	1
2	0	1	1	0
1	0	1	1	0
0	0	1	1	0

Figura 12.5.6 Declaração de vetores multidimensionais.

O Quadro 12.5.7 contém uma descrição ilustrando a transferência de valores entre vetores multidimensionais. A seleção de um elemento do vetor é feita através de índices entre parênteses separados por vírgulas. Por exemplo, um único elemento de um objeto declarado segundo o tipo " vetor_2m " pode ser selecionado na forma " objeto(indice_v2, indice_v1) ". De modo semelhante, um elemento de um objeto declarado segundo o tipo " vetor_3m " pode ser selecionado na forma " objeto(indice_v3, indice_v2, indice_v1) ".

```
 1  ENTITY teste_f1 IS
 2  END teste_f1;
 3
 4  ARCHITECTURE teste OF teste_f1 IS
 5    TYPE vetor_2m  IS ARRAY          (0 TO 7, 3 DOWNTO 0) OF BIT;
 6    TYPE vetor_3m  IS ARRAY (0 TO 2, 0 TO 7, 3 DOWNTO 0) OF BIT;
 7
 8    SIGNAL   s_2m, t_2m : vetor_2m;
 9    CONSTANT c_2m: vetor_2m := (0 TO 2 =>('0','0','0','0'), OTHERS => ('1','0','1','1'));
10    SIGNAL   s_3m, t_3m:  vetor_3m;
11
12  BEGIN
13    s_2m(7,2) <= c_2m(7,1);                    -- 1 elemento
14    t_2m <= c_2m;                              -- vetor completo
15
16    s_3m(2,7,3) <= c_2m(7,1);                   -- 1 elemento
17    t_3m <= s_3m;
18  END teste;
```

Quadro 12.5.7 Declaração de um vetor multidimensional — elementos do tipo " bit ".

12.6 Definição de tipos registro

O tipo registro, ou *record*, é um tipo composto de elementos nomeados. Os elementos que compõem um objeto do tipo " RECORD " não necessitam ser do mesmo tipo, conforme se observa na classe vetor. Um objeto do tipo registro é referenciado por um nome, e cada elemento do objeto pode ser referenciado na forma " nome_objeto.nome_elemento ". O modo de atribuição de valores é similar ao empregado nos tipos da classe vetor.

A Figura 12.6.1 ilustra a declaração de um tipo registro denominado " instante ". Esse tipo é composto dos elementos " hora ", " minuto " e " segundo ". A seguir na figura, têm-se três exemplos de objetos declarados segundo esse tipo: " inicio ", " agora " e " tempo ". O elemento " minuto " da constante " inicio ", por exemplo, pode ser referenciado no formato " inicio.minuto ".

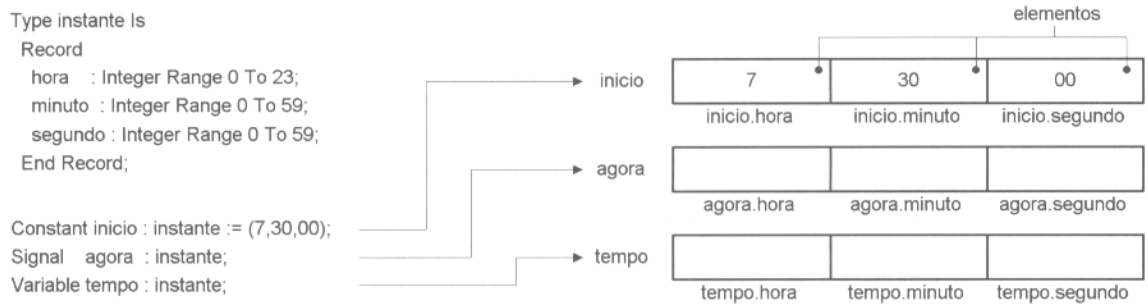


Figura 12.6.1 Declaração de tipos " record ".

O Quadro 12.6.1 contém uma descrição empregando um tipo registro, denominado " sinais_controle ", composto de três elementos. O elemento " r_dado " contém um tipo " INTEGER "; o elemento " r_tempo " contém um tipo " TIME "; e o elemento " r_atraso " contém um tipo " BOOLEAN ". O sinal " estimulo " é declarado segundo o tipo " sinais_controle " (ver linha 14).

Na descrição, uma série de valores é aplicada através da atribuição de sinal que inicia na linha 21. A cada novo valor aplicado, o procedimento é executado e realiza uma soma entre os sinais " dado " e " soma ". Conforme o valor do sinal " atraso ", o resultado da soma é transferido com ou sem atraso para o sinal " soma ". O valor do atraso é dado pelo elemento " r_tempo " do sinal " estimulo ".

A Figura 12.6.2 ilustra a operação da descrição. Observe a seqüência de atribuições de valores ao sinal " estimulo " (ver linhas 21 a 23). Essa seqüência de atribuições pode ser vista como um exemplo de operações de teste de entidades, onde cada elemento do tipo registro corresponde ao valor de um sinal da entidade sob teste.

```
 1  ENTITY rec_ts1a IS
 2  END rec_ts1a;
 3
 4  ARCHITECTURE teste OF rec_ts1a IS
 5  TYPE sinais_controle IS
 6    RECORD r_dado   : INTEGER RANGE 0 TO 15;
 7           r_tempo  : TIME;
 8           r_atraso : BOOLEAN;
 9    END RECORD;
10
11  SIGNAL dado, soma : INTEGER RANGE 0 TO 31;
12  SIGNAL tempo      : TIME;
13  SIGNAL atraso     : BOOLEAN;
14  SIGNAL estimulo   : sinais_controle;
15
16  BEGIN
17    dado   <= estimulo.r_dado;
18    tempo  <= estimulo.r_tempo;
19    atraso <= estimulo.r_atraso;
20
21    estimulo <= ( 2, 50 ns, TRUE),
22                ( 1, 35 ns, FALSE) AFTER 100 ns,
23                ( 5, 70 ns, TRUE)  AFTER 200 ns;
24
25    abc: PROCESS(dado, tempo, atraso)
26    BEGIN
27      IF  atraso THEN soma <= dado + soma AFTER tempo;
28      ELSE            soma <= dado + soma;
29      END IF;
30    END PROCESS;
31  END teste;
```

Quadro 12.6.1 Aplicação de um tipo registro.

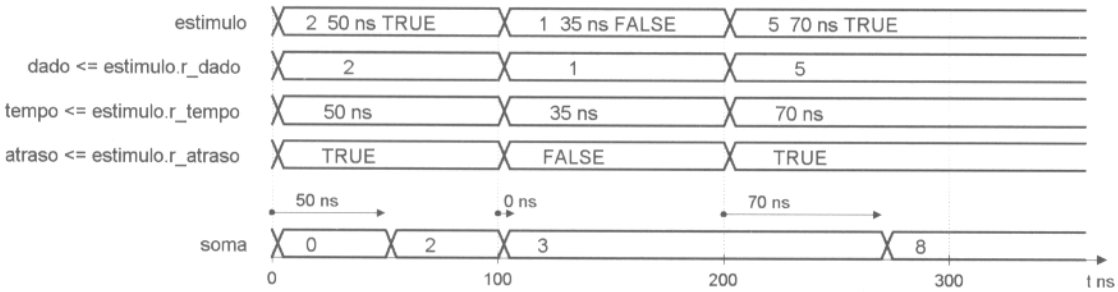


Figura 12.6.2 Simulação da entidade apresentada no Quadro 12.6.1.

12.7 Pseudônimos

Na versão VHDL-1987, um pseudônimo, ou *alias*, é uma designação adicional para um objeto das classes constante, variável ou sinal. O pseudônimo pode ser todo um objeto ou uma parte dele, e tem como objetivo facilitar o acesso ao objeto e melhorar a leitura do código. Deve-se observar que um pseudônimo não é um novo objeto, mas somente uma designação alternativa para todo o objeto ou parte do objeto. Pseudônimos não são válidos para vetores multidimensionais.

Na versão VHDL-1993, o escopo de um pseudônimo é ampliado para um nome alternativo para um item nomeado. O pseudônimo de um objeto, *object aliasis*, corresponde ao conceito da versão VHDL-1987 incluindo a classe arquivo. Os novos elementos incluídos na versão VHDL-1993 são os itens nomeados não-objeto, *nonobject aliasis*, como nome de subprogramas, nome de tipos, nome de atributos, pacotes, etc. Pseudônimos não são permitidos nos casos de rótulos e identificadores de comandos " LOOP " e " GENERATE ".

No Quadro 12.7.1 é apresentado um código empregando pseudônimos. No caso, são propostos dois nomes alternativos para parte do vetor " dado " de oito bits (ver linhas 8 e 9). O nome " dado_alto " corresponde aos quatro bits mais significativos de " dado ", e o nome " dado_baixo " corresponde aos quatro bits restantes.

```
1  ENTITY alias_a IS
2    PORT(h, l    : IN  BIT_VECTOR(3 DOWNTO 0);
3         h_l     : OUT BIT_VECTOR(7 DOWNTO 0));
4  END alias_a;
5
6  ARCHITECTURE teste OF alias_a IS
7    SIGNAL dado       : BIT_VECTOR(7 DOWNTO 0);
8    ALIAS  dado_alto  : BIT_VECTOR(3 DOWNTO 0) IS dado(7 DOWNTO 4);
9    ALIAS  dado_baixo : BIT_VECTOR(3 DOWNTO 0) IS dado(3 DOWNTO 0);
10 BEGIN
11   dado_alto  <= NOT h;  -- equivalente a: dado(7 DOWNTO 4) <= NOT h;
12   dado_baixo <= NOT l;  -- equivalente a: dado(3 DOWNTO 0) <= NOT l;
13   h_l <= dado_alto & dado_baixo;
14 END teste;
```

Quadro 12.7.1 Exemplo de pseudônimos, *alias*, em objetos.

No Quadro 12.7.2 é apresentado um exemplo de pseudônimos em itens que não são objetos. No caso, são dadas designações alternativas para dois subprogramas. A função " logica_funcao " pode ser chamada pelo seu pseudônimo " lg_func " (ver linha 24), e o procedimento " logica_proced " pode ser invocado pelo pseudônimo " lg_proc ". Pseudônimos de subprogramas requerem a identificação dos tipos segundo a ordem como eles foram declarados (ver linhas 20 e 21). A palavra reservada " RETURN " é necessária somente para funções.

```
1  -- VHDL-1993 somente
2  ENTITY alias_b IS
3    PORT(x, y    : IN  BIT_VECTOR(3 DOWNTO 0);
4         s, t, u : OUT BIT_VECTOR(3 DOWNTO 0));
5  END alias_b;
6
7  ARCHITECTURE teste OF alias_b IS
8  FUNCTION logica_funcao (a, b : BIT_VECTOR) RETURN BIT_VECTOR IS
9    BEGIN
10     RETURN a AND NOT b;
11 END logica_funcao;
12
13 PROCEDURE logica_proced(a, b : IN BIT_VECTOR; SIGNAL d, e : OUT BIT_VECTOR) IS
14   BEGIN
15     d <= a  OR NOT b;
16     e <= a XOR NOT b;
17  END logica_proced;
18
19 -- identificador    item      [            assinatura                          ];
20 ALIAS lg_func IS logica_funcao [BIT_VECTOR, BIT_VECTOR  RETURN BIT_VECTOR];
21 ALIAS lg_proc IS logica_proced [BIT_VECTOR, BIT_VECTOR, BIT_VECTOR, BIT_VECTOR];
22
23 BEGIN
24  s <= lg_func(x, y);
25  lg_proc(x, y, t, u);
26 END teste;
```

Quadro 12.7.2 Exemplo de pseudônimos em não-objetos — versão VHDL-1993 somente.

12.8 Síntese de memórias

A inserção de uma memória em um projeto deve considerar a tecnologia escolhida para a implementação do circuito. Normalmente, uma memória é um recurso de grande impacto no custo final, seja em área ou elementos disponíveis no dispositivo empregado. Algumas famílias de FPGA's contêm, além de registradores e circuitos combinacionais, unidades de memória RAM. Nesse caso, as ferramentas de síntese podem lançar mão desse recurso disponível na tecnologia escolhida quando for inferida uma memória na descrição, evitando, assim, o uso dos outros elementos do dispositivo. A organização dessas unidades de memória depende do fabricante e da família do dispositivo. A Figura 12.8.1 ilustra o emprego de recursos em dispositivos com e sem unidades de memória RAM na sua arquitetura.

Uma unidade de memória RAM disponível em um determinado dispositivo pode ser empregada, também, para implementar memórias ROM descritas no código. A ferramenta de síntese define os dados armazenados em

uma ou mais unidades de memória RAM do dispositivo. Essas unidades têm o conteúdo estabelecido no instante da programação do dispositivo, que não é alterado ao longo da operação.

A descrição de memórias emprega, normalmente, vetores compostos com elementos do tipo vetor, embora outras opções sejam possíveis. Um exemplo é o caso de memórias ROM descritas por construções " CASE WHEN " ou " WITH SELECT ". A expressão de escolha corresponde ao endereço da memória, e cada condição identifica o valor armazenado no endereço.

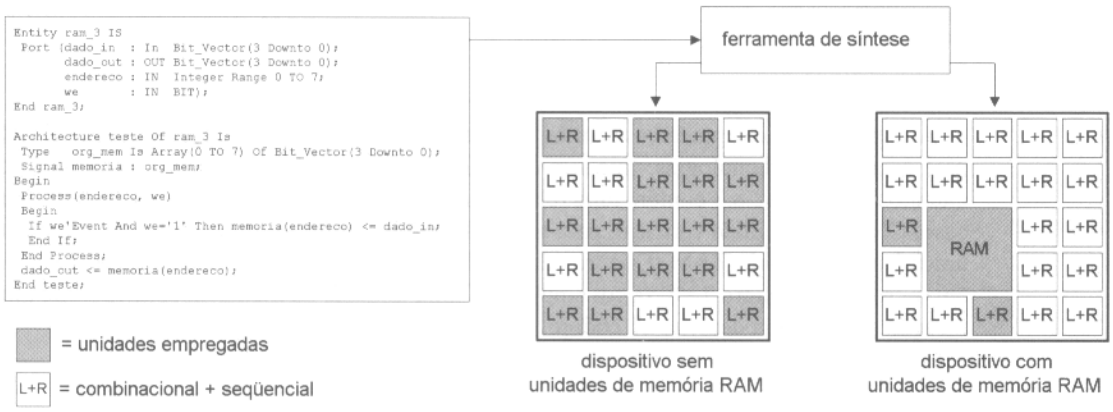


Figura 12.8.1 Emprego de recursos — " L+R " combinacional e registradores, " ram " = memória.

Memórias ROM

O Quadro 12.8.1 apresenta a descrição de uma memória ROM de oito bits de dados por dezesseis posições de endereço. A operação da memória é ilustrada na Figura 12.8.2. A linha " endereco " define a posição dos dados; a linha " saida " representa a informação armazenada; e a linha " ce " habilita a memória quando estiver em nível lógico baixo.

A organização da memória é estabelecida na linha 12 na declaração do tipo " arranjo_memoria ". Note que apenas o número de bits é fixado em oito, o número de endereços é definido pelo objeto que emprega o tipo, no caso a constante " dados " na linha 13. A constante " dados " é um vetor com dezesseis elementos, com os valores definidos entre as linhas 14 e 16. A seleção de um elemento desse vetor equivale a uma operação de leitura do valor de um endereço da memória. Como a posição do vetor é dada por um tipo inteiro, na realidade o subtipo " NATURAL ", é necessário converter o tipo " UNSIGNED " do sinal " endereco " para " INTEGER ". Isso é realizado pela função " to_integer " do pacote " numeric_std " (ver linha 18). Ainda na linha 18, é definida a habilitação da saída da memória. No caso " ce = 0 ", o sinal " saida " é habilitado; e para " ce = 1 " o sinal " saida " é levado ao estado de alta impedância.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.ALL;
 3  USE ieee.numeric_std.ALL;
 4
 5  ENTITY rom_1 IS
 6    PORT(endereco : IN  UNSIGNED(3 DOWNTO 0); -- valores de 0 a 15
 7         ce       : IN  STD_LOGIC;            -- habilita memoria
 8         saida    : OUT UNSIGNED(7 DOWNTO 0));-- saida de dados da memoria
 9  END rom_1 ;
10
11  ARCHITECTURE teste OF rom_1 IS
12    TYPE     arranjo_memoria IS ARRAY (NATURAL RANGE <>) OF UNSIGNED(7 DOWNTO 0);
13    CONSTANT dados : arranjo_memoria(0 TO 15) :=
14    ("00000011","10011111","00100101","00001101","10011000", -- valores posicoes 0,1,2,3,4
15     "01001001","01000001","00011111","00011001","00001001", -- valores posicoes 5,6,7,8,9
16     "01001001","01000001","00010111","00000001","01001001","01000110"); -- posicoes restantes
17  BEGIN
18    saida <= dados(to_integer(endereco)) WHEN ce='0' ELSE (OTHERS=>'Z');
19  END teste;
```

Quadro 12.8.1 Descrição de uma memória ROM.

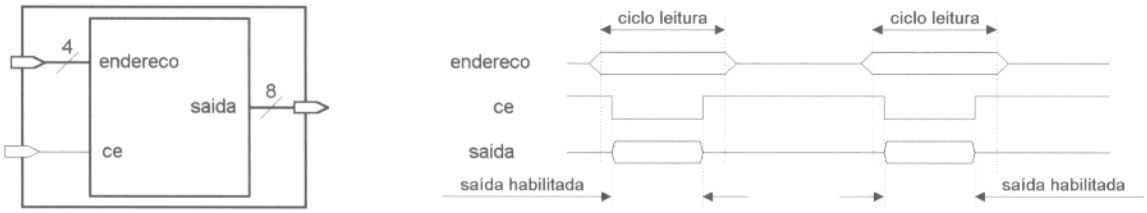


Figura 12.8.2 Operação da memória descrita no Quadro 12.8.1.

A seguir, são apresentados os resultados referentes à síntese da descrição da memória ROM do Quadro 12.8.1 para dois tipos de tecnologias, uma sem unidades de memória RAM e outra com unidades de memória RAM. Os Quadros 12.8.2 e 12.8.3 contêm uma relação parcial das mensagens geradas pela ferramenta de síntese em cada caso. Conforme se pode verificar, para ambas as condições foi inferida uma memória ROM. No dispositivo sem unidades de memória RAM (Quadro 12.8.2), foram empregadas 8 células lógicas, LCs, para a implementação. No outro caso (Quadro 12.8.3), foi empregada uma unidade de memória com 128 bits que corresponde à organização da memória descrita no código: 16×8.

```
"/vhdl/rom_1.vhd", line 18:Info, Inferred rom instance 'ix96' of type 'rom_memenab_8_4_16_4'
Total accumulated area :
 Number of LCs :                          8
 Number of TRIs :                         8
 Number of accumulated instances :       58
```

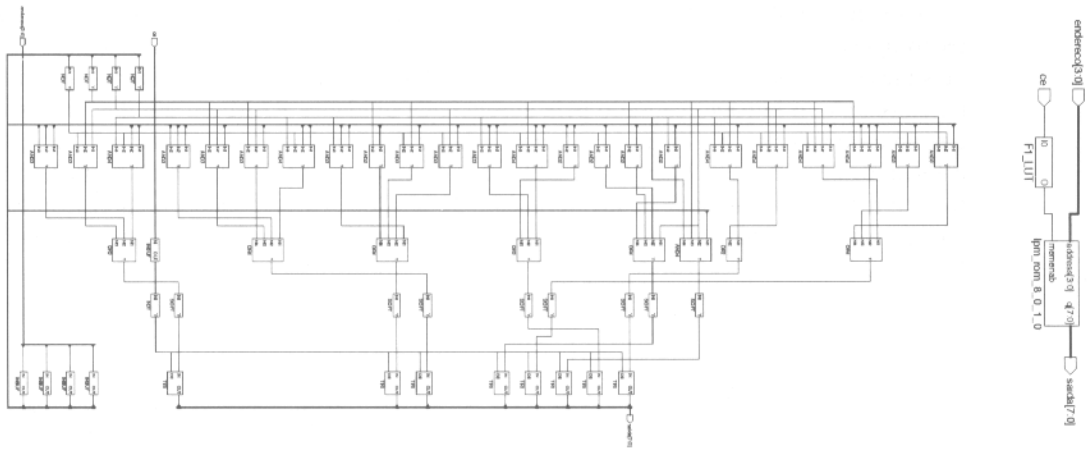
Quadro 12.8.2 Mensagens da ferramenta de síntese — tecnologia-alvo sem unidades de memória.

```
"/vhdl/rom_1.vhd", line 18:Info, Inferred rom instance 'ix96' of type 'rom_memenab_8_4_16_3'

Total accumulated area :
 Number of LCs :                          0
 Number of Memory Bits :                128
 Number of accumulated instances :        2
```

Quadro 12.8.3 Mensagens da ferramenta de síntese — tecnologia-alvo com unidades de memória.

A Figura 12.8.3 contém os diagramas nível porta lógica dos circuitos implementados. Nesses diagramas fica evidenciado, novamente, o emprego de recursos para ambos os casos. Deve-se notar que a ferramenta de síntese pode optar por uma implementação sem a utilização das unidades de memória RAM. Caso a memória identificada seja de pequenas dimensões, o circuito gerado pode ser um simples decodificador empregando os recursos combinacionais do dispositivo. Isso ocorre, por exemplo, se a memória descrita no Quadro 12.8.1 for reduzida para dez endereços.



tecnologia: sem unidades de memória com unidades de memória

Figura 12.8.3 Resultado da síntese da descrição do Quadro 12.8.1 — nível porta lógica.

Memórias RAM

O Quadro 12.8.4 apresenta a descrição de uma memória RAM de quatro bits de dados com oito posições de endereço. A sua operação é ilustrada na Figura 12.8.4. O sinal " endereco " define a posição dos dados; o sinal " saida " representa a informação armazenada; o sinal " ce " habilita a escrita na memória quando estiver em nível lógico baixo; e o sinal " we " identifica se a operação é de leitura ou escrita. Note que a saída está sempre habilitada.

A organização da memória, nesse caso, é estabelecida completamente na declaração do tipo " arranjo_memoria " (ver linha 13). Tanto o número de bits quanto o número de endereços são fixos. O sinal " memoria " é o objeto que contém as informações da memória. Da mesma forma que no exemplo anterior, a seleção de um elemento desse vetor identifica o endereço da memória.

A escrita de um dado na memória ocorre na borda de subida do sinal " ce " com " we = 0 " (ver linhas 18 e 19). A informação armazenada em um determinado endereço está sempre disponível no sinal " dado_saida ".

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3 USE ieee.numeric_std.ALL;
 4
 5 ENTITY ram_1 IS
 6    port (dado_entrada : IN  UNSIGNED(3 DOWNTO 0);
 7          dado_saida   : OUT UNSIGNED(3 DOWNTO 0);
 8          endereco     : IN  UNSIGNED(2 DOWNTO 0);
 9          we, ce       : IN  STD_LOGIC);
10 END ram_1;
11
12 ARCHITECTURE teste OF ram_1 IS
13   TYPE   arranjo_memoria IS ARRAY(0 TO 7) OF UNSIGNED(3 DOWNTO 0);
14   SIGNAL memoria : arranjo_memoria;
15 BEGIN
16   PROCESS(ce, endereco)
17   BEGIN
18     IF rising_edge(ce) THEN   -- dado armazenado na subida de "ce" com "wr=0"
19       IF we = '0' THEN  memoria(to_integer(endereco)) <= dado_entrada;
20       END IF;
21    END IF;
22    END PROCESS;
23    dado_saida <= memoria(to_integer(endereco));
24 END teste;
```

Quadro 12.8.4 Descrição de uma memória RAM.

A seguir, é feita uma comparação, similar ao caso anterior da memória ROM, considerando dois tipos de tecnologias: com e sem unidades de memória RAM. A relação parcial das mensagens geradas pela ferramenta de síntese é apresentada nos Quadros 12.8.5 e 12.8.6. Novamente, as memórias foram inferidas corretamente; entretanto, o impacto causado pela ausência de unidades RAM em um dos dispositivos foi muito maior que no caso anterior. No dispositivo sem unidades de memória RAM (Quadro 12.8.5), foram empregados 32 células lógicas, LCs, e 32 *flip flops*, DFFs. No dispositivo com unidades de memória RAM (Quadro 12.8.3), foi empregada uma unidade de memória com 32 bits, que corresponde à organização da memória descrita no código: 8×4. O impacto fica mais evidente quando se observam os diagramas esquemáticos apresentados na Figura 12.8.6.

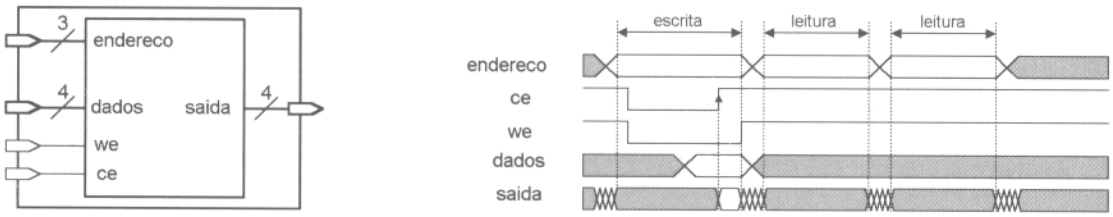


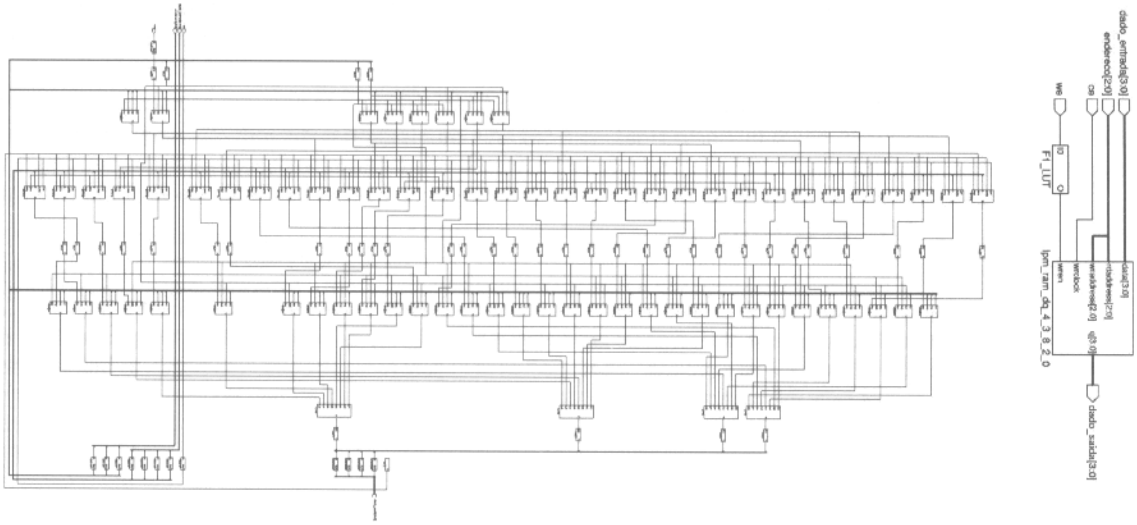
Figura 12.8.4 Operação da memória descrita no Quadro 12.8.4.

```
"/vhdl/ram_1.vhd",line 14: Info, RAM detected for 'memoria'.
"/vhdl/ram_1.vhd",line 14: Info, created RAM instance 'ix20' for 'memoria'.
Total accumulated area :
 Number of DFFs :                          32
 Number of EXPs :                          16
 Number of LCs :                           32
 Number of accumulated instances :        130
```

Quadro 12.8.5 Mensagens da ferramenta de síntese — tecnologia-alvo sem unidades de memória.

```
"/vhdl/ram_1.vhd",line 14: Info, RAM detected for 'memoria'.
"/vhdl/ram_1.vhd",line 14: Info, created RAM instance 'ix20' for 'memoria'.
Total accumulated area :
 Number of LCs :                            0
 Number of Memory Bits :                   32
 Number of accumulated instances :          2
```

Quadro 12.8.6 Mensagens da ferramenta de síntese — tecnologia-alvo com unidades de memória.



tecnologia: sem unidades de memória com unidades de memória

Figura 12.8.5 Resultado da síntese da descrição do Quadro 12.8.4 — nível porta lógica.

12.9 Principais pontos abordados

- Declaração de tipos enumerados:
  - os valores do tipo são apresentados ordenadamente na especificação;
  - o primeiro elemento corresponde ao menor valor; o último elemento, ao maior valor.

- Declaração de subtipos:
  - estabelece uma nova gama de valores;
  - operações e transferência de valores entre objetos de um tipo e subtipo são permitidas.

- Declaração de um tipo vetor:
  - são declarados o nome, os limites do vetor e o tipo de elementos que compõem o vetor;
  - os limites do vetor são definidos por tipos discretos.

- Declaração de um vetor com os limites em aberto, ou *unconstrained array*:
  - a declaração do tipo não estabelece o número de elementos;
  - os limites são definidos na declaração de um objeto que emprega esse tipo, na declaração de um subtipo do tipo, ou em um subprograma.

- Declaração de vetores compostos com elementos do tipo vetor:
  - criação de objetos com uma organização equivalente a um arranjo;
  - suportados pela maioria das ferramentas de síntese;
  - empregados na descrição de memórias.

- Declaração de vetores multidimensionais:
  - criação de objetos com uma organização de um arranjo;
  - menos flexíveis nas opções de atribuição de valores, em se comparando com vetores compostos de elementos do tipo vetor;
  - não são suportados pela maioria das ferramentas de síntese.

- Declaração de tipo registro, ou *record*:
  - tipo composto de elementos nomeados;
  - os elementos não necessitam ser do mesmo tipo.

- Pseudônimo, ou *alias*, na versão VHDL-1987, é uma designação adicional para um objeto. Na versão VHDL-1993, o escopo de um pseudônimo é ampliado para um nome alternativo para um item nomeado.

- Descrições contendo uma memória devem considerar a tecnologia escolhida. Essas unidades podem ter um grande impacto no custo final, seja em área ou elementos disponíveis, conforme o dispositivo empregado.

12.10 Exercícios

12.10.1 Apresente o código para a síntese de uma memória " RAM " com uma palavra de 4 bits e 16 posições de endereço. O terminal " wr " define se a operação é de leitura ou escrita, e o terminal " ce " habilita a unidade. A Figura 12.10.1 ilustra a operação da memória. A escrita de um dado ocorre na borda de subida de " ce " como " wr " em nível baixo. A leitura de um dado é executada na condição " ce=0 " e " wr=1 "; para todas as outras condições, as linhas de dados devem permanecer no estado de alta impedância.

