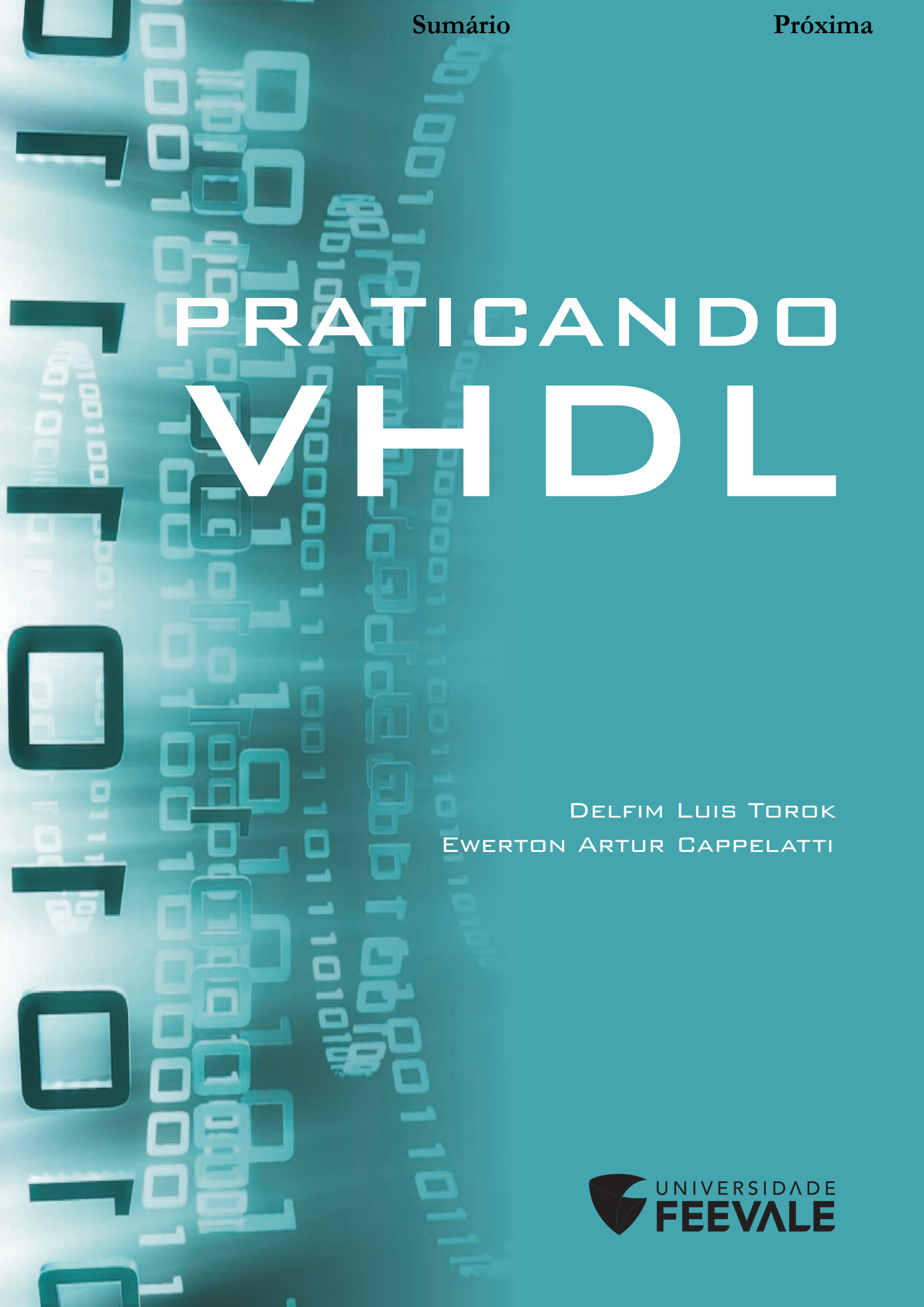


PRATICANDO VHDL

Delfim Luis Torok
Ewerton Artur Cappelatti

UNIVERSIDADE FEEVALE

Associação Pró-ensino Superior em Novo Hamburgo - ASPEUR
Universidade Feevale

PRATICANDO VHDL

Delfim Luis Torok
Ewerton Artur Cappelatti

UNIVERSIDADE FEEVALE

Novo Hamburgo - Rio Grande do Sul - Brasil
2011

EXPEDIENTE

PRESIDENTE DA ASPEUR
Argemi Machado de Oliveira

REITOR DA UNIVERSIDADE FEEVALE
Ramon Fernando da Cunha

PRÓ-REITORA DE ENSINO
Inajara Vargas Ramos

PRÓ-REITOR DE PESQUISA E INOVAÇÃO
João Alcione Sganderla Figueiredo

PRÓ-REITOR DE PLANEJAMENTO E ADMINISTRAÇÃO
Alexandre Zeni

PRÓ-REITORA DE EXTENSÃO E ASSUNTOS COMUNITÁRIOS
Gladis Luisa Baptista

COORDENAÇÃO EDITORIAL
Inajara Vargas Ramos

EDITORA FEEVALE
Celso Eduardo Stark
Daiane Thomé Scariot
Gislaine Aparecida Madureira Monteiro

CAPA
Gislaine Aparecida Madureira Monteiro

EDITORAÇÃO ELETRÔNICA
Gislaine Aparecida Madureira Monteiro
Celso Eduardo Stark

Dados Internacionais de Catalogação na Publicação (CIP)
Universidade Feevale, RS, Brasil
Bibliotecária Responsável: Susana Fernandes Pfarrius Ladeira - CRB 10/1484

Torok, Delfim Luis
Praticando VHDL / Delfim Luis Torok , Ewerton Cappelatti . – Novo Hamburgo : Feevale, 2010.
113 p. ; il. ; 30 cm.

Inclui bibliografia e apêndice.
ISBN 978-85-7717-120-0

l. VHDL (Linguagem descritiva de hardware). 2. Eletrônica digital. 3. Circuitos lógicos. I. Torok, Delfim Luis. II. Título.

CDU 004.312

UNIVERSIDADE FEEVALE
Editora Feevale
Campus II: RS 239, 2755 - CEP: 93352-000 - Vila Nova - Novo Hamburgo - RS
Fone: (51) 3586.8819 - Homepage: www.feevale.br/editora

SUMÁRIO

LISTA DE FIGURAS

PREFÁCIO

A quem é destinado este livro

A ideia de escrever um livro abordando VHDL surgiu da necessidade de elaborar material didático para disciplinas do curso de graduação em Engenharia Eletrônica da Universidade Feevale. Eletrônica Digital, Arquitetura e Organização de Computadores, Programação Aplicada e Microeletrônica estão entre as disciplinas nas quais VHDL é abordada. Acadêmicos de Engenharia Eletrônica e Ciência da Computação encontrarão neste volume inicial apoio para a aplicação prática em descrição de *hardware*, bem como a modelagem e simulação de dispositivos eletrônicos reconfiguráveis. Os próximos volumes destinar-se-ão a uma abordagem mais profunda, incluindo a síntese física de dispositivos com maior complexidade.

SOBRE OS AUTORES

Delfim Luis Torok é Engenheiro Eletrônico graduado pela Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS. Possui especialização em Automação Industrial pela Universidade Federal de Santa Catarina - UFSC e mestrado em Ciência da Computação pela PUCRS.

Ewerton Artur Cappelatti é Engenheiro Eletrônico graduado pela Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS. Possui especialização em Processamento de Sinais pela PUCRS e mestrado em Ciência da Computação pela PUCRS.

AGRADECIMENTOS

Agradecemos à Universidade Feevale pela oportunidade, pelo incentivo e pelo apoio para a realização desta obra.

INTRODUÇÃO

Very High Speed Hardware Description Language - VHDL é uma linguagem de descrição de *hardware* (Hardware Description Language - HDL) que foi concebida na década de 80 a partir da necessidade de uma ferramenta computacional para projetos e documentação do Departamento de Defesa dos Estados Unidos da América (Defense Advanced Research Projects Agency - DARPA).

A primeira versão da VHDL data de 1987, tendo sido atualizada em 1993. Foi a primeira linguagem de descrição de *hardware* padronizada pelo Institute of Electrical and Electronics Engineers – IEEE, recebendo a denominação de IEEE 1076-83 e IEEE1076-93, respectivamente. Um padrão adicional, o IEEE 1164, foi estabelecido posteriormente para introduzir sistemas lógicos mutivariáveis.

Linguagens de descrição de *hardware*, assim como VHDL, trazem consigo a vantagem de códigos independentes de tecnologia e fabricante, sendo portáveis e reutilizáveis.

ESTRUTURA DO VOLUME

Praticando VHDL está dividido em duas partes. Inicia com um tutorial sobre o *software* ModelSim®, utilizado como ferramenta computacional para descrição de dispositivos digitais. Aborda fluxo de projeto, síntese funcional, bancada de testes virtual e simulação de circuitos descritos em VHDL. Em sua segunda parte, são apresentados projetos de circuitos combinacionais básicos, que evoluem gradativamente em complexidade e funcionalidade, bem como são introduzidos dispositivos que combinam arquiteturas combinacionais e sequenciais.

1 Introdução à Modelagem e Simulação

1.1 Modelagem

VHDL é uma linguagem de descrição de *hardware* na qual as atribuições deste são executadas na sequência em que estão declaradas (Figura 1.1).

Há dois tipos básicos de declarações:

Sequencial - são declarações executadas uma após a outra, como na programação em linguagens formais (C, Pascal, etc.) e as declarações anteriores são ignoradas após sua execução.

Concorrente - são declarações continuamente ativas. Portanto, a sua ordem não é relevante. Declarações concorrentes são especialmente adaptadas ao modelo de *hardware* paralelo.

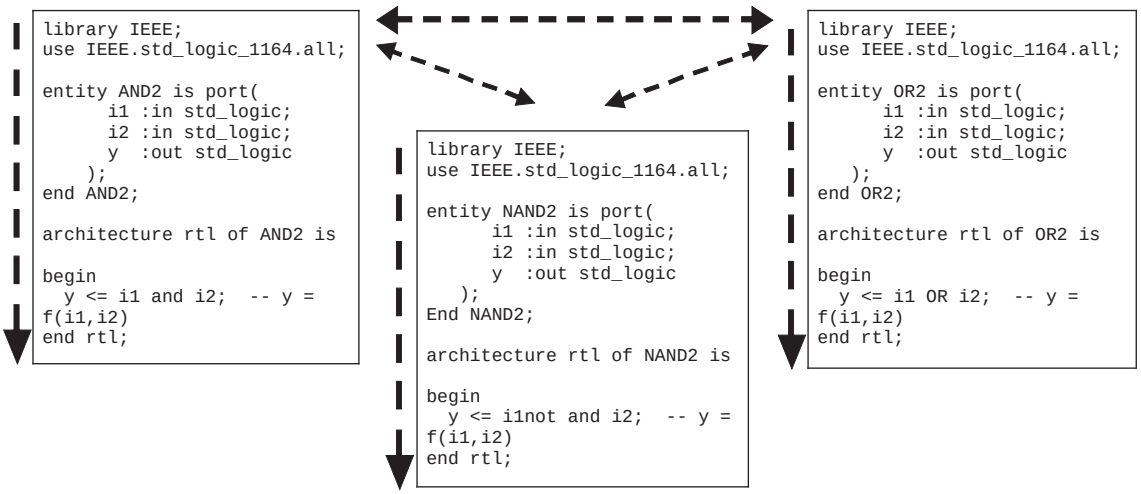


Figura 1.1 - Execução de atribuições em VHDL

VHDL também utiliza uma metodologia baseada em três importantes características técnicas de modelagem:

Abstração - permite a descrição de diferentes partes de um sistema com diferentes níveis de detalhes. Em módulos utilizados para simulação, não há necessidade de serem descritos com o mesmo detalhamento dos módulos que serão sintetizados[1].

Modularidade - permite ao projetista dividir o projeto em blocos funcionais e após descrevê-los como um bloco único contendo vários blocos funcionais interconectados (Figura 1.2).

Hierarquia - permite ao projetista construir módulos individuais e cada um destes pode ser composto de vários submódulos. Cada nível da hierarquia pode conter módulos de diferentes níveis abstração. Um submódulo em um determinado nível da hierarquia maior pode estar presente em um módulo de nível hierárquico menor.

[1]Síntese é o processo de "tradução" ou compilação de um código VHDL para uma descrição abstrata, em uma linguagem mais próxima da implementação. A síntese lógica é ainda um processo independente da tecnologia. O resultado obtido é uma representação do dispositivo em nível de transferência entre registradores, na qual se definem os registradores, suas entradas e saídas e a lógica combinacional entre eles.