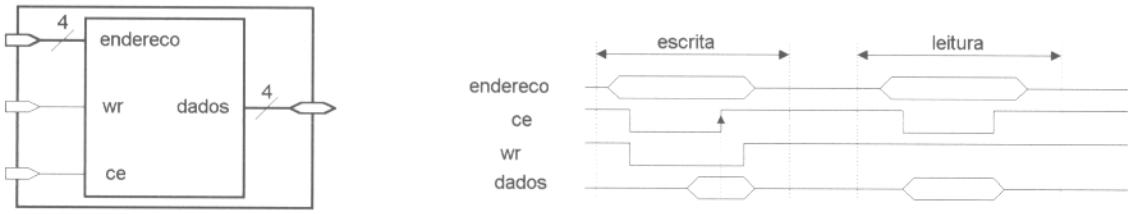


Figura 12.10.1 Operação da memória " RAM " proposta para o Exercício 12.10.1.

12.10.2 Sintetize a descrição do Exercício 12.10.1 em uma FPGA que contenha unidades de memória RAM, e verifique se o código foi inferido corretamente pela ferramenta de síntese.

12.10.3 Apresente o código para descrição de uma memória " RAM " operacionalmente similar à memória proposta no Exercício 12.10.1. Na entidade proposta, entretanto, não deve ser fixado o número de bits para os sinais de endereço e dados; essas informações devem ser deixadas em aberto. A definição desses parâmetros é estabelecida pela entidade que solicitar esse componente. Desse modo, a entidade " RAM " proposta pode ser empregada por mais de uma entidade em diferentes configurações. Para o teste da descrição proposta, crie uma entidade que empregue a memória " RAM " como um componente. Na Figura 12.10.2 é ilustrado o problema solicitado.

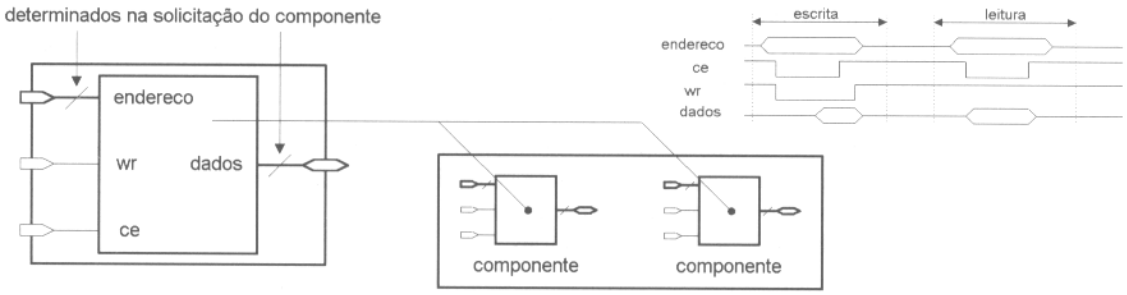


Figura 12.10.2 Memória " RAM " empregada como componente.

12.10.4 Sintetize a descrição do Exercício 12.10.3 em uma FPGA que contenha unidades de memória RAM, e verifique se o código foi inferido corretamente pela ferramenta de síntese.

12.10.5 A descrição de uma memória ROM pode ser realizada, também, com construções " CASE WHEN ". Proponha a descrição de uma memória ROM contendo dezesseis endereços e oito bits de dados, empregando uma construção " CASE WHEN ". As informações armazenadas podem ser quaisquer, e a saída é controlada pelo sinal " ce ". Com esse sinal em nível alto, a saída da memória permanece em estado de alta impedância; em nível baixo, a saída é habilitada.

12.10.6 Apresente um código que realize a função de um relógio com as unidades de hora, minuto e segundo. A contagem é realizada por meio de um sinal do tipo registro, sendo o tipo composto dos seguintes elementos: " hora ", " minuto " e " segundo ". O código deve incrementar um objeto declarado conforme esse tipo, de modo a implementar as operações de um relógio. Note que a descrição não necessita ser sintetizável.

12.10.7 Apresente o código para o teste da memória " RAM " descrita no Quadro 12.8.4 na forma de um componente. Declare um tipo " RECORD " agrupando os sinais " dado_entrada ", " endereco ", " we " e " ce " para a aplicação dos estímulos de teste. A descrição deve conter uma seqüência de estímulos que verifique uma operação de escrita na memória.

12.10.8 Altere o código proposto no Quadro 12.7.2, substituindo cada subprograma por uma entidade. Nessa nova proposta, cada entidade que substitui um subprograma é solicitada na forma de componente por uma outra entidade. Empregue um pseudônimo para cada componente solicitado.

12.10.9 No código proposto no Quadro 12.10.1, é declarado um vetor de três dimensões denominado " vetor3_dm ". Esse vetor é um vetor multidimensional de duas dimensões composto de elementos do tipo " BIT_VECTOR ". As expressões de transferências de valores, colocadas na forma de comentário nas linhas 11, 13, 15, 17 e 19, são coerentes para um vetor composto de elementos do tipo vetor (ver Quadro 12.5.6). Proponha expressões equivalentes, de modo a adequar a nova declaração do vetor.

```
 1  ENTITY teste_g1 IS
 2  END teste_g1;
 3
 4  ARCHITECTURE teste OF teste_g1 IS
 5    TYPE vetor_2d  IS ARRAY (0 TO 7) OF BIT_VECTOR(3 DOWNTO 0);
 6    TYPE vetor_3dm IS ARRAY (0 TO 2, 0 TO 7) OF BIT_VECTOR(3 DOWNTO 0);
 7    CONSTANT c_2d: vetor_2d := (0 TO 2 =>('0','0','0','0'), OTHERS => ('1','0','1','1'));
 8    SIGNAL   s_3d, t_3d: vetor_3dm;
 9
10  BEGIN
11  --s_3d(2)(7)(3) <= c_2d(7)(1);                    -- 1 elemento
12
13  --s_3d(0)(1)(2 DOWNTO 0) <= c_2d(3)(3 DOWNTO 1); -- faixa
14
15  --s_3d(1)(2 TO 3) <= c_2d(5 TO 6);                -- faixa
16
17  --t_3d(2) <= c_2d;                                -- faixa
18
19  --t_3d(0 TO 1) <= c_2d & c_2d;                    -- faixa
20
21  END teste;
```

Quadro 12.10.1 Código para o Exercício 12.10.9.

CAPÍTULO

13 ARQUIVOS

As opções fornecidas para operações com arquivos em VHDL são limitadas, quando comparadas com linguagens convencionais de programação. As informações podem ser armazenadas no formato binário ou no formato texto. No formato texto, o tratamento das informações é feito com o auxílio do pacote " TEXTIO ", que define tipos e subprogramas para as operações.

As versões VHDL-1987 e VHDL-1993 possuem diferenças significativas com relação à sintaxe para o tratamento de arquivos. Na versão VHDL-1993, o leque de operações possíveis foi ampliado com a definição de novos subprogramas que permitem abrir e fechar arquivos durante a simulação.

13.1 Emprego de arquivos

As operações com arquivos em VHDL não visam à criação de unidades de projeto sintetizáveis. Operações como leitura ou escrita de dados em arquivos são por demais abstratas para síntese. Elas podem, entretanto, ser de grande auxílio nas operações de teste e validação de entidades mais complexas. Os estímulos aplicados podem ter como origem informações armazenadas em arquivos, permitindo a troca de dados entre uma entidade em teste e o mundo exterior. Nessas operações, unidades funcionais especialmente criadas para teste são empregadas no controle de uma entidade em teste, gerando estímulos, fornecendo dados e armazenando resultados (ver Figura 13.1.1).

Conforme o manual de referência da linguagem, " o tipo arquivo é usado para definir objetos representando arquivos no ambiente do sistema hospedeiro. O valor de um objeto tipo arquivo é uma seqüência de valores armazenados em um arquivo ". A manipulação de informações armazenadas em arquivos necessita de:

- declaração de um tipo arquivo;
- objetos declarados segundo um tipo arquivo definido.

A leitura e escrita de informações em objetos do tipo arquivo são feitas através de procedimentos especialmente definidos. Os valores são recuperados ou armazenados de modo seqüencial, e o final de um arquivo é identificado por uma marca de final de arquivo.

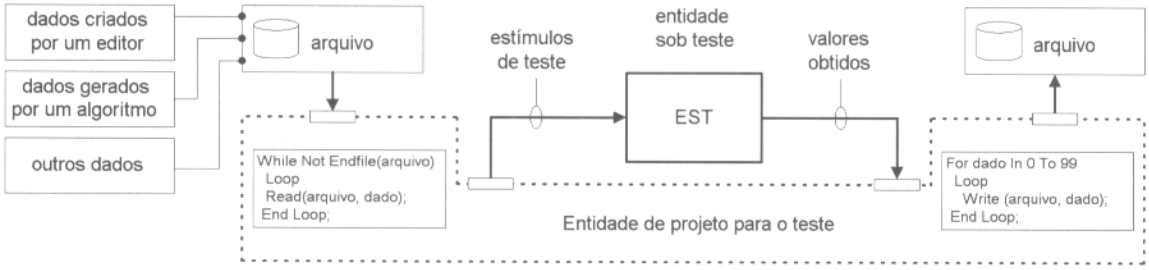


Figura 13.1.1 Operações envolvendo arquivos.

Declaração de um tipo arquivo

A definição de um tipo arquivo é feita através de uma declaração, conforme mostrado na parte superior do Quadro 13.1.1. O " tipo_contido " determina o tipo que um objeto declarado como tipo " arquivos_de_tipo_x " pode conter.

Em outras palavras, um objeto do tipo " arquivos_de_tipo_x " representa um arquivo contendo elementos do tipo " tipo_contido ". Ainda no Quadro 13.1.1, são apresentados alguns exemplos de declarações de um tipo " FILE ".

```
TYPE arquivos_de_tipo_x  IS FILE OF tipo_contido;   -- declaracao de um tipo arquivo
```

```
TYPE arqv_inteiro     IS FILE OF INTEGER;      -- tipo FILE para objetos armazenando INTEGER
TYPE arqv_bit_vector IS FILE OF BIT_VECTOR;   -- tipo FILE para objetos armazenando BIT_VECTOR
TYPE arqv_caractere  IS FILE OF CHARACTER;    -- tipo FILE para objetos armazenando CHARACTER

TYPE arqv_byte IS FILE OF BIT_VECTOR(0 TO 7); -- tipo FILE p/objetos armazenando vetor de 8 bits

SUBTYPE sub_inteiro IS INTEGER RANGE 0 TO 99;
TYPE arqv_inteiro_0_99   IS FILE OF sub_inteiro; -- tipo FILE para objetos armazenando um sub-tipo
```

Quadro 13.1.1 Exemplos de declaração tipo arquivo.

Declaração de objetos tipo arquivo - versão VHDL-1987

A declaração de um objeto tipo arquivo deve conter um identificador, " nome_objeto " e o tipo arquivo que especifica os elementos contidos nesse objeto, " arquivo_de_tipo_x " (ver a parte superior do Quadro 13.1.2). É necessário, também, declarar o modo do arquivo e o nome externo do arquivo no sistema hospedeiro. Para um arquivo de leitura o modo é " IN ", e para um arquivo de escrita o modo é " OUT ". No Quadro 13.1.2 são apresentados, também, dois exemplos de declaração de objetos do tipo arquivo. O primeiro define um arquivo de leitura, e o segundo, um arquivo de escrita.

Na versão VHDL-1987 não é possível fechar um arquivo durante a simulação. O arquivo é aberto no momento da declaração de um objeto do tipo arquivo e fechado ao término da simulação.

```
FILE nome_objeto : arquivos_de_tipo_x IS modo "nome_externo";
```

```
FILE leitura     : arqv_inteiro       IS  IN  "/vhdl/dados_r.dat"; -- objeto: arquivo leitura
FILE escrita     : arqv_caractere     IS OUT  "/vhdl/dados_w.dat"; -- objeto: arquivo escrita
```

Quadro 13.1.2 Declaração de um objeto do tipo arquivo — versão VHDL-1987.

Operações implicitamente declaradas - versão VHDL-1987

As operações com os arquivos são executadas por meio de subprogramas implicitamente declarados após uma declaração de um tipo arquivo (ver Quadro 13.1.3). O procedimento " READ " executa uma operação de leitura, recuperando o próximo valor de um arquivo. O valor obtido no arquivo " arquivo_leitura " retorna em " dado ". O procedimento " WRITE " insere um novo valor em um arquivo. O valor contido em " dado " é escrito no arquivo " arquivo_leitura ". A função " ENDFILE " retorna o valor " TRUE ", caso o valor lido no arquivo seja a marca de final de arquivo. Essa função retorna sempre o valor verdadeiro para arquivos de escrita, modo " OUT ". A chamada da função " ENDFILE " somente pode ocorrer para um arquivo já aberto.

No caso da declaração de um tipo arquivo cujos elementos são vetores sem limites, o procedimento de leitura " READ " implicitamente declarado corresponde ao subprograma da segunda linha do Quadro 13.1.3. Nesse caso, o procedimento retorna no campo " tamanho " as dimensões reais do vetor recuperado na operação de leitura. Note que o valor lido depende das dimensões do objeto associado ao parâmetro formal do argumento " dado ". Se as dimensões forem menores do que o valor retornado em " tamanho ", parte da informação é perdida; se as dimensões forem maiores, parte da informação em " dado " não é alterada.

```
READ (arquivo_leitura, dado);           -- procedimento para leitura
READ (arquivo_leitura, dado, tamanho); -- proc. p/ leitura, arq. contendo vetor sem limites def.
```

```
WRITE (arquivo_escrita, dado);          -- procedimento para escrita
```

```
ENDFILE(arquivo_c)                      -- funcao fim de arquivo, retorna tipo boleano
```

Quadro 13.1.3 Procedimentos e função disponíveis — versão VHDL-1987.

Exemplos - versão VHDL-1987

O código apresentado no Quadro 13.1.4 cria um arquivo denominado " dado_a.dat " contendo os valores numéricos entre " 48 " e " 57 ". A cada iteração do laço da linha 12, um valor é escrito no arquivo. O tipo " arqv_int " declarado na linha 9 define um tipo " FILE " para elementos do tipo " INTEGER ". O objeto " arquivo_wr " define um objeto do tipo arquivo cujos valores armazenados seguem a especificação de " arqv_int ", isto é, valores do tipo inteiro. Como o arquivo criado é binário, foram escolhidos os valores " 48 " a " 57 ", pois eles correspondem aos caracteres " 0 " a " 9 " no código ASCII. Assim, ao se observar o arquivo criado através de um editor de texto, deve-se esperar que ele contenha os caracteres " 0 ", " 1 " ... " 9 ".

```
 1  -- versao VHDL-1987
 2  -- termine a execucao da simulacao para fechar arquivo
 3  ENTITY arqv_a IS
 4  END arqv_a;
 5
 6  ARCHITECTURE teste OF arqv_a IS
 7  BEGIN
 8    escreve: PROCESS
 9      TYPE arqv_int IS FILE OF INTEGER;
10      FILE arquivo_wr : arqv_int IS OUT "/vhdl/dado_a.dat";
11    BEGIN
12      FOR dado IN 48 TO 57 LOOP    -- valor 48 ASCII corresponde a caractere "0"
13        WRITE (arquivo_wr, dado); -- escreve valores 48 a 57 no arquivo
14      END LOOP;                   -- correspondem aos caracteres "0" a "9" ASCII
15    WAIT;
16    END PROCESS escreve;
17  END teste;
```

Quadro 13.1.4 Teste para escrita de dados em um arquivo — versão VHDL-1987.

O código apresentado no Quadro 13.1.5 realiza uma operação de leitura em um arquivo, e permite, por exemplo, a observação dos dados armazenados no arquivo criado pelo código anterior. A função " ENDFILE " (linha 14) é empregada para verificar se o final do arquivo foi atingido. Observe que foi criada, também, a variável " dado_v " no processo de leitura para a transferência do valor para o sinal " dado ". Isso é necessário, porque os parâmetros formais dos subprogramas " READ " e " WRITE " são da classe variável, e, portanto, os parâmetros reais devem ser da mesma classe. Note, ainda, que os valores recuperados no arquivo correspondem àqueles armazenados na forma binária, isto é, " 48 ", " 49 " ... " 57 ". O comando " WAIT " na linha 17 permite apresentar esses valores na janela do simulador em intervalos de tempos de 10 ns.

```
 1  -- versao VHDL-1987
 2  -- termine a execucao da simulacao para fechar arquivo
 3  ENTITY arqv_d IS
 4    PORT(dado   : OUT INTEGER RANGE 0 TO 100);  -- valores numericos esperados
 5  END arqv_d;
 6
 7  ARCHITECTURE teste OF arqv_d IS
 8    BEGIN
 9    le: PROCESS
10      TYPE arqv_int IS FILE OF INTEGER;
11      FILE arquivo_rd : arqv_int IS IN "/vhdl/dado_a.dat";
12      VARIABLE dado_v : INTEGER;
13    BEGIN
14      WHILE NOT ENDFILE(arquivo_rd) LOOP -- testa fim do arquivo
15        READ(arquivo_rd, dado_v);        -- le os valores na forma binaria
16        dado <= dado_v;                  -- argumento do procedimento tipo variavel
17        WAIT FOR 10 ns;                  -- aguarda 10 ns para proxima leitura
18      END LOOP;
19    WAIT;
20    END PROCESS le;
21  END teste;
```

Quadro 13.1.5 Teste para leitura de dados em um arquivo — versão VHDL-1987.

No Quadro 13.1.6, é apresentado um exemplo de um objeto tipo arquivo contendo vetores do tipo " STRING ". Os limites dos vetores não foram definidos na declaração do tipo " FILE " (ver linha 11). Desse modo, o procedimento " READ " é declarado com um terceiro parâmetro que retorna com as dimensões reais do vetor recuperado na operação de leitura (ver definição dos procedimentos no Quadro 13.1.3).

O valor recuperado no arquivo retorna na variável " dado_lido_v ", do tipo " STRING ". As dimensões de " dado_lido_v " são definidas pelo genérico " n " (ver linhas 4 e 15). A dimensão do vetor recuperado retorna na variável " tamanho_v ". Os valores de ambas as variáveis são transferidos para as portas de saída (ver linhas 20 e 21). O emprego do genérico " n " nessa descrição tem como objetivo explorar a operação do procedimento " READ " com três parâmetros, conforme proposto a seguir.

```
 1 -- versao VHDL-1987
 2 -- termine a execucao da simulacao para fechar arquivo
 3 ENTITY arqv_k IS
 4   GENERIC(n : INTEGER := 7);
 5   PORT(dado_lido  : OUT STRING(1 TO n);
 6        tamanho    : OUT NATURAL);              -- valor retornado do procedimento READ
 7 END arqv_k;
 8
 9 ARCHITECTURE teste OF arqv_k IS
10   -- informacao contida no arquivo dado_k.dat = abcdefghijklmnopqrstuvxz
11   TYPE arquivo IS FILE OF STRING;               -- limites do vetor nao definidos
12   FILE arq_wr_rd : arquivo IS IN "/vhdl/dado_k.dat";
13  BEGIN
14   escreve_le: PROCESS
15     VARIABLE dado_lido_v : STRING(1 TO n);
16     VARIABLE tamanho_v : NATURAL;
17   BEGIN
18     WHILE NOT ENDFILE(arq_wr_rd) LOOP           -- testa fim do arquivo
19       READ(arq_wr_rd, dado_lido_v, tamanho_v); -- le os valores na forma binaria
20       dado_lido <= dado_lido_v;                 -- argumento do procedimento tipo variavel
21       tamanho <= tamanho_v;
22       WAIT FOR 10 ns;                           -- aguarda 10 ns para proxima leitura
23     END LOOP;
24   WAIT;
25   END PROCESS escreve_le;
26 END teste;
```

Quadro 13.1.6 Leitura — arquivo contendo vetores com limites não-definidos — VHDL-1987.

Na Figura 13.1.2, é apresentado o resultado da simulação para duas compilações diferentes, uma com " n=3 " e outra com " n=7 ". Suponha o conteúdo do arquivo " dado_k.dat " uma seqüência de caracteres entre " a " e " z ", conforme inserido na linha 10 do código na forma de comentário.

No caso do valor genérico " n " igual a três, o tamanho dos vetores recuperados é sempre igual às dimensões definidas para a variável " dado_lido_v ". Isso porque o arquivo contém vinte e quatro caracteres, e em oito operações de leitura todos os valores são recuperados no arquivo.

No segundo caso, como a variável " dado_lido_v " é um vetor de dimensão igual a sete, apenas as três primeiras operações de leitura retornam com sete novos caracteres. A última operação de leitura retorna com somente três novos caracteres. Note que o conteúdo dos quatro elementos da posição mais à direita de " dado_lido_v " permanece inalterado, conforme a definição da linguagem.

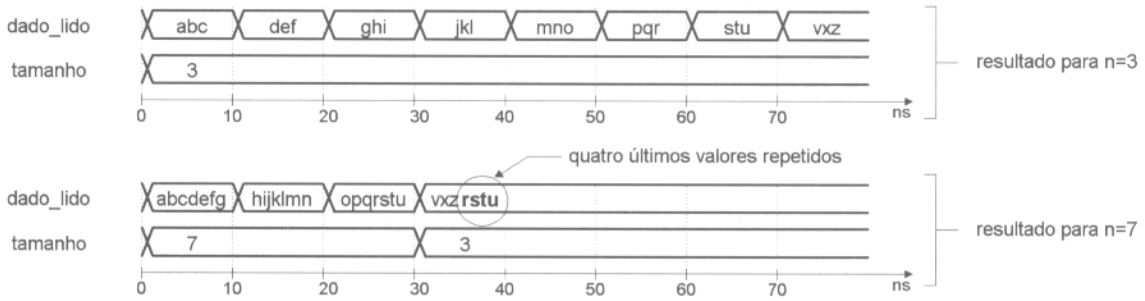


Figura 13.1.2 Resultado da simulação da descrição do Quadro 13.1.6 — duas condições.

Declaração de objetos tipo arquivo - versão VHDL-1993

Na versão VHDL-1987, a abertura do arquivo ocorre na declaração de um objeto do tipo arquivo. Esta opção é possível, também, na versão VHDL-1993 (ver a parte superior do Quadro 13.1.7). Nesse caso, a sintaxe da declaração de um objeto é semelhante à sintaxe da versão VHDL-1987. Ela contém um identificador, " nome_

objeto ", o tipo do arquivo, o modo do arquivo e o nome externo do arquivo no sistema hospedeiro. O modo é " Read_Mode " para um arquivo de leitura, e " Write_Mode " para um arquivo de escrita. Note que a palavra reservada " OPEN " explicita que uma operação de abertura de arquivo está ocorrendo na definição do tipo arquivo. O Quadro 13.1.7 contém exemplos nas duas versões, comparando as declarações para o caso de um arquivo de leitura, e para o caso de um arquivo de escrita.

```
--FILE nome_objeto : arqv_de_tipo_x IS   modo    "nome_externo";              -- VHDL-1987
  FILE nome_objeto : arqv_de_tipo_x OPEN modo IS "nome_externo";
```

```
--FILE leitura     : arqv_inteiro    IS    IN          "/vhdl/dados_r.dat";     -- VHDL-1987
  FILE leitura     : arqv_int       OPEN Read_Mode IS "/vhdl/dados_r.dat";

--FILE escrita     : arqv_caractere  IS   OUT          "/vhdl/dados_w.dat";     -- VHDL-1987
  FILE escrita     : arqv_int       OPEN Write_Mode IS "/vhdl/dados_w.dat";
```

Quadro 13.1.7 Declaração de um objeto do tipo arquivo — versões VHDL-1987 e VHDL-1993.

A abertura de um arquivo com a declaração de um objeto do tipo arquivo, conforme apresentada no Quadro 13.1.7, é útil para a conversão de códigos para a versão VHDL-1993, devido à sua similaridade com a versão VHDL-1987.

Do mesmo modo que na versão anterior, na versão VHDL-1993 a declaração de um tipo arquivo leva à declaração implícita de uma série de subprogramas para as operações com arquivos. Uma das operações disponíveis é a abertura de arquivo. Assim, na declaração de um objeto arquivo, podem ser omitidas as informações referentes à abertura do arquivo, conforme ilustrado no Quadro 13.1.8.

```
FILE nome_objeto : arqv_de_tipo_x;  -- Objeto declarado, mas o arquivo nao foi aberto

FILE arquivo_a   : arqv_inteiro;    -- idem
```

Quadro 13.1.8 Declaração de um objeto do tipo arquivo sem informação de abertura — VHDL-1993.

Operações implicitamente declaradas - versão VHDL-1993

Os subprogramas implicitamente definidos após uma declaração de tipo arquivo na versão VHDL-1993 são apresentados no Quadro 13.1.9. A operação dos procedimentos " READ " e " WRITE " e da função " ENDFILE " é similar à operação dos subprogramas com mesmo nome da versão anterior. Os procedimentos " FILE_OPEN " são destinados à abertura de um arquivo cujo nome é especificado no campo " nome_externo ". O arquivo é associado ao objeto especificado no campo " arquivo_a ", e o modo de operação é especificado no campo " modo ". O procedimento " FILE_CLOSE " fecha o arquivo especificado.

Três valores podem ser passados para o campo " modo " nos procedimentos " FILE_OPEN ":

- " READ_MODE ";
- " WRITE_MODE ";
- " APPEND_MODE ".

O valor " READ_MODE " identifica um arquivo somente de leitura, e a operação de leitura inicial será executada no primeiro valor contido no arquivo. Os valores " WRITE_MODE " e " APPEND_MODE " identificam arquivos de escrita. No primeiro caso, o arquivo é limpo na operação de abertura. No segundo caso, a informação contida no arquivo é preservada, e os valores são inseridos no final do arquivo.

O procedimento " FILE_OPEN " com o campo " status " permite determinar o resultado da operação de abertura do arquivo. Os valores de retorno podem ser:

- " OPEN_OK ";
- " STATUS_ERROR ";
- " NAME_ERROR ";
- " MODE_ERROR ".

O primeiro indica que a operação foi realizada com sucesso; o segundo indica que o objeto do tipo arquivo especificado em " arquivo_a " já tem um arquivo associado a ele. O valor " NAME_ERROR " informa que ocor-

reu um erro com relação ao nome do arquivo devido à inexistência do arquivo ou impossibilidade de se criar o arquivo. O valor " MODE_ERROR " indica que o arquivo não pode ser aberto conforme o modo requisitado.

Cabe observar que os valores " READ_MODE ", " WRITE_MODE " e " APPEND_MODE " pertencem ao tipo " FILE_OPEN_KIND ", e os valores " OPEN_OK ", " STATUS_ERROR ", " NAME_ERROR " e " MODE_ERROR " pertencem ao tipo " FILE_OPEN_STATUS ". Ambos são tipos predefinidos no pacote padrão da versão VHDL-1993.

```
READ   (arquivo_a, dado);                                  -- procedimento para leitura
READ   (arquivo_a, dado, tamanho);                         -- procedimento para leitura

WRITE  (arquivo_a, dado);                                  -- procedimento para escrita

FILE_OPEN(        arquivo_a, "nome_externo", modo);        -- abre um arquivo
FILE_OPEN(status, arquivo_a, "nome_externo", modo);        -- abre um arquivo

FILE_CLOSE(arquivo_a);                                     -- fecha um arquivo

ENDFILE(arquivo_a)                          -- funcao fim de arquivo, retorna tipo boleano
```

Quadro 13.1.9 Procedimentos e função disponíveis — versão VHDL-1993.

Exemplos - versão VHDL-1993

Nos Quadros 13.1.10 e 13.1.11, são apresentados dois códigos para escrita de valores em arquivos. O primeiro é equivalente ao código do Quadro 13.1.4 para a versão VHDL-1987, onde a abertura do arquivo ocorre na declaração de um objeto do tipo arquivo. O segundo código emprega os subprogramas " FILE_OPEN " e " FILE_CLOSE " para abrir e fechar o arquivo (ver linhas 11 e 15). Note que, nesse caso, a definição do objeto arquivo não contém a informação de abertura. Compare a linha 9 com a linha 10 do Quadro 13.1.10.

```
 1  -- versao VHDL-1993
 2  -- o arquivo e' fechado apos o termino da simulacao
 3  ENTITY arqv_b IS
 4  END arqv_b;
 5
 6  ARCHITECTURE teste OF arqv_b IS
 7  BEGIN
 8    escreve: PROCESS
 9      TYPE arqv_int IS FILE OF INTEGER;
10      FILE arquivo_wr : arqv_int OPEN Write_Mode IS "/vhdl/dado_b.dat";
11     BEGIN
12     FOR dado IN 48 TO 57 LOOP  -- valor 48 ASCII corresponde a caractere "0"
13       WRITE(arquivo_wr, dado); -- escreve valores 48 a 57 no arquivo
14     END LOOP;                  -- correspondem aos caracteres "0" a "9" ASCII
15    WAIT;
16    END PROCESS escreve;
17  END teste;
```

Quadro 13.1.10 Escrita de dados num arquivo — arquivo aberto na declaração — VHDL-1993.

```
1  -- versao VHDL-1993
2  ENTITY arqv_c IS
3  END arqv_c;
4
5  ARCHITECTURE teste OF arqv_c IS
6  BEGIN
7    escreve: PROCESS
8      TYPE arqv_int IS FILE OF INTEGER;
9      FILE arquivo_wr : arqv_int;
10   BEGIN
11     FILE_OPEN(arquivo_wr, "/vhdl/dado_c.dat",Write_Mode);
12     FOR dado IN 48 TO 57 LOOP   -- valor 48 ASCII corresponde a caractere "0"
13       WRITE(arquivo_wr, dado);  -- escreve valores 48 a 57 no arquivo
14     END LOOP;                   -- correspondem aos caracteres "0" a "9" ASCII
15     FILE_CLOSE(arquivo_wr);     -- fecha o arquivo
16   WAIT;
17   END PROCESS escreve;
18 END teste;
```

Quadro 13.1.11 Escrita de dados num arquivo — procedimento abre arquivo — VHDL-1993.

13.2 Pacote "TEXTIO"

O pacote " TEXTIO " fornece subprogramas para operação com arquivos no formato texto, sendo disponível em qualquer implementação da linguagem. Os arquivos criados através do pacote permitem uma interpretação direta das informações e podem ser manipulados por um editor de texto. Um outro aspecto importante é a compatibilidade entre plataformas; arquivos no formato texto possibilitam a troca de informações entre aplicativos de diferentes fornecedores. Entretanto, comparados com arquivos no formato binário, os arquivos no modo texto são maiores e a execução é mais lenta, pois ela necessita de mais operações.

Os arquivos no formato texto são compostos por linhas, sendo cada linha constituída por uma série de caracteres. Cada linha tem no seu final um caractere de retorno de carro. Para tornar o pacote visível à entidade, é necessário incluir a cláusula " USE STD.TEXTIO.ALL ".

No pacote são definidos os tipos " TEXT ", " LINE " e " SIDE ", e o subtipo " WIDTH " (ver Quadro 13.2.1). O tipo " LINE " é um ponteiro para um valor do tipo " STRING ". As operações de escrita e leitura de dados em arquivos são executadas com objetos definidos segundo esse tipo. O tipo " TEXT " é um tipo de arquivo composto por elementos do tipo " STRING ".

Objetos do tipo " LINE " são empregados nas operações com arquivos no formato texto (ver Figura 13.2.1). As operações de leitura e escrita de dados são executadas em duas etapas. Para o armazenamento de uma informação em um arquivo, deve-se primeiro transferir um ou mais dados para um objeto do tipo " LINE ". Uma vez transferidos os valores, o conteúdo do objeto tipo " LINE " é armazenado no arquivo. De maneira semelhante, na operação de leitura os valores são transferidos para um objeto do tipo " LINE ", e posteriormente os dados são extraídos do objeto.

O tipo " SIDE " é empregado nos procedimentos de escrita para definir a justificação dos dados no campo de escrita. O valor " RIGHT " indica que o valor está alinhado pelo lado direito do campo, e o valor " LEFT " pelo lado esquerdo. O subtipo " WIDTH " é utilizado para especificar o tamanho do campo empregado na escrita de um dado.

```
TYPE    LINE IS ACCESS STRING;      --  LINE e' um ponteiro para um tipo STRING
TYPE    TEXT IS FILE OF STRING;     --  Tipo FILE composto de elementos do tipo STRING
TYPE    SIDE IS (RIGHT, LEFT);      --  Valores empregados para o posicionamento do dado
SUBTYPE WIDTH IS NATURAL;           --  Empregado para definir o tamanho de uma informacao
```

Quadro 13.2.1 Tipos definidos no pacote " TEXTIO ".

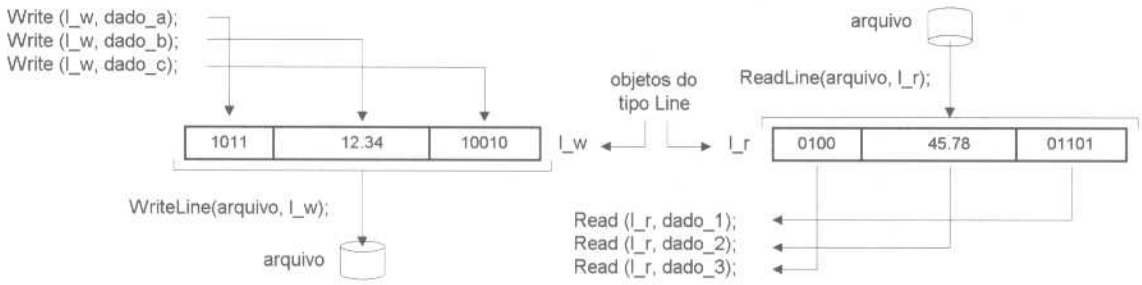


Figura 13.2.1 Operações envolvidas na escrita e leitura de dados — pacote " TEXTIO ".

O Quadro 13.2.2 apresenta, de modo simplificado, os procedimentos definidos no pacote " TEXTIO ". O procedimento " WRITE " insere o valor de " dado " no objeto do tipo " LINE " " linha_wr ". O procedimento " WRITELINE " escreve a informação contida no objeto tipo " LINE " " linha_wr ", no arquivo.

O procedimento " READLINE " executa a leitura em arquivo e retorna com a informação no objeto, tipo " LINE " " linha_rd ". O procedimento " READ " extrai um dos valores presentes no objeto, tipo " LINE " " linha_rd ", e retorna com o valor em " dado ".

Um objeto tipo " LINE " pode conter vários valores, pois esse tipo é um ponteiro para um valor " STRING ". A cada chamada do procedimento " WRITE ", um novo dado é inserido no objeto tipo " LINE ", e o ponteiro é movido para uma nova posição. A chamada do procedimento " WRITELINE " causa o retorno do ponteiro à posição inicial, e os valores contidos no objeto tipo " LINE " são perdidos. De modo semelhante, a cada chamada do procedimento " READ " um dado é removido do objeto tipo " LINE ", e o ponteiro é deslocado.

Os procedimentos " READ " e " WRITE " definidos no pacote " TEXTIO " sobrecarregam os procedimentos de mesmo nome definidos na linguagem para arquivos binários. Esses procedimentos do pacote " TEXTIO " são invocados quando um objeto do tipo " LINE " é empregado no campo identificado por " linha_wr " ou " linha_rd ". Em ambos os procedimentos, " dado " pode ser do tipo " BIT ", " BIT_VECTOR ", " BOOLEAN ", " CHARACTER ", " INTEGER ", " REAL ", " STRING " e " TIME ".

```
WRITE     (linha_wr, dado);
WRITELINE (arquivo_escrita, linha_wr);
```

```
READLINE  (arquivo_leitura, linha_rd);
READ      (linha_rd, dado);
```

Quadro 13.2.2 Subprogramas para operações com arquivos do pacote " TEXTIO ".

A Tabela 13.2.1 relaciona todos os procedimentos relativos a escrita definidos na linguagem. Comparando com o Quadro 13.2.2, verifica-se que os procedimentos " WRITE " são definidos com mais de dois parâmetros. Note que esses outros parâmetros já têm um valor inicial estabelecido; assim, eles podem ser omitidos na chamada do procedimento, conforme exposto no Quadro 13.2.2. O parâmetro " JUSTIFIED " especifica se o dado é alinhado à esquerda ou à direita do campo reservado para a escrita. Os valores válidos nesse parâmetro são do tipo " SIDE ": " RIGHT " ou " LEFT ". O parâmetro " FIELD " especifica o tamanho do campo reservado para o dado escrito. O valor " 0 " especifica um campo do tamanho exato para escrita sem nenhuma margem. O parâmetro " DIGITS " determina o número de dígitos à direita do ponto decimal que deve constar na escrita de um dado do tipo " REAL ". Um valor " 0 " indica que o valor é expresso com a mantissa normalizada e expoente, como, por exemplo, " 3.1416E00 ". No caso de um valor do tipo " TIME ", o parâmetro " UNITS " especifica a unidade empregada na representação do valor escrito.

Tabela 13.2.1 Procedimentos para escrita — pacote " TEXTIO " — tipos do pacote " STD ".

`WRITELINE (file F: TEXT; L: INOUT LINE);`
`WRITE (L: INOUT LINE; V: IN BIT;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN BIT_VECTOR;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN BOOLEAN;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN CHARACTER;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN INTEGER;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN STRING;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0);`
`WRITE (L: INOUT LINE; V: IN REAL;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0;  D: IN NATURAL:= 0);`
`WRITE (L: INOUT LINE; V: IN TIME;  J: IN SIDE:= RIGHT;  F: IN WIDTH := 0;  U: IN TIME:= ns);`
Designação não abreviada dos parâmetros formais: " V = Value ", " J = Justified ", " F = Field ", " D = Digits ", " U = Units "

As Figuras 13.2.2 e 13.2.3 ilustram a formatação resultante para valores do tipo " BIT ", " BIT_VECTOR ", " INTEGER ", " STRING ", " REAL " e " TIME ". Ao lado de cada procedimento " WRITE ", são apresentados o valor e tipo da constante empregada na operação. Na primeira figura, os dados foram alinhados à esquerda do campo. Note que o valor do tipo real, " n_real ", foi convertido para um formato não-normalizado, e o valor do tipo " TIME " foi convertido de " us " para unidade especificada; no caso, " ms ".

Na Figura 13.2.3 os procedimentos " WRITE " empregaram os valores iniciais definidos para os parâmetros " JUSTIFIED ", " FIELD , " DIGITS " e " UNITS ". Desse modo, o alinhamento é à direita, e o tamanho do campo é o necessário para a representação do valor. Com relação ao valor do tipo real, ele é apresentado no formato normalizado, e o valor do tipo " TIME " é ajustado para a unidade " ns ".

```
Write (linha, bits,       Left, 2);      -- bits := 1            tipo Bit
Write (linha, vetor_bit, Left, 5);       -- vetor_bit :=1010     tipo Bit_Vector
Write (linha, inteiro,    Left, 6);      -- inteiro := 98765     tipo Integer
Write (linha, carac,      Left, 8);      -- carac := exemplo     tipo String
Write (linha, n_real,     Left, 8, 5);   -- n_real := 12.3E-4    tipo Real
Write (linha, tempo,      Left, 9, ms);  -- tempo := 23 us       tipo Time
```

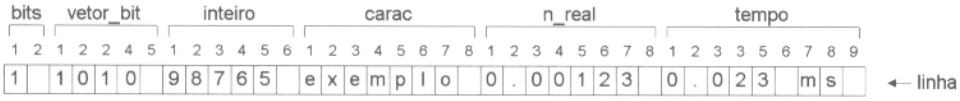


Figura 13.2.2 Procedimentos " WRITE " — pacote " TEXTIO " — exemplo 1.

```
Write (linha, bits);          -- bits := 1            tipo Bit
Write (linha, vetor_bit);     -- vetor_bit :=1010     tipo Bit_Vector
Write (linha, inteiro);       -- inteiro := 98765     tipo Integer
Write (linha, carac);         -- carac := exemplo     tipo String
Write (linha, n_real);        -- n_real := 12.3E-4    tipo Real
Write (linha, tempo);         -- tempo := 23 us       tipo Time
```

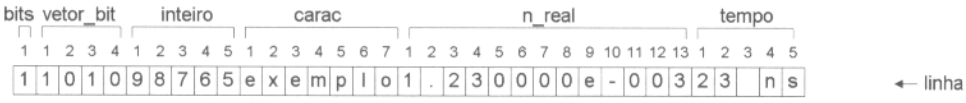


Figura 13.2.3 Procedimentos " WRITE " — pacote " TEXTIO " — exemplo 2.

Na Tabela 13.2.2 são apresentados todos os procedimentos definidos no padrão " TEXTIO " para operações de leitura. Conforme se pode observar, os procedimentos " READ " podem ter dois ou três parâmetros. O procedimento com três parâmetros permite identificar o resultado da operação no parâmetro " GOOD ". Ele retorna o

valor " TRUE ", caso a operação tenha sido realizada com êxito. O valor " FALSE " indica uma discrepância de formato entre o dado retornado em " VALUE " e o tipo de " VALUE ". Por exemplo, um valor contendo caracteres de texto, como " xyz ", retornando em um tipo numérico caracteriza um erro na operação, que pode ser verificado através do parâmetro " GOOD ".

A composição de um valor nos procedimentos " READ " é executada pela remoção de caracteres contidos em " L ". Se um caractere não pertencer ao conjunto de caracteres empregados na representação de valores do tipo de " VALUE ", o caractere é mantido em " L ", e a operação de composição do valor termina. Exceto para os tipos " CHARACTER " e " STRING ", os procedimentos " READ " pulam os caracteres espaço e tabulação presentes no parâmetro " L ". No caso dos tipos " INTEGER " e " REAL ", são aceitos os caracteres representando sinal, desde que justapostos ao valor numérico. Note que a leitura de um objeto do tipo linha conforme o formato apresentado no Quadro 13.2.3 pode ser problemática, devido à dificuldade de ser identificado o término de um valor.

Tabela 13.2.2 Procedimentos para leitura — pacote " TEXTIO " — tipos do pacote " STD ".

READLINE (file F: TEXT; L: OUT LINE);
READ (L: INOUT LINE; VALUE: OUT BIT);
READ (L: INOUT LINE; VALUE: OUT BIT_VECTOR);
READ (L: INOUT LINE; VALUE: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT CHARACTER);
READ (L: INOUT LINE; VALUE: OUT INTEGER);
READ (L: INOUT LINE; VALUE: OUT REAL);
READ (L: INOUT LINE; VALUE: OUT STRING);
READ (L: INOUT LINE; VALUE: OUT TIME);
READ (L: INOUT LINE; VALUE: OUT BIT; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT BIT_VECTOR; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT BOOLEAN; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT CHARACTER; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT INTEGER; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT REAL; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT STRING; GOOD: OUT BOOLEAN);
READ (L: INOUT LINE; VALUE: OUT TIME; GOOD: OUT BOOLEAN);

Versão VHDL-1987

O emprego do pacote " TEXTIO " na versão VHDL-1987 segue as restrições dessa versão: não é possível fechar um arquivo durante a simulação; o arquivo é aberto no momento da declaração de um objeto do tipo arquivo; e o arquivo é fechado ao término da simulação. Os objetos do tipo arquivo são declarados como tipo " TEXT " (ver parte superior do Quadro 13.2.3). Ainda no mesmo quadro, são apresentados exemplos de declaração de objetos do tipo " TEXT ". O primeiro define um arquivo de leitura, e o segundo, um arquivo de escrita.

```
FILE nome_objeto : TEXT   IS modo "nome_externo";

FILE leitura     : TEXT   IS  IN   "/vhdl/dados_r.dat"; -- objeto: arquivo leitura
FILE escrita     : TEXT   IS OUT   "/vhdl/dados_w.dat"; -- objeto: arquivo escrita
```

Quadro 13.2.3 Declaração de um objeto do tipo arquivo texto — pacote " TEXTIO " — VHDL-1987.

A descrição do Quadro 13.2.4 armazena os valores entre " 48 " e " 57 " em um arquivo. A cada iteração do laço, um dado é escrito na variável tipo " LINE " denominada " linha ". Ao final da iteração, o procedimento " WRITELINE " é executado, e o conteúdo da variável " linha " é armazenado no arquivo. Essa descrição é si-

milar à proposta no Quadro 13.1.4; o conteúdo dos arquivos criados, entretanto, é diferente. No caso do arquivo no formato binário, um editor de texto apresenta os caracteres ASCII correspondentes: " 0 1 ... 9 ". No arquivo criado, os valores são armazenados no formato texto, resultando em: " 48 49 ... 57 ". Note que cada valor de " dado " é separado por um caractere de espaço (ver linha 17).

```
 1 -- versao VHDL 1987
 2 -- termine a execucao da simulacao para fechar arquivo
 3 ENTITY arqv_a1 IS
 4 END arqv_a1;
 5
 6 USE STD.TEXTIO.ALL;  -- inclui o pacote TEXTIO
 7
 8 ARCHITECTURE teste OF arqv_a1 IS
 9 BEGIN
10   escreve: PROCESS
11     FILE arquivo_wr  : TEXT IS OUT "/vhdl/dado_a1.dat";
12     VARIABLE linha   : LINE;
13     VARIABLE espaco  : CHARACTER := ' ';
14   BEGIN
15     FOR dado IN 48 TO 57 LOOP
16       WRITE (linha, dado);       -- escreve o texto 48 49 ...57
17       WRITE (linha, espaco);     -- separados por um espaco
18     END LOOP;
19     WRITELINE (arquivo_wr, linha); -- armazena no arquivo
20     WAIT;
21   END PROCESS escreve;
22 END teste;
```

Quadro 13.2.4 Exemplo para escrita de dados em um arquivo no formato texto.

A descrição do Quadro 13.2.5 realiza a leitura de um arquivo no formato texto. A operação é semelhante à realizada na descrição do Quadro 13.1.5; nesse caso, entretanto, o formato do arquivo é do tipo texto. Note que o resultado no sinal " dado " será o mesmo em ambos os casos. Isso ocorre, porque os valores armazenados na forma de texto são convertidos para o tipo inteiro no procedimento " READ ".

Com relação ao código, observe que dois laços foram empregados. O primeiro, definido na linha 17, verifica se o fim do arquivo foi atingido. O segundo, definido na linha 20, executa um teste para detectar o fim dos dados presentes no objeto tipo " LINE ". A condição de fim é armazenada na variável " leitura_ok ", associada ao parâmetro " GOOD " do procedimento " READ ". Ela retorna com um valor falso, caso a operação de leitura não tenha sido realizada com sucesso. Novamente, foi inserido um comando " WAIT " apenas para permitir a observação dos dados no simulador (ver linha 23).

```
 1 -- versao VHDL 1987
 2 -- termine a execucao da simulacao para fechar arquivo
 3 USE STD.TEXTIO.ALL;
 4
 5 ENTITY arqv_b1 IS
 6   PORT (dado : OUT INTEGER);
 7 END arqv_b1;
 8
 9 ARCHITECTURE teste OF arqv_b1 IS
10 BEGIN
11   le: PROCESS
12     FILE     arquivo_rd : TEXT IS IN "/vhdl/dado_a1.dat";
13     VARIABLE linha      : LINE;
14     VARIABLE valor      : INTEGER;
15     VARIABLE leitura_ok : BOOLEAN;
16   BEGIN
17     WHILE NOT ENDFILE(arquivo_rd) LOOP
18       leitura_ok := TRUE;                -- habilita laco linha 20
19       READLINE (arquivo_rd, linha);      -- leitura de uma linha
20       WHILE leitura_ok LOOP
21         READ (linha, valor, leitura_ok); -- leitura dos dados de uma linha
22         dado <= valor;
23         WAIT FOR 10 ns;
24       END LOOP;
25     END LOOP;
26     WAIT;
27   END PROCESS le;
28  END teste;
```

Quadro 13.2.5 Exemplo para leitura de dados em um arquivo no formato texto — VHDL 1987.

A Figura 13.2.4 ilustra a simulação da descrição do Quadro 13.2.5 supondo um arquivo contendo dados, conforme mostrado no topo da figura. Note que o final dos dados na variável " linha " é detectado na décima primeira iteração do laço da linha 20. Nessa iteração ocorre uma repetição de valor apresentado na saída " dado ".

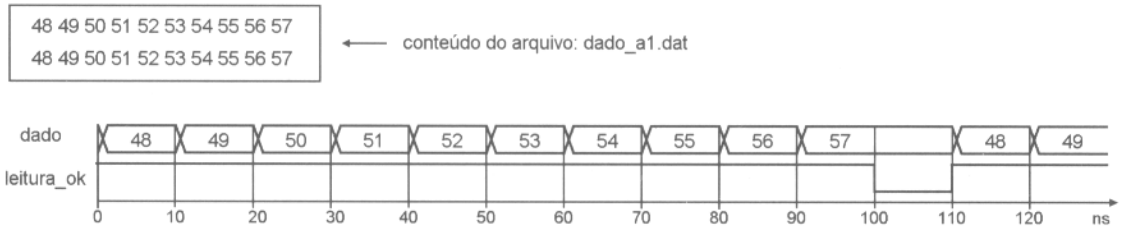


Figura 13.2.4 Simulação da descrição do Quadro 13.2.5.

Versão VHDL-1993

O emprego do pacote " TEXTIO " na versão VHDL-1993 pode lançar mão das novas operações disponíveis nessa atualização. No Quadro 13.2.6 são ilustradas as opções para abertura do arquivo na declaração de um objeto tipo arquivo texto, e a opção para apenas declarar um objeto tipo arquivo texto; primeiro, segundo e terceiro exemplos, respectivamente.

```
FILE leitura     : TEXT    OPEN Read_Mode IS  "nome_externo";
FILE escrita     : TEXT    OPEN Write_Mode IS "nome_externo";
FILE nome_objeto : TEXT;
```

Quadro 13.2.6 Declaração de um objeto tipo arquivo texto — versão VHDL-1993.

No Quadro 13.2.7 é apresentado um código que executa a escrita de uma linha, contendo os caracteres de " a " a " z ", em um arquivo. O objeto tipo arquivo texto é definido na linha 9, e o arquivo é aberto para escrita na linha 16. O laço na linha 17 define o conteúdo da variável tipo " LINE " que será inserido no arquivo. Após o armazenamento, o arquivo é fechado, e posteriormente aberto, novamente, no modo leitura para recuperação da informação armazenada. Para leitura da informação no arquivo, o procedimento " READLINE " é executado uma única vez, sem teste de final de arquivo (ver linha 24). Os valores recuperados são extraídos um a um através do laço da linha 25. A construção " WAIT FOR " foi incluída, assim como em descrições anteriormente discutidas, para permitir a visualização dos resultados no sinal " dado ".

```
 1  -- versao VHDL-1993
 2  USE STD.TEXTIO.ALL;
 3
 4  ENTITY arqv_e1 IS
 5    PORT(dado  : OUT CHARACTER);  -- caracteres recuperados no arquivo
 6  END arqv_e1;
 7
 8  ARCHITECTURE teste OF arqv_e1 IS
 9     FILE arq_wr_rd : TEXT;
10   BEGIN
11    escreve_le: PROCESS
12      VARIABLE linha      : LINE;             -- objeto tipo LINE para armazenar texto
13      VARIABLE leitura_ok : BOOLEAN := TRUE;  -- verificacao do procedimento READ
14      VARIABLE valor      : CHARACTER;        -- dado recuperado no arquivo
15    BEGIN
16      FILE_OPEN(arq_wr_rd, "/vhdl/dado_e1.dat",Write_Mode);
17      FOR ch IN 'a' TO 'z' LOOP
18        WRITE(linha, ch);                   -- escreve valores na linha
19      END LOOP;                             --
20      WRITELINE (arq_wr_rd, linha);         -- armazena no arquivo
21      FILE_CLOSE(arq_wr_rd);                -- fecha o arquivo
22
23      FILE_OPEN(arq_wr_rd, "/vhdl/dado_e1.dat",Read_Mode);
24      READLINE (arq_wr_rd, linha);          -- leitura de uma linha
25      WHILE leitura_ok LOOP
26        READ (linha, valor, leitura_ok);    -- leitura dos dados de uma linha
27        dado <= valor;
28        WAIT FOR 10 ns;                     -- temporizacao para mostra "dado" no simulador
29      END LOOP;
30      FILE_CLOSE(arq_wr_rd);                -- fecha o arquivo
31      WAIT;
32    END PROCESS escreve_le;
33  END teste;
```

Quadro 13.2.7 Exemplo de leitura e escrita em um arquivo no formato texto — VHDL-1993.

13.3 Principais pontos abordados

- A declaração de objetos representando arquivos emprega um tipo arquivo. Na declaração de um tipo arquivo é identificado o tipo dos elementos que o arquivo pode armazenar. Exemplo:
  " TYPE arquivo_de_tipo_x IS FILE OF tipo_contido; ".
- A declaração do objeto tipo arquivo associa um identificador ao nome do arquivo no sistema hospedeiro.
  Exemplos para versão VHDL-1987:
  - " FILE abc : arquivo_de_tipo_x IS IN "/vhdl/dados_r.dat"; ";
  - " FILE def : arquivo_de_tipo_x IS OUT "/vhdl/dados_w.dat"; ".
  Exemplos para versão VHDL-1993:
  - " FILE abc : arquivo_de_tipo_x OPEN Read_Mode IS "/vhdl/dados_r.dat"; ";
  - " FILE def : arquivo_de_tipo_x OPEN Write_Mode IS "/vhdl/dados_w.dat"; ";
  - " FILE ghi : arquivo_de_tipo_x ".
- A abertura de um arquivo na versão VHDL-1987 ocorre na declaração de um objeto do tipo arquivo, e o arquivo é fechado somente no final da simulação. Na versão VHDL-1993 é possível abrir um arquivo na declaração de um objeto do tipo arquivo, ou através do subprograma " FILE_OPEN ". O subprograma " FILE_CLOSE " permite fechar um arquivo durante a simulação.
- Arquivos no formato texto podem ser criados com o auxílio do pacote " TEXTIO ". Os arquivos são estruturados na forma de linhas contendo caracteres.
- No pacote " TEXTIO " é definido o tipo " LINE ": um ponteiro para um valor do tipo " STRING . Um objeto do tipo " LINE " contém os caracteres que serão armazenados em um arquivo numa operação de escrita, ou os caracteres que foram recuperados de um arquivo numa operação de leitura.
- Em arquivos do formato binário, as operações de leitura e escrita de dados em arquivos são executadas pelos procedimentos " READ " e " WRITE ".
- Em arquivos do formato texto, as operações de leitura e escrita de dados em arquivos são executadas pelos procedimentos " READLINE " e " WRITELINE ". Os procedimentos " READ " e " WRITE " são sobrecarregados. Eles removem ou inserem dados em objetos do tipo " LINE ".

13.4 Exercícios

13.4.1 Apresente um código para armazenar os valores contidos na constante " dado_escrito " em um arquivo na forma binária (ver Quadro 13.4.1). Na mesma descrição, após a escrita dos dados, realize uma operação de leitura dos valores armazenados, e apresente-os numa porta de saída. Para visualizar os dados, execute a operação de leitura em intervalos de tempo de 10 ns. Empregue a versão VHDL-1993.

```
TYPE vetor_2d IS ARRAY(0 TO 7) OF STRING(1 TO 3);
CONSTANT dado_escrito : vetor_2d := ("abc","def","ghi","jkl","mno","pqr","stu","vxz");
```

Quadro 13.4.1 Exercício 13.4.1.

13.4.2 Altere o código proposto no Exercício 13.4.1, definindo um objeto tipo arquivo contendo vetores cujos limites não foram definidos na declaração do tipo. O objetivo dessa descrição é verificar a operação do procedimento de leitura " READ ". Nessas condições, o procedimento retorna o dado e as dimensões reais do vetor recuperado na operação de leitura (ver Quadro 13.1.3). Apresente em portas de saída distintas os valores recuperados no arquivo e o tamanho do vetor. Simule a descrição, e compare o resultado com a Figura 13.1.2, para diferentes dimensões do objeto da classe vetor que retorna com o dado lido no arquivo.

13.4.3 Altere a posição do procedimento " WRITELINE " do código apresentado no Quadro 13.2.4 para o interior do laço, conforme apresentado no Quadro 13.4.2. Simule e compare os arquivos gerados em ambos os casos, e discuta o motivo da alteração do formato dos arquivos.

```
14 BEGIN
15     FOR dado IN 48 TO 57 LOOP
16       WRITE (linha, dado);
17       WRITE (linha, espaco);
18       WRITELINE (arquivo, linha); -- posicao alterada
19     END LOOP;
20     WAIT;
21   END PROCESS escreve;
22 END teste;
```

Quadro 13.4.2 Trecho do código do Quadro 13.2.4 alterado para o Exercício 13.4.3.

13.4.4 A simulação da descrição do Quadro 13.2.5, apresentada na Figura 13.2.4, mostra que ocorre uma repetição do último valor contido no objeto do tipo " LINE ", quando uma nova linha é recuperada no arquivo. Isso pode resultar em incorreções, caso a descrição seja empregada para a realização de testes. Altere a descrição de modo a remover esse problema, conforme ilustrado na Figura 13.4.1, onde os dados são apresentados continuamente.

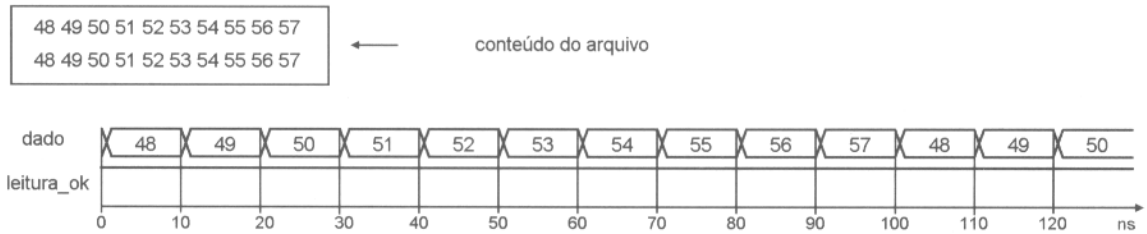


Figura 13.4.1 Resultado da alteração proposta da descrição do Quadro 13.2.5.

13.4.5 Escreva uma descrição que permita armazenar e recuperar as informações em três arquivos. Um arquivo deve conter informações do tipo " BIT ", o outro informações do tipo " BIT_VECTOR ", e o último informações do tipo " RECORD ". A Figura 13.4.2 apresenta uma sugestão de valores, dimensões dos vetores, e elementos para o tipo " RECORD ".

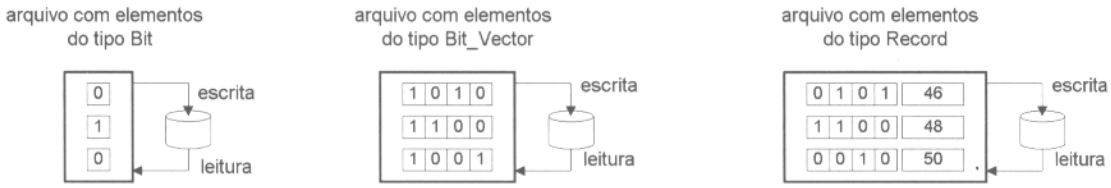


Figura 13.4.2 Ilustração para o Exercício 13.4.5.

13.4.6 Desenvolva o código de uma entidade que converta um arquivo no formato texto para um arquivo na forma binária (ver Figura 13.4.3). A entidade de conversão recupera os dados correspondentes ao tipo " BIT " no arquivo texto. Conjuntamente com essa operação, os dados devem ser armazenados no arquivo binário como dados do tipo " BIT ".

Para verificar se o arquivo foi criado corretamente, proponha uma segunda descrição que recupere os dados armazenados no formato binário. Para o arquivo no formato texto, crie um arquivo com um editor de texto contendo, por exemplo, quatro colunas de caracteres " 0 " e " 1 ".

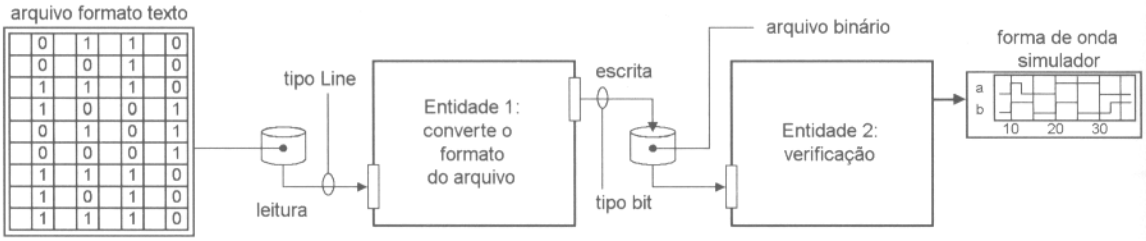


Figura 13.4.3 Ilustração para o Exercício 13.4.6.