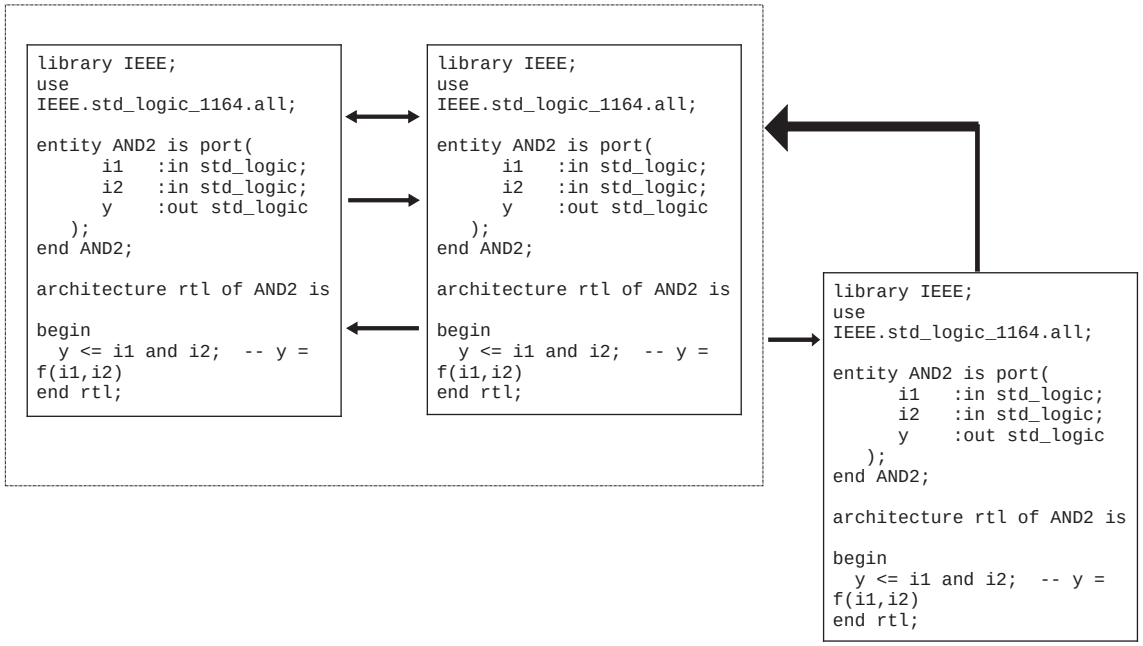


Figura 1.2 - Interconexão de módulos individuais

Uma descrição em VHDL pode conter diferentes níveis de abstração: comportamental (ou algorítmico); transferência entre registradores; funcional em nível de portas lógicas com atraso unitário ou funcional em nível de portas lógicas com atrasos detalhados.

O nível mais alto de abstração é o comportamental (*behavioral*), que permite descrever o comportamento do circuito através de laços (*loops*) e processos. Na descrição comportamental, faz-se uso de texto ou equações para descrever como o dispositivo eletrônico deve se comportar. Neste nível de abstração, o circuito é definido na forma de um algoritmo, utilizando construções similares àquelas de linguagens de programação formais.

Já o nível intermediário de abstração possibilita descrever o funcionamento do circuito em termos de lógica combinacional (booleana), englobando a representação do dispositivo em nível de transferência entre registradores (Register Transfer Level – RTL), que consiste na utilização de funções lógicas combinacionais e registradores.

No nível mais baixo de abstração (estrutural), faz-se uma representação do circuito semelhante a uma lista, descrevendo a rede de portas lógicas e suas interconexões. Neste nível de abstração, o circuito é descrito mais próximo da implementação real, podendo ser definidas portas lógicas com atrasos unitários ou com atrasos detalhados.

1.2 Fluxo de projeto em VHDL

Seguir um fluxo de projeto é básico para um desenvolvimento em VHDL. É o que permite chegar à síntese de um circuito, ou seja, a geração de uma lista otimizada de portas lógicas e registradores (RTL), configurando um sistema eletrônico sobre um dispositivo programável (PLD[2] ou FPGA[3]) e/ou posteriormente sua implementação como ASIC[4] (Figura 1.3).

[2] PLD – Programmable Logic Device - dispositivo lógico programável.
[3] FPGA – Field Programmable Gate Array - dispositivo lógico configurado pelo usuário (*fabless*).
[4] ASIC - Application Specific Integrated Circuit – circuito integrado projetado para executar uma aplicação específica.

Na metodologia *Top-Down*, o projetista inicia a descrição do *hardware* no nível de abstração mais elevado e simula o sistema. Posteriormente, ele pode descrevê-lo com maiores detalhes (descendo níveis) e voltar a simulá-lo. O fluxo diagrama da mostra os passos básicos para modelagem de um projeto em HDL, iniciando por criar o projeto e seus módulos, simular e depurar os resultados obtidos pela simulação dos seus módulos, bem como o projeto como um todo.

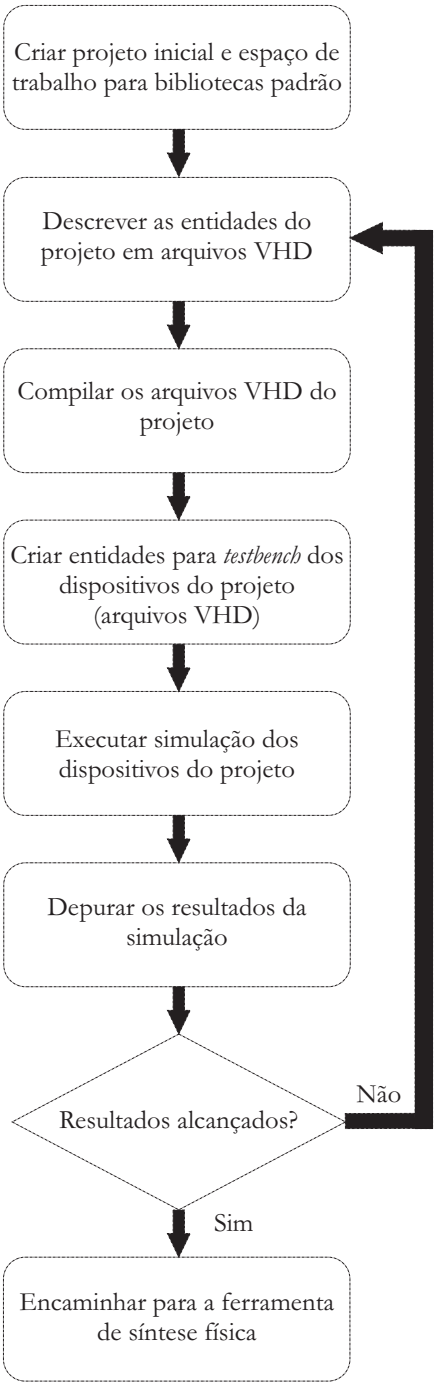


Figura 1.3 - Fluxo básico para modelagem de um projeto em HDL

1.3 Ferramentas para projetos em VHDL

Para se trabalhar com linguagens de descrição de *hardware*, utilizam-se ambientes de projeto denominados Electronic Design Automation (EDA), que permitem modelar, descrever, verificar, simular e compilar a descrição do *hardware* sob projeto. Adotou-se como EDA para este livro o ModelSim[5], que permite projetos em VHDL, Verilog, System Verilog e linguagens mistas de descrição de *hardware*.

No ModelSim, todos os projetos são compilados em uma biblioteca. Normalmente, inicia- se uma modelagem através da criação de uma biblioteca de trabalho, denominada "Work". A biblioteca "Work" é utilizada pelo compilador do ModelSim como diretório padrão, destino das unidades compiladas do projeto.

Para iniciar um novo projeto, siga o fluxo básico da Figura 1.3 e carregue o ModelSim, previamente instalado no computador. O Anexo A contém informações detalhadas de como obter e instalar este EDA. Após carregar o programa, inicie um novo projeto, conforme ilustrado na tela capturada do ModelSim (Figura 1.4).

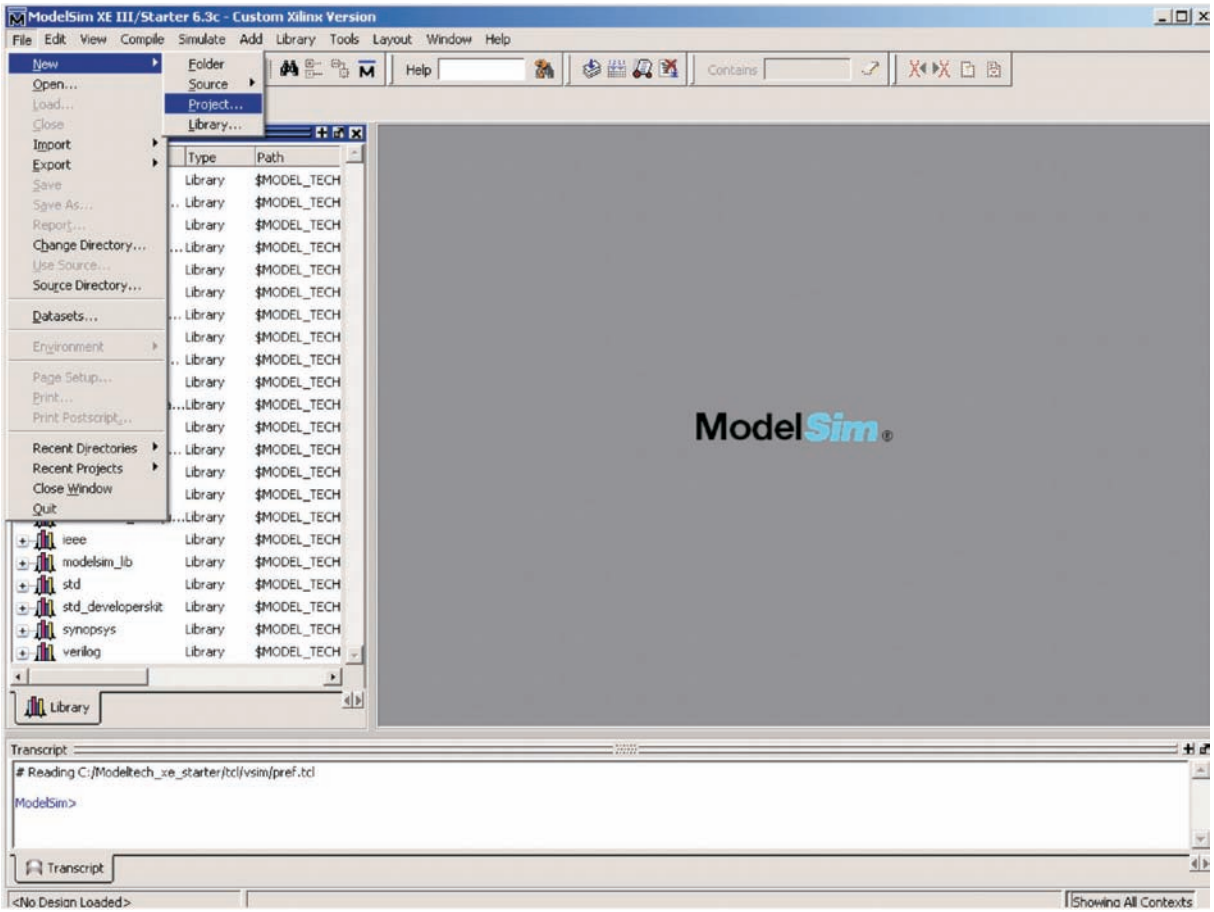


Figura 1.4 - Tela inicial do ModelSim

Aponte o *mouse* e clique na barra do menu inicial na sequência <File→ New→ Project>, chegando à tela "Create Project" (Figura 1.5). Determine um nome para o projeto, por exemplo, "Proj_Cap1" e, após escolha sua localização (pasta), que pode ser identificada com o mesmo nome do projeto para facilitar a identificação do diretório no computador.

[5]ModelSim é um produto da Mentor Graphics.

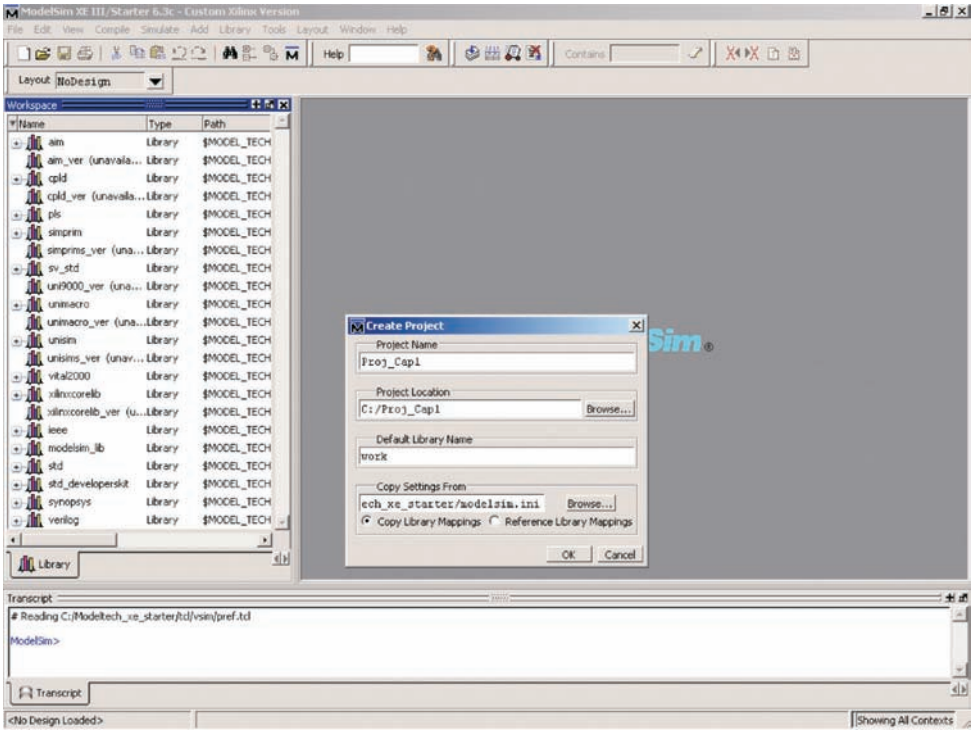


Figura 1.5 - Janela para a criação do "Proj_Cap1"

Uma vez digitados os nomes do projeto e do respectivo diretório, os demais campos serão assumidos por definição padrão (*default*) do ModelSim e, desta forma, encerra-se a criação do projeto, confirmando com "OK".

A próxima tela, "Add items to the Project", permite a criação de um novo arquivo fonte que será adicionado ao projeto. Esta tela também é utilizada para a inserção de novos itens ao projeto, conforme ilustrado na Figura 1.6.

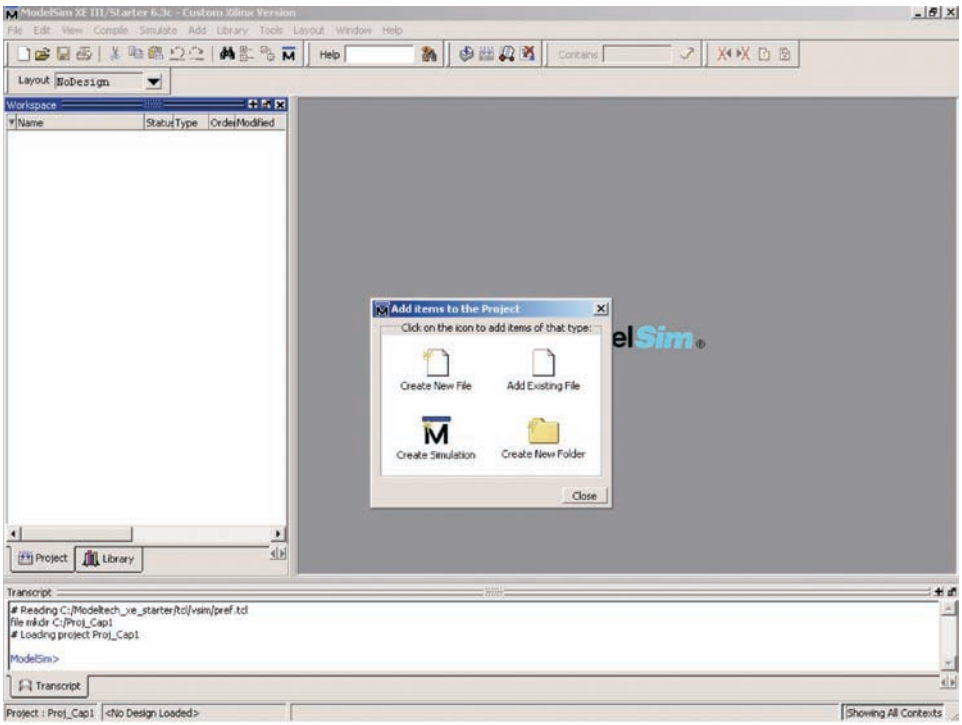


Figura 1.6 - Janela para a inserção de novos itens ao projeto

A escolha do item "Create New File" permite introduzir o nome de um novo arquivo que será anexado ao projeto e determina uma extensão de acordo com o tipo da linguagem escolhida para desenvolver o projeto. Dado que a linguagem de descrição de *hardware* VHDL é o objeto de estudo adotado neste livro, todas as descrições de circuitos para a prática estão em VHDL.

Conforme ilustra a Figura 1.7, a tela "Create Project File", na sequência do item "Create New File", determina o nome de um arquivo fonte, que poderá conter desde a descrição de uma simples porta lógica até um sistema complexo, por exemplo, um núcleo processador, tal como o PicoBlaze[6]. Para este tutorial, sugere-se "and2" para o nome do arquivo que irá conter a descrição da entidade e respectiva arquitetura de uma porta lógica AND de duas entradas. Na tela "Create Project File", o campo "ADD File as type" determina a extensão que é agregada ao arquivo de trabalho (and2.vhd) e o campo "Folder" determina o "Top Level", a mais alta hierarquia de compilação[7] dos arquivos do projeto. Esta hierarquia poderá ser reordenada de acordo com a adição de novos arquivos ao projeto. A ordenação poderá ser determinada de forma manual ou automática de acordo com o desenvolvimento das partes do projeto.

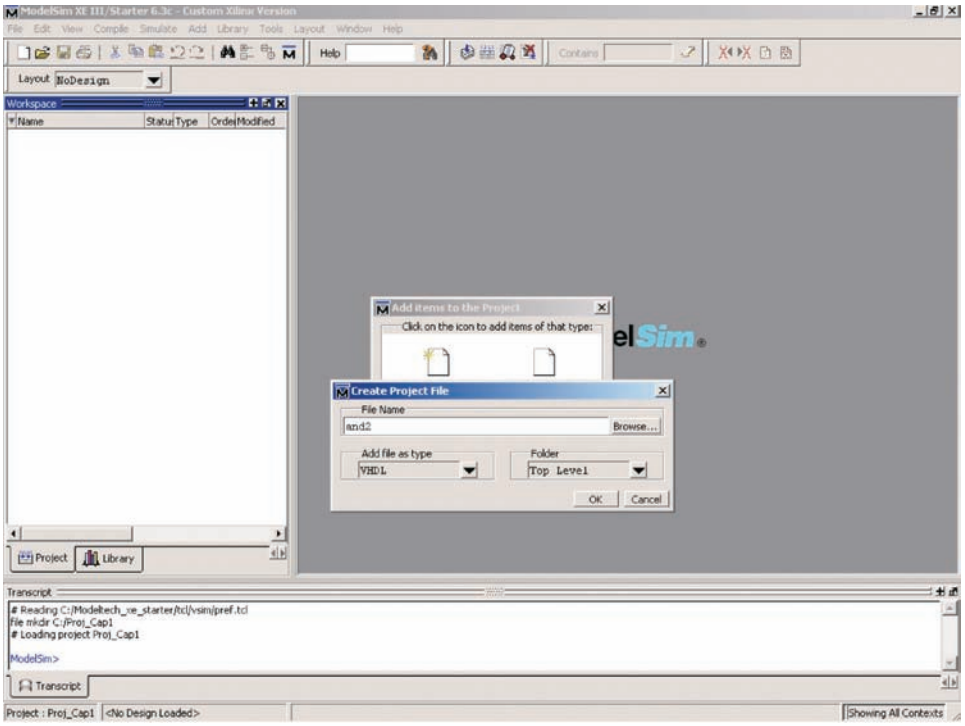


Figura 1.7 - Janela para criar um novo arquivo fonte do projeto

Uma vez digitado nome do arquivo que irá conter a descrição da porta lógica AND de duas entradas, os demais campos serão assumidos por *default* e, dessa forma, encerra-se a criação do arquivo fonte, confirmando com "OK".

Na próxima tela, ilustrada na Figura 1.8, observa-se que o novo arquivo fonte é adicionado ao projeto com a extensão "vhd". A coluna estado (*status*) apresenta o caractere "?" em função de o arquivo estar vazio, não contendo uma descrição em VHDL.

---

[6]Microprocessador desenvolvido em VHDL para as famílias de FPGAs Spartan™-3, Virtex™-4, Virtex-II e Virtex-II Pro da Xilinx.

[7]O código fonte VHDL é compilado, gerando uma biblioteca de componentes que descrevem algebricamente a lógica dos componentes e uma listagem de conexões em nível de portas (*Netlist*).

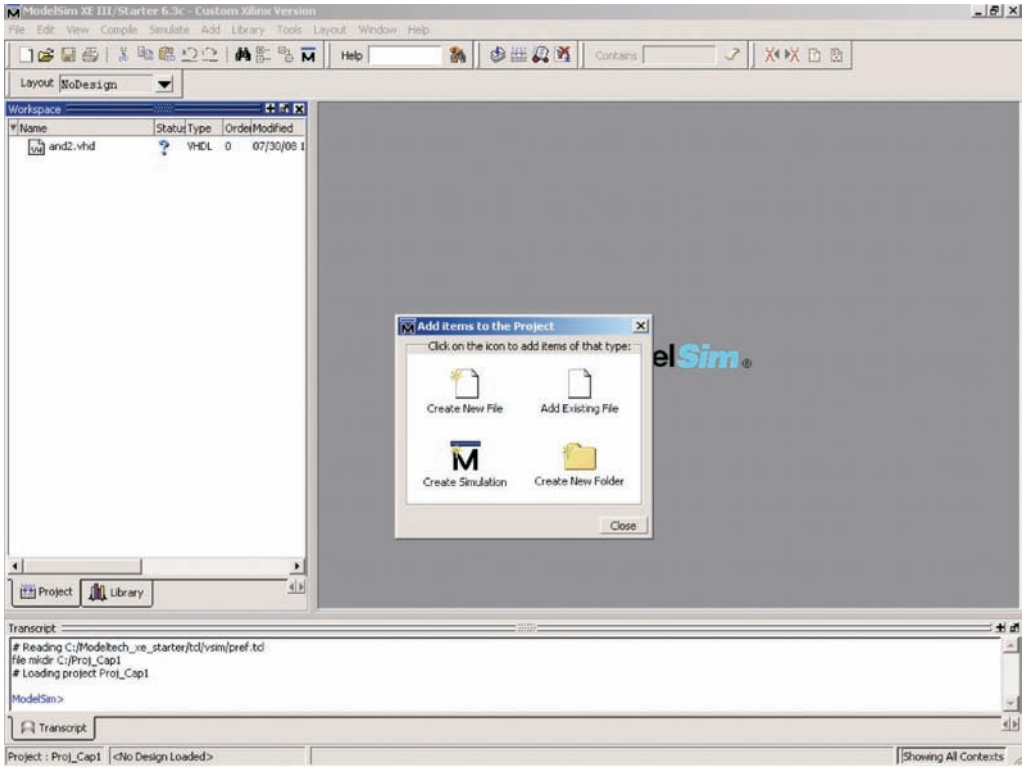


Figura 1.8 - Tela apresentada após criar um novo arquivo fonte

Após o fechamento da tela "Add itens to the project", confirmando-se com "Close", é necessário abrir o arquivo de trabalho "and2.vhd" por meio de um duplo clique, como apresenta a tela da Figura 1.9.

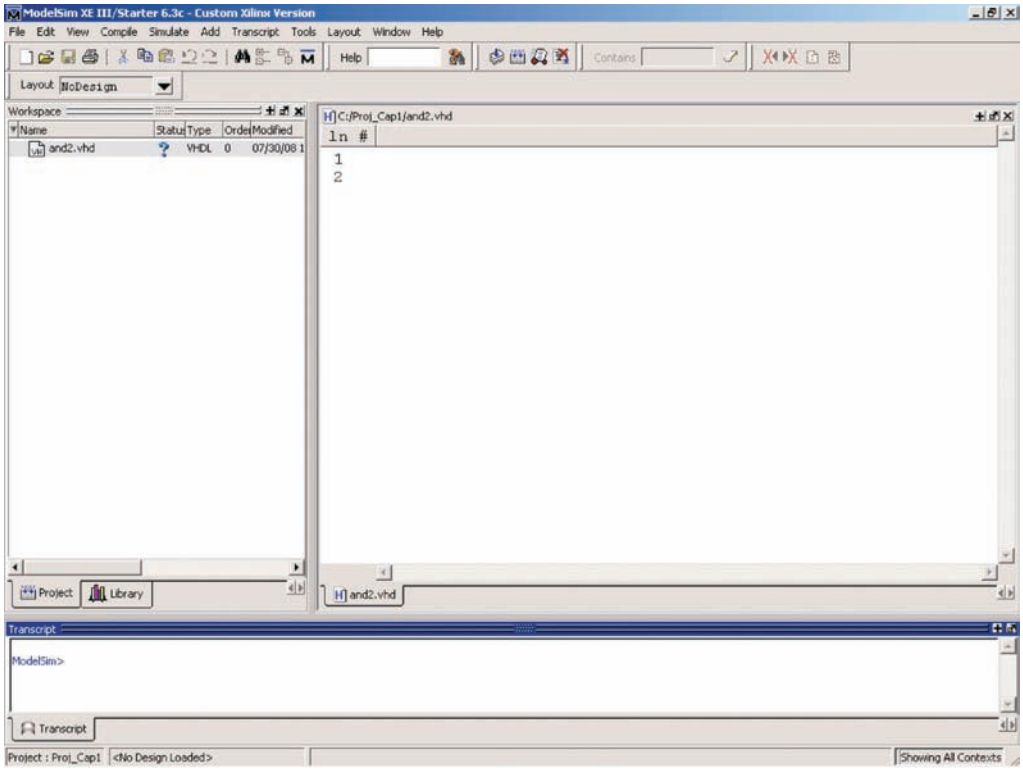


Figura 1.9 - Tela apresentada após abrir o arquivo de trabalho "and2.vhd"

O novo arquivo de trabalho "and2.vhd" é aberto com duas linhas em branco, pronto para ser digitada a descrição da porta lógica. Contudo, se for solicitada uma compilação deste arquivo vazio, com um clique sobre ícone localizado na barra de ferramentas para compilação, certamente ocorrerá um erro, conforme ilustrado na janela da Figura 1.10.