13.4.7 Desenvolva o código de uma entidade que converta um arquivo no formato texto para um arquivo no formato binário, operando de forma similar ao código do exercício anterior. Nesse caso, entretanto, as informações contidas no arquivo texto correspondem a dados de tipos " STD_LOGIC " e " STD_LOGIC_VECTOR ".

Duas dificuldades devem ser encontradas neste exercício. A primeira advém do fato de que o procedimento " READ " para objetos do tipo " LINE " pode retornar apenas tipos definidos no pacote padrão (ver Tabela 13.2.2). Assim, é necessário implementar um subprograma para converter, por exemplo, um valor do tipo " CHARACTER " originado do procedimento " READ " para um valor do tipo " STD_LOGIC ". A segunda está relacionada com o arquivo binário. Como devem ser armazenados tipos " STD_LOGIC " e " STD_LOGIC_VECTOR ", o arquivo binário deve ser composto por elementos do tipo " RECORD " para comportar os tipos diferentes.

Novamente, proponha uma segunda descrição para certificar que o arquivo foi criado corretamente. A Figura 13.3.4 contém uma sugestão para o arquivo no formato texto. Ele é composto por quatro colunas: duas com dados correspondentes a valores do tipo " STD_LOGIC ", e duas com dados correspondentes a valores do tipo " STD_LOGIC_VECTOR ".

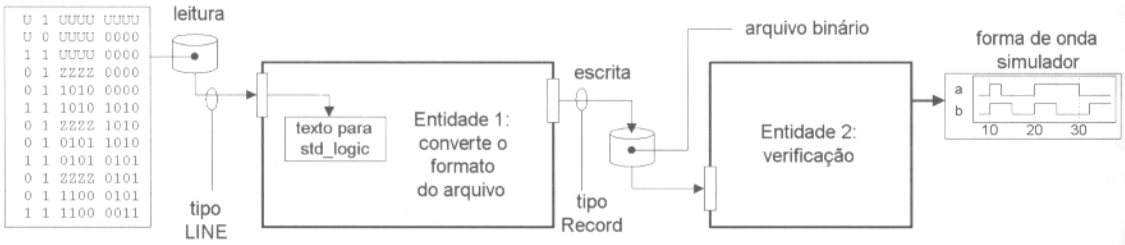


Figura 13.4.4 Ilustração para o Exercício 13.4.7.

CAPÍTULO

14 TÓPICOS AVANÇADOS

Este capítulo contém tópicos que refinam o conhecimento da linguagem: comandos para apresentar mensagens durante a simulação, a criação de funções de resolução, configurações e comandos pospostos.

O conhecimento das operações envolvidas na definição de uma função de resolução e tipos com função de resolução auxilia a compreensão dos tipos definidos no padrão IEEE-1164. Configuração é um recurso avançado da linguagem que permite definir o elo entre componentes e entidades num projeto hierárquico. Comandos pospostos têm a sua execução condicionada ao término das iterações que ocorrem na avaliação dos comandos concorrentes.

14.1 Comando " ASSERT "

O comando " ASSERT " é um comando seqüencial usado para notificar condições ilegais. Ele pode ser empregado em conjunto com as cláusulas " REPORT " e " SEVERITY ". O formato da construção é apresentado no Quadro 14.1.1. A condição de teste do comando corresponde ao requisito correto, isto é, quando a condição é verdadeira, nenhuma ação é tomada. Nessa condição, as cláusulas " REPORT " e " SEVERITY ", se existentes, não são consideradas.

A cláusula " REPORT " é opcional, e especifica uma mensagem a ser apresentada pelo simulador. A mensagem deve ser uma expressão do tipo " STRING ". Na ausência da cláusula, é retornada a mensagem: " Assertion violation ".

A cláusula " SEVERITY " é opcional, e identifica o nível de gravidade da condição atingida. A expressão deve ser um tipo " SEVERITY_LEVEL " definido com os valores: " FAILURE ", " ERROR ", " WARNING " e " NOTE ". Na ausência da cláusula, é assumido o nível " ERROR ". Os simuladores possuem, normalmente, opções quanto à ação a ser tomada nesses casos. É possível, por exemplo, abortar a simulação, se um determinado nível for atingido.

```
ASSERT condicao                          -- condicao = TRUE (nada ocorre); FALSE (mensagem exibida)
  REPORT expressao_tipo_texto            -- opcional, expressao que retorna um tipo STRING
    SEVERITY expressao_de_severidade;    -- opcional: FAILURE, ERROR, WARNING, NOTE
```

Quadro 14.1.1 Formato do comando " ASSERT ".

O código do Quadro 14.1.2 permite testar a operação do comando " ASSERT ". Ele contém dois processos. O primeiro descreve o comportamento de um *flip flop*, e o segundo confere se ocorreu uma violação de temporização. A Figura 14.1.1 ilustra a temporização que deve ser respeitada: a informação na entrada " d " deve estar estável por um período de tempo maior do que " 30 ns " com relação à borda de subida de " ck ".

O comando " WAIT " da linha 17 condiciona a execução do processo " verifica " a cada borda de subida do sinal " ck ". Assim, toda vez que o processo sai do estado suspenso, é possível determinar o tempo decorrido entre a borda de subida de " ck " e o último evento do sinal " d " através do atributo " LAST_EVENT ".

O requisito correto do comando " ASSERT " é " d'LAST_EVENT > set_up ". Nessa condição, a temporização não foi violada, e nenhuma mensagem é gerada. Supondo uma seqüência de estímulos, conforme apresentado na

Figura 14.1.1, uma violação de temporização ocorre no instante " 250ns ". O Quadro 14.1.3 ilustra a mensagem gerada nessa condição pelo simulador quando a violação é detectada. Note que são apresentados a mensagem contida na linha 19 e o nível de gravidade declarado na linha 20.

```
 1  ENTITY ass_3a IS
 2    GENERIC (set_up  : TIME := 30 ns);
 3    PORT (d, ck  : IN  BIT;
 4          q      : OUT BIT);
 5  END ass_3a;
 6
 7  ARCHITECTURE teste OF ass_3a IS
 8  BEGIN
 9    operacao: PROCESS(ck, d)
10    BEGIN
11      IF ck'EVENT AND ck='1' THEN q <= d;
12      END IF;
13    END PROCESS;
14
15    verifica: PROCESS                          -- processo e' executado a cada borda subida de ck
16    BEGIN                                      -- d'LAST_EVENT = intervalo de tempo ocorrido entre
17      WAIT UNTIL (ck'EVENT AND ck='1');        --    instante atual e ultimo evento em "d"
18      ASSERT d'LAST_EVENT > set_up             -- d'last_event > set_up condicao correta,
19        REPORT " Tempo set_up nao respeitado " --      se falso mensagem e' apresentada
20        SEVERITY ERROR;
21    END PROCESS;
22  END teste;
```

Quadro 14.1.2 Código para teste do comando " ASSERT ".

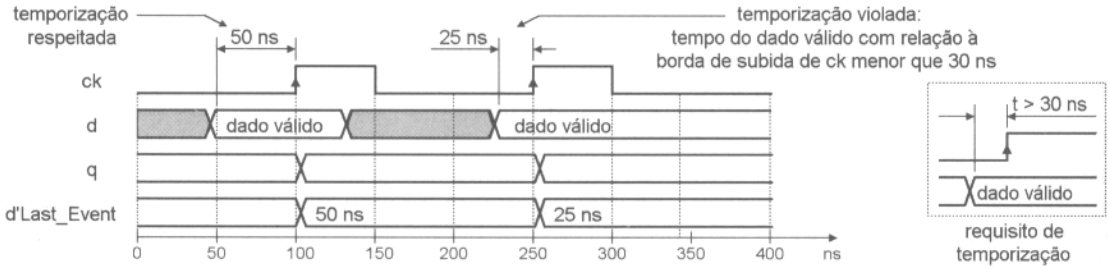


Figura 14.1.1 Requisito de temporização estabelecido na descrição do Quadro 14.1.2.

```
# ** Error:  Tempo set_up nao respeitado
#    Time: 250 ns  Iteration: 0  Instance: /ass_3a
```

Quadro 14.1.3 Mensagem na janela do simulador — simulação da entidade " ass_3a ".

O comando " ASSERT " pode ser inserido em uma região de código concorrente. Nesse caso, existe um processo equivalente contendo um comando " ASSERT ". A estrutura do comando segue o mesmo formato apresentado no Quadro 14.1.1. Se a condição da asserção for falsa, a mensagem de erro é notificada pelo simulador.

No Quadro 14.1.4 é apresentada uma descrição para o teste do comando " ASSERT " numa região de código concorrente. Essa descrição verifica se os valores dos sinais " t1 " e " t2 " têm o mesmo nível lógico, empregando a função " std_match " do pacote " numeric_std ". Se a função retornar o valor falso, a mensagem é apresentada pelo simulador. Note, na linha 12, que após a mensagem foram introduzidos os caracteres de retorno de carro, " CR ", e avanço de linha " LF ". Esses caracteres levam à apresentação das mensagens em linhas sucessivas na janela do simulador.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3 USE ieee.numeric_std.ALL;
 4
 5 ENTITY ass_2 IS
 6   PORT (t1, t2 : IN  STD_LOGIC);
 7 END ass_2;
 8
 9 ARCHITECTURE teste OF ass_2 IS
10  BEGIN                                --(nivel de t1= nivel de t2) condicao correta,
11   ASSERT std_match(t1,t2)             --         caso falso mensagem e' apresentada
12    REPORT "Nivel logico de t1 diferente do nivel logico de t2" & CR & LF
13    SEVERITY ERROR;
14 END teste;
```

Quadro 14.1.4 Código para teste do comando " ASSERT " — região de código concorrente.

14.2 Comando " REPORT "

O comando " REPORT " é um comando seqüencial usado para relatar mensagens, disponível na versão VHDL-1993. Ele pode ser empregado em conjunto com a cláusula opcional " SEVERITY ", conforme apresentado no Quadro 14.2.1. Como nenhuma condição é associada à execução do comando, o comando " REPORT " somente é permitido em regiões de código seqüencial. O objetivo dessa nova opção na versão VHDL-1993 é evitar o uso do comando " ASSERT " com a condição " FALSE " quando se necessita apresentar sempre uma mensagem.

```
REPORT expressao_tipo_texto            --  expressao que retorna um tipo STRING
  SEVERITY expressao_de_severidade;    -- opcional: FAILURE, ERROR, WARNING, NOTE
```

Quadro 14.2.1 Formato do comando " REPORT ".

O Quadro 14.2.2 contém um exemplo para a verificação da operação do comando " REPORT ". A simulação do código tem como resultado a apresentação do valor da porta " dado ". Observe que a mensagem é uma expressão contendo a concatenação entre um texto e o valor retornado pela função " para_string ". Essa função (ver linhas 10 a 17) converte o valor tipo " STD_LOGIC " de " dado " para o tipo " STRING ". Os valores " CR " e " LF " pulam uma linha e retornam o cursor na janela onde as informações são apresentadas pelo simulador (ver exemplo de uma mensagem gerada, no Quadro 14.2.3).

```
 1 -- versao VHDL-1993
 2 LIBRARY ieee;
 3 USE ieee.std_logic_1164.ALL;
 4
 5 ENTITY rep_0 IS
 6   PORT (dado : IN  STD_LOGIC);
 7 END rep_0;
 8
 9 ARCHITECTURE teste OF rep_0 IS
10   FUNCTION para_string (a : STD_LOGIC) RETURN string IS
11     BEGIN
12      CASE a IS
13        WHEN '0' => RETURN " 0"; WHEN 'L' => RETURN " L";
14        WHEN '1' => RETURN " 1"; WHEN 'H' => RETURN " H";
15        WHEN 'X' => RETURN " X"; WHEN 'U' => RETURN " U"; WHEN OTHERS => RETURN " -";
16       END CASE;
17   END para_string;
18 BEGIN
19   operacao: PROCESS(dado)
20   BEGIN
21     REPORT "Valor dado= " & para_string(dado) & CR & LF
22     SEVERITY NOTE;
23   END PROCESS;
24 END teste;
```

Quadro 14.2.2 Código para teste do comando " REPORT ".

```
# ** Note: Valor dado=  U
#
#
#    Time: 0 ns  Iteration: 0  Instance: /rep_0
```

Quadro 14.2.3 Mensagem na janela do simulador — simulação da entidade " rep_0 ".

14.3 Tipo com função de resolução

Uma função de resolução determina o valor resultante de um sinal acionado por mais de um controlador, ou *driver*. Essa função é necessária, porque não é permitido mais de um controlador acionar um mesmo sinal sem uma função de resolução.

O Quadro 14.3.1 contém o exemplo de uma seqüência de comandos para a definição de um tipo com uma função de resolução. O primeiro passo é definir um tipo enumerado contendo os valores que o tipo com função de resolução pode assumir (" Sem_resolucao ", no exemplo). Em seguida, é declarado um vetor com os limites em aberto, contendo elementos do tipo recém-declarado (" Sem_resolucao_vetor ", no exemplo). Note que esses dois tipos não têm uma função de resolução associada.

Uma vez declarados esses dois tipos, é declarada a função de resolução. Ela tem um único argumento de entrada (função " resolver ", no exemplo). O parâmetro formal da função é um vetor unidimensional com os limites em aberto da classe constante. Cada elemento desse vetor corresponde a um dos possíveis controladores que acionam o sinal; daí a necessidade de um vetor sem limites definidos, conforme declarado no segundo passo. A função de resolução é invocada, toda vez que um sinal com função de resolução é ativo. Ela analisa o valor de todos os elementos do vetor e toma a decisão quanto ao valor resultante.

A função de resolução retorna um argumento sem uma função de resolução associada. Isso porque ainda não foi definido o tipo com função de resolução. O tipo com função de resolução (" Com_resolucao ", no exemplo) é declarado através de um subtipo que inclui o nome da função de resolução associada (ver quarto passo).

O Quadro 14.3.2 apresenta exemplos de declarações de subtipos. Os exemplos da parte superior foram abordados no Capítulo 12. O exemplo da parte inferior é uma nova opção empregada na definição de sinais com função de resolução. Qualquer sinal declarado como um subtipo contendo uma função de resolução é um tipo com uma função de resolução.

```
-- primeiro passo: tipo sem funcao de resolucao
TYPE Sem_resolucao IS (valor_1, valor_2, valor_n);

-- segundo passo: vetor sem funcao de resolucao
TYPE Sem_resolucao_vetor IS ARRAY (NATURAL RANGE <>) OF Sem_resolucao;

-- terceiro passo: declaracao da funcao de resolucao
FUNCTION resolver(a : Sem_resolucao_vetor) RETURN Sem_resolucao;      -- funcao de resolucao

-- quarto passo: declaracao do tipo com funcao de resolucao
SUBTYPE Com_resolucao IS resolver Sem_resolucao;                      -- subtipo com resolucao
```

Quadro 14.3.1 Definição de um tipo com função de resolução.

```
-- Definicoes de subtipos abordadas no capitulo 12
--     identificador    IS    tipo_a    faixa_no_tipo_a
SUBTYPE maiuscula       IS  CHARACTER  RANGE 'A' TO 'Z';
SUBTYPE octeto          IS  BIT_VECTOR(7 DOWNTO 0);

--      identificador       tipo        faixa_nao_definida
SUBTYPE mesmo_limite_1 IS  maiuscula;
SUBTYPE mesmo_limite_2 IS  octeto;
```

```
-- Nova opcao de declaracao de um subtipo. Note, funcao_de_resolucao declarada anteriormente
-- Detalhamento do quarto passo:
--
--      identificador   funcao_de_resolucao   tipo
SUBTYPE Com_resolucao IS      resolver      Sem_resolucao;
```

Quadro 14.3.2 Definição de subtipos e opção de subtipo com função de resolução.

No Quadro 14.3.3 é apresentado o pacote " dre_ab_a " para declaração de um tipo com função de resolução que modela o comportamento de portas lógicas com saídas em dreno aberto, ou saídas em coletor aberto. Nessas portas lógicas, o nível lógico alto é imposto por uma resistência (ver Figura 14.3.1). Esse nível lógico, comparado com o nível lógico baixo, tem um nível de força menor. Assim, num circuito contendo várias saídas interligadas, o nível lógico resultante somente será alto se nenhuma saída estiver em nível lógico baixo.

O entendimento do código pode ser facilitado com uma analogia entre os tipos " STD_ULOGIC " e " STD_LOGIC " definidos no pacote " std_logic_1164 ". Conforme visto, o tipo " STD_LOGIC " possui função de resolução, e é um subtipo de " STD_ULOGIC ". No pacote definido no Quadro 14.3.3, " R_aberto " é um tipo com função de resolução cujo tipo base é " U_aberto ", um tipo sem função de resolução.

Como primeiro passo, é declarado o tipo " U_aberto " para definição dos valores que o tipo pode assumir: " H " e " 0 ". O valor " H " possui um nível de força mais baixo e corresponde ao nível lógico alto; o valor " 0 " possui um nível de força mais elevado e corresponde ao nível lógico baixo.

No segundo passo, é declarado o tipo " U_aberto_vetor ", um vetor de elementos " U_aberto ". A seguir, é declarada a função de resolução " resolver " com um parâmetro formal do tipo " U_aberto_vetor ". A operação da função é descrita no corpo do pacote (ver linhas 9 a 19). Finalmente, na linha 6, é declarado o subtipo com função de resolução " R_aberto ".

A atividade em um sinal declarado como " R_aberto " leva à execução da função de resolução. O valor transferido à função é um vetor, contendo os valores de todos os controladores do sinal. A função contém um laço que analisa todos os elementos do vetor, acumulando o resultado temporário na variável " final ". A variável é inicializada com o valor de menor força (no caso, " H "). O teste dos tipos declarados no pacote " dre_ab_a " pode ser realizado através da descrição do Quadro 14.3.4.

```
 1 PACKAGE dre_ab_a IS
 2   TYPE U_aberto IS ('H','0');                                  -- tipo sem funcao de resolucao
 3   TYPE U_aberto_vetor IS ARRAY (NATURAL RANGE <>) OF U_aberto; -- tipo sem funcao de resolucao
 4
 5   FUNCTION resolver(a : U_aberto_vetor) RETURN U_aberto;       -- funcao de resolucao
 6   SUBTYPE R_aberto IS resolver U_aberto;                       -- tipo com funcao de resolucao
 7 END dre_ab_a;
 8
 9 PACKAGE BODY dre_ab_a IS
10    FUNCTION resolver(a : U_aberto_vetor) RETURN U_aberto IS
11      VARIABLE final : R_aberto := 'H';
12    BEGIN
13      FOR i IN a'RANGE LOOP
14        IF a(i) /= 'H' THEN final := a(i);
15        END IF;
16      END LOOP;
17      RETURN final;
18    END resolver;
19 END dre_ab_a;
```

Quadro 14.3.3 Pacote com a definição de um tipo com função de resolução.

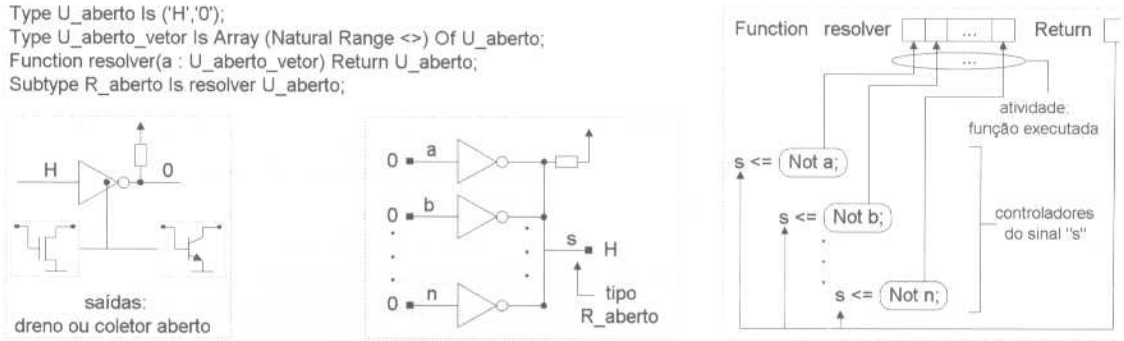


Figura 14.3.1 Ilustração das operações envolvidas no pacote do Quadro 14.3.3.

```
 1  USE WORK.dre_ab_a.ALL;
 2
 3  ENTITY dre_ab_b IS
 4    PORT (a, b : IN  R_aberto;
 5          s    : OUT R_aberto);
 6  END dre_ab_b;
 7
 8  ARCHITECTURE teste OF dre_ab_b IS
 9   BEGIN
10    s <= a;
11    s <= b;
12  END teste;
```

Quadro 14.3.4 Descrição para teste do pacote " dre_ab_a ".

Nem sempre o modo como a função de resolução é descrita no pacote do Quadro 14.3.3 é o mais conveniente. O Quadro 14.3.5 utiliza uma abordagem diferente, empregando tabelas para a descrição da função. A idéia é armazenar todas as condições em uma constante, e referenciá-la como uma memória a cada execução da função de resolução. O código dos dois pacotes é idêntico até a linha 6. Na linha 8 é declarado um tipo vetor de duas dimensões, que tem como índice elementos do tipo " U_aberto ", denominado " resolve_aberto ". Em seguida, é declarada uma constante empregando esse tipo que contém o valor resultante para cada possível combinação de valores do tipo " U_aberto ".

A função de resolução executa uma operação similar à descrita no pacote anterior. Nesse caso, entretanto, o valor resultante não é avaliado por um comando de teste a cada iteração do laço. O valor é determinado diretamente através do par de índices da constante " tabela ". O índice à esquerda corresponde ao valor acumulado na iteração, variável " final ", e o índice à direita corresponde ao valor atual do elemento de posição " i " do vetor " a ".

```
 1  PACKAGE dre_ab_c IS
 2    TYPE U_aberto IS ('H','0');                                  -- tipo sem funcao de resolucao
 3    TYPE U_aberto_vetor IS ARRAY (NATURAL RANGE <>) OF U_aberto; -- tipo sem funcao de resolucao
 4
 5    FUNCTION resolver(a : U_aberto_vetor) RETURN U_aberto;       -- funcao de resolucao
 6    SUBTYPE  R_aberto IS resolver U_aberto;                      -- tipo com funcao de resolucao
 7
 8    TYPE resolve_aberto IS ARRAY (U_aberto, U_aberto) OF U_aberto;
 9    CONSTANT tabela : resolve_aberto :=
10   -- ordem na tabela: do indice de menor valor para indice de maior valor (H para 0)
11   --  H    0
12    (('H', '0'),  -- H
13     ('0', '0')); -- 0
14  END dre_ab_c;
15
16  PACKAGE BODY dre_ab_c IS
17     FUNCTION resolver(a : U_aberto_vetor) RETURN U_aberto IS
18       VARIABLE final : R_aberto := 'H';
19     BEGIN
20       FOR i IN a'RANGE LOOP
21         final := tabela(final,a(i)); -- valor extraido da tabela
22       END LOOP;
23       RETURN final;
24     END resolver;
25  END dre_ab_c;
```

Quadro 14.3.5 Pacote com a definição de um tipo com função de resolução.

O Quadro 14.3.6 contém uma descrição para teste do pacote " dre_ab_c ". A descrição serve, também, para apresentar a opção de inserir o nome de uma função de resolução na declaração de um sinal. Nesse caso, a função de resolução é associada ao sinal, do mesmo modo que um subtipo declarado com uma função de resolução.

As portas de entrada e saída são do tipo " U_aberto ", sem função de resolução, portanto. A declaração do sinal " s_res " (linha 10) contém o nome da função de resolução " resolver ", resultando em um sinal com uma função de resolução. A declaração da linha 10 é equivalente à declaração " SIGNAL s_res : R_aberto; "

```
1  USE WORK.dre_ab_c.ALL;
2
3  ENTITY dre_ab_d IS
4    PORT (a, b : IN  U_aberto;  -- sem funcao de resolucao
5          s    : OUT U_aberto); -- sem funcao de resolucao
6  END dre_ab_d;
7
8  ARCHITECTURE teste OF dre_ab_d IS
9  --   identificador : funcao_de_resolucao
10   SIGNAL    s_res  : resolver    U_aberto;  -- sinal declarado com uma funcao de resolucao
11  BEGIN
12   s_res <= a;
13   s_res <= b;
14   s     <= s_res;
15 END teste;
```

Quadro 14.3.6 Descrição para teste do pacote " dre_ab_c ".

14.4 Configurações

O objetivo de uma configuração é estabelecer um elo entre a declaração de um componente e uma entidade de projeto. Conforme visto no Capítulo 7, o elo é naturalmente estabelecido se o nome do componente declarado e o nome da entidade de projeto forem os mesmos (ver Figura 14.4.1). Nesse caso, se a entidade tiver mais de uma arquitetura, o elo é estabelecido com a última arquitetura compilada.

Em projetos mais complexos, uma entidade de projeto pode ter várias arquiteturas, e o nome do componente solicitado pode ser diferente do nome da entidade com a qual se deseja estabelecer a interligação. Além disso, durante o desenvolvimento do projeto, pode-se desejar uma alteração das associações entre as solicitações de componentes e entidades, sem que seja necessário recompilar as entidades que solicitam esses elementos. A linguagem fornece dois mecanismos para atender esses requisitos denominados especificação de configuração e declaração de configuração.

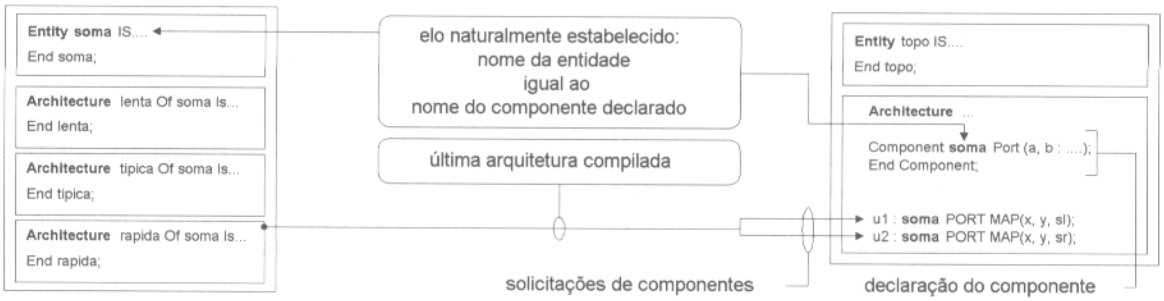


Figura 14.4.1 Elo natural entre o nome da entidade de projeto e o rótulo do componente.

Especificação de configuração

A especificação de configuração define o elo entre uma solicitação de componente e uma entidade de projeto. Ela é inserida na região de declaração de uma arquitetura que solicita os componentes. A Figura 14.4.2 ilustra o elo estabelecido entre dois componentes solicitados e uma entidade de projeto através da especificação de configuração. Nesse exemplo, é estabelecido um elo entre a entidade " soma " e as solicitações do componente " add " de rótulos " u1 " e " u2 ".

O Quadro 14.4.1 contém exemplos de especificação de uma configuração. Ela é composta pela especificação do componente e pela indicação do elo.

A especificação do componente contém uma lista de rótulos e o nome do componente. A lista de rótulos pode conter um ou mais rótulos, bem como as palavras reservadas " ALL " e " OTHERS ". Cada rótulo representa a solicitação de um componente declarado.

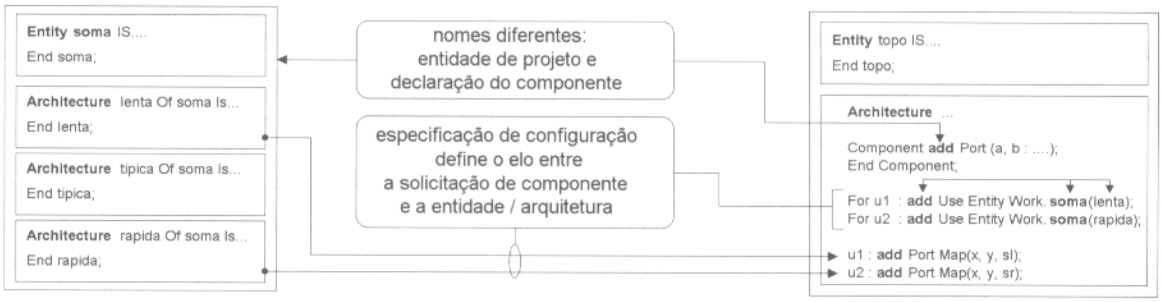


Figura 14.4.2 Elo definido através de uma especificação.

A indicação do elo associa o componente a uma entidade. Essa associação permite identificar a biblioteca em que a entidade está armazenada, o nome da entidade, a sua arquitetura, bem como permite definir mapas de genéricos e portas. Esses mapas são necessários quando os nomes empregados para os genéricos e portas na entidade são diferentes dos nomes empregados na declaração do componente.

```
   --- especificacao_do_componente ---   -------------- indicacao_do_elo ------------------------
FOR lista_de_rotulos : nome_componente   USE ENTITY WORK.nome_entidade(arquitetura);
FOR ALL              : porta_ou          USE ENTITY WORK.or_gate(lenta) GENERIC MAP(n => k);
FOR OTHERS           : inversor_tri      USE ENTITY WORK.inv_gate(ttl) PORT MAP(x1 => y1, x2 => y2);
FOR u1               : ram_rapida        USE ENTITY WORK.ram PORT MAP(adr => end);
FOR u2, u3           : ttl_138           USE PORT MAP (x1 => y1, x2 => y2);
```

Quadro 14.4.1 Especificação de configuração.

O Quadro 14.4.2 contém a entidade " soma " e suas três arquiteturas, as quais são empregadas na forma de componentes pela descrição do Quadro 14.4.3. Na entidade " somadores1 " são declarados dois componentes: " add " e " adx ". Na declaração do componente " add ", os nomes das portas coincidem com os nomes das portas da entidade " soma "; o mesmo não ocorre com o componente " adx " (ver linhas 9 a 15).

A especificação da configuração é definida entre as linhas 18 e 20. Para a solicitação " u1 ", que emprega a declaração de componente " add ", é estabelecido um elo com a entidade " soma ". Como não foi especificada a arquitetura, é empregada a última arquitetura compilada. A solicitação " u2 " emprega, também, a declaração " add "; para essa solicitação é estabelecido um elo com a arquitetura " lenta " da entidade " soma ". A solicitação " u3 " emprega a declaração de componente " adx "; para essa solicitação é estabelecido um elo com a arquitetura " tipica " de " soma ". Como neste último caso o nome das portas é diferente, é necessário definir a correspondência entre nomes através do " PORT MAP ".