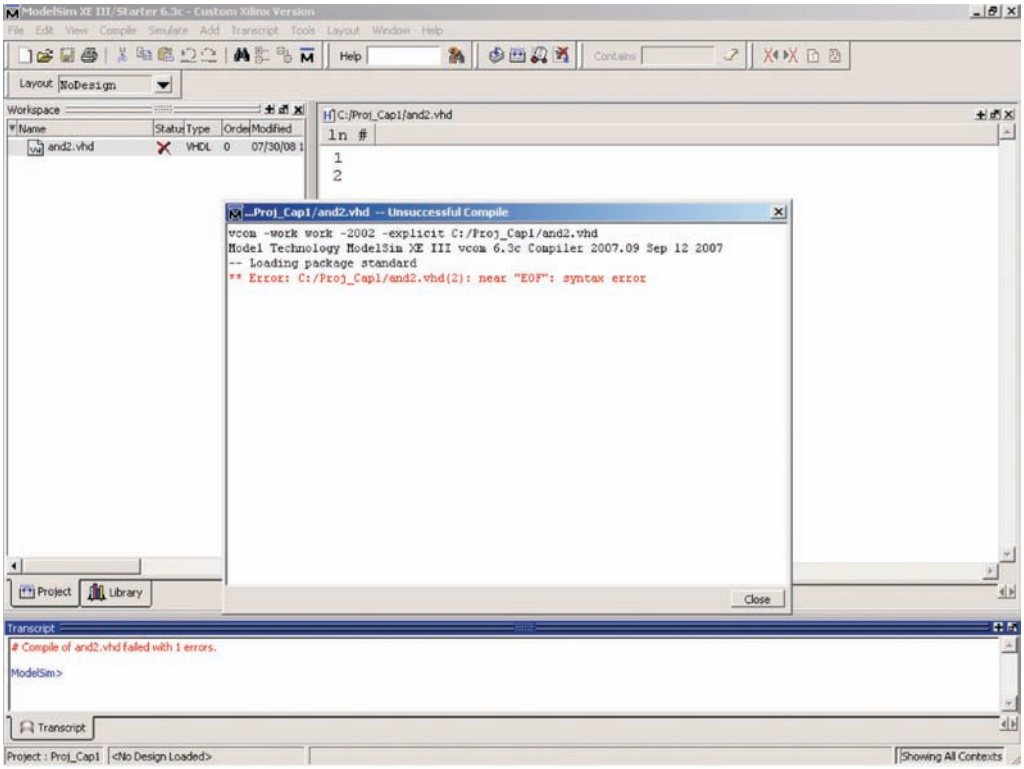


Figura 1.10 - Janela apresentada após a compilação do "arquivo and2.vhd"

Sempre que a compilação resultar com erro, informado na janela "Transcript" localizada na parte inferior da tela principal, é necessário apontar sobre o resultado:

```
"# Compile of and2.vhd failed with 1 errors."
```

E com duplo clique obtém-se uma nova janela com mais detalhes sobre o erro ocorrido.

Para se obter uma compilação com sucesso do *hardware* objeto do projeto, no caso do nosso exemplo uma porta lógica AND de duas entradas, é necessário digitar uma descrição completa e adequadamente formatada definida pelo padrão IEEE[8] para linguagem VHDL.

Inicia-se uma descrição de *hardware* pela declaração do pacote (*package*), contendo as constantes e bibliotecas que serão utilizadas na arquitetura da entidade objeto do projeto. A descrição deve seguir rigorosamente a formatação ilustrada pela Tabela 1, determinada pelo padrão IEEE 1164[9].

[8]IEEE - Institute of Electrical and Electronics Engineers - Instituto americano de normalização para sistemas aeroespaciais, computadores e telecomunicações, engenharia biomédica, eletricidade e eletroeletrônicos.

[9]Ao padrão IEEE 1076 de1987, primeiro padrão industrial, foi acrescentada a norma IEEE 1164, de 1993, adicionando novos tipos de dados, tais como: "std_logic e std_logic_vector".

Tabela 1 - Padrão IEEE 1164 para formatação da descrição VHDL

`LIBRARY IEEE;`
`USE IEEE.STD_LOGIC_1164.all;`
`USE IEEE.STD_LOGIC_UNSIGNED.all;`
	PACKAGE
	(Bibliotecas)

`ENTITY exemplo IS`
`PORT (`
`  <descrição dos pinos de entrada e saída>`
`  );`
`END exemplo;`
	ENTITY
	(Pinos de Entrada/Saída)

`ARCHITECTURE teste OF exemplo IS`
`BEGIN`
`  PROCESS( <pinos de entrada e sinal > )`
`  BEGIN`
`  < descrição do dispositivo >`
`  END PROCESS;`
`END teste;`
	ARCHITECTURE
	(Arquitetura)

Uma entidade (*entity*) é uma abstração que descreve um sistema, uma placa, um *chip*, uma função ou uma simples porta lógica. Define a interface do sistema digital descrito com o mundo externo,

$$y = f(i1, i2)$$

i1	i2	y
0	0	0
0	1	0
1	0	0
1	1	1

→ y = i1. i2

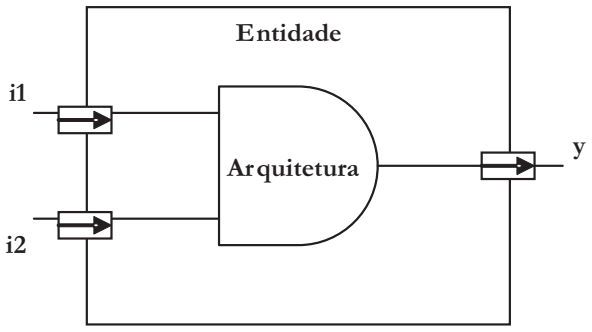


Figura 1.11 - Tabela verdade, equação booleana e diagrama da entidade

O modelo completo para descrição da porta lógica AND de duas entradas é transcrito como segue.

O modelo completo e sua respectiva compilação com sucesso estão ilustrados na Figura 1.13. Nessa edição da descrição da porta lógica AND, é possível determinar claramente os três blocos básicos

Na Figura 1.12, observa-se que, no arquivo de trabalho, foi digitado, da linha 1 até a 6, um cabeçalho sob forma de comentário. Comentários em VHDL devem ser precedidos por dois hífens (--). As linhas 7 e 8 contêm a indicação do pacote a ser utilizado e determinado pelo padrão IEEE 1164.

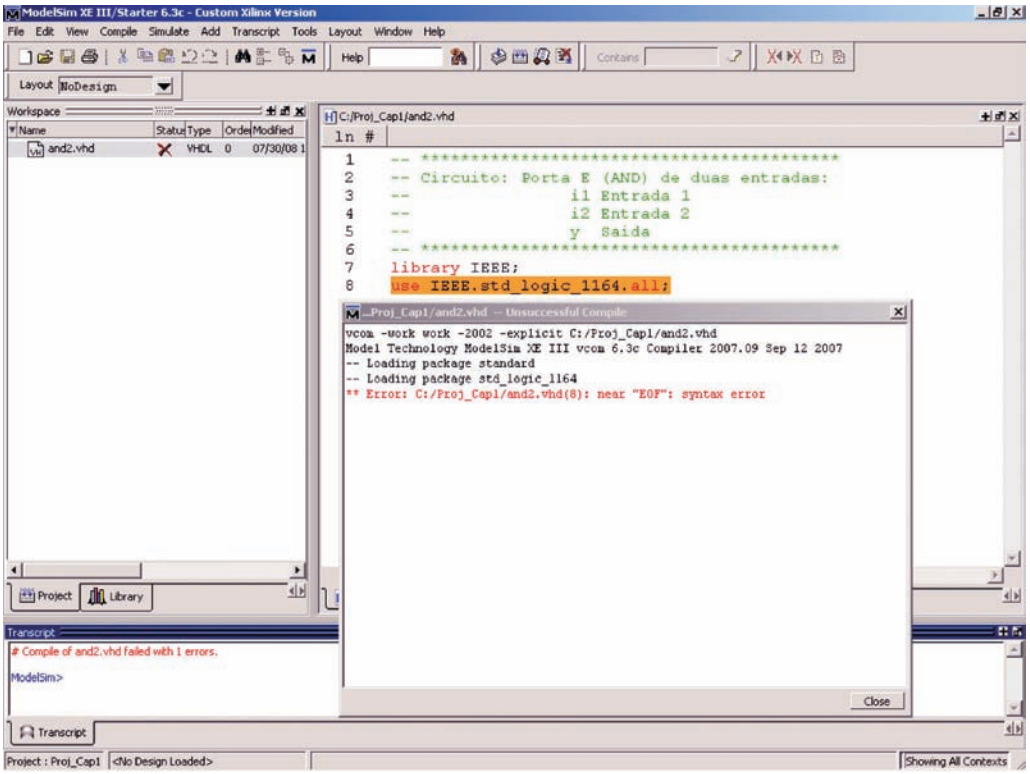


Figura 1.12 - Tela apresentada após a compilação

necessários para modelar qualquer sistema digital.

Quando for necessário utilizar algo não definido nas bibliotecas do VHDL padrão, faz-se uso do pacote (*package*). A única restrição é que o pacote deve ser previamente definido, antes do início da entidade (*entity*).

O uso do pacote é feito por meio de duas declarações: "library" e "use" (linhas 7 e 8 do código VHDL - Figura 1.13). Dos vários pacotes existentes, o mais conhecido e utilizado é o "STD_LOGIC_1164", que contém a maioria dos comandos adicionais mais usados em VHDL. O uso desse pacote é dado por:

```
library IEEE;
```

```
-- ****************************************
--  Circuito: Porta E (AND) de duas entradas:
--                i1 Entrada 1
--                i2 Entrada 2
--                y  Saída
-- ****************************************
```

Cabeçalho contendo uma breve descrição do dispositivo modelado (Comentário opcional)

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

Package (Pacote)
- constantes e bibliotecas

```
entity AND2 is port(
i1   :in std_logic;
i2   :in std_logic;
y    :out std_logic
);
end AND2;
```

Entity (Entidade)
- pinos de entrada e saída

```
architecture rtl of AND2 is

begin   -- a definicao inicia por begin
y <= i1 and i2;  -- y = f(i1,i2)
end rtl; -- a definicao termina por end
```

Architecture (Arquitetura)
- implementações do projeto

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

A entidade é a parte principal de qualquer projeto, pois descreve a interface do sistema. Tudo que é

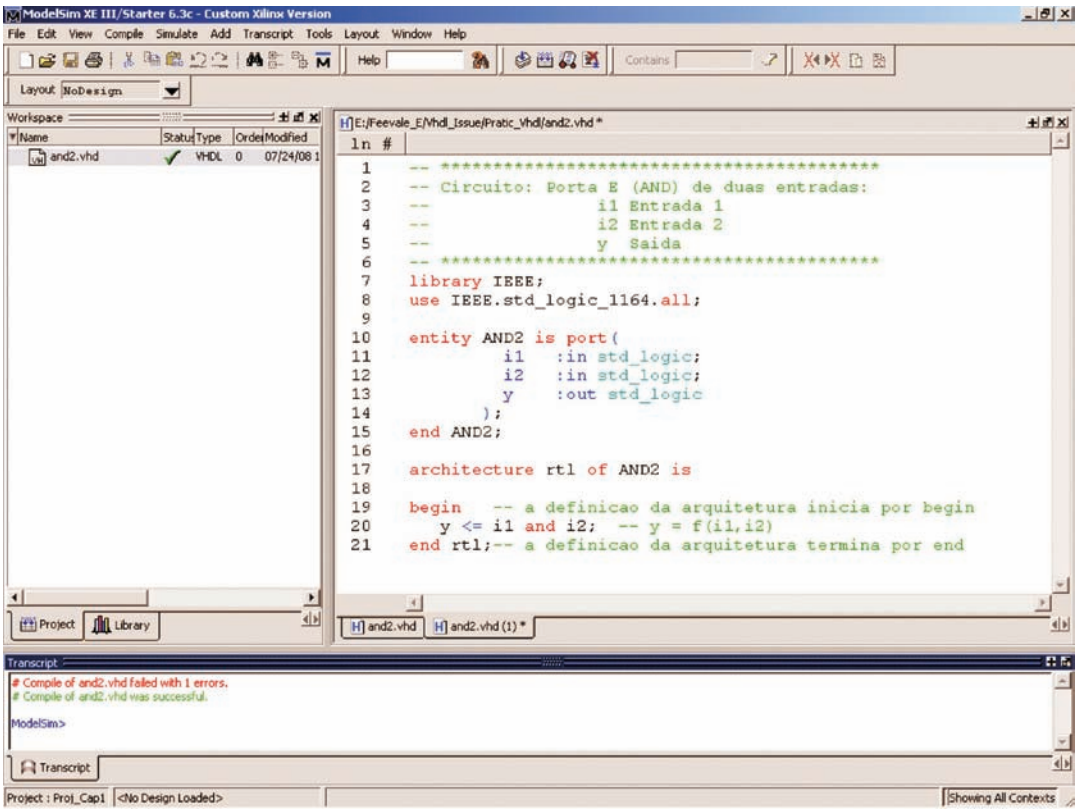


Figura 1.13 - Modelo completo e compilação com sucesso na janela "Transcript"

descrito na entidade fica automaticamente visível a outras unidades associadas com ela. O nome do sistema é o próprio nome da entidade, assim, deve-se sempre iniciar um projeto em VHDL pela entidade, como, por exemplo, "entity AND2 is", descrito na linha 10 do código VHDL e ilustrado na Figura 1.13.

Uma entidade (*entity*) é composta de duas partes; parâmetros (*parameters*) e conexões (*connections*). Os parâmetros referem-se a dimensões, valores e constantes vistos do mundo externo, tais como largura de barramento e frequência de operação, e são declarados como genéricos (*generics*). Conexões referem-se a onde e como ocorre a transferência de informações para dentro e fora do sistema e são declaradas por portas (*ports*).

A entidade de um sistema é tão importante que a própria arquitetura (*architecture*) é especificada na forma de arquitetura da entidade ("architecture rtl of AND2 is", linha 17 do código VHDL - Figura 1.13).

Um sistema pode ser descrito em termos da sua funcionalidade, isto é, o que o sistema faz, ou em termos de sua estrutura, isto é, como o sistema é composto. A descrição funcional especifica as respostas nas saídas em termos dos sinais aplicados nas entradas. Neste caso, não há nenhuma informação de como o sistema deverá ser implementado. A descrição estrutural, por sua vez, especifica quais componentes devem ser usados e como devem ser ligados. Essa descrição é mais facilmente sintetizada, porém exige mais experiência do projetista. Desta forma, pode-se ter várias arquiteturas capazes de implementar um mesmo circuito. Uma entidade pode ser formada por mais de uma arquitetura.

1.4 Bancada de Testes Virtual

Uma vez modelado o dispositivo (porta lógica AND), é necessário realizar sua simulação funcional, para comprovar e validar a função lógica implementada pela descrição VHDL.

A simulação funcional é uma estratégia que inicia por gerar sinais e dados necessários para estimular o dispositivo sob teste, de forma que este possa executar as operações modeladas e compiladas com sucesso nas etapas anteriores do projeto.

A estratégia de validação funcional do dispositivo é baseada no uso de *testbenches*, conforme apresentando no diagrama de blocos ilustrada na Figura 1.14.

Um *testbench* é uma bancada de testes virtual, implementada como uma descrição também em VHDL, que por sua vez contém uma instância VHDL do dispositivo a testar. Essa estrutura é apresentada na Figura 1.15, ilustrando, também, blocos geradores de estímulos, capturadores de saídas e formatadores de resultados.

Conforme a Figura 1.15, um *testbench* é um sistema autônomo, que descreve o comportamento do ambiente externo instanciando o módulo sob teste e interagindo com este. Os estímulos são produzidos a partir da especificação de uma sucessão de tarefas previamente preparadas para o módulo em teste funcional. Em geral as tarefas são sequências de dados que serão processados pelo módulo sob teste e devem ser criteriosamente elaboradas para representar as ações que o protótipo, no futuro, deverá realmente processar, conforme especificado no projeto.

Testbenches são projetados para executar testes de um dispositivo de forma automática ou semi-automática. O dispositivo sob teste funcional, em geral, necessita de sinais de relógio (*clock*) e inicialização para o sincronismo e sequenciamento de operações, fornecidos pelo bloco "Gerador",

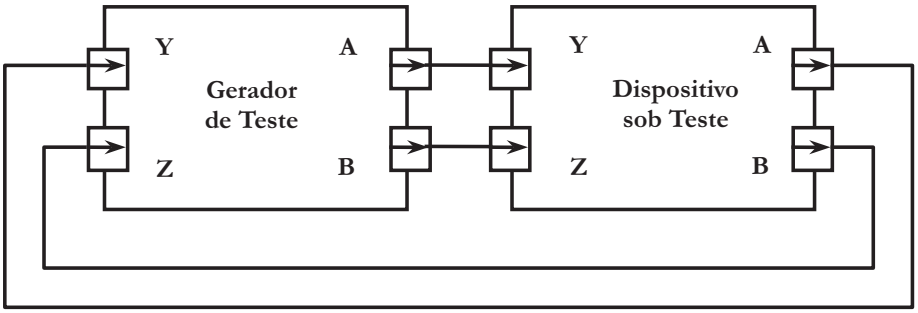


Figura 1.14 - Diagrama de blocos da estratégia para simulação

conforme a Figura 1.15. As partes referentes aos "Dados de Entrada e Comandos" e "Resultados para análise posterior" representam, em geral, informação armazenada em arquivos texto. Essa estrutura permite construir *testbenches* parametrizáveis. Os resultados do processamento, depois de adaptados pelo "Capturador" de sinais, são armazenados em arquivos para análise. Os blocos "Gerador" e

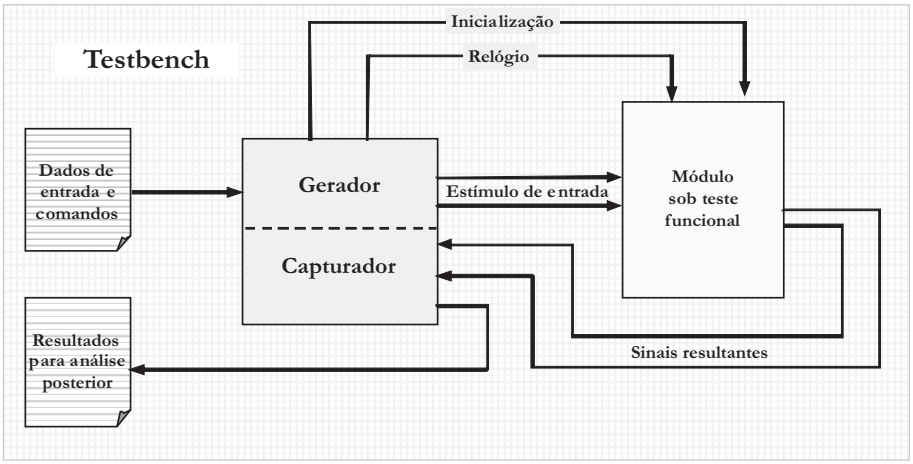


Figura 1.15 - Diagrama de blocos do método *testbench*

"Capturador", para serem executados, também necessitam de estímulos de entrada adequados para se obter as respostas, em conformidade com o projeto, na saída dos módulos em teste, como exemplificado na Figura 1.15.

A Figura 1.16 ilustra o projeto da entidade *testbench1* para simulação funcional da porta lógica AND de duas entradas.

A entidade *testbench1* é implementada como uma descrição em VHDL, que por sua vez contém uma instância VHDL do dispositivo a testar, no caso o componente AND2.

Para iniciar o desenvolvimento de um *testbench*, é necessário criar um arquivo fonte para conter sua descrição, conforme ilustrado pela Figura 1.17. Para tanto, clique com o botão direito do *mouse*, apontando no interior da janela "Workspace" para obter o menu "<Add to Project→New File...>" e, dessa forma, abrir a janela "Create Project File".

Na janela "Create Project File", apresentada na Figura 1.18, é então digitado o nome do arquivo que irá conter a descrição da entidade *testbhech1* e o respectivo componente instanciado - entidade porta lógica AND de duas entradas. Os demais campos serão assumidos por definição inicial do ModelSim e, dessa forma, encerra-se a criação do arquivo fonte para o *testbench* confirmando em "OK".

Na próxima tela, ilustrada na Figura 1.19, observa-se que o novo arquivo fonte "tb_end2" é

adicionado ao projeto com a extensão "vhd" e seu estado (*status*) apresenta um caractere "?", dado que ele está vazio, não contendo uma descrição em VHDL.

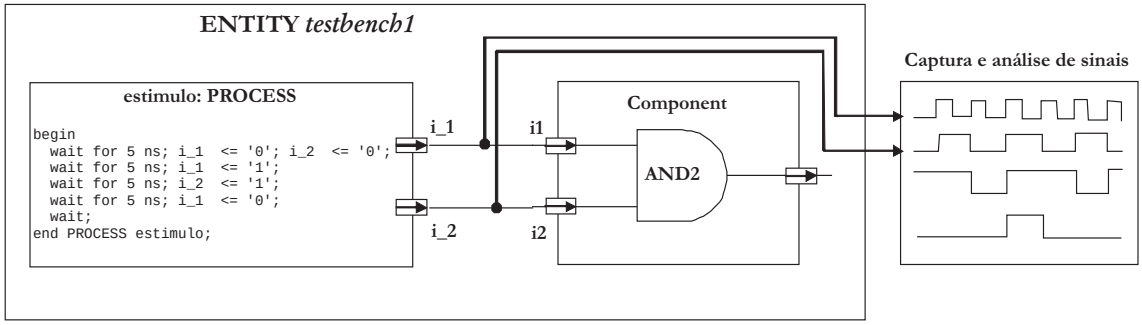


Figura 1.16 - Diagrama de blocos do *testbench1*

Definida a arquitetura da entidade *testbench1*, inicia-se a sua descrição, sempre a partir de um cabeçalho sob forma de comentário para a identificação do objetivo do conteúdo descrito em VHDL, conforme ilustra a tela da janela de projeto da Figura 1.20.

Uma vez modelada a entidade *testbench1*, é necessário realizar uma compilação com sucesso, para se obter a validação de sua descrição VHDL (Figura 1.21).

A descrição completa da entidade *testbench1* é transcrita como segue.

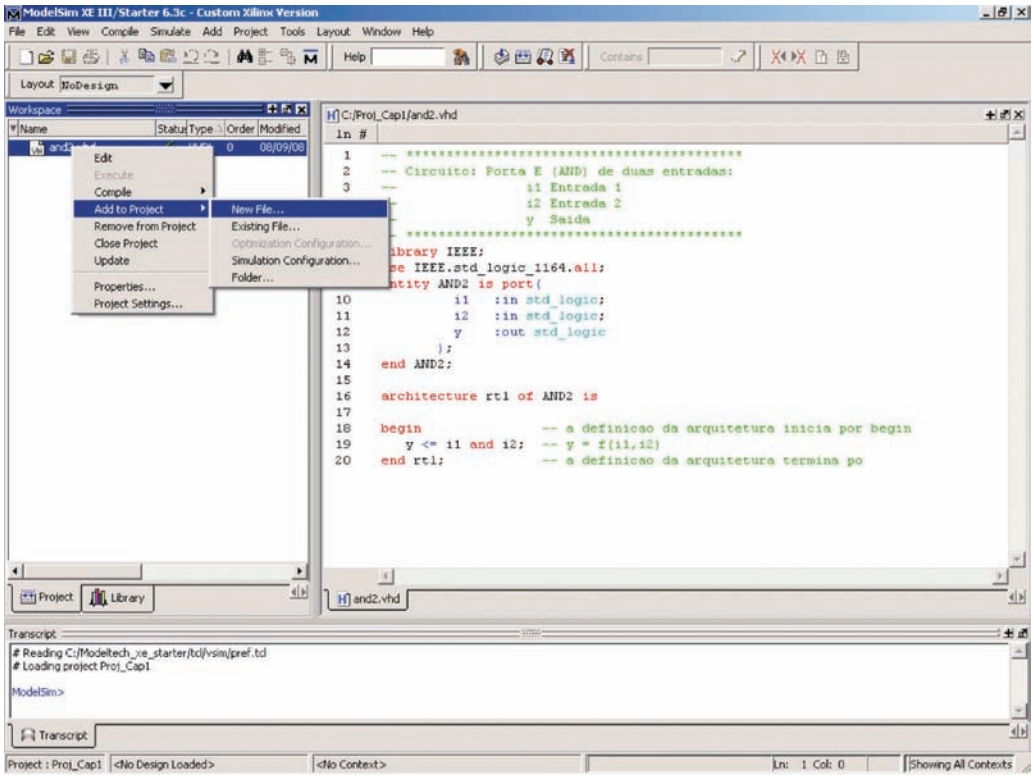


Figura 1.17 - Menu para criação de novo arquivo

Não há limite para o número de instâncias a serem determinadas, basta identificar cada nova instância por um rótulo diferente. Dessa forma, é possível identificar outro componente idêntico a "AND2", por exemplo:

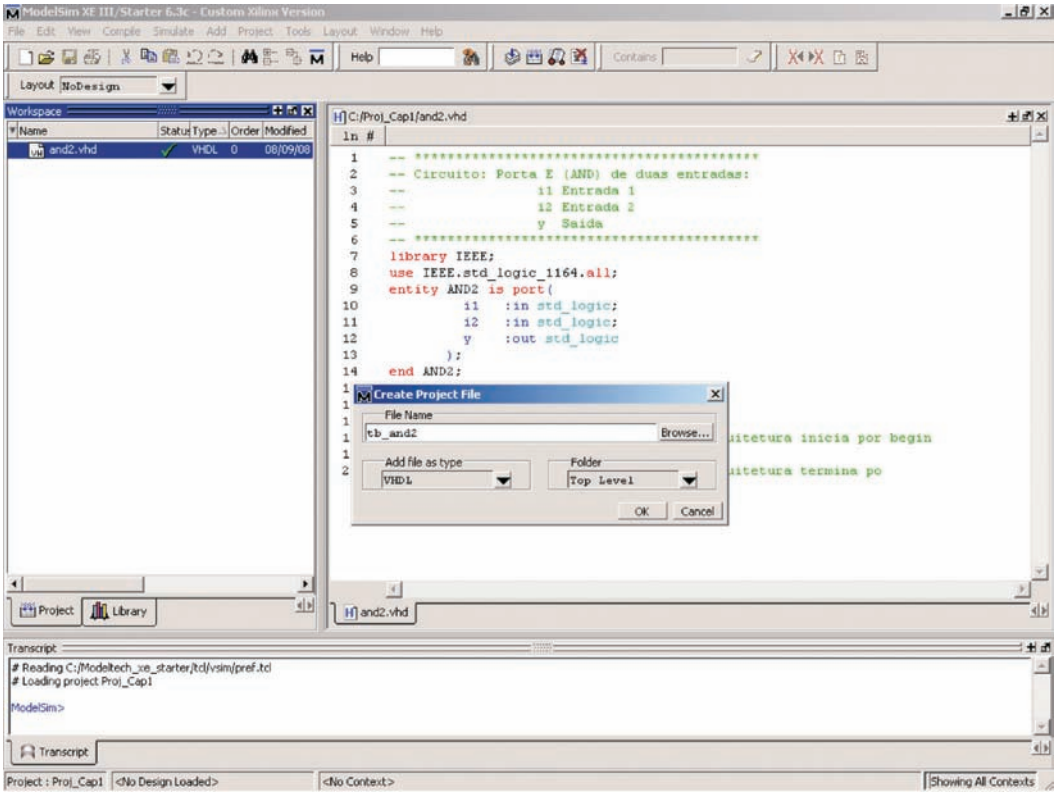


Figura 1.18 - Janela para nomear o arquivo de trabalho do *testbench*

```
"and2: AND2 PORT MAP (i1 =>...);"
```

E assim por diante, quantos forem necessários para o desenvolvimento do projeto.