```
 1  -- arquivo conf_00.vhd
 2  -- declaracao da entidade "soma" e suas 3 arquiteturas
 3  ENTITY soma IS
 4   PORT (a, b : IN INTEGER RANGE 0 TO 7; s : OUT INTEGER RANGE 0 TO 15);
 5  END soma;
 6
 7  ARCHITECTURE lenta OF soma IS
 8  BEGIN   s <= a + b AFTER 50 ns;
 9  END lenta;
10
11  ARCHITECTURE tipica OF soma IS
12  BEGIN   s <= a + b AFTER 25 ns;
13  END tipica;
14
15  ARCHITECTURE rapida OF soma IS
16  BEGIN   s <= a + b AFTER 10 ns;
17  END rapida;
```

Quadro 14.4.2 Entidade de projeto " soma " e suas três arquiteturas.

```
1  -- arquivo conf_1.vhd
2  -- Especificacao da configuracao
3  ENTITY somadores1 IS
4    PORT (x,  y  : IN  INTEGER RANGE 0 TO 7;
5          s1, st, sr : OUT INTEGER RANGE 0 TO 15);
6  END somadores1;
7
8  ARCHITECTURE teste OF somadores1 IS
9  --          "add": nome_local do componente -- portas com mesma designacao
10   COMPONENT add PORT (a, b : IN  INTEGER RANGE 0 TO 7; s : OUT INTEGER RANGE 0 TO 15);
11   END COMPONENT;
12
13 --          "adx": nome_local do componente -- portas com designacao diferente
14   COMPONENT adx PORT (k, l : IN  INTEGER RANGE 0 TO 7; m : OUT INTEGER RANGE 0 TO 15);
15   END COMPONENT;
16
17 -- associacao entre "add" e "adx" e as entidades de projeto "soma"
18   FOR u1 : add USE ENTITY WORK.soma;          -- arquitetura = ultima compilada
19   FOR u2 : add USE ENTITY WORK.soma(lenta);
20   FOR u3 : adx USE ENTITY WORK.soma(tipica) PORT MAP(a => k, b =>l, s => m);
21 BEGIN
22   u1 : add PORT MAP(x, y, sr);
23   u2 : add PORT MAP(x, y, s1);
24   u3 : adx PORT MAP(x, y, st);
25 END teste;
```

Quadro 14.4.3 Exemplo de especificação de configuração.

Declaração da configuração

Conforme visto, a especificação de configuração define o elo entre a solicitação de um componente e uma entidade de projeto, na arquitetura que emprega o componente. A declaração de configuração fornece meios para que essa configuração seja realizada, não mais na arquitetura que emprega o componente, mas, posteriormente, em uma outra unidade de projeto.

A Figura 14.4.3 ilustra as cinco unidades de projeto da linguagem. As quatro unidades de projeto discutidas em capítulos anteriores são: declaração de entidade, declaração de pacote, corpo da arquitetura e corpo do pacote. A declaração de configuração é uma outra unidade de projeto.

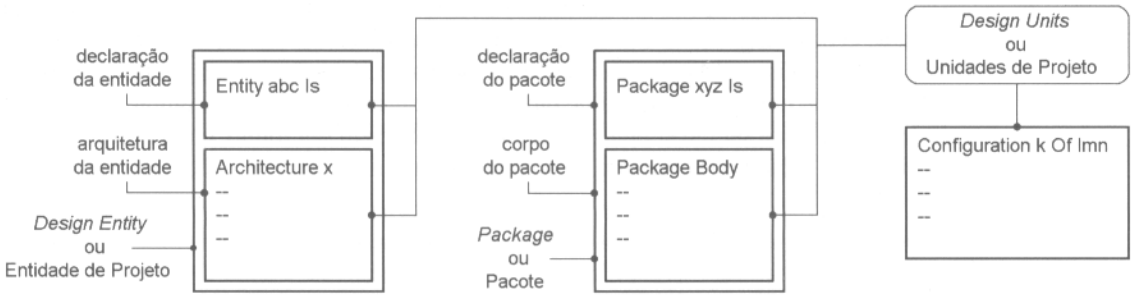


Figura 14.4.3 Unidades de projeto em VHDL.

A Figura 14.4.4 ilustra o elo estabelecido entre as solicitações de componentes e uma entidade através da declaração da configuração. Como a declaração de uma configuração é uma unidade de projeto, ela deve ser compilada para a execução da simulação. Note que ela está no nível mais alto da hierarquia do projeto, e a simulação é executada a partir dessa unidade de projeto.

A grande vantagem de a declaração de configuração ser uma unidade de projeto é a possibilidade de se alterar a configuração do projeto sem a necessidade de recompilar as unidades dependentes da configuração. No exemplo da Figura 14.4.4, os elos das solicitações de componentes na entidade " divisor " podem ser modificados recompilando-se, apenas, a configuração " versao_x ".

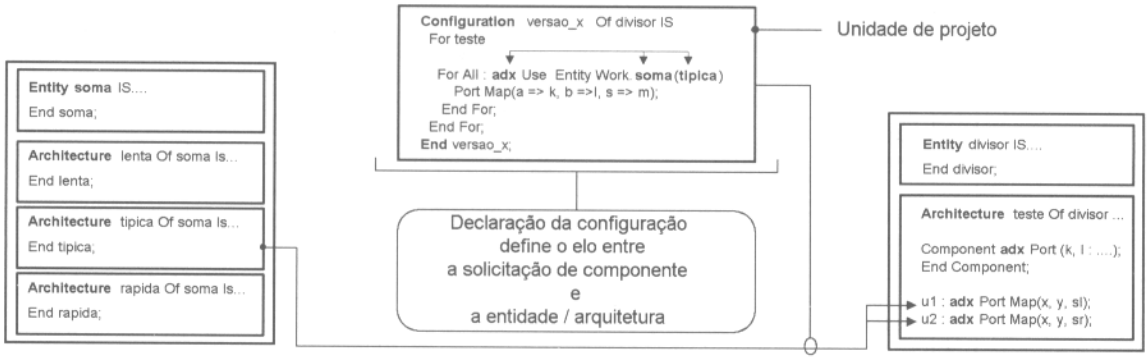


Figura 14.4.4 Elo definido através de uma declaração de configuração.

O Quadro 14.4.4 contém exemplos de declaração de configuração. A declaração contém dois nomes. O primeiro identifica o nome da configuração, e o segundo identifica a entidade de projeto a que a configuração se aplica. No interior da configuração podem ser inseridas configurações de bloco e cláusulas " USE ".

```
CONFIGURATION identificador_configuracao OF entidade_configurada IS
  -- Clausula USE.                    Exemplo: USE.WORK.ALL
  -- Configuracao de bloco.           Ver quadro seguinte
END identificador_configuracao;
```

```
CONFIGURATION versao_x OF divisor IS    -- "versao_x": unidade de projeto contendo uma configuracao
  --                                    -- "divisor":  unidade de projeto configurada por "versao_x"
  --
END versao_x;
```

Quadro 14.4.4 Exemplos de declaração de configuração.

Exemplos de opções para configuração de blocos são apresentados no Quadro 14.4.5. A configuração de bloco inicia com a palavra reservada " FOR " acompanhada do nome que identifica o bloco. Se a configuração do bloco seguir a declaração de configuração, o bloco a ser identificado é a arquitetura da entidade a que a configuração se aplica. Por exemplo, na Figura 14.4.4, a configuração " versao_x " configura a arquitetura " teste " da entidade " divisor ".

O interior de uma configuração de bloco pode conter uma outra configuração de bloco. Essa outra configuração de bloco pode ser um simples bloco identificado pelo seu rótulo, ou um rótulo que identifica um comando " GENERATE ". No interior de um bloco pode ser inserida, também, a configuração de um componente.

```
FOR nome_arquitetura
  -- Clausula use
  -- Configuracao de um outro bloco
  -- Configuracao de um componente.      Ver quadro seguinte
END FOR;
```

```
FOR nome_arquitetura
  FOR rotulo_de_um_bloco
    -- Configuracao de um outro bloco
    -- Configuracao de um componente.     Ver quadro seguinte
  END FOR;
END FOR;
```

```
FOR nome_arquitetura
  FOR rotulo_de_um_comando_GENERATE (especificacao_de_indice)
    -- Configuracao de um outro bloco
    -- Configuracao de um componente.     Ver quadro seguinte
  END FOR;
END FOR;
```

Quadro 14.4.5 Exemplos de configuração de bloco.

A configuração de um componente estabelece a configuração dos componentes contidos no bloco. Uma configuração de um componente inicia com a palavra reservada " FOR " seguida da especificação do componente (ver Quadro 14.4.6). No interior da configuração, podem ser inseridas a definição do elo e uma nova configuração de bloco. A especificação do componente e a definição do elo seguem o mesmo modelo da especificação da configuração (ver Quadro 14.4.1). A inserção de um novo bloco no interior da configuração de um componente é empregada quando um componente contém outros componentes no seu interior.

```
FOR rotulo_do_componente : nome_componente              -- especificacao_do_componente
  USE ENTITY biblioteca.nome_entidade(arquitetura);     -- definicao do elo
END FOR;
```

```
FOR u1 : Local_def                -- Inicia configuracao de "u1" associado ao componente "Local_def"
  USE ENTITY WORK.Externo_def;    -- define elo entre a solicitacao "u1" e a entidade "Externo_def"
END FOR;                          -- fim configuracao componente de "u1"
```

```
FOR rotulo_do_componente : nome_componente              -- especificacao_do_componente
   USE ENTITY biblioteca.nome_entidade(arquitetura);    -- definicao do elo
   -- configuracao_de_um_outro_bloco
END FOR;
```

Quadro 14.4.6 Exemplos de configuração de componente.

A Figura 14.4.5 e o Quadro 14.4.7 ilustram a configuração de uma entidade com dois níveis de hierarquia. Nesse exemplo, a arquitetura " topo " solicita um componente de rótulo " x1 ". Esse componente é declarado com o nome " local_Nb ". A configuração deve estabelecer um elo entre " x1 " e a arquitetura " media " da entidade " Nb ". A arquitetura " media ", por sua vez, solicita o componente de rótulo " y1 " de nome " local_Nc ". A configuração deve estabelecer um elo entre " y1 " e a arquitetura " baixa " de " Nc ".

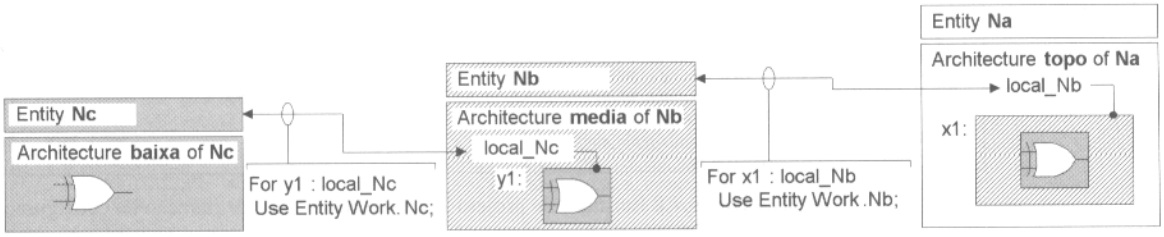


Figura 14.4.5 Configuração com níveis de hierarquia.

A configuração inicia com a declaração definindo o nome da unidade de projeto (" versao_x ") e o nome da entidade configurada (" Na "). A linha 2 inicia a configuração de um bloco que estabelece a configuração da arquitetura " topo ". Na linha 3 tem-se a configuração da solicitação " x1 " do componente " local_Nb ", e na linha seguinte é estabelecido o elo com a entidade " Nb ". A linha 5 inicia a configuração de um novo bloco, identificando a arquitetura " media " da entidade " Nb ". Na linha 6 é iniciada a configuração da solicitação " y1 " de " local_Nc ", e na linha seguinte é estabelecido o elo de " y1 " com a entidade " Nc ".

```
 1  CONFIGURATION versao_x OF Na IS -- Declaracao da configuracao
 2   FOR topo                       -- Configuracao de um bloco, define arquitetura configurada
 3    FOR x1: local_Nb              --{a} Configuracao de componente, rotulo : nome_componente
 4      USE ENTITY WORK.Nb;         --    define elo entre solicitacao "x1" e entidade "Nb"
 5      FOR media                   --{b} Inicia bloco p/ configurar sol. interna a "x1"
 6        FOR y1 : local_Nc         --{c} Inicia configuracao solicitacao interna a "x1"
 7          USE ENTITY WORK.Nc;     --    define elo da solicitacao "y1"
 8        END FOR;                  --    fim configuracao da sol. "y1"              {c}
 9      END FOR;                    --    fim configuracao bloco interno a  "x1"     {b}
10    END FOR;                      --    fim configuracao da solicitacao "x1"       {a}
11   END FOR;                       --    fim da configuracao bloco
12  END;                            --    fim declaracao de configuracao
```

Quadro 14.4.7 Configuração com níveis de hierarquia.

A seguir, é apresentada uma seqüência de exemplos para ilustrar três casos de declaração de configuração: descrição sem blocos, descrição contendo blocos e descrição contendo o comando " GENERATE ". Nos exemplos são empregadas as arquiteturas da entidade " soma " descritas no Quadro 14.4.2.

Os Quadros 14.4.8 e 14.4.9 ilustram a declaração de configuração aplicada a uma entidade com três componentes. A entidade " somadores2 " necessita de uma configuração para estabelecer o elo entre as solicitações de componentes e as unidades de projeto disponíveis. No exemplo, a configuração está contida na unidade de projeto " versao1 ".

A declaração da configuração inicia na linha 5, onde é definida a configuração da entidade de projeto " somadores2 ". Na linha 6 tem-se a configuração de um bloco identificando qual arquitetura de " somadores2 " está sendo configurada (no caso, arquitetura " teste "). Na linha 7 inicia-se a configuração de um componente que estabelece o elo entre todas as solicitações do componente " add " (no caso, " u1 " e " u2 "), e a entidade " soma ". Como a arquitetura foi omitida, o elo é estabelecido com a última arquitetura compilada. Na linha 9 é concluída a configuração dos componentes. De modo semelhante, o elo da solicitação " u3 " com a arquitetura " tipica " da entidade " soma " é estabelecido entre as linhas 10 e 13. Como o nome das portas da declaração de componente " adx " é diferente, é necessário definir a correspondência entre nomes através do " PORT MAP " (ver linha 12). Na linha 14 é concluída a configuração da arquitetura " teste " de " somadores2 ".

```
 1  -- arquivo conf_2.vhd
 2  -- Mensagem de aviso esperada: No default binding for component: "add" "adx".
 3  -- Necessita da declaracao de configuracao. Sugestoes em: conf_3.vhd
 4  ENTITY somadores2 IS
 5    PORT (x,  y  : IN  INTEGER RANGE 0 TO 7;
 6          s1, st, sr : OUT INTEGER RANGE 0 TO 15);
 7  END somadores2;
 8
 9  ARCHITECTURE teste OF somadores2 IS
10    COMPONENT add PORT (a, b : IN  INTEGER RANGE 0 TO 7; s : OUT INTEGER RANGE 0 TO 15);
11    END COMPONENT;
12
13    COMPONENT adx PORT (k, l : IN  INTEGER RANGE 0 TO 7; m : OUT INTEGER RANGE 0 TO 15);
14    END COMPONENT;
15  BEGIN
16    u1 : add PORT MAP(x, y, sr);
17    u2 : add PORT MAP(x, y, s1);
18    u3 : adx PORT MAP(x, y, st);
19  END teste;
```

Quadro 14.4.8 Entidade de projeto " somadores2 " sem declaração de configuração.

```
 1  -- arquivo conf_3.vhd
 2  -- Nao e' necessario compilar novamente a entidade somadores2 para o teste das configuracoes
 3
 4  -- Unidade de projeto versao_1. Simule: versao_1 para configuracao que segue
 5  CONFIGURATION versao_1 OF somadores2 IS
 6    FOR teste                          --{1} FOR nome_arquitetura
 7      FOR ALL : add                    --{2} FOR rotulo_componente : nome_componente
 8        USE ENTITY WORK.soma;          --       definicao do elo
 9      END FOR;                         --    fim configuracao componente   {2}
10      FOR u3 : adx                     --{3} FOR rotulo_componente : nome_componente
11        USE ENTITY WORK.soma(tipica)   --       definicao do elo
12        PORT MAP(a => k, b =>l, s => m);--      definicao do elo
13      END FOR;                         --    fim configuracao componente   {3}
14    END FOR;                           --    fim arquitetura teste         {1}
15  END versao_1;
```

Quadro 14.4.9 Opção de configuração para a entidade de projeto " somadores2 ".

Os Quadros 14.4.10 e 14.4.11 ilustram como executar a configuração no caso de uma descrição contendo blocos. A arquitetura " teste " de " somadores6 " é dividida em dois blocos (" bloco_1 " e " bloco_2 "). A unidade de projeto para configurar " somadores6 " é denominada " ver_2 " (ver linha 5 do Quadro 14.4.11).

A diferença com relação à estrutura da configuração apresentada anteriormente, que não continha blocos, localiza-se nas linhas 7 e 12. Note que nessas linhas são abertas configurações de bloco, para os blocos " bloco_1 " e " bloco_2 ".

```
 1 -- arquivo conf_6.vhd
 2 -- mensagem de aviso esperada: No default binding for component: "add".
 3 ENTITY somadores6 IS
 4   PORT (x,  y  : IN  INTEGER RANGE 0 TO 7;
 5         sl, st, sr : OUT INTEGER RANGE 0 TO 15);
 6 END somadores6;
 7
 8 ARCHITECTURE teste OF somadores6 IS
 9   COMPONENT add PORT (a, b : IN  INTEGER RANGE 0 TO 7; s : OUT INTEGER RANGE 0 TO 15);
10   END COMPONENT;
11
12 BEGIN
13   bloco_1: BLOCK
14   BEGIN
15     u1 : add PORT MAP(x, y, sl);
16     u2 : add PORT MAP(x, y, sr);
17   END BLOCK;
18
19   bloco_2: BLOCK
20   BEGIN
21     u1 : add PORT MAP(x, y, st);
22   END BLOCK;
23 END teste;
```

Quadro 14.4.10 Entidade de projeto " somadores6 " sem declaração de configuração.

```
 1 -- arquivo conf_7.vhd
 2 -- Nao e' necessario compilar novamente a entidade somadores6 para o teste das configuracoes
 3
 4 -- Unidade de projeto ver_2. Simule: ver_2 para configuracao que segue:
 5 CONFIGURATION ver_2 OF somadores6 IS
 6   FOR teste                          --{1} FOR nome_arquitetura
 7     FOR bloco_1                      --{2} FOR rotulo_de_um_bloco
 8       FOR OTHERS: add                --{3} FOR rotulo_componente : nome_componente
 9         USE ENTITY WORK.soma(lenta); --       definicao do elo
10       END FOR;                       --    fim configuracao componente   {3}
11     END FOR;                         --    fim bloco_1                   {2}
12     FOR bloco_2                      --{4} FOR rotulo_de_um_bloco
13       FOR u1: add                    --{5} FOR rotulo_componente : nome_componente
14         USE ENTITY WORK.soma(tipica); --       definicao do elo
15       END FOR;                       --    fim configuracao componente   {5}
16     END FOR;                         --    fim bloco_2                   {4}
17   END FOR;                           --    fim arquitetura teste         {1}
18 END ver_2;
```

Quadro 14.4.11 Configuração para a entidade de projeto " somadores6 ".

Os Quadros 14.4.12 e 14.4.13 ilustram a configuração no caso de uma descrição que contém comandos " GENERATE ". A interligação dos componentes na arquitetura " teste " de " somadores4 " é executada entre as linhas 15 e 17 por meio do comando " GENERATE ". A unidade de projeto para configurar " somadores4 " é designada como " ver_1 " (ver linha 5 do Quadro 14.4.13). Note que as configurações de bloco das linhas 7 e 12 estão identificadas pelo rótulo do comando " GENERATE ", e é possível definir elos diferentes para cada solicitação de componente da iteração " GENERATE ". Para a iteração " 1 ", o elo estabelecido é com " soma(rapida) ", e para as iterações " 2 " e " 3 " o elo é estabelecido com " soma(tipica) ".

```
1  -- arquivo conf_4.vhd
2  -- Mensagem de aviso esperada: No default binding for component: "add".
3  ENTITY somadores4 IS
4    PORT (x,  y  : IN  INTEGER RANGE 0 TO 7;
5          sl, st, sr : OUT INTEGER RANGE 0 TO 15);
6  END somadores4;
7
8  ARCHITECTURE teste OF somadores4 IS
9    COMPONENT add PORT (a, b : IN  INTEGER RANGE 0 TO 7; s : OUT INTEGER RANGE 0 TO 15);
10   END COMPONENT;
11
12   TYPE saida IS ARRAY(1 TO 3) OF INTEGER RANGE 0 TO 15;
13   SIGNAL sx : saida;
14 BEGIN
15   rotulo_gen: FOR i IN 1 TO 3 GENERATE
16    ux : add PORT MAP(x, y, sx(i));
17   END GENERATE;
18   sl <= sx(1); st <= sx(2); sr <= sx(3);
19 END teste;
```

Quadro 14.4.12 Entidade de projeto " somadores4 " sem declaração de configuração.

```
1  -- arquivo conf_5.vhd
2  -- Nao e' necessario compilar novamente a entidade somadores4 para o teste das configuracoes
3
4  -- Unidade de projeto ver_1. Simule: ver_1 para configuracao que segue:
5  CONFIGURATION ver_1 OF somadores4 IS
6    FOR teste                          --{1} FOR nome_arquitetura
7      FOR rotulo_gen(1)                --{2} FOR rotulo_do_comando_Generate(indice)
8        FOR ux : add                   --{3} FOR rotulo_componente : nome_componente
9          USE ENTITY WORK.soma(rapida); --       definicao do elo
10       END FOR;                       --    fim configuracao componente   {3}
11     END FOR;                         --    fim rotulo_gen(1)             {2}
12     FOR rotulo_gen(2 TO 3)           --{4} FOR rotulo_do_comando_Generate(indice)
13       FOR ux : add                   --{5} FOR rotulo_componente : nome_componente
14         USE ENTITY WORK.soma(tipica); --       definicao do elo
15       END FOR;                       --    fim configuracao componente   {5}
16     END FOR;                         --    fim rotulo_gen(2 TO 3)        {4}
17   END FOR;                           --    fim arquitetura teste         {1}
18 END ver_1;
```

Quadro 14.4.13 Configuração para a entidade de projeto " somadores4 ".

Síntese contendo configuração

A especificação de configuração e a declaração de configuração nem sempre são suportadas pelas ferramentas de síntese. Algumas ferramentas aceitam apenas um subconjunto das opções disponíveis para configuração. A seguir é apresentado o exemplo de uma descrição contendo uma configuração submetida a uma operação de síntese. Nesse caso, são empregados genéricos na configuração, tópico ainda não discutido na forma de exemplo.

O Quadro 14.4.14 contém a descrição de dois registradores onde o número de bits é definido pelo genérico " n ". O registrador " reg_lat " é composto de elementos sensíveis a nível, e o registrador " reg_fli " de elementos sensíveis a borda. No Quadro 14.4.15, as solicitações " u1 " e " u2 " da entidade " reg_uso " empregam os componentes " latch " e " fli flop ", respectivamente.

Para se estabelecer um elo entre " u1 " e " reg_lat ", e um elo entre " u2 " e " reg_fli ", é empregada uma declaração de configuração, conforme ilustra o Quadro 14.4.16. Como o nome das portas de entrada e saída é o mesmo, pode ser omitida a lista de associação de portas. A designação dos genéricos, entretanto, é diferente; assim, é necessária a declaração do mapa de genéricos (ver linhas 9 e 13).

A Figura 14.4.6 contém o resultado da síntese da descrição " reg_uso " configurada por " sintese ". Conforme se pode observar, os sinais de entrada " controle ", " dado " e " reset ", assim como os sinais de saída " saida_l " e " saida_f " estão interligados corretamente aos blocos " reg_lat_3 " e " reg_fli_3 ". O interior desses blocos é apresentado na Figura 14.4.7, onde se verifica a correspondência com as entidades " reg_lat " e " reg_fli " descritas no Quadro 14.4.14.

```
1  -- arquivo reg_des.vhd
2  ENTITY reg_lat IS
3    GENERIC (n : INTEGER);
4    PORT (en, rst : IN  BIT; d : IN BIT_VECTOR(n DOWNTO 0); q : OUT BIT_VECTOR(n DOWNTO 0));
5  END reg_lat;
6
7  ARCHITECTURE teste OF reg_lat IS
8  BEGIN
9    abc: PROCESS (en, rst, d) BEGIN
10     IF (rst = '1') THEN q <= (OTHERS => '0');   ELSIF (en = '1') THEN q <= d;   END IF;
11   END PROCESS abc;
12 END teste;
13
14 ENTITY reg_fli IS
15   GENERIC (n : INTEGER);
16   PORT (ck, rst : IN  BIT; d : IN BIT_VECTOR(n DOWNTO 0); q : OUT BIT_VECTOR(n DOWNTO 0));
17 END reg_fli;
18
19 ARCHITECTURE teste OF reg_fli IS
20 BEGIN
21   abc: PROCESS (ck) BEGIN
22     IF (ck'EVENT AND ck='1') THEN
23       IF (rst = '1') THEN q <= (OTHERS => '0');   ELSE  q <= d;   END IF;
24     END IF;
25   END PROCESS abc;
26 END teste;
```

Quadro 14.4.14 Descrição de dois registradores de " n " bits - " reg_lat " e " reg_fli ".

```
1  ENTITY reg_uso IS
2    GENERIC (k : INTEGER := 3 );
3    PORT (controle, reset: IN BIT; dado : IN BIT_VECTOR(k DOWNTO 0);
4          saida_l, saida_f : OUT BIT_VECTOR(k DOWNTO 0));
5  END reg_uso;
6
7  ARCHITECTURE topo OF reg_uso IS
8  COMPONENT latch
9    GENERIC (k : INTEGER := k);
10   PORT (en, rst: IN  BIT; d: IN BIT_VECTOR(k DOWNTO 0); q: OUT BIT_VECTOR(k DOWNTO 0));
11 END COMPONENT;
12
13 COMPONENT flipflop
14   GENERIC (k : INTEGER);  -- necessita mapa de generico
15   PORT (ck, rst: IN  BIT; d: IN BIT_VECTOR(k DOWNTO 0); q: OUT BIT_VECTOR(k DOWNTO 0));
16 END COMPONENT;
17
18 BEGIN
19   u1 : latch                    PORT MAP(controle, reset, dado, saida_l);
20   u2 : flipflop GENERIC MAP(k) PORT MAP(controle, reset, dado, saida_f);
21 END topo;
```

Quadro 14.4.15 Descrição empregando os componentes " latch " e " flip flop ".

```
1  -- arquivo reg_conf.vhd
2  -- Nao e' necessario compilar novamente a entidade "reg_uso" para o teste das configuracoes
3
4  -- Unidade de projeto "sintese". Simule: "sintese" para configuracao que segue
5  CONFIGURATION sintese OF reg_uso IS
6    FOR topo                          --{1} FOR nome_arquitetura
7      FOR u1 : latch                  --{2} FOR rotulo_componente  : nome_componente
8        USE ENTITY WORK.reg_lat       --       definicao do elo
9        GENERIC MAP(n => k);          --       definicao do elo (continuacao)
10     END FOR;                        --    fim configuracao componente   {2}
11     FOR u2 : flipflop               --{3} FOR rotulo_componente  : nome_componente
12       USE ENTITY WORK.reg_fli       --       definicao do elo
13       GENERIC MAP(n => k);          --       definicao do elo (continuacao)
14     END FOR;                        --    fim configuracao componente   {3}
15   END FOR;
16 END sintese;
```

Quadro 14.4.16 Configuração para a entidade de projeto " reg_uso ".

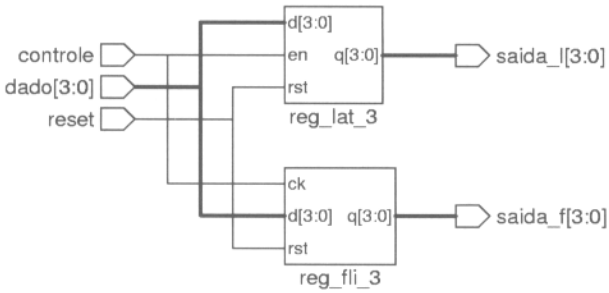


Figura 14.4.6 Resultado da síntese empregando configuração — nível RTL.

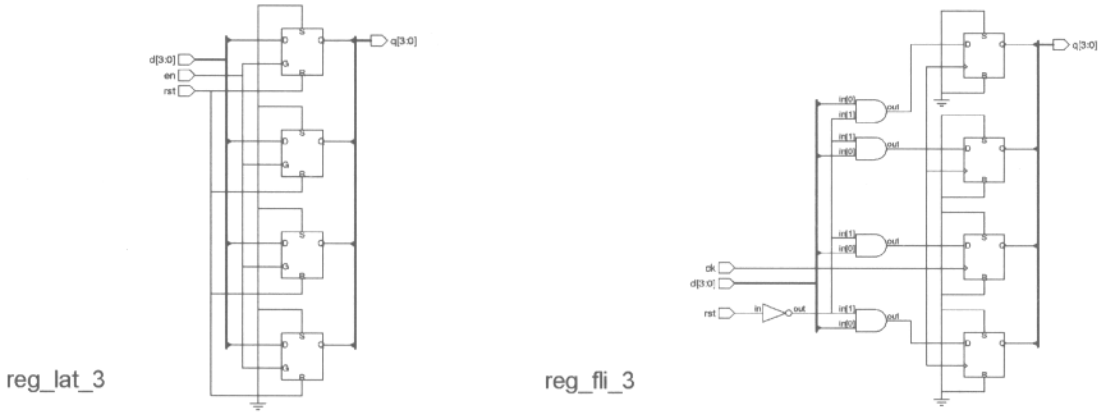


Figura 14.4.7 Visão do interior dos blocos.

14.5 Comandos concorrentes pospostos

Conforme visto no Capítulo 3, a avaliação de comandos concorrentes necessita de uma série de iterações até atingir um resultado final. Cada iteração é processada em um intervalo de tempo nulo denominado Δ. Um sinal pertencente à lista de sensibilidade de um processo, por exemplo, causa a reativação do processo a cada alteração de seu valor. Isso pode ser um inconveniente no caso de operações com arquivos e teste.

O objetivo de pospor a execução de um comando concorrente é condicionar a sua execução ao final das iterações pendentes, de modo que elas sejam avaliadas com os valores finais do instante atual da simulação. Os comandos concorrentes pospostos são os processos e comandos concorrentes que possuem um processo equivalente com o mesmo significado semântico. Os comandos pospostos não são disponíveis na versão VHDL-1987.

Exemplos de comandos pospostos são apresentados no Quadro 14.5.1. Um processo posposto, *postponed process*, contém a palavra reservada " POSTPONED " após o rótulo de identificação, e somente é executado após o término de todas as iterações do ciclo atual de simulação. Um procedimento concorrente, *concurrent procedure*, é um procedimento invocado em uma região de código concorrente. Para cada procedimento concorrente, existe um processo equivalente que pode ser posposto, se na sua chamada for incluída a palavra reservada " POSTPONED ". De modo semelhante, para um comando de asserção concorrente, *concurrent assertion*, existe um processo equivalente, e ele pode ser posposto, se incluída a palavra reservada " POSTPONED " anteriormente ao comando " ASSERT ". No caso de transferência de valores para sinal, existem, também, processos equivalentes. Assim, é possível pospor a transferência incondicional de um sinal, bem como os casos condicionais: construções " WHEN ELSE " e " WITH SELECT ".

Considere o código do Quadro 14.5.2. Ele contém a descrição de um *latch* RS acionado por um sinal de relógio, e um processo que notifica quando o valor do sinal " qx " difere do valor de " qy ". A Figura 14.5.1 ilustra a operação da descrição do *latch* tomando em conta as iterações que ocorrem. Conforme se pode observar, cada mudança de valor em " ck " leva a uma troca de valores em " qx " e " qy ". Ao final das iterações, o valor de " qx " é sempre o complemento de " qy ".