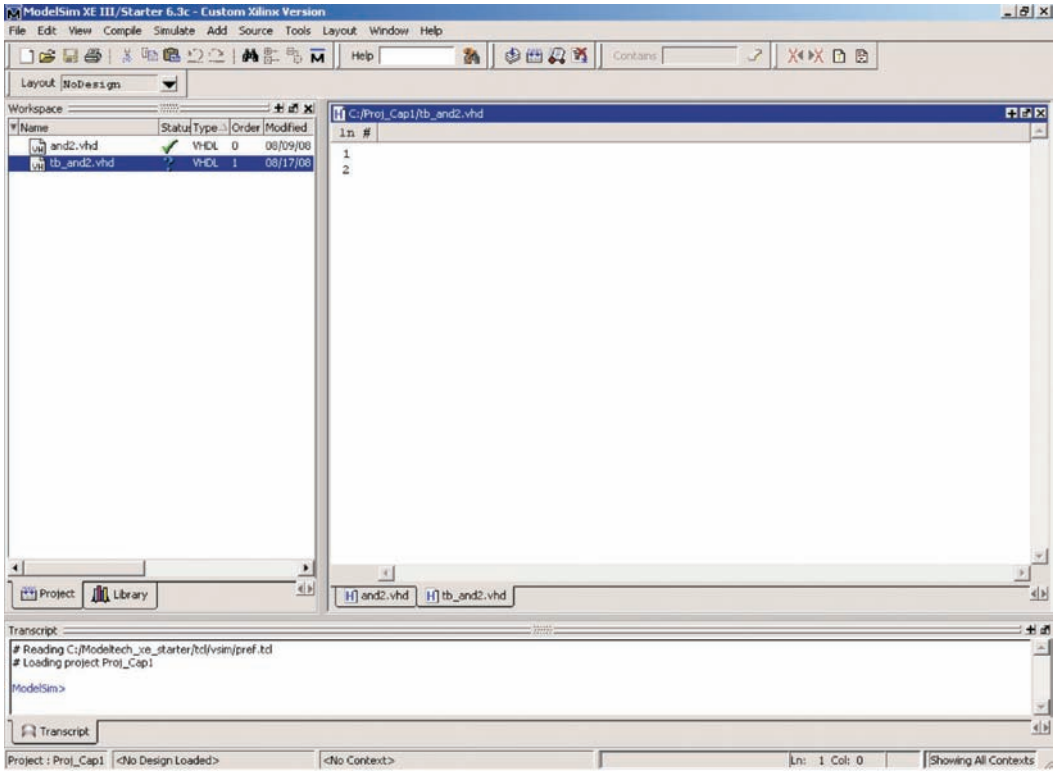


Figura 1.19 - Tela apresentada após criar um novo arquivo fonte

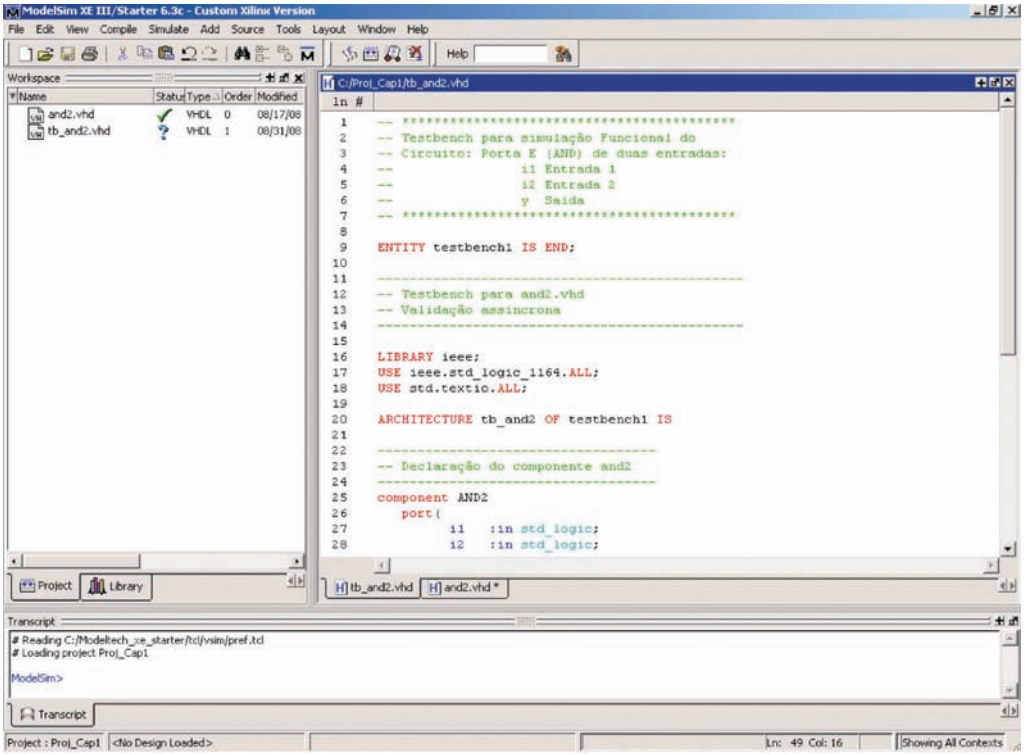


Figura 1.20 - Janela de projeto para o arquivo "tb_and2.vhd"

1.5 Simulação

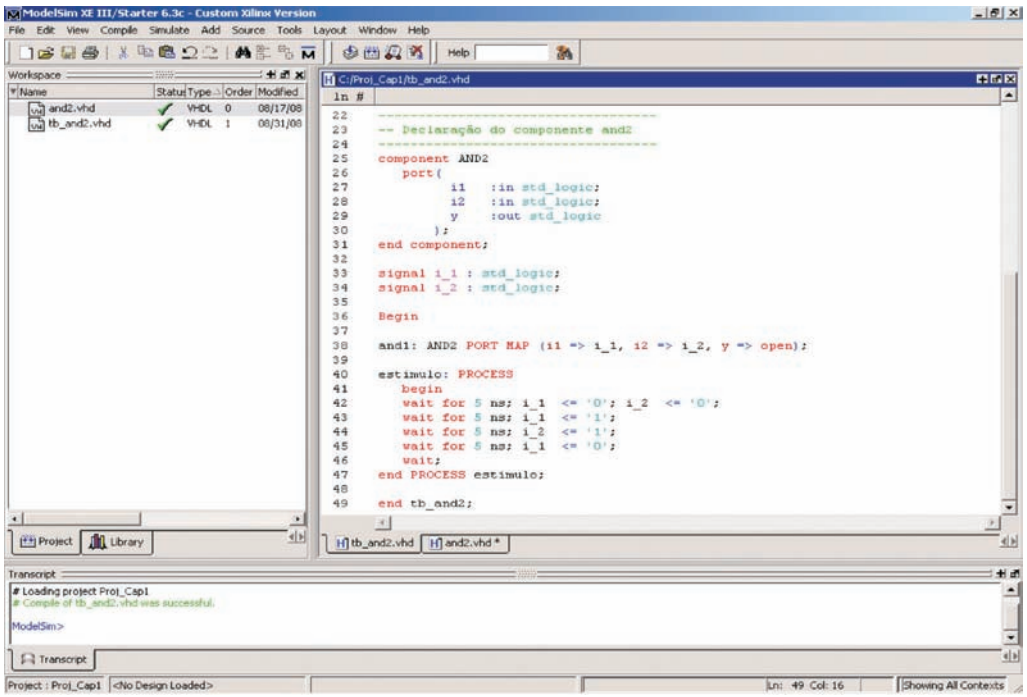


Figura 1.21 - Modelo completo do *testbench1* e compilação com sucesso

```
-- ****************************************
--  Testbench para simulacao Funcional do
--  Circuito: Porta E (AND) de duas entradas:
--                    i1 Entrada 1
--                    i2 Entrada 2
--                    y  Saida
-- ****************************************

ENTITY testbench1 IS END;


-- Testbench para and2.vhd
-- Validacao assincrona

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_and2 OF testbench1 IS

-- Declaracao do componente and2
component AND2
   port(
       i1   :in std_logic;
       i2   :in std_logic;
       y    :out std_logic
         );
end component;

signal i_1 : std_logic;
signal i_2 : std_logic;

Begin

and1: AND2 PORT MAP (i1 => i_1, i2 => i_2, y => open);

estimulo: PROCESS
begin
   wait for 5 ns; i_1  <= '0'; i_2  <= '0';
   wait for 5 ns; i_1  <= '1';
   wait for 5 ns; i_2  <= '1';
   wait for 5 ns; i_1  <= '0';
   wait;
end PROCESS estimulo;

end tb_and2;
```

Cabeçalho contendo uma breve descrição do dispositivo modelado (comentário opcional)

ENTITY (Entidade)
- testbench1 é uma entidade sem pinos de entrada e saída

Package (Pacote)
- constantes e bibliotecas

Declaração do componente and2, referente à sua arquitetura descrita no arquivo and2.vhd

Sinais auxiliares para a interconexão ao processo de estímulo

Instância do componente and2 e interconexão do componente ao processo de estímulo

Implementação do processo de estímulo

Observe a linha 38, na Figura 1.21:

```
"and1: AND2 PORT MAP (i1 => i_1, i2 => i_2, y => open);"
```

Na janela do projeto (Figura 1.21), está descrita uma única instância do componente "AND2" e a respectiva interconexão do componente ao processo de estímulo, que é rotulada por:

```
"and1:"
```

O modelo completo e sua respectiva compilação estão ilustrados na Figura 1.22. Nesta edição da descrição, somente uma instância do componente AND2 e a interconexão do componente ao processo de estímulo foram declaradas na linha anterior à declaração do bloco processo de estímulo.

Observar que na janela "Workspace" (Figura 1.22) é informada a ordem para compilação. Em um projeto, o ModelSim compila os arquivos de forma padrão, na ordem em que foram adicionados ao projeto. Por exemplo, o arquivo "and2.vhd" possui ordem zero, pois foi o primeiro a ser adicionado ao projeto. Os demais arquivos são ordenados sequencialmente.

Existem duas alternativas para a mudança na ordem de compilação:

- selecionar e compilar cada arquivo individualmente, ou
- especificar uma ordem para compilação personalizada.

Para especificar uma ordem para compilação personalizada, selecione "<Compile → Compile Order...>" na barra de menu principal do ModelSim, obtendo a janela "Compiler Order", conforme ilustrado pela Figura 1.23.

A opção "Auto Generate" é utilizada somente em projetos em VHDL. Essa opção determina a ordem correta de compilação dos arquivos do projeto, realiza múltiplos passes ao longo dos arquivos e determina a ordem para compilação a partir do topo. Caso a compilação de um arquivo falhe, devido a

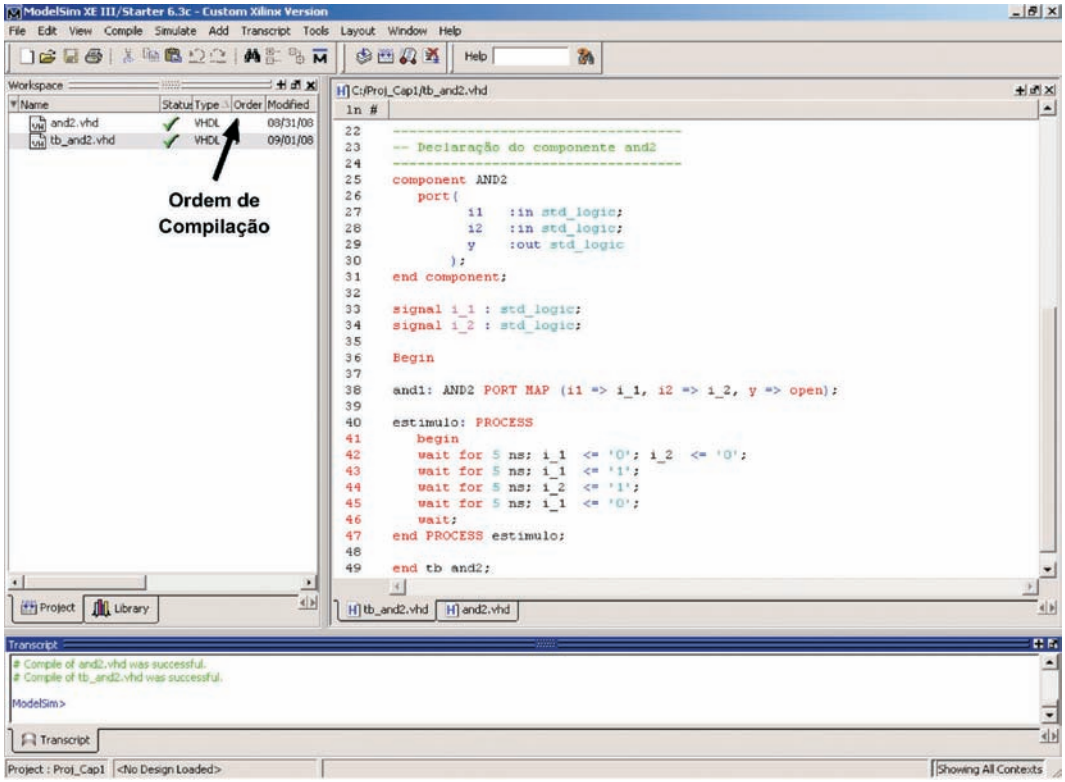


Figura 1.22 - Janela "Workspace" informando a ordem para compilação

dependências relativas aos outros arquivos, é movido para baixo e, em seguida, é compilado novamente após a compilação de todos os arquivos de ordem superior. Esse processo permanece em execução até que todos os arquivos sejam compilados com sucesso, ou até que um arquivo não possa ser compilado por outros motivos que não suas dependências. A ordem observada na guia projeto não é, necessariamente, a ordem na qual os arquivos serão compilados.

Resolvidas todas as dependências para as compilações das entidades AND2 e respectiva "tb_end2", pode-se então iniciar a fase de simulação em "<Simulate→Start Simulation...>" na barra de menus da janela principal, conforme ilustrado na Figura 1.24.