```
abc: POSTPONED PROCESS (sinal_a, sinal_b)              -- processo posposto
BEGIN
   ...
END PROCESS;
```

```
POSTPONED procedimento_a(sinal_a, sinal_b, sinal_c);  -- procedimento posposto
```

```
POSTPONED ASSERT condicao                              -- condicao = TRUE (nada ocorre);
   REPORT expressao_tipo_texto                         -- opcional
     SEVERITY expressao_de_severidade;                 -- opcional: FAILURE, ERROR, WARNING, NOTE
```

```
POSTPONED x <= y;                                      -- transferencia
```

```
POSTPONED x <= y  WHEN condicao_1 ELSE z;              -- transferencia construcao WHEN ELSE
```

```
POSTPONED WITH k SELECT                                -- transferencia construcao WITH SELECT
          x <= y WHEN condicao_1,
               z WHEN OTHERS;
```

Quadro 14.5.1 Exemplos de comandos pospostos.

Supondo que a finalidade do processo " iguais " é encontrar uma condição de falha, isto é, valor de " qx " igual ao valor de " qy ". Se o processo não fosse posposto, durante as iterações seriam apresentadas mensagens de erro indevidas, pois os valores finais de " qx " e " qy " ainda não foram atingidos. O processo posposto nessa descrição somente sai do seu estado suspenso ao final das iterações, com " qx " e " qy " estáveis. Assim, a mensagem de erro nunca é exibida.

```
 1  -- versao VHDL-1993
 2  ENTITY posp_0 IS
 3    PORT (ck      : IN  BIT;
 4          qx, qy : BUFFER BIT);
 5  END posp_0;
 6
 7  ARCHITECTURE teste OF posp_0 IS
 8    SIGNAL ck_n : BIT;
 9  BEGIN
10    qx <= ck   NOR qy;
11    qy <= ck_n NOR qx;
12    ck_n <= NOT ck;
13
14    iguais: POSTPONED PROCESS(qx, qy)  -- processo aguarda todas iteracoes pendentes
15    BEGIN
16     ASSERT qx /= qy  --(qx /=qy) condicao correta, caso falso mensagem e' apresentada
17        REPORT "Saida qx = qy " SEVERITY NOTE;
18    END PROCESS;
19  END teste;
```

Quadro 14.5.2 Descrição para teste de um processo posposto.

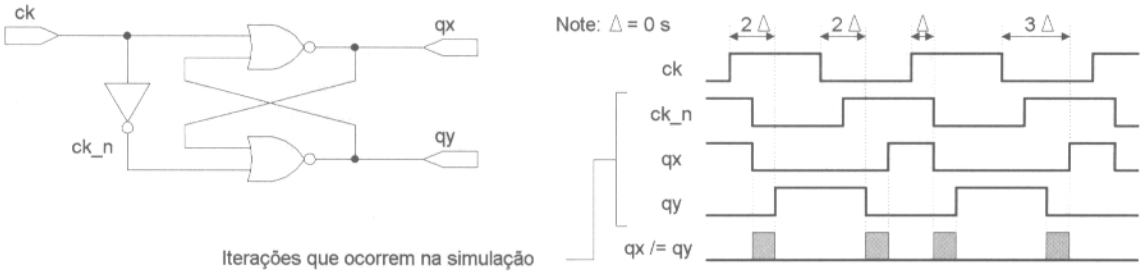


Figura 14.5.1 Ilustração para a descrição do Quadro 14.5.2.

14.6 Principais pontos abordados

- " ASSERT ": comando seqüencial para notificar condições ilegais. A sua inserção em regiões de código concorrente é, também, possível. Pode ser empregado em conjunto com as cláusulas " REPORT " e " SEVERITY ". A condição de teste corresponde ao requisito correto, onde a mensagem não é apresentada.
- " REPORT ": comando seqüencial usado para relatar mensagens. Pode ser empregado em conjunto com a cláusula " SEVERITY ". Não é disponível na versão VHDL-1987.
- A definição de um tipo com uma função de resolução deve conter os seguintes passos:
  - definir um tipo enumerado " en " incluindo todos os valores que o tipo com função de resolução pode assumir;
  - declarar um vetor com os limites em aberto contendo elementos do tipo " en ";
  - declarar a função de resolução " rs " que retorna um tipo " en ";
  - declarar um subtipo de " en " associado à função de resolução " rs ".
- Uma configuração permite definir o elo entre uma solicitação de um componente e uma entidade. A definição pode ser estabelecida por uma especificação de configuração ou declaração de configuração. Uma especificação de configuração faz parte de uma arquitetura; uma declaração de configuração emprega uma unidade de projeto específica para a configuração.
- Um comando concorrente posposto tem a sua execução condicionada ao término das iterações pendentes de um instante de simulação. Eles não são disponíveis na versão VHDL-1987.

14.7 Exercícios

14.7.1 Conforme visto no exemplo do Quadro 14.2.2, é necessária uma função de conversão para um tipo " STRING " em um comando " ASSERT ", quando se deseja apresentar uma mensagem cujos valores pertençam a um outro tipo.

Crie uma descrição que contenha uma porta de entrada do tipo " std_logic_vector " e empregue o comando " ASSERT ". A cada novo valor imposto à porta, ele deve ser convertido para o tipo " STRING " e apresentado na forma de mensagem pelo simulador.

14.7.2 Proponha um pacote com a definição de um tipo com uma função de resolução contendo os valores apresentados na Tabela 14.7.1. Cada valor corresponde a uma cor, onde " vermelho ", " verde " e " azul " estão associados às cores primárias. Nesse tipo, a atribuição de valor a um sinal por mais de um controlador é equivalente a uma mistura de cores.

A mistura de valores associados a duas cores primárias resulta no valor de outra cor; exemplo: o valor " verde " misturado com o valor " vermelho " deve resultar no valor " amarelo ". A mistura de valores contendo os valores relacionados às três cores primárias deve resultar no valor " branco "; exemplo: " amarelo " misturado com " azul ". O valor " preto " corresponde à ausência de cor.

Crie uma descrição empregando o pacote e verifique a operação do tipo criado. Ela deve conter um sinal acionado por mais de um controlador, e cada controlador deve impor um dos oito valores definidos no tipo, de modo a varrer todas as combinações possíveis de valores.

Tabela 14.7.1 Exercício 14.7.2

ausência de cor	cores primárias	cores compostas
preto	vermelho	amarelo = verde + vermelho
	verde	ciano = verde + azul
	azul	violeta = vermelho + azul
		branco = vermelho + verde + azul

14.7.3 O objetivo do exercício é criar uma declaração de configuração para a entidade " xor_3 ". A descrição dessa entidade emprega componentes com dois níveis de hierarquia (ver Figura 14.7.1).

No Quadro 14.7.1, " xor_3 " solicita o componente " m1 " declarado como " local_xor_2 ". A configuração deve criar um elo entre " m1 " e " xor2 ". Por sua vez, " xor_2 " solicita dois componentes " u1 " e " u2 " declarados como " local_xor_1 ". A configuração deve prever, também, um elo entre " u1 " e " u2 " e " xor_1 ".

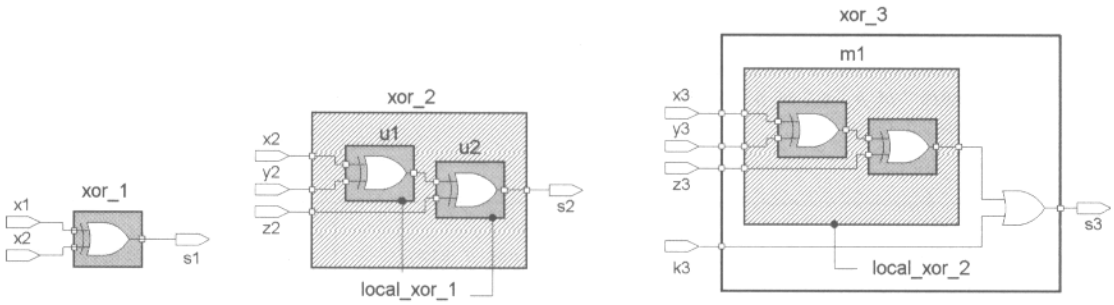


Figura 14.7.1 Níveis de hierarquia das descrições do Quadro 14.7.1.

```
1  -- arquivo conf_e1.vhd
2
3  -- Unidade nivel mais baixo
4  ENTITY xor_1 IS PORT (x1,  y1  : IN  BIT; s1  : OUT BIT);
5  END xor_1;
6  ARCHITECTURE baixa OF xor_1 IS
7  BEGIN
8    s1 <= x1 XOR y1;
9  END baixa;
10
11 -- Unidade nivel intermediario.
12 -- Referencia entidade "local_xor_1", necessita de configuracao para criar elo com "xor_1"
13 ENTITY xor_2 IS PORT (x2, y2, z2 : IN BIT; s2  : OUT BIT);
14 END xor_2;
15 ARCHITECTURE media OF xor_2 IS
16   COMPONENT local_xor_1 PORT(x1, y1 : IN BIT; s1 : OUT BIT); END COMPONENT;
17   SIGNAL k2 : BIT;
18 BEGIN
19   u1 : local_xor_1  PORT MAP(x2, y2, k2);  -- criar elo com "xor_1"
20   u2 : local_xor_1  PORT MAP(z2, k2, s2);  -- criar elo com "xor_1"
21 END media;
22
23 -- Unidade nivel mais alto.
24 -- Referencia entidade "local_xor_2", necessita configuracao para criar elo com "xor_2"
25 ENTITY xor_3 IS PORT (x3, y3, z3, k3 : IN BIT; s3 : OUT BIT);
26 END xor_3;
27
28 ARCHITECTURE alta OF xor_3 IS
29   COMPONENT local_xor_2 PORT (x2, y2, z2 : IN BIT; s2  : OUT BIT); END COMPONENT;
30   SIGNAL t3 : BIT;
31 BEGIN
32   m1 : local_xor_2 PORT MAP(x3, y3, z3, t3);  -- criar elo com "xor_2"
33   s3 <= t3 OR k3;
34 END alta;
```

Quadro 14.7.1 Descrição hierárquica que necessita de configuração.

14.7.4 Sintetize a configuração proposta para o Exercício 14.7.3. Verifique se o código é aceito pela ferramenta de síntese e interpretado corretamente.

14.7.5 Converta a declaração de configuração da entidade de projeto " reg_uso ", apresentada no Quadro 14.4.16, para uma especificação de configuração. Para tal, insira a especificação na arquitetura da descrição " reg_uso ".

14.7.6 Sintetize a configuração proposta para o Exercício 14.7.5. Verifique se o código é aceito pela ferramenta de síntese e interpretado corretamente.

14.7.7 O Quadro 14.7.2 contém dois códigos incompletos que descrevem um circuito de seleção, conforme ilustrado na Figura 14.7.2. Proponha uma descrição que permita verificar se o resultado nos sinais " ot_a " e " ot_b " é coincidente, notificando, com uma mensagem, caso o valor seja diferente. A mensagem não deve ser apresentada, se a discrepância de valores entre " ot_a " e " ot_b " ocorrer durante as iterações dos comandos concorrentes. Somente o valor final dos sinais deve ser considerado.

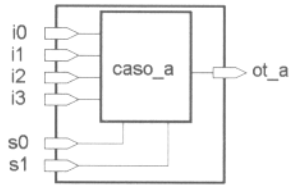
```
-- circuito de selecao caso_a
   SIGNAL int0, int1, int2, int3, ns0, ns1 : BIT;

   ot_a <= int0 OR int1 OR int2 OR int3;
   ns1  <= NOT s1;
   ns0  <= NOT s0;
   int0 <= i0 AND ns1 AND ns0;
   int1 <= i1 AND ns1 AND  s0;
   int2 <= i2 AND  s1 AND ns0;
   int3 <= i3 AND  s1 AND  s0;
```

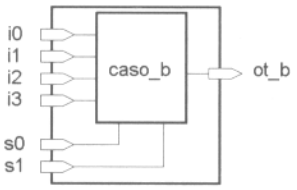
```
-- circuito de selecao caso_b
   SIGNAL sel : BIT_VECTOR (1 DOWNTO 0);

   sel <= s1 & s0;
   WITH sel SELECT
     ot_b <= i0 WHEN "00",
             i1 WHEN "01",
             i2 WHEN "10",
             i3 WHEN "11";
```

Quadro 14.7.2 Códigos incompletos: dois circuitos de seleção.



s1	s0	ot_a
0	0	ot_a = i0
0	1	ot_a = i1
1	0	ot_a = i2
1	1	ot_a = i3



s1	s0	ot_b
0	0	ot_b = i0
0	1	ot_b = i1
1	0	ot_b = i2
1	1	ot_b = i3

Figura 14.7.2 Ilustração para o Exercício 14.7.7.

CAPÍTULO

15 TESTE

Um teste é um conjunto de ações destinadas a verificar a operação de um sistema. O teste pode realizar uma comparação entre a especificação do sistema e o sistema projetado, bem como detectar problemas na implementação física do mesmo.

O objetivo deste capítulo é apresentar técnicas para o teste de erros de projeto em descrições VHDL. Após uma pequena discussão sobre os objetivos de um teste, são propostas opções para geração de formas de onda com a finalidade de estimular uma descrição. Em seguida são discutidos os modos de aplicação de estímulos e a verificação dos resultados.

15.1 Introdução

A execução de um teste consiste em exercitar o sistema sob teste com uma série de estímulos e observar o resultado. Inconsistências com a especificação são erros de projeto, e devem ser detectadas na fase de simulação. Problemas na implementação ocorrem devido a falhas na fabricação, e são detectados após a manufatura do sistema com o auxílio de equipamentos de teste específicos, ou através do próprio sistema. Neste último caso, o sistema incorpora recursos para autoteste.

Normalmente, os estímulos aplicados para verificação de erros de projeto, e os estímulos para detecção de falhas de fabricação são distintos. Na fase de arquitetura do sistema, o projetista pode lançar mão de recursos que não são disponíveis no teste do sistema implementado. Isolar uma unidade do sistema para aplicar estímulos e observar o resultado é uma tarefa trivial na fase de projeto. No sistema físico, a mesma operação pode ser muito complexa. Isso se deve ao fato de que, num sistema implementado, não são disponíveis pontos de acesso às unidades no interior do sistema. Além disso, o teste procura exercitar todos os nós do circuito para a detecção de problemas que podem ocorrer na fabricação, como curtos-circuitos, abertos, componentes defeituosos; essa preocupação não ocorre na fase de arquitetura.

Estímulos de teste para verificar a consistência de uma descrição podem ser criados com o auxílio da ferramenta de simulação empregada. Arquiteturas mais complexas, entretanto, necessitam de um número elevado de estímulos e, muitas vezes, são necessárias as tomadas de decisão ao longo do teste, bem como a verificação dos resultados para torná-lo mais eficiente.

Uma opção para geração dos estímulos de teste é a criação de uma entidade em VHDL que execute as operações de teste. Entidades de teste especialmente criadas para essa operação geram os estímulos através de dados armazenados, ou por meio de comandos da própria linguagem. O dados podem ser armazenados em constantes definidas na descrição, ou na forma de arquivos, tendo em vista que operações de leitura e escrita de dados em arquivos são disponíveis na linguagem.

15.2 Geração de estímulos

Nos procedimentos de teste, o ponto de partida é a geração dos estímulos que simulem as condições de entrada da descrição sob teste, em todas as condições que o projetista julgar conveniente. Os simuladores dispõem de facilidades que permitem impor uma condição a um sinal ou variável, bem como gerar estímulos periódicos. Considerando a potencialidade da linguagem, formas de onda das mais complexas podem ser sintetizadas de modo eficiente por uma descrição, sem o emprego de recursos da ferramenta de simulação. Essa abordagem é muito vantajosa, pois torna o procedimento de teste independente do aplicativo empregado.

Sinais aleatórios

A geração de valores aleatórios é a condição mais geral na síntese de estímulos. O Quadro 15.2.1 apresenta duas opções: a primeira utilizando a cláusula " AFTER " em uma região de código concorrente, e a segunda empregando o comando " WAIT " em uma região de código seqüencial.

As duas construções sintetizam a mesma forma de onda (ver Figura 15.2.1). No caso da atribuição de sinal empregando a cláusula " AFTER ", são indicados os valores e os correspondentes atrasos na transferência dos valores para o sinal. No caso da construção empregando " WAIT ", os intervalos de tempo são acumulados por cada comando, e a transferência do valor para o sinal ocorre na execução do comando " WAIT ".

Na primeira atividade do processo " abc ", o valor " 01 " é agendado para ser transferido para " s1 " na próxima iteração dos comandos concorrentes. Isso ocorre em " t=0 ns ", na execução da construção " WAIT FOR 20 ns ", e o processo é suspenso por 20 ns. No instante " t=20 ns ", o valor " 11 " é agendado para ser transferido para " s1 ", e na execução da construção " WAIT FOR 50 ns " o valor é transferido. De modo semelhante ocorrem as outras transferências de valores.

Note que a atribuição de sinal com a cláusula " AFTER " pode ser inserida em regiões de código seqüencial, conforme mostra o trecho de código contido no Quadro 15.2.2. Nesse caso, a forma de onda definida pelas cláusulas " AFTER " fica agendada para a próxima iteração dos comandos concorrentes por parte do simulador, que ocorre na execução do comando " WAIT ".

```
1  ENTITY forma_4 IS
2    PORT (s0, s1  : BUFFER BIT_VECTOR(1 DOWNTO 0));
3  END forma_4;
4
5  ARCHITECTURE teste OF forma_4 IS
6  BEGIN
7  -- valor atribuido em:
8  --      t=0ns  t=20 ns            t=50 ns            t=100ns
9    s0 <= "01", "11" AFTER 20 ns, "10" AFTER 70 ns, "00" AFTER 100 ns;
10
11   abc: PROCESS BEGIN             -- iteracao ocorre na execucao da decl. WAIT
12     s1 <= "01"; WAIT FOR 20 ns; -- s1 = 01 em 0ns,      e aguarda 20ns
13     s1 <= "11"; WAIT FOR 50 ns; -- s1 = 11 em 0+20ns,   e aguarda 50ns
14     s1 <= "10"; WAIT FOR 30 ns; -- s1 = 10 em 20+50ns,  e aguarda 30ns
15     s1 <= "00"; WAIT;           -- s1 = 00 em 70+30ns,  e para
16   END PROCESS;
17 END teste;
```

Quadro 15.2.1 Geração de sinais aleatórios.

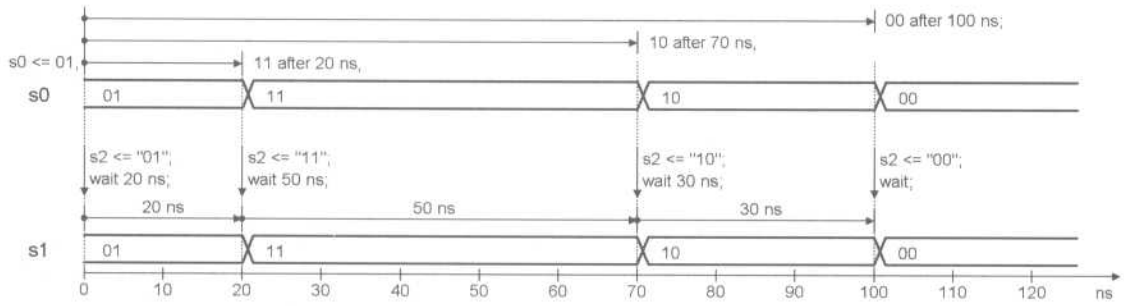


Figura 15.2.1 Carta de tempos resultante do código do Quadro 15.2.5.

```
7  xyz: PROCESS BEGIN
8    s0 <= "01", "11" AFTER 20 ns, "10" AFTER 70 ns, "00" AFTER 100 ns;
9    WAIT;
10 END PROCESS;
```

Quadro 15.2.2 Cláusula " AFTER " em região de código seqüencial.

Uma outra abordagem para a geração de sinais aleatórios é a utilização de vetores para armazenar os valores que a forma de onda pode assumir. A temporização é determinada através de cláusulas " AFTER " e comandos " WAIT ".

O Quadro 15.2.3 contém o exemplo de um código que permite sintetizar formas de onda a partir de um vetor contendo elementos do tipo registro. No exemplo, " sinais " é um tipo de registro composto de elementos do tipo " integer " e " bit_vector ", nomeados respectivamente como " nt " e " dt " (ver linhas 7 a 10). O tipo " conjunto_sinais " é um vetor sem limites definidos, composto de elementos do tipo " sinais " (ver linha 11). Finalmente, a constante " vet ", do tipo " conjunto_sinais ", armazena os valores desejados para a forma de onda.

Os sinais " s0 " e " s1 " geram a mesma forma de onda apresentada na Figura 15.2.1. O primeiro emprega a cláusula " AFTER " numa região de código concorrente; e o segundo, o comando " WAIT " em uma região de código seqüencial. Os sinais " n0 " e " n1 " seguem a mesma temporização de " s0 " e " s1 "; apenas os valores gerados são diferentes.

```
 1  ENTITY forma_5 IS
 2    PORT (s0, s1  : OUT BIT_VECTOR(1 DOWNTO 0);
 3          n0, n1  : OUT INTEGER);
 4  END forma_5;
 5
 6  ARCHITECTURE teste OF forma_5 IS
 7    TYPE sinais IS
 8      RECORD nt : INTEGER;
 9             dt : BIT_VECTOR(1 DOWNTO 0);
10      END RECORD;
11   TYPE conjunto_sinais IS ARRAY (natural RANGE <>) OF sinais;
12   CONSTANT vet : conjunto_sinais := ((1,"01"),(3,"11"),(2,"10"),(0,"00"));
13
14  BEGIN
15  -- valor atribuido em:
16  --          t=0ns  t=20 ns                t=50 ns                t=100ns
17    s0 <= vet(0).dt, vet(1).dt AFTER 20 ns, vet(2).dt AFTER 70 ns, vet(3).dt AFTER 100 ns;
18    n0 <= vet(0).nt, vet(1).nt AFTER 20 ns, vet(2).nt AFTER 70 ns, vet(3).nt AFTER 100 ns;
19
20    abc: PROCESS BEGIN                    -- iteracao ocorre na execucao da decl. WAIT
21      s1 <= vet(0).dt; n1 <= vet(0).nt;
22      WAIT FOR 20 ns;                     -- s1 =01, n1=1 em 0ns,     e aguarda 20ns
23      s1 <= vet(1).dt; n1 <= vet(1).nt;
24      WAIT FOR 50 ns;                     -- s1 =11, n1=3 em 0+20ns,  e aguarda 50ns
25      s1 <= vet(2).dt; n1 <= vet(2).nt;
26      WAIT FOR 30 ns;                     -- s1 =10, n1=2 em 20+50ns, e aguarda 30ns
27      s1 <= vet(3).dt; n1 <= vet(3).nt;
28      WAIT;                               -- s1 =00, n1=0 em 70+30ns, e para
29    END PROCESS;
30  END teste;
```

Quadro 15.2.3 Emprego de vetores para geração de sinais.

Sinais periódicos

Um problema muito freqüente em operações de teste é a geração de sinais periódicos no controle de circuitos síncronos, como registradores, contadores, máquinas de estado etc. O código do Quadro 15.2.4 contém algumas opções para geração de sinais periódicos em regiões de código concorrente. Os exemplos empregam a cláusula " AFTER " na atribuição de valores em sinais. A Figura 15.2.2 ilustra as formas de onda geradas.

Na forma de onda do sinal " s0 ", o intervalo de tempo em nível alto é igual ao intervalo de tempo em nível baixo, e o período é definido pelo genérico " t_ck ". A forma de onda dos sinais " s1 ", " s2 " e " s3 " não é simétrica. No caso dos sinais " s1 " e " s2 ", os intervalos de tempo são definidos pelos genéricos " t_alto " e " t_baixo ". A forma de onda do sinal " s3 " é gerada pelo deslocamento do sinal " dado " que contém o padrão de valores que " s3 " pode assumir. A cada intervalo de tempo definido pelo genérico " t_rol ", a informação armazenada em " dado " sofre uma rotação, e o valor do elemento de posição mais à esquerda do vetor é transferido para " s3 ". Essa técnica é muito útil para a geração de padrões complexos de forma periódica.

```
 1  -- VHDL 1993
 2  ENTITY forma_1 IS
 3    GENERIC(t_ck    : TIME := 30 ns;                          -- sinal s0
 4            t_alto  : TIME := 10 ns; t_baixo : TIME := 20 ns; -- sinais s1 e s2
 5            t_rol   : TIME := 10 ns);                         -- sinal s3
 6    PORT (s0, s1, s2, s3  : BUFFER BIT);
 7  END forma_1;
 8
 9  ARCHITECTURE teste OF forma_1 IS
10    SIGNAL dado : BIT_VECTOR(2 DOWNTO 0) := "001";
11  BEGIN                                          --
12    s0 <= NOT s0 AFTER t_ck/2;                   --   s0     ____|     |____|    |
13
14    s1 <= NOT s1 AFTER t_baixo WHEN s1 ='0' ELSE
15          NOT s1 AFTER t_alto;                   --
16                                                 -- s1 s2 s3 _____|   |_____|   |
17    s2 <= '0' AFTER t_alto  WHEN s2 ='1' ELSE
18          '1' AFTER t_baixo;
19
20    dado <= dado ROL 1 AFTER t_rol;
21    s3 <= dado(2);
22  END teste;
```

Quadro 15.2.4 Sinais periódicos — região de código concorrente.

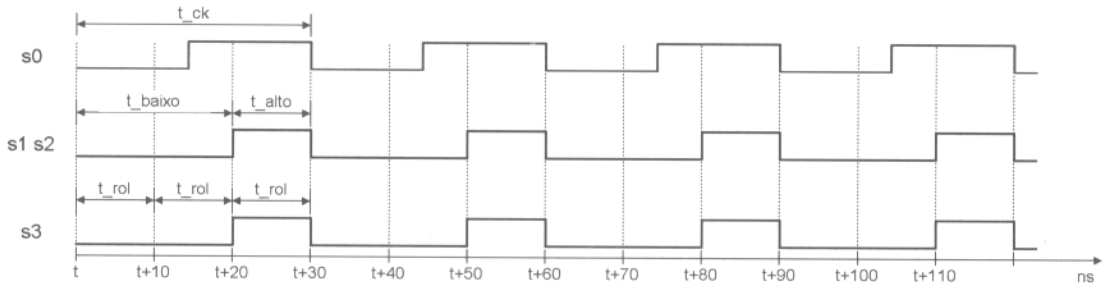


Figura 15.2.2 Formas de onda geradas pelo código do Quadro 15.2.1.

O Quadro 15.2.5 contém uma alternativa para geração de sinais " s1 " e " s2 " em uma região de código seqüencial. Como nos exemplos anteriores, o perfil da forma de onda é definido através de uma cláusula " AFTER "; o período de repetição do perfil, entretanto, é estabelecido por um comando " WAIT ".

A atribuição dos valores agendados segundo a cláusula " AFTER " toma efeito na execução do comando " WAIT ". No processo " ghi ", por exemplo, os valores " 0 " e " 1 " estão agendados para " ta " e " ta +t_baixo ", onde " ta " é o instante atual da simulação. Na execução do comando " WAIT " ocorre uma iteração dos comandos concorrentes, e o processo fica suspenso por um intervalo de tempo correspondente ao período da forma de onda, isto é, " t_alto +t_baixo ". Passado esse período, as operações se repetem gerando o mesmo perfil novamente. De modo semelhante, pode ser descrita a seqüência de eventos no sinal " s1 " dessa descrição.

```
 1  ENTITY forma_1a IS
 2    GENERIC(t_alto : TIME := 10 ns; t_baixo : TIME := 20 ns); -- sinais s1 e s2
 3    PORT (s1, s2   : BUFFER BIT := '0');
 4  END forma_1a;
 5
 6  ARCHITECTURE teste OF forma_1a IS
 7  BEGIN
 8    def: PROCESS BEGIN
 9      s1 <= NOT s1 AFTER t_baixo, s1 AFTER t_baixo + t_alto;
10      WAIT FOR t_alto + t_baixo;
11    END PROCESS;
12
13    ghi: PROCESS BEGIN
14      s2 <= '0', '1' AFTER t_baixo;
15      WAIT FOR t_alto + t_baixo;
16    END PROCESS;
17  END teste;
```

Quadro 15.2.5 Sinais periódicos — região de código seqüencial empregando cláusula " AFTER ".

O Quadro 15.2.6 contém novos exemplos para a geração dos sinais " s0 ", " s1 " e " s2 " descritos na Figura 15.2.2. O perfil das formas de onda e o período são definidos através dos comandos " WAIT " sem o emprego da cláusula " AFTER ".

Considerando, por exemplo, o sinal " s0 ". Em " t=0 ns " o valor de " s0 " é igual a " 0 ". Após " t_ck/2 " o processo sai do estado suspenso, e o valor de " s0 " complementado fica agendado para ser transferido na próxima iteração do simulador. Essa iteração ocorre em " t=0+t_ck/2 ns ", na execução do comando " WAIT ". De modo semelhante pode ser analisada a operação dos outros sinais.

```
1  -- VHDL 1993
2  ENTITY forma_2c IS
3    GENERIC(t_ck   : TIME := 30 ns;                          -- sinal s0
4            t_alto : TIME := 10 ns; t_baixo : TIME := 20 ns; -- sinais s1 e s2
5            t_rol  : TIME := 10 ns);                         -- sinal s3
6    PORT (s0, s1, s2, s3  : BUFFER BIT := '0');              -- Note: valor inicial =0
7  END forma_2c;
8
9  ARCHITECTURE teste OF forma_2c IS
10   SIGNAL dado : BIT_VECTOR(2 DOWNTO 0) := "001";
11 BEGIN
12   abc: PROCESS BEGIN
13     WAIT FOR t_ck/2; s0 <= NOT s0;
14   END PROCESS;
15
16   def: PROCESS BEGIN
17     WAIT FOR t_baixo; s1 <= NOT s1;
18     WAIT FOR t_alto; s1 <= NOT s1;
19   END PROCESS;
20
21   ghi: PROCESS BEGIN
22     s2 <= '0'; WAIT FOR t_baixo;
23     s2 <= '1'; WAIT FOR t_alto;
24   END PROCESS;
25
26   jkl: PROCESS BEGIN
27     dado <= dado ROL 1; s3 <= dado(2); WAIT FOR t_rol;
28   END PROCESS;
29 END teste;
```

Quadro 15.2.6 Sinais periódicos — região de código seqüencial.

Registradores dinâmicos e memórias necessitam, muitas vezes, ser controlados por sinais de relógio com duas fases, sem a ocorrência de superposição. Esses sinais devem respeitar sempre a condição " ck1 AND ck2 = 0 ", onde " ck1 " e " ck2 " são os sinais de controle ativos em nível alto (ver Figura 15.2.3). Isso significa que nunca os dois sinais podem estar ativos conjuntamente.

Uma descrição para sintetizar essas formas de onda é apresentada no Quadro 15.2.7. O sinal " ck " é uma base de tempo interna com período definido pelo genérico " t_ck ", e os sinais " ck1 " e " ck2 " são gerados a partir dessa base. Os genéricos " t_subida_ck1 " e " t_subida_ck2 " estabelecem o intervalo de tempo em que não ocorre a sobreposição. O primeiro define o intervalo de tempo entre a borda de subida do sinal " ck1 " e a borda de descida de " ck2 ". De modo semelhante, " t_subida_ck2 " estabelece a relação entre as bordas de subida e descida de " ck1 " e " ck2 ". Uma outra forma de gerar esses sinais é através da rotação de dois vetores contendo o padrão desejado, de forma semelhante à empregada no caso do sinal " s3 " da descrição contida no Quadro 15.2.4.

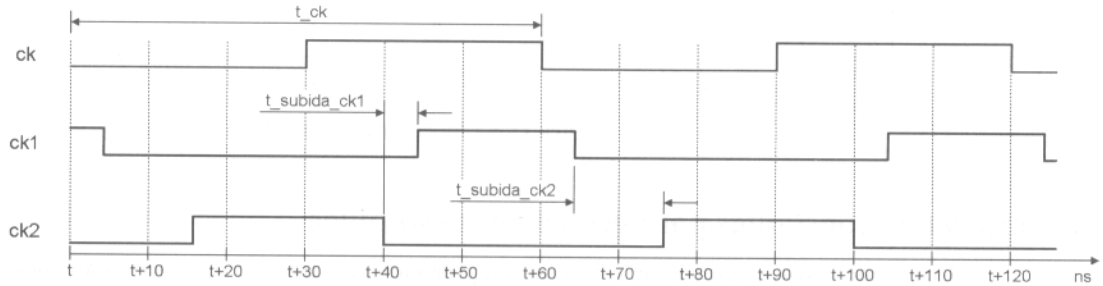


Figura 15.2.3 Sinais periódicos sem sobreposição.

```
1  ENTITY forma_3 IS
2    GENERIC(t_ck          : TIME := 60 ns; -- periodo da forma de onda
3            t_subida_ck1  : TIME :=  5 ns; -- interv. de tempo: subida de ck1 - descida de ck2
4            t_subida_ck2  : TIME := 10 ns);-- interv. de tempo: subida de ck2 - descida de ck1
5    PORT (ck, ck1, ck2     : BUFFER BIT);
6  END forma_3;
7
8  ARCHITECTURE teste OF forma_3 IS
9  BEGIN
10   ck <= NOT ck AFTER t_ck/2;  -- base de tempo
11
12   ck1 <= (ck2 NOR NOT ck) AFTER t_subida_ck1;
13   ck2 <= (ck1 NOR   ck)   AFTER t_subida_ck2;
14 END teste;
```

Quadro 15.2.7 Sinais periódicos sem sobreposição.

15.3 Teste de uma descrição

A execução do teste de uma descrição pode empregar diferentes métodos, e a escolha da abordagem mais adequada deve considerar aspectos, como: facilidade na elaboração, compatibilidade entre ferramentas de diferentes fornecedores, tempo de execução, necessidade de armazenar resultados etc.

O emprego dos recursos da linguagem é uma maneira eficiente de realizar parte ou toda operação de teste de uma entidade de projeto. A Figura 15.3.1 ilustra o caso geral de uma estrutura de teste. Os estímulos de teste podem ser gerados por uma constante do tipo vetor, por um conjunto de comandos próprios da linguagem, ou por um arquivo contendo os dados. Naturalmente, é possível, e certamente vantajosa, uma composição desses métodos para uma obtenção mais eficiente dos estímulos.

Os valores obtidos com a simulação da entidade de projeto podem ser observados através de uma janela própria da ferramenta de simulação. Entretanto, a verificação das formas de onda em descrições complexas exige a análise de muitas condições, o que dificulta a detecção de falhas de forma visual. Além disso, a documentação nesses casos é problemática e dependente da ferramenta empregada. Uma outra opção é realizar uma comparação entre os valores obtidos e os valores corretos segundo a especificação do projeto. Nesse caso, os valores corretos podem ser gerados de modo semelhante ao formato dos estímulos de teste: constantes, comandos, arquivos ou uma composição desses. A descrição que controla as operações se encarrega da comparação dos resultados, gerando mensagens de alerta e armazenando os resultados em arquivos. A vantagem dessa opção é a independência com a ferramenta empregada, permitindo que o procedimento seja aplicado por qualquer simulador VHDL de maneira automática. Além disso, a documentação fica disponível na forma de arquivos.

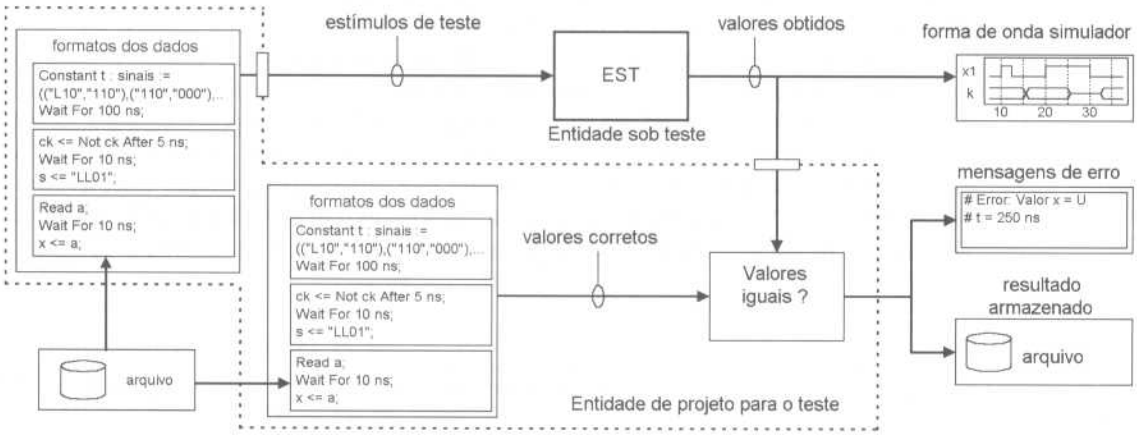


Figura 15.3.1 Procedimento de teste — caso geral.

Teste empregando constantes para geração dos estímulos

Conforme visto, a geração de estímulos empregando constantes é uma maneira simples de realizar um teste. Os valores são armazenados em uma ou mais constantes do tipo vetor, e um processo controlado por coman-

dos " WAIT " realiza a aplicação dos estímulos nos instantes desejados. As desvantagens desse método são a necessidade de recompilar a descrição a cada alteração de estímulos e uma maior demanda de memória. A Figura 15.3.2 ilustra o exemplo de um teste executado segundo essa sistemática, e o código correspondente é apresentado no Quadro 15.3.1.

A entidade sob teste é um registrador tipo D com inicialização assíncrona (ver Quadro 15.3.2). Doze estímulos de teste são propostos na constante " vetores ". Essa constante é um vetor composto de elementos do tipo " RECORD ", onde cada elemento contém os estímulos que devem ser aplicados e o resultado esperado. A aplicação dos estímulos é definida pelo laço entre as linhas 32 e 38 que percorre todos os elementos do vetor. A cadência da aplicação é definida pelo comando " WAIT " no interior do laço. Note que ele foi inserido para permitir a observação das formas de onda na simulação; poderia ser removido se a visualização não fosse necessária.

A operação de teste proposta na descrição verifica a ocorrência de erros. O comando " ASSERT " (linhas 43 e 44) verifica se os valores obtidos, " q ", e os valores corretos, " q_td ", são iguais. Na discordância entre os valores, é gerada uma mensagem de erro. A Figura 15.3.3 ilustra a seqüência de estímulos resultante da descrição. Note que foi introduzido um erro no décimo primeiro estímulo para observação de uma condição de falha. A mensagem gerada pelo comando " ASSERT " é apresentada no Quadro 15.3.3.

Deve ser observado um detalhe importante com relação ao sincronismo na comparação entre os valores corretos e os valores obtidos (ver novamente a Figura 15.3.2). Os estímulos, sinais " ck_t ", " rs_t " e " d_t ", são aplicados ao componente, por exemplo, no intervalo de tempo " t1 ". Entretanto, o valor obtido " q " somente estará disponível após uma iteração, ou seja, em " t1+Δ ". Assim, uma comparação entre " q " e " q_t ", valor correto e valor fornecido pelo vetor, geraria uma mensagem de erro indevida, em virtude da defasagem de " 1Δ " entre os sinais " q " e " q_td ". A fim de evitar esse problema, o valor correto, " q_t ", foi transferido para o sinal " q_td " para imprimir um atraso de um " 1Δ " e sincronizar a comparação entre os dois valores.

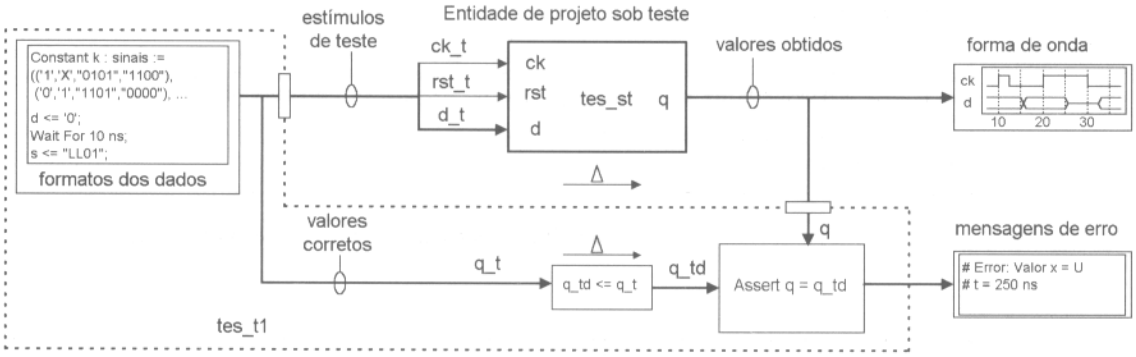


Figura 15.3.2 Proposta de teste empregando constantes para geração dos estímulos.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY tes_t1 IS
 5   PORT (ck_t, rst_t : BUFFER STD_LOGIC;
 6         d_t, q_t, q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
 7 END tes_t1;
 8
 9 ARCHITECTURE teste OF tes_t1 IS
10   TYPE sinais IS
11     RECORD ck, rst   : STD_LOGIC;
12            d, q      : STD_LOGIC_VECTOR(3 DOWNTO 0);
13     END RECORD;
14  TYPE conjunto_sinais IS ARRAY (natural RANGE <>) OF sinais;
15  CONSTANT vetores : conjunto_sinais :=
16 -- ck   rst   d     q      ck  rst   d      q      ck  rst    d     q
17  (('U','1',"UUUU","UUUU"),('U','0',"UUUU","0000"),('1','1',"UUUU","0000"),   -- reset
18   ('0','1',"ZZZZ","0000"),('0','1',"1010","0000"),('1','1',"1010","1010"),   -- armazena 1010
19   ('0','1',"ZZZZ","1010"),('0','1',"0101","1010"),('1','1',"0101","0101"),   -- armazena 0101
20   ('0','1',"ZZZZ","0101"),('0','1',"1100","0101"),('1','1',"1100","0011")); -- armazena 1100,
21                                                                  --     \
22  COMPONENT tes_st                                                --      \__ valor de teste errado
23    PORT (ck, rst : IN STD_LOGIC; d : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
24          q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
25  END COMPONENT;
26
27 SIGNAL q_td : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- empregado para atrasar de 1 delta o dado q_t
28
29 BEGIN
30   operacao: PROCESS
31     BEGIN
32       FOR i IN 0 TO  vetores'LENGTH-1 LOOP
33         ck_t   <= vetores(i).ck;
34         rst_t  <= vetores(i).rst;
35         d_t    <= vetores(i).d;
36         q_t    <= vetores(i).q;
37         WAIT FOR 10 ns;
38       END LOOP;
39     WAIT;
40   END PROCESS;
41
42   q_td <= q_t;      -- sincroniza valor "q_t" com "q", (q_t e' atrasado de 1 delta)
43   ASSERT q = q_td  -- (q=q_td) condicao correta, caso falso mensagem e' apresentada
44       REPORT "Valor da saida q difere do proposto" SEVERITY ERROR;
45
46   x1: tes_st PORT MAP(ck_t, rst_t, d_t, q);
47 END teste;
```

Quadro 15.3.1 Descrição para o teste de um registrador.

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 ENTITY tes_st IS
 5   PORT (ck, rst : IN STD_LOGIC;
 6         d       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
 7         q       : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
 8 END tes_st;
 9
10 ARCHITECTURE teste OF tes_st IS
11 BEGIN
12   abc: PROCESS (ck, rst, d)
13   BEGIN
14     IF    rst = '0'        THEN  q <= "0000";
15     ELSIF rising_edge(ck) THEN  q <= d;
16     END IF;
17   END PROCESS abc;
18 END teste;
```

Quadro 15.3.2 Entidade sob teste — registrador de quatro bits.

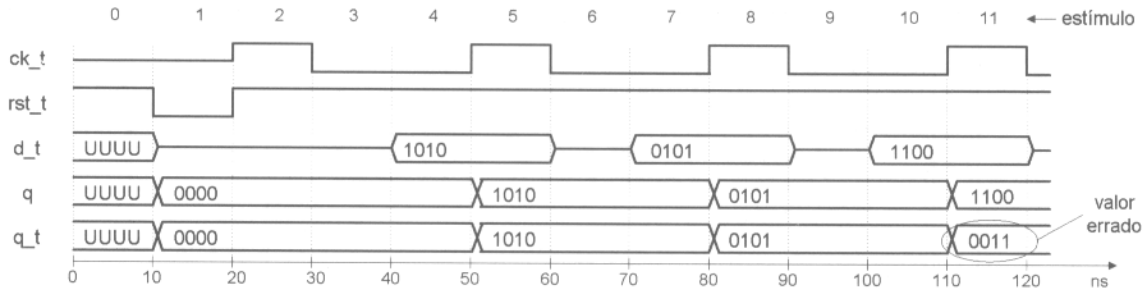


Figura 15.3.3 Seqüência de estímulos proposta para o teste — note: 11° estímulo com erro.

```
# ** Error: Valor da saida q difere do proposto
#    Time: 110 ns  Iteration: 2  Instance: /tes_t1
```

Quadro 15.3.3 Mensagem na janela do simulador — simulação da entidade " tes_t1 ".

A solução para sincronização dos sinais apresentada na descrição do Quadro 15.3.1 não pode ser aplicada em qualquer caso. Se a execução da simulação do componente necessitar de mais de uma iteração, ou se o número de iterações for dependente das condições de estímulo, serão geradas mensagens de erro a cada iteração, até que o valor correto dos sinais analisados atinja o valor final. A maneira mais adequada para resolver esse problema é pospor a execução da comparação para o final das iterações (ver Figura 15.3.4). O Quadro 15.3.4 apresenta um novo código que incorpora essa solução mais geral. Note que processos pospostos não são definidos na versão VHDL-1987.

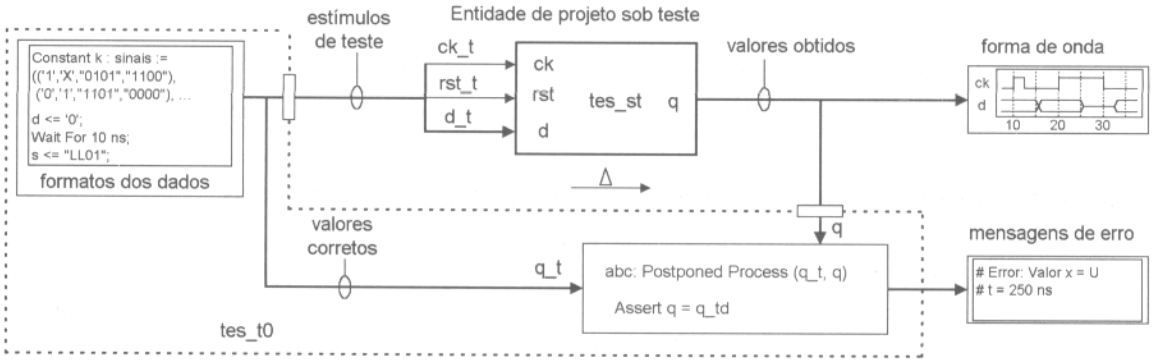


Figura 15.3.4 Proposta de teste alterada.

```
 1  -- versao VHDL-1993
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.ALL;
 4
 5  ENTITY tes_t0 IS
 6    PORT (ck_t, rst_t : BUFFER STD_LOGIC;
 7          d_t, q_t, q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
 8  END tes_t0;
 9
10
11  ARCHITECTURE teste OF tes_t0 IS
12    TYPE sinais IS
13      RECORD ck, rst   : STD_LOGIC;
14             d, q      : STD_LOGIC_VECTOR(3 DOWNTO 0);
15      END RECORD;
16   TYPE conjunto_sinais IS ARRAY (natural RANGE <>) OF sinais;
17   CONSTANT vetores : conjunto_sinais :=
18  -- ck   rst   d     q       ck  rst   d       q       ck  rst   d     q
19   (('U','1',"UUUU","UUUU"),('U','0',"UUUU","0000"),('1','1',"UUUU","0000"),   -- reset
20    ('0','1',"ZZZZ","0000"),('0','1',"1010","0000"),('1','1',"1010","1010"),   -- armazena 1010
21    ('0','1',"ZZZZ","1010"),('0','1',"0101","1010"),('1','1',"0101","0101"),   -- armazena 0101
22    ('0','1',"ZZZZ","0101"),('0','1',"1100","0101"),('1','1',"1100","0011")); -- armazena 1100,
23                                                                        --     \
24   COMPONENT tes_st                                                     --      \__ valor de teste errado
25     PORT (ck, rst : IN STD_LOGIC; d : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
26           q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
27   END COMPONENT;
28
29  BEGIN
30    operacao: PROCESS
31      BEGIN
32        FOR i IN 0 TO  vetores'LENGTH-1 LOOP
33          ck_t    <= vetores(i).ck;  rst_t  <= vetores(i).rst;
34          d_t     <= vetores(i).d;   q_t    <= vetores(i).q;
35        WAIT FOR 10 ns;
36        END LOOP;
37      WAIT;
38    END PROCESS;
39
40    iguais: POSTPONED PROCESS(q, q_t)  -- processo aguarda todas iteracoes pendentes
41    BEGIN
42     ASSERT q = q_t  -- (q=q_d) condicao correta, caso falso mensagem e' apresentada
43        REPORT "Valor da saida q difere do valor proposto" SEVERITY ERROR;
44    END PROCESS;
45
46    x1: tes_st PORT MAP(ck_t, rst_t, d_t, q);
47  END teste;
```

Quadro 15.3.4 Descrição para o teste de um registrador — processo posposto — VHDL-1993.

Teste empregando dados em arquivo para geração dos estímulos

O emprego de arquivos é uma outra maneira de realizar operações de teste. Esse modo é interessante, pois permite a alteração da seqüência de estímulos sem que seja necessário recompilar a entidade responsável pelo teste.

O formato dos arquivos contendo os dados pode ser binário ou texto. O formato binário tem como vantagem uma maior rapidez na simulação; o arquivo texto, por sua vez, tem como benefícios a compatibilidade com diferentes plataformas e a possibilidade de leitura direta da informação. O emprego de arquivos no formato binário é justificado em simulações complexas, onde o ganho na velocidade de simulação compense conversões de formato de arquivo que venham a ser necessárias. Deve ser lembrado, ainda, que as operações de leitura e escrita em arquivos no formato texto são definidas somente para os tipos do pacote padrão. Assim, operações com tipos definidos nos pacotes " STD_LOGIC_1164 " e " NUMERIC_STD " necessitam de funções de conversão, aumentando, ainda mais, o tempo de execução de um teste.

A Figura 15.3.5 ilustra uma proposta de teste empregando dados fornecidos por um arquivo no formato texto. O Quadro 15.3.5 contém a descrição completa da entidade de teste. Os dados, armazenados segundo uma formatação estabelecida, são recuperados periodicamente até o encontro do final de arquivo (ver linhas 44 a 51). Como os dados estão no formato texto, são empregados os procedimentos " READLINE " e " READ ". Observe que as portas de entrada e saída da entidade sob teste foram declaradas como tipos " STD_LOGIC " e " STD_LOGIC_VECTOR ". Como os procedimentos " READ " do pacote " TEXTIO " retornam valores apenas para os tipos definidos no pacote padrão, é necessária a inclusão de um subprograma para conversão de tipo. No caso, foi definida a função " para_std " (linhas 13 a 30) para executar essa operação.

Note, também, que a formatação dos dados no arquivo é importante. Para cada linha recuperada no arquivo, os dois primeiros dados devem conter dois caracteres, e os dois últimos dados devem conter cinco caracteres

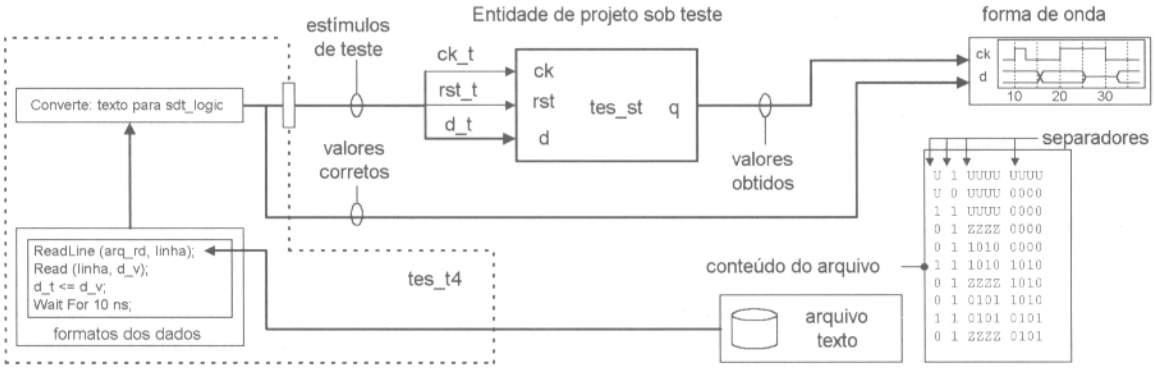


Figura 15.3.5 Proposta de teste empregando arquivos.

```
 1 -- versao VHDL 1993
 2 LIBRARY ieee;
 3 USE ieee.std_logic_1164.ALL;
 4 USE STD.TEXTIO.ALL;
 5
 6 ENTITY tes_t4 IS
 7   PORT (ck_t, rst_t : BUFFER STD_LOGIC;
 8         d_t, q_t, q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
 9 END tes_t4;
10
11 ARCHITECTURE teste OF tes_t4 IS
12
13 FUNCTION para_std(dado_stg : STRING) RETURN STD_LOGIC_VECTOR IS
14    VARIABLE dado_std : STD_LOGIC_VECTOR(dado_stg'LENGTH-1 DOWNTO 0);
15 BEGIN
16   FOR i IN 1 TO dado_stg'LENGTH LOOP  -- valido para mesma ordem
17     CASE dado_stg(i) IS
18       WHEN '-'     => dado_std(i-1) := '-';  --  dado_stg:
19       WHEN 'Z'     => dado_std(i-1) := 'Z';  --  n      ...     1
20       WHEN 'W'     => dado_std(i-1) := 'W';  -- | |H|L|...|U|1|X|  string
21       WHEN 'L'     => dado_std(i-1) := 'L';  --  \
22       WHEN 'H'     => dado_std(i-1) := 'H';  --   \_ espaco
23       WHEN '0'     => dado_std(i-1) := '0';  --
24       WHEN '1'     => dado_std(i-1) := '1';  --  retorna: dado_std
25       WHEN 'U'     => dado_std(i-1) := 'U';  -- n-1      ...     0
26       WHEN OTHERS => dado_std(i-1) := 'X';  -- |X|H|L|...|U|1|X|  std_logic_vector
27      END CASE;                                --   \
28    END LOOP;                                  --    \_ espaco convertido para X
29    RETURN dado_std;
30 END;
31
32  COMPONENT tes_st
33    PORT (ck, rst : IN STD_LOGIC; d : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
34          q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
35  END COMPONENT;
36
37 BEGIN
38   le: PROCESS
39     FILE     arquivo_rd  : TEXT OPEN READ_MODE IS "estim_t3.dat"; -- arquivo formato:
40     VARIABLE linha       : LINE;                          -- | | | | | | | | | | | | | | |
41     VARIABLE ck_v, rst_v : STRING(2 DOWNTO 1);            --    U   1   U U U U   U U U U
42     VARIABLE d_v,  q_v   : STRING(5 DOWNTO 1);            --    U   0   U U U U   0 0 0 0
43   BEGIN                                                   --    1   1   U U U U   0 0 0 0
44     WHILE NOT ENDFILE(arquivo_rd) LOOP
45      READLINE (arquivo_rd, linha);                        -- leitura de uma linha
46      READ (linha, ck_v);  ck_t  <= para_std(ck_v)(0);  -- leitura dos dados, remove espaco
47      READ (linha, rst_v); rst_t <= para_std(rst_v)(0); --                    "
48      READ (linha, d_v);   d_t   <= para_std(d_v)(3 DOWNTO 0); --             "
49      READ (linha, q_v);   q_t   <= para_std(q_v)(3 DOWNTO 0); --             "
50      WAIT FOR 10 ns;                                      -- intervalo de tempo entre estimulos
51     END LOOP;
52     FILE_CLOSE(arquivo_rd);
53     WAIT;
54   END PROCESS le;
55
56   x1: tes_st PORT MAP(ck_t, rst_t, d_t, q);
57 END teste;
```

Quadro 15.3.5 Descrição para o teste de um registrador — dados em arquivo texto — VHDL-1993.

(ver o segundo comando das linhas 46 a 49). O primeiro caracter de cada dado é considerado um separador e ignorado na transferência para os sinais tipo " STD_LOGIC " e " STD_LOGIC_VECTOR ". Subprogramas mais elegantes, que incluem a remoção de linhas e caracteres em branco, podem ser desenvolvidos com objetivo de tornar a formatação do arquivo de dados mais versátil. Deve ser lembrado, entretanto, que essas operações extras comprometem o desempenho de testes mais complexos, devido ao maior tempo de execução.

Teste empregando técnica mista

Uma outra opção para a execução do teste do registrador proposto anteriormente é apresentada no Quadro 15.3.6. Nesse código, as informações para o sinal de relógio " ck " e sinal de inicialização " rst " não estão armazenadas em tabelas ou arquivos. Os estímulos desses dois sinais são gerados através de comandos no próprio código.

Os dados de entrada do registrador em teste e os valores corretos esperados são armazenados numa constante, de modo semelhante ao empregado no código do Quadro 15.3.1. Para os sinais " ck " e " rst ", os valores dos dois primeiros estímulos são definidos diretamente no código (ver linhas 32 e 36). Após o segundo estímulo, o valor de " rst " é mantido constante, e os valores restantes de " ck " são gerados através da informação armazenada na