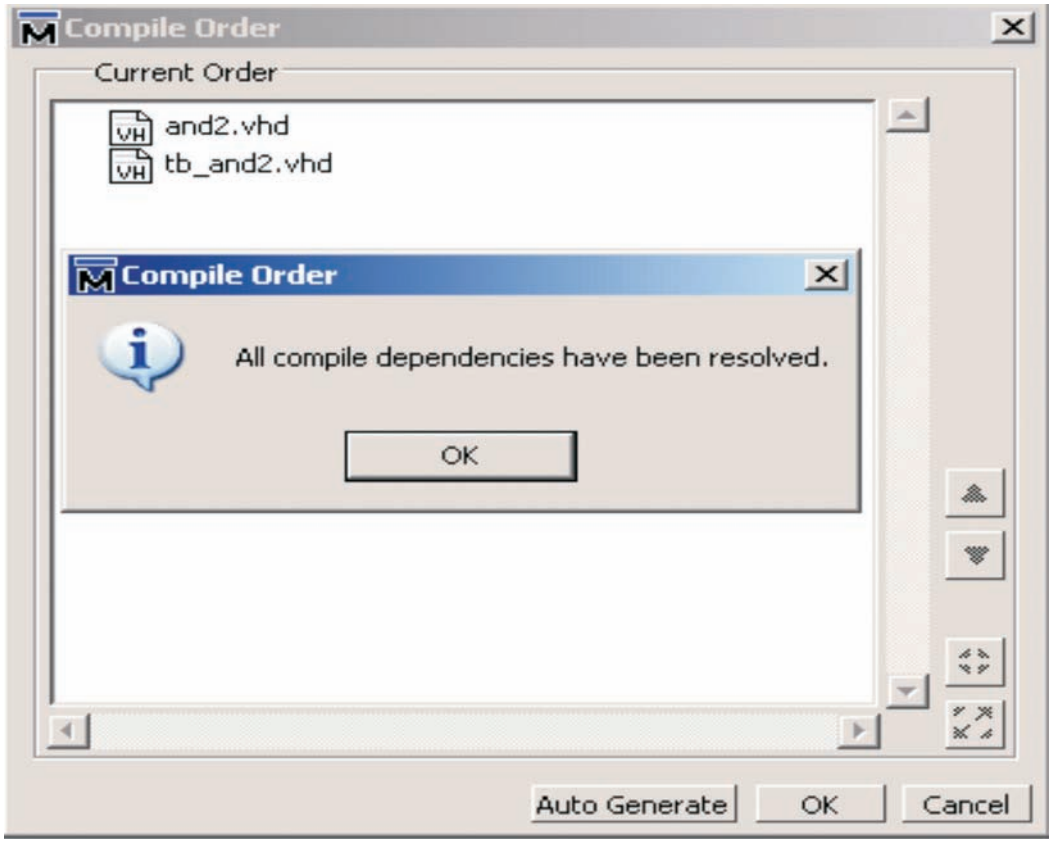


Figura 1.23 - Opção "Auto Generate" ativada

A partir desse comando, uma nova janela denominada "Start Simulation" é aberta, que na sequência é obtida expandindo-se o diretório <Work→ *testbench1*> (Figura 1.25), permitindo a escolha da entidade *testbench1* como alvo da simulação.

Confirmando com "OK" na janela "Start Simulation", o ModelSim determina a entidade *testbench1* como o projeto-alvo para a simulação e, dessa forma, obtém-se no "Workspace" a janela "sim" para a escolha de qual instância dos componentes que poderá ser analisada.

A escolha, neste exemplo, será a instância do componente "and1", conforme ilustrado na Figura 1.26. Neste ponto da fase de simulação é importante observar a janela "Transcript", ilustrada na parte inferior da Figura 1.26, onde são descritas as mensagens de carregamento para o simulador das entidades que compõem o projeto e que, neste caso, ocorreu de forma normal.

As ocorrências de erros, nesta fase de carregamento das entidades que compõem o projeto, não

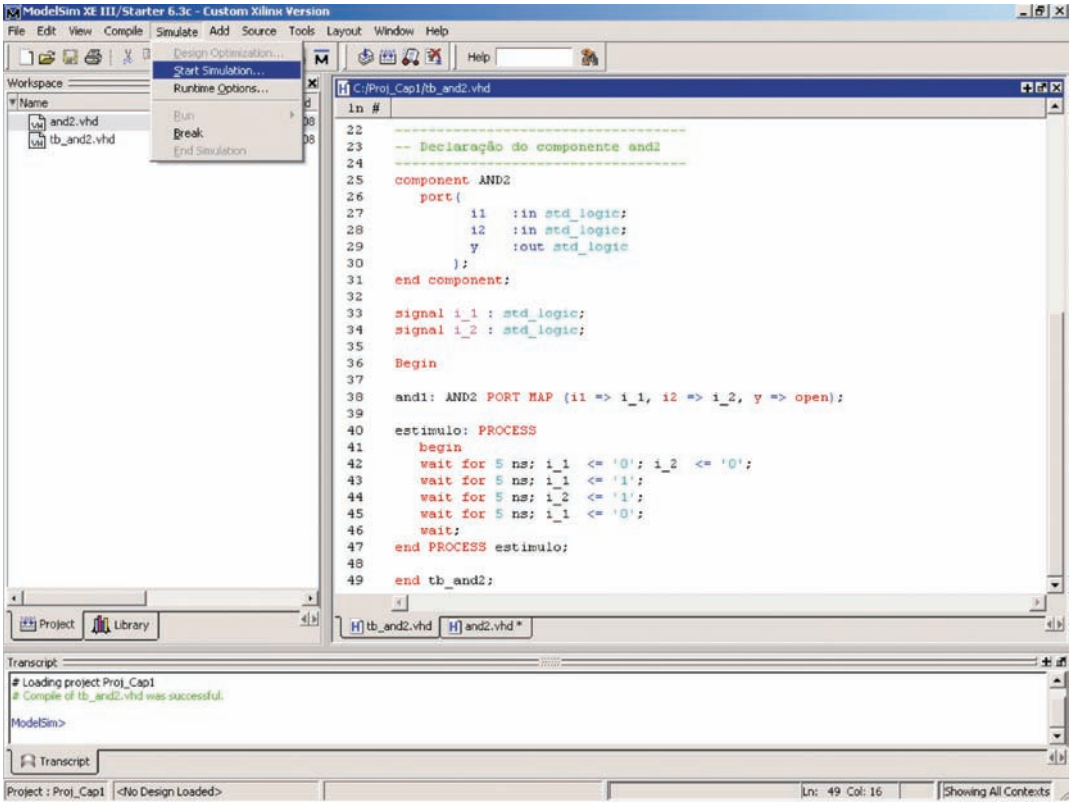


Figura 1.24 - Menu para iniciar a simulação

ocorrem com a mesma frequência com que os erros durante a fase de modelagem. Contudo, quando ocorrem, são difíceis de serem localizados, pois não é resultado da lógica funcional dos diversos dispositivos sob simulação.

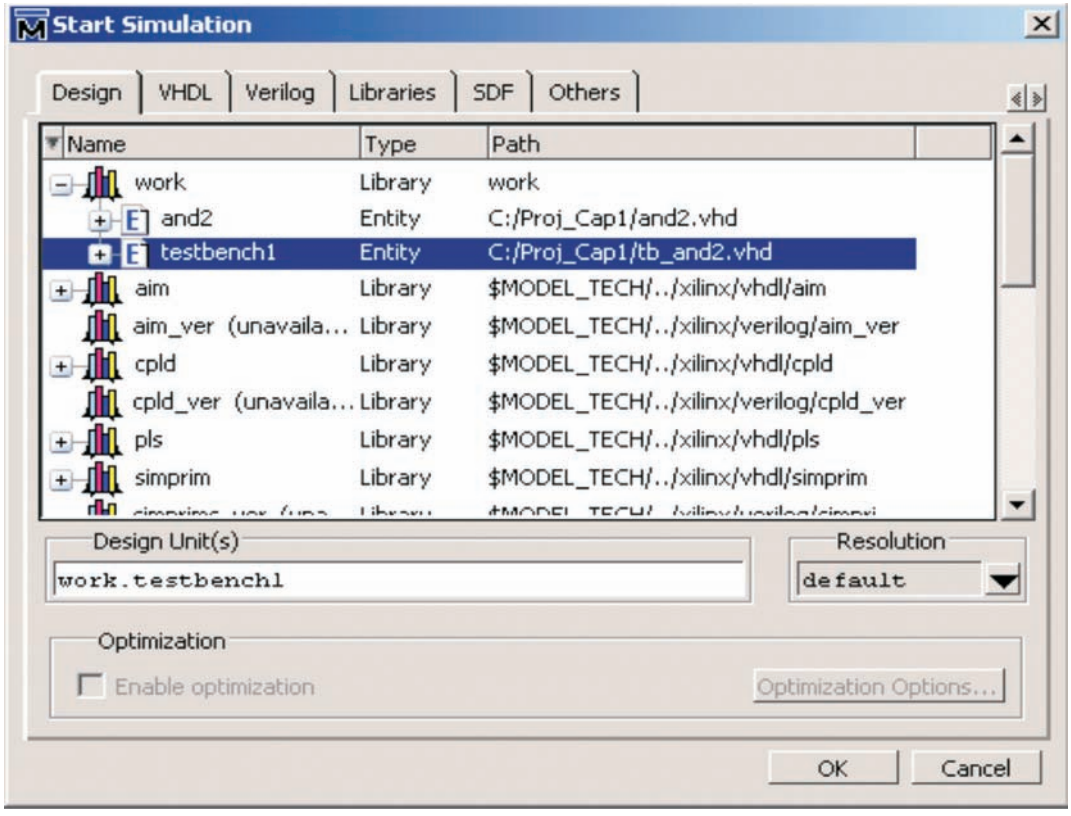


Figura 1.25 - Janela para escolha da entidade-alvo da simulação

A próxima etapa da simulação é a da escolha dos sinais para a captura e análise, conforme ilustrado na Figura 1.27. Nesta mesma figura observa-se que, para se analisar a lógica do componente AND2, é necessário capturar os sinais "i_1" e "i_2" da entrada do componente AND2, bem como o sinal "y" da

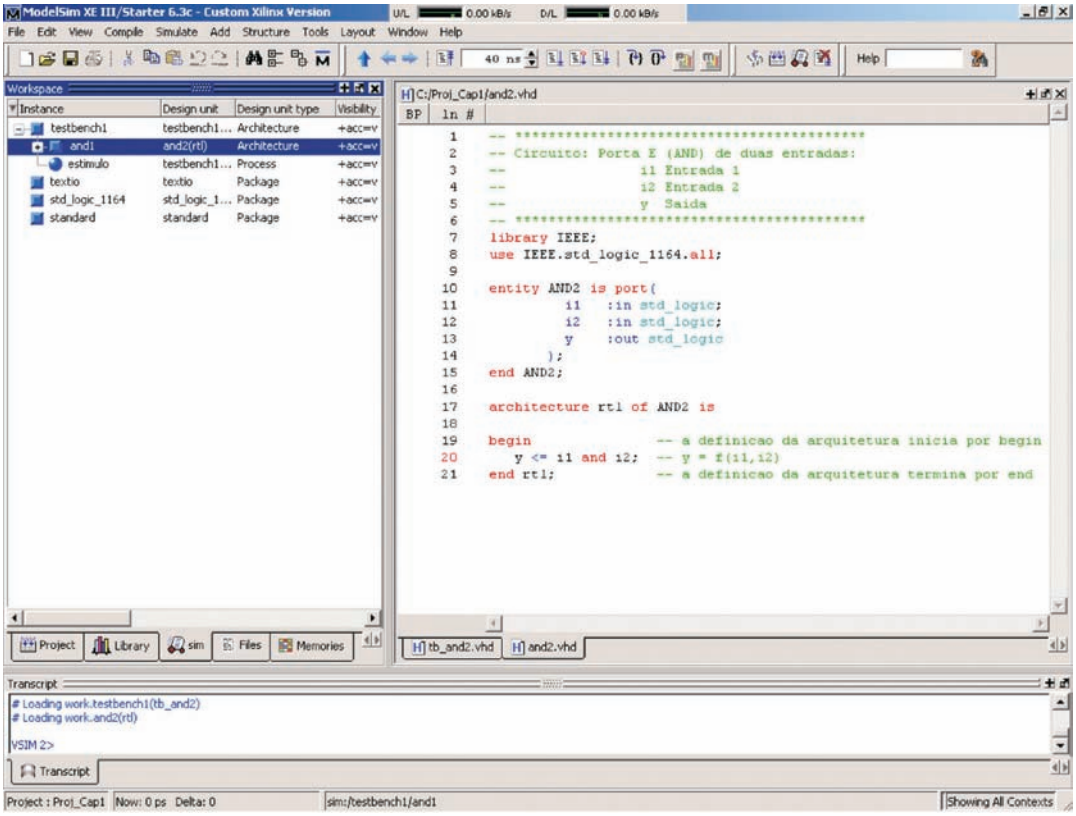


Figura 1.26 - Janela para escolha da entidade-alvo da simulação

saída e analisá-los para validar a operação lógica.

Para análise dos sinais no ModelSim do componente "and1", seleciona-se sua instância na janela "Workspace" (botão esquerdo do *mouse*). Após, aponte para a tarja azul e clique com o botão direito do *mouse* para obter o menu <Add→ Add to Wave>. Novamente, com o botão esquerdo do *mouse,* abre-se a janela "Wave – default", conforme ilustrado na Figura 1.28.

A janela "Wave – default" abre-se na tela principal do ModelSim, como mostra a Figura 1.29. Esta janela pode ser desvinculada da janela principal, bastando clicar sobre ícone ("unlock") na janela "Wave – default" à direita, conforme indicado (Figura 1.29).

Observe na Figura 1.30, a janela "Wave – default" está sobre a janela principal e apresenta uma barra de menu, semelhante à da janela principal, onde se deve definir qual o tempo de execução da simulação ("Run Length"). Neste caso, 100 ns é um tempo adequado, visto que os estímulos gerados estão na ordem de 5 ns, totalizando 20 ns.