```
 1 -- versao VHDL-1993
 2 LIBRARY ieee;
 3 USE ieee.std_logic_1164.ALL;
 4
 5 ENTITY tes_t6 IS
 6   PORT (ck_t, rst_t : BUFFER STD_LOGIC;
 7         d_t, q_t, q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
 8 END tes_t6;
 9
10 ARCHITECTURE teste OF tes_t6 IS
11  TYPE sinais IS  RECORD  d, q : STD_LOGIC_VECTOR(3 DOWNTO 0);
12  END RECORD;
13  TYPE conjunto_sinais IS ARRAY (natural RANGE <>) OF sinais;
14  CONSTANT vetores : conjunto_sinais :=
15 --    d      q          d       q         d      q
16  (("UUUU","UUUU"),("UUUU","0000"),("UUUU","0000"),   -- reset
17   ("ZZZZ","0000"),("1010","0000"),("1010","1010"),   -- armazena 1010
18   ("ZZZZ","1010"),("0101","1010"),("0101","0101"),   -- armazena 0101
19   ("ZZZZ","0101"),("1100","0101"),("1100","0011"));  -- armazena 1100,
20                                                      --     \
21  COMPONENT tes_st                                    --      \__ valor de teste errado
22    PORT (ck, rst : IN STD_LOGIC; d : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
23          q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
24  END COMPONENT;
25
26 BEGIN
27   operacao: PROCESS
28     VARIABLE ck_v     : std_logic;
29     VARIABLE sequencia_ck : BIT_VECTOR(2 DOWNTO 0) := "100"; -- sequencia de valores p/ ck
30   BEGIN
31     -- aplica primeiro conjunto de estimulos
32     ck_t <= 'U';  rst_t <= '1';  d_t <= vetores(0).d;  q_t <= vetores(0).q;
33     WAIT FOR 10 ns;
34
35     -- aplica segundo conjunto de estimulo
36     ck_t <='U';   rst_t <= '0';  d_t <= vetores(1).d;  q_t   <= vetores(1).q;
37     WAIT FOR 10 ns;
38
39     -- aplica estimulos restantes, sinal de relogio gerado na linha 44
40     rst_t <= '1' ;
41     FOR i IN 2 TO  vetores'LENGTH-1 LOOP
42       ck_t  <= To_StdULogic(sequencia_ck(2));  d_t <= vetores(i).d; q_t <= vetores(i).q;
43       WAIT FOR 10 ns;
44       sequencia_ck := sequencia_ck rol 1; -- circula p/ esquerda conjunto de valores
45     END LOOP;
46     WAIT;
47   END PROCESS;
48
49   iguais: POSTPONED PROCESS(q, q_t)  -- processo aguarda todas iteracoes pendentes
50   BEGIN
51    ASSERT q = q_t  -- (q=q_d) condicao correta, caso falso mensagem e' apresentada
52       REPORT "Valor da saida q difere do valor proposto" SEVERITY ERROR;
53   END PROCESS;
54
55   x1: tes_st PORT MAP(ck_t, rst_t, d_t, q);
56 END teste;
```

Quadro 15.3.6 Descrição para o teste de um registrador — técnica mista.

variável do tipo vetor " sequencia_ck " (ver linha 29). A cada iteração do laço com início na linha 41, o elemento de posição mais à esquerda do vetor é transferido para " ck ", e o valor contido em " sequencia_ck " sofre uma rotação. A cada iteração, ainda, novos valores são atribuídos aos sinais " d_t " e " q_t ".

15.4 Principais pontos abordados

- Teste: conjunto de ações destinadas a verificar a operação de um sistema.
- Objetivos de um teste:
  - verificar a consistência entre a especificação do sistema e o sistema projetado;
  - detectar problemas na implementação física do sistema.
- Execução de um teste: aplicar ao sistema sob teste uma série de estímulos e observar o resultado.
- A criação dos estímulos de teste para verificar a consistência de uma descrição pode ser realizada através:
  - da própria ferramenta de simulação empregada (descrições simples);
  - de um código especialmente criado para gerar as operações de teste (descrições complexas).
- Os códigos criados para geração de estímulos podem mesclar diferentes técnicas, como:
  - varredura de um conjunto de constantes definidas no código;
  - leitura de dados armazenados em arquivos;
  - comandos da linguagem.

15.5 Exercícios

15.5.1 Altere o código do Quadro 15.2.3 de modo a gerar uma forma de onda periódica, repetindo a seqüência de valores proposta na Figura 15.2.1 a cada 140 ns.

15.5.2 Apresente a descrição de uma entidade para o teste de um registrador de 4 bits com a operação de *reset* síncrona, conforme proposto na Figura 15.5.1. Os estímulos de teste devem estar contidos na descrição na forma de uma constante, e deve ser gerada uma mensagem de erro, se o valor obtido na simulação for diferente do valor correto.

A seqüência de estímulos é apresentada na Figura 15.5.2. Ela verifica a operação de *reset* e armazenamento de dados, introduzindo um erro em t=110 ns com objetivo de verificar se a descrição detecta falhas.

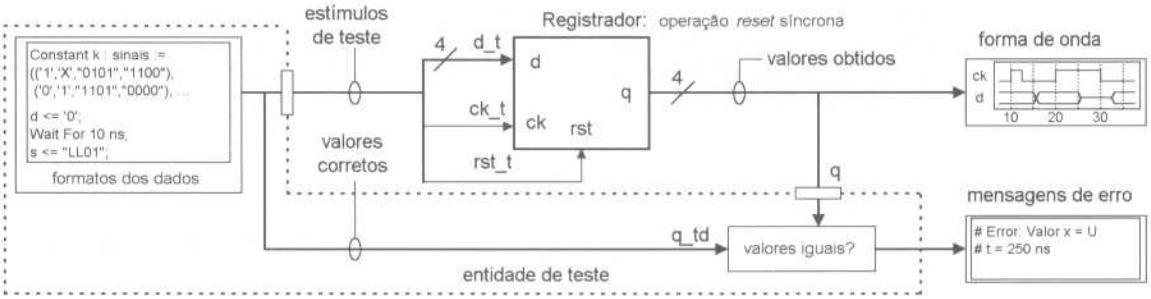


Figura 15.5.1 Exercício 15.5.2.

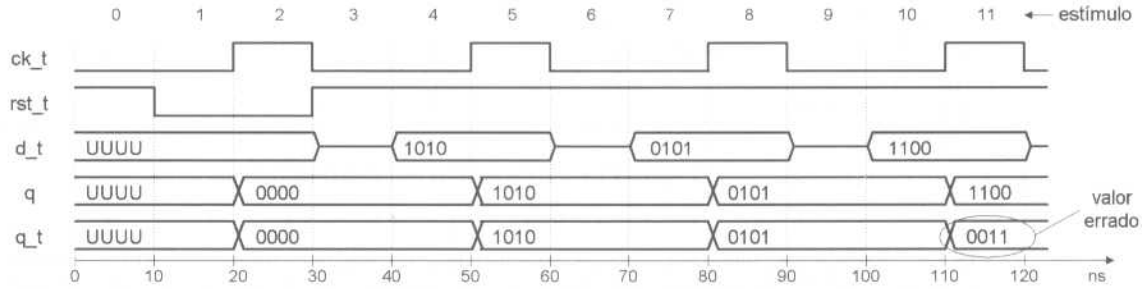


Figura 15.5.2 Seqüência de estímulos de teste.

15.5.3 Altere a descrição proposta para o Exercício 15.5.2 de modo a apresentar uma mensagem quando detectado um erro. A mensagem deve conter o valor esperado e o valor obtido na simulação. Será necessário um subprograma para converter os valores no formato " std_logic_vector " para o formato " string ".

15.5.4 Altere a descrição proposta para o Exercício 15.5.2 removendo o sinal " ck_t " da constante que contém os estímulos. Esse sinal deve ser gerado na descrição através de comandos, sintetizando a forma de onda descrita na Figura 15.5.2. Note que a forma de onda não é simétrica: 20 ns em nível lógico baixo, e 10 ns em nível lógico alto.

15.5.5 Proponha uma descrição que gere um arquivo binário contendo pares de valores, de modo a permitir o teste exaustivo de uma entidade com duas portas de entrada. Por teste exaustivo entende-se a aplicação de todas as combinações possíveis de valores. Os dados devem ser do tipo inteiro, e o arquivo deve conter as combinações de pares de valores. Os valores iniciais e finais dos estímulos armazenados devem ser definidos por genéricos: valores iniciais e finais da entrada " a " são " ini_a " e " max_a ". Os valores iniciais e finais da entrada " b " são " ini_b " e " max_b ". (Ver a parte inferior da Figura 15.5.3.)

15.5.6 O objetivo deste exercício é implementar o teste de um somador com o arquivo binário criado no Exercício 15.5.5. A Figura 15.5.3 ilustra o formato do teste. Note que os dados no arquivo não podem ser visualizados na forma apresentada na figura, tendo em vista que eles não estão armazenados no formato texto.

A entidade de teste deve executar a leitura no arquivo que contém os estímulos de teste. Os dados devem ser aplicados ao somador em intervalos de tempo regulares, de modo a permitir a visualização no simulador. Crie, também, a descrição de um somador para a realização do teste.

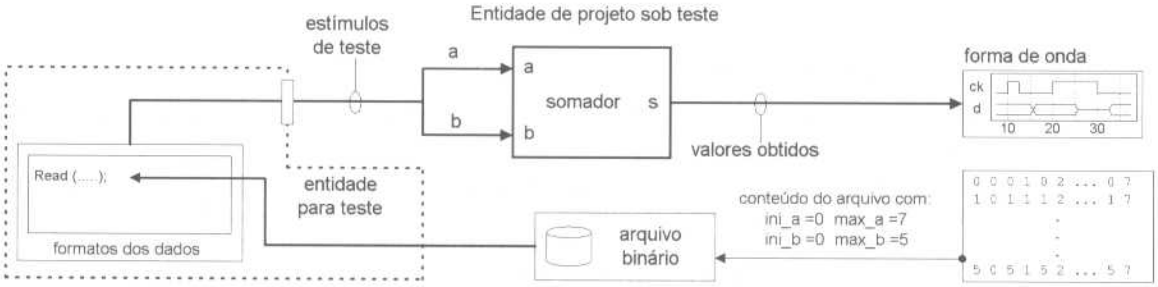


Figura 15.5.3 Exercícios 15.5.5 e 15.5.6.

15.5.7 Repita os Exercícios 15.5.5 e 15.5.6 alterando o formato do arquivo de binário para texto.

15.5.8 Proponha um código para teste do Exercício 7.6.9 do Capítulo 7, conforme proposto na Figura 15.5.4. Para o teste do " bloco 1 ", a entidade de teste deve armazenar quatro valores distintos nos registradores " reg 0 " a " reg 3 ". No teste do " bloco 2 ", ela deve gerar, também, o sinal " ck " para uma varredura contínua dos registradores, de modo a verificar a operação dos decodificadores.

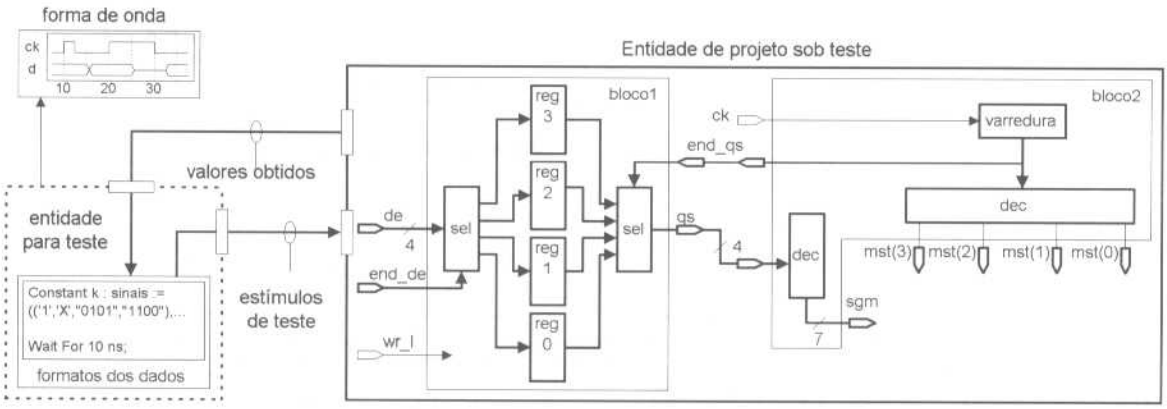


Figura 15.5.4 Teste de uma entidade desenvolvida no Capítulo 7.

15.5.9 Apresente um código para teste de parte do conversor analógico digital proposto no Exercício 7.6.10 do Capítulo 7 (ver Figura 15.5.5). No exercício, cada dígito do conversor necessita de um contador, um registrador e um decodificador. Efetue o teste de uma entidade contendo três blocos, de modo a verificar a operação do conjunto. Gere um sinal de reló-

gio para incrementar os contadores e, após um determinado período de tempo, transfira o resultado da contagem para os registradores.

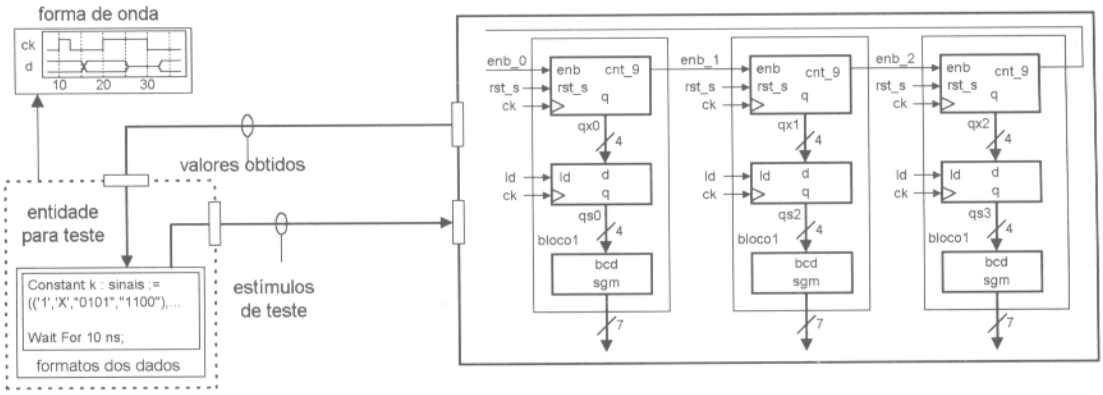


Figura 15.5.5 Teste de parte do conversor analógico digital desenvolvido no Capítulo 7.

ÍNDICE

Delimitadores

= operador relacional, 17-21
/= operador relacional, 17-21, 170-172
< operador relacional, 17-21, 170-172
<= operador relacional, 17-21, 170-172
<= atribuição de sinal, 17-21, 30-31, 170-172
<> box, 190-196
:= atribuição de variável, 12-14
> operador relacional, 17-21, 170-172
>= operador relacional, 17-21, 170-172
+ operador adição, 17-21, 170-172
+ operador sinal, 17-21
- operador adição, 17-21, 170-172
- operador sinal, 17-21, 170-172
& operador adição, 17-21
* operador multiplicação, 17-21, 170-172
/ operador multiplicação, 17-21, 170-172
** operadores diversos, 17-21
| símbolo, 32-34, 48-49

A

Abs: operador aritmético, 17-21, 170-172
Access: tipo, 188
Active: atributo sinal, 65-73
Actual parameter. *Veja* Parâmetro real
Adição: operador, 17-21
After: cláusula, 62-63, 242
Aggregate. *Veja* Agregado
Agregado, 14-17
Alias. *Veja* Pseudônimo
Alta impedância: inferência, 159-161
And: operador lógico, 17-21, 153, 170-175
Aninhamento, 43-44
Architecture
    declaração, 12
    palavra reservada, 2, 5-8
Arquivo
    formato binário, 212-218
    formato texto, 212-215
    operações de teste, 246-253
Array. *Veja* Vetor
    unconstrained. *Veja* Vetor limites em aberto
ASIC: síntese, 3-7
Assert: comando, 221, 236
Atraso
    com inércia, 62-63
    com transporte, 62-63
    modelagem, 62-63
    Δ, 28-29, 44-46, 63-64
Atribuição de valor
    sinal, 30-31, 44-46, 63-64
    variável, 63-64
Atributo
    Active: sinal, 65-73
    definido pelo usuário, 71-73
    Delayed: sinal, 65-73
    Event: sinal, 65-73
    High: tipo, 65-73
    High: vetor, 65-73
    Last_Event: sinal, 65-73
    Last_Value: sinal, 65-73
    Left: tipo, 65-73
    Left: vetor, 65-73
    LeftOf: tipo, 65-73
    Length: vetor, 65-73
    Low: tipo, 65-73
    Low: vetor, 65-73
    Pos: tipo, 65-73
    Pred: tipo, 65-73
    Quiet: sinal, 65-73
    Range: vetor, 65-73
    Referência, 65-73
    Reverse_Range: vetor, 65-73
    Right: tipo, 65-73
    Right: vetor, 65-73
    RightOF: tipo, 65-73
    Stable: sinal, 65-73
    Succ: tipo, 65-73
    Transaction: sinal, 65-73
    Val: tipo, 65-73

B

Bibliotecas
    Definida pelo usuário, 135-137
    IEEE, 151-153, 156
    Inclusão em um projeto, 135-137
    Introdução, 135-137
    STANDARD, 135-137
    TEXTIO, 135-137, 212-215
    WORK, 135-137
Bit: tipo, 14-17
Bit_vector: tipo, 14-17
Block. *Veja* Bloco
Bloco
    comando, 34-36
    configuração, 227-236
Boolean: tipo, 14-17
Box: delimitador <>, 190-196
Buffer: modo de uma porta, 11-12
Bus: palavra reservada, 34-36, 161

C

Case: construção Case When, 48-49
Character: tipo, 14-17
Classe: objetos, 12-14
Coletor aberto, 151-153
Comparação entre construções
    "When Else" "If Else", 51-52
    "With Select" "Case When", 51-52
Compilação: ordem de análise, 141-145
Composite. *Veja* Composto
Composto: tipo, 14-17
Configuração
    Declaração, 229-234
    Especificação, 227-229
Configuration. *Veja* Configuração
Constant. *Veja* Constante
Constante
    Deferida, 145
    Objeto, 12-14
Contadores: inferência, 91-93
Controlador, 151-153, 224-227

D

Declaração
    Componente, 100
    Constante, 12-14
    Entidade, 11-12
    Genéricos, 103-105
    objeto tipo Arquivo (VHDL-1987), 207-209
    objeto tipo Arquivo (VHDL-1993), 209-212
    Pacotes, 137-141
    Sinal, 12-14
    Sinal global, 146
    subtipo, 189
    tipo Arquivo, 206
    tipo Enumerado, 188
    tipo Físico,189
    tipo Registro, 196
    tipo Vetor, 190-196
    Variável, 12-14
    Variável compartilhada, 128-131
Deferred Constant. *Veja* Constante deferida
Definição de tipo
    máquina de estados, 86-90
    novos tipos, 188-197
Delayed: atributo sinal, 65-73
Desconectado: comando Block, 34-36
Design
    Entity, 2, 5-8, 10-12
    Library, 135
    Unit, 135-150
Deslocamento: operador, 17-21
Diversos: classe de operadores, 17-21
Downto: limites, 17-21, 106, 108-114, 127
Dreno aberto, 151-153
Driver. *Veja* Controlador

E

EDIF: síntese, 3-7
Else
    construção IF Else, 46-47

construção When Else, 31-32
Elsif: construção IF Else, 46-47
Endfile: arquivo, 207
Entidade de projeto, 10-12
Entity. *Veja* Entidade de projeto
Enumerado
declaração tipo, 86-90, 188-189
tipo, 14-17, 188
Enumerated. *Veja* Enumerado
Error: valor tipo Severity_Level, 221-224
Escalar: tipo, 14-17, 188
Event: atributo sinal, 65-73
Exit: comando, 113

F

Failure: valor tipo Severity_Level, 221-224
Falling_edge: função std_logic_1164, 153-156
False: valor, 14-17
File. *Veja* Arquivo
tipo, 208
File_Close: arquivo, 210-212
File_Open: arquivo, 210-212
Físico
definição de tipo, 189
tipo, 14-17, 188
Flip Flop
definição, 78
inferência, 81-85, 90-91
For
comando Generate, 106-108
comando Loop, 108-114
comando Wait, 49-51
configuração, 229-234
Formal parameter. *Veja* Parâmetro formal
FPGA: síntese, 3-7
fs: unidade tipo time, 14-17
Função
comportamento de variáveis, 128-131
corpo, 121-123
de resolução, 151-153, 224-227
declaração, 121-123
introdução, 121-123
pseudônimos, 197
Function. *Veja* Função

G

Generate: comando, 106-108
Generic. *Veja* Genérico
Generic Map, 104
Genérico, 10-12, 103-105, 227-239
Gray: atribuição de estados, 86-90
Guarded
comando Block, 34-36
inferência de alta impedância, 159
inferência de registrador, 93

H

HDL, 1-2
High
atributo tipo, 65-73
atributo vetor, 65-73
High impedance. *Veja* Alta impedância
hr: unidade tipo time, 14-17

I

IEEE, 1-2, 151-153, 170-187
If
comando Generate, 106
construção IF Else, 46-47
In
esquema de geração FOR, 106
esquema de iteração FOR, 108-114
porta, 11-12
Inércia atraso, 62-63
Inertial. *Veja* Atraso com inércia
Inferência de *flip flop*
atributos, 121-125
comando Block, 93
comando Wait, 93
construção If Else, 81-86
construção When Else, 93
função, 121-123
procedimento, 123-125
Inferência de *latch*
comando Block, 34-36
comando Null, 51-52
construção If Else, 79-80
construção When Else, 37-40
construção With Select, 37-40
Inicialização
assíncrona, 82
síncrona, 81
Inout: porta, 11-12
Instantiation. *Veja* Solicitação
Integer: tipo, 14-17
Inteiro: tipo, 14-17
Is_X: função std_logic_1164, 153-156

L

Label. *Veja* Rótulo
Last_event: atributo sinal, 65-73
Last_value: atributo sinal, 65-73
Latch. *Veja* Inferência de *latch*
Left
atributo tipo, 65-73
atributo vetor, 65-73
Leftof: atributo tipo, 65-73
Length: atributo vetor, 65-73
Library. *Veja* Biblioteca
Line: pacote Textio, 212-218
Lista de sensibilidade, 43-44, 94
Lógico: operador, 17-21
Loop: comando, 108-114
Low
atributo tipo, 65-73
atributo vetor, 65-73
LRM, 198-199

M

Mapa
de genéricos, 104
de portas, 100
Máquina de estados
codificação de estados, 86-90
inferência, 86-90
Mealy definição, 86-90
Moore definição, 86-90
Memórias
RAM, 198-203
ROM, 198-203
síntese, 198-203
min: unidade tipo time, 14-17
Mod: operador multiplicação, 17-21, 170-172
ms: unidade tipo time, 14-17
Multiplicação: operador, 17-21

N

Named notation. *Veja* Notação nomeada
Nand: operador lógico, 17-21, 153, 170-175
Natural: subtipo, 14-17
Next: comando Loop, 108-114
Next When: comando Loop, 108-114
Nor: operador lógico, 17-21, 153, 170-175
Not: operadores diversos, 17-21, 153, 170-175
Notação
nomeada, 12-14, 100-103
posicional, 12-14, 100-103
Note: valor tipo Severity_Level, 221-224
ns: unidade tipo time, 14-17
Null: comando, 51-52, 165
Numeric. *Veja* Numérico
Numeric_bit: pacote, 170
Numeric_std: pacote, 170
Numérico: tipo, 14-17

O

Object. *Veja* Objeto
Objeto, 10-12
On: comando Wait, 49-51
One hot: atribuição de estados, 86-90
Open collector. *Veja* Coletor aberto
Open drain. *Veja* Dreno aberto
Open: palavra reservada, 100
Operadores, 17-21
Or: operador lógico, 17-21, 170-175
Ordem da análise, 141-145
Others
agregados, 14-17
construção "Case When", 48-49
construção "With Select", 32-34
Out: porta, 11-12
Overflow. *Veja* Transbordo
Overloading. *Veja* Sobrecarregamento

P

Package. *Veja* Pacotes
Pacotes
corpo, 137-141
declaração, 137-141
Padrão IEEE 1076, 170-187
Padrão IEEE 1164, 151-169
Parâmetro
formal, 119-121
real, 119-121
Physical. *Veja* Físico
Place and route: síntese, 3-7
Port. *Veja* Porta
Port Map, 100, 228
Porta
cláusula, 11-12
de passagem, 158-166
Pos: atributo tipo, 65-73
Positional notation. *Veja* Notação posicional
Positive: subtipo, 14-17
Posposto: comandos concorrentes, 236-237
Postponed. *Veja* Posposto
Precedência: em operadores, 17-21
Pred: atributo tipo, 65-73
Procedimento
chamada região código concorrente, 128-131
comportamento de variáveis, 128-131
corpo, 123-125
declaração, 123-125
introdução, 123-125

posposto, 236-237
pseudônimos, 197
Procedure. *Veja* Procedimento
Process: comando, 36, 236-237
ps: unidade tipo time, 14-17
Pseudônimo, 197

Q

Quiet: atributo sinal, 65-73

R

Range: atributo vetor, 65-73
Read
arquivo binário VHDL-1987, 207-209
arquivo binário VHDL-1993, 210-212
arquivo texto VHDL-1987, 208-209
arquivo texto VHDL-1993, 210-212
Readline
arquivo texto VHDL-1987, 208-209
arquivo texto VHDL-1993, 210-212
Real
parâmetro real, 119-121
tipo predefinido, 14-17
Record. *Veja* Registro
Register transfer level: síntese, 3-7
Registrador sensível a borda. *Veja* Flip Flop
Registrador sensível a nível. *Veja* Latch
Registro
declaração tipo, 196
operações de teste com o tipo, 241-253
tipo composto, 188
Relacional operador, 17-21
Rem: operador multiplicação, 17-21, 172
Report
cláusula, 221
comando, 223
Reset: definição, 79
Resize: função escalonamento, 170-175
Resolution function. *Veja* Função de resolução
Reverse_range: atributo vetor, 65-73
Right
atributo tipo, 65-73
atributo vetor, 65-73
Rightof: atributo tipo, 65-73
rising_edge: função std_logic_1164, 153-156
Rol: operador rotação, 17-21, 170-175
Ror: operador rotação, 17-21, 170-175
Rotate_left: operador rotação, 170-175
Rotate_right: operador rotação, 170-175
Rótulo: componente, 100, 227-236
RTL: síntese, 3-7

S

Scalar. *Veja* Escalar
SDF: síntese, 3-7
sec: unidade tipo time, 14-17
Select: construção With Select, 32-34, 236-237
Set: definição, 79
set-up: teste, 221-222
Severity: cláusula, 221
Severity_Level: tipo, 221-224
Shared variable. *Veja* Variável compartilhada
Shift_left: operador deslocamento, 170-175
Shift_right: operador deslocamento, 170-175
Side: pacote Textio, 210-212
Signal. *Veja* Sinal
Signed: tipo, 170
Sinal
classe de objeto, 12-14
global, 146
Síntese. *Veja* Inferência
alta impedância (saídas), 158-166
atributo, 65-73
circuito síncrono, 79-80, 82-86, 90-91
circuito síncrono "numerid_std", 181-1832
codificador de prioridade, 37-40
comando Block, 34-36, 93
comando Generate, 106
comando Loop, 108-114
comando Null, 51-52, 165
comando Wait, 93
comparação com valor '-', 158-166
componente, 106-108, 227-236
configuração, 234-236
constante deferida, 145
construção Case When, 51-52, 86-90, 158-166
construção If Else, 78-80, 82-86, 90-91, 158-166, 182-183
construção When Else, 37-40, 93-94
construção With Select, 37-40, 178-183
contadores, 91-93, 178-183
conversão de tipo, 108-114
detecção de bordas "std_logic_1164", 158-166
empregando pacote "std_logic_1164", 158-166
função, 123-125
lista de sensibilidade, 43-44, 94
mapa de genéricos, 227-236
máquina de estados, 86-90
memória, 198-203
minimização de tabelas com valor "-", 158-166
procedimento, 123-125
registrador de deslocamento, 172-173
sinal global, 146
sobrecarregamento, 125-127
somador, 106-114, 121-127, 178-183
subtrator, 123-125
valor "-", 158-166
variável compartilhada, 128-131
Sla: operador deslocamento, 17-21
Sll: operador deslocamento, 17-21, 173
Sobrecarregamento, 125-127
Solicitação
de componente, 100
direta de componente, 102
Sra: operador deslocamento, 17-21
Srl: operador deslocamento, 17-21, 173
Stable: atributo sinal, 65-73
Standard IEEE 1076. *Veja* Padrão IEEE 1076
Standard IEEE 1164. *Veja* Padrão IEEE 1164
Standard: pacote, 135-137
Std: biblioteca, 135-137
Std_logic: tipo, 151-153
Std_logic_1164: pacote, 151-153
Std_logic_vetor: tipo, 151-153
Std_match: f. numeric_bit numeric_std, 170-175
Std_ulogic: tipo, 151-153
Std_ulogic_vetor: tipo, 151-153
String: tipo, 14-17
Subprograma, 119-121
Subtipo
declaração, 189
introdução, 14-17
SubType. *Veja* Subtipo
Succ: atributo tipo, 65-73

T

Teste
de descrições, 246-253
sinais de estímulos, 241-246
Text: pacote Textio, 210-212
Textio: pacote, 135-137
Tipo(s).
composto, 14-17
declaração de, 14-17, 188-196
escalar, 14-17
predefinidos, 14-17
To: limites, 17-21, 106, 108-114, 127
To_Bit: função std_logic_1164, 153-156
To_Bitvector: função std_logic_1164, 153-156
To_Integer: f. numeric_bit numeric_std, 170-175
To_Signed: f. numeric_bit numeric_std, 170-175
To_StdLogicVector: f. std_logic_1164, 153-156
To_StdULogic: função std_logic_1164, 153-156
To_StdULogicVector: f. std_logic_1164, 153-156
To_Unsigned: f. numeric_bit numeric_std, 170-175
To_UX01: função std_logic_1164, 153-156
To_X01: função std_logic_1164, 153-156
To_X01Z: função std_logic_1164, 153-156
To_01: f. numeric_bit numeric_std, 170-175
Transaction: atributo sinal, 65-73
Transbordo, 179-181
Transmission gate. *Veja* Porta de passagem
Transport: cláusula, 62-63
True: valor, 14-17
Type
declaração, 14-17, 188-205
definição, 14-17, 188

U

Unaffected: palavra reservada, 37
Unsigned: tipo, 170
Until: comando Wait, 49-51
us: unidade tipo time, 14-17
Use: cláusula, 137-141, 175-183, 212-218, 227-236

V

Val: atributo tipo, 65-73
Variable. *Veja* Variável
Variável
compartilhada, 128-131
inferência de registrador, 90
objeto, 12-14, 63-64
Vetor
composto de elementos tipo vetor, 193
declaração, 190-192
introdução, 14-17
limites em aberto, 192-193
multidimensional, 195
VHSIC, 1-2
VITAL: síntese, 3-7

W

Wait
comando, 49-51
For, 49-51
inferência de registrador, 93
On, 49-51
Until, 49-51
Warning: valor tipo Severity_Level, 221-224
When
comando Exit, 108-114
comando Next, 108-114
construção When Else, 31-32
inferência de registrador When Else, 93
While: comando Loop, 108-114
Width: pacote Textio, 210-212

With: construção With Select, 32-34
Work: biblioteca, 135-137
Write
arquivo binário VHDL-1987, 207-209
arquivo binário VHDL-1993, 210-212
arquivo texto VHDL-1987, 215-217
arquivo texto VHDL-1993, 217-218
Writeline
arquivo texto VHDL-1987, 215-217
arquivo texto VHDL-1993, 217-218

X

Xnor: operador lógico, 17-21, 153, 170-175
Xor: operador lógico, 17-21, 153, 170-175