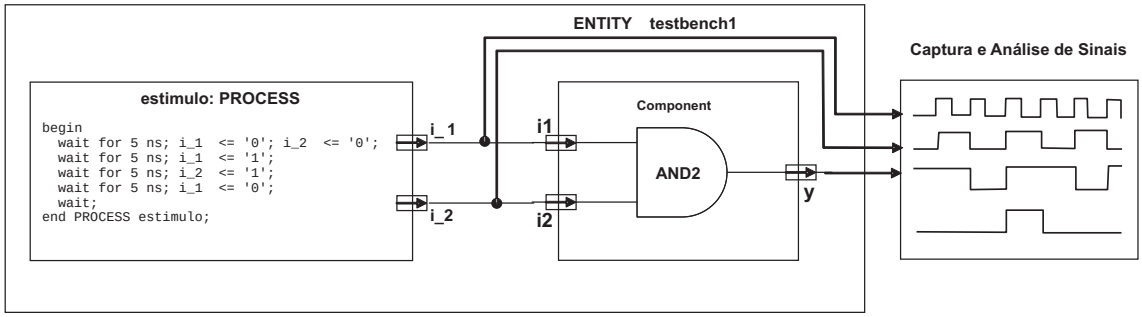


Figura 1.27 - Captura e análise dos sinais

Uma vez determinado o tempo de execução da simulação, aponte sobre o ícone ("run") e execute uma rodada de simulação por um intervalo de tempo previamente ajustado no "Run Length", obtendo as formas de onda dos sinais "i_1" e "i_2" da entrada da AND2, bem como o sinal "y" na saída, conforme ilustrado na Figura 1.31 .

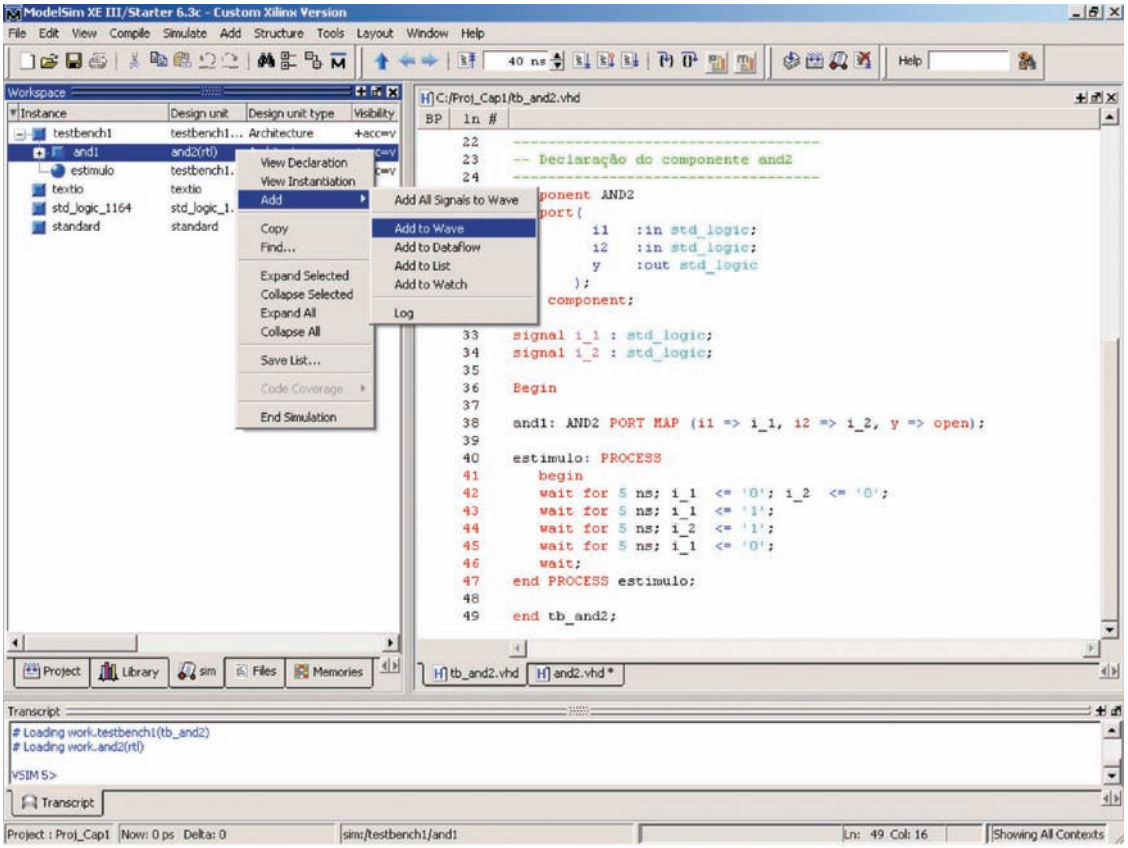


Figura 1.28 - Adicionando os sinais para análise

Na Figura 1.32, detalhe da Figura 1.31, há um detalhamento da simulação da porta lógica utilizada como exemplo neste tutorial.

Após 5 ns (linha 42 do código VHDL - Figura 1.31) do início da simulação (t1), as entradas "i1" e "i2"

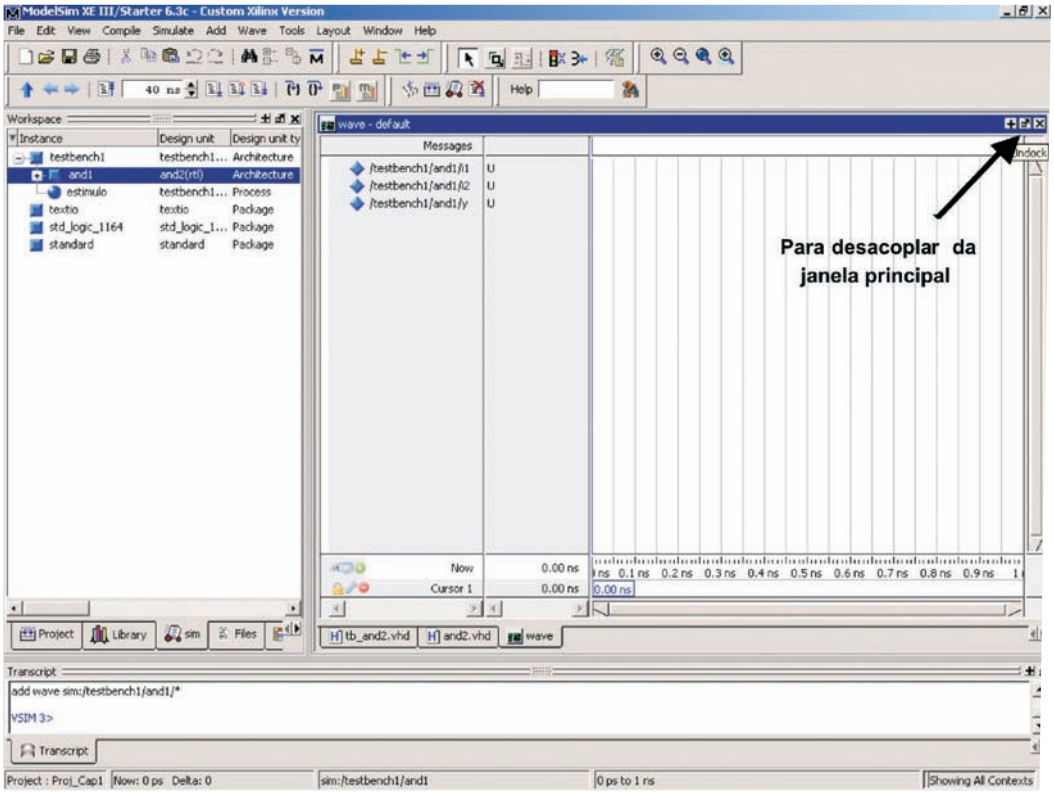


Figura 1.29 - Desvinculando a janela "Wave - default" da janela principal

são colocadas em nível lógico "0", resultando na saída do circuito o nível lógico "0" (0 AND 0 = 0). Em "t2" (linha 43 do código VHDL), a entrada "i1" é colocada em nível lógico 1, logo a saída "y" permanece em "0" (1 AND 0 = 0). No instante "t3", tem-se "i1=1" e "i2=1", resultando em "y="1" (1 AND 1 = 1)" (linha 44 do código VHDL). A partir de "t4", a saída "y" permanece em nível lógico "0" (0 AND 1 = 0),

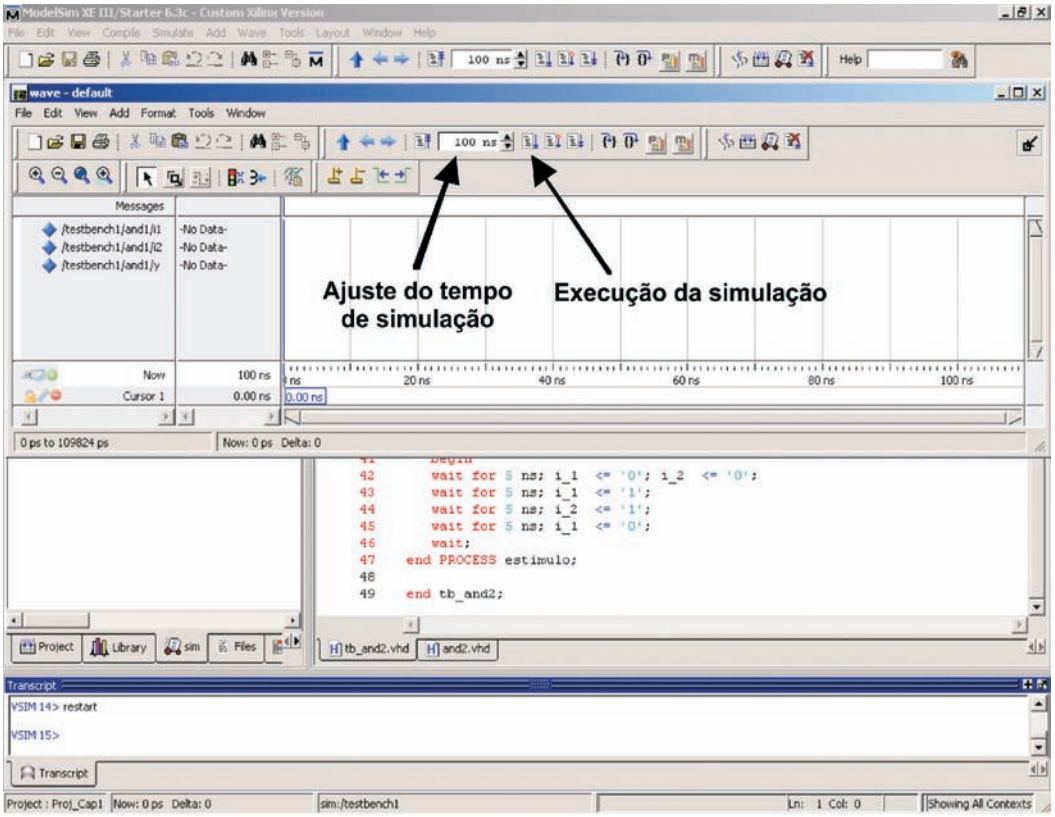


Figura 1.30 - Determinando o tempo de simulação

dado que "i1" passa a ter o nível lógico "0" (linha 45 do código VHDL - Figura 1.31).

Após os resultados esperados terem sido validados pela simulação funcional, chega-se ao final do fluxo básico de projetos em VHDL apresentado na Figura 1.3 deste capítulo. Caso os resultados obtidos através da simulação não tivessem sido alcançados, haveria necessidade de modificação no código fonte

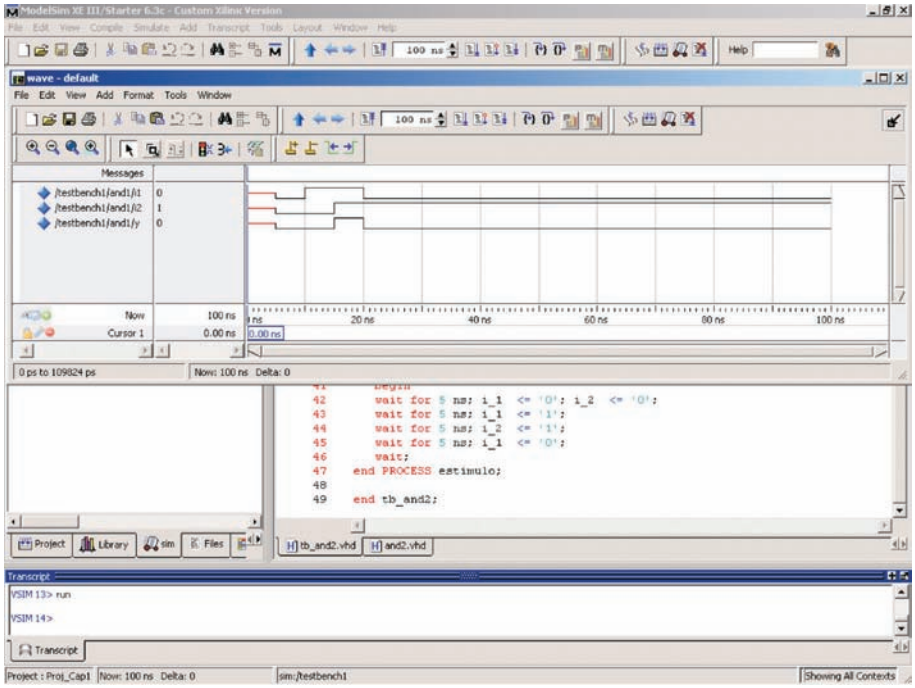


Figura 1.31 - Janela "Wave - default" após a execução de uma rodada de simulação

e repetição da simulação até atingir os resultados corretos.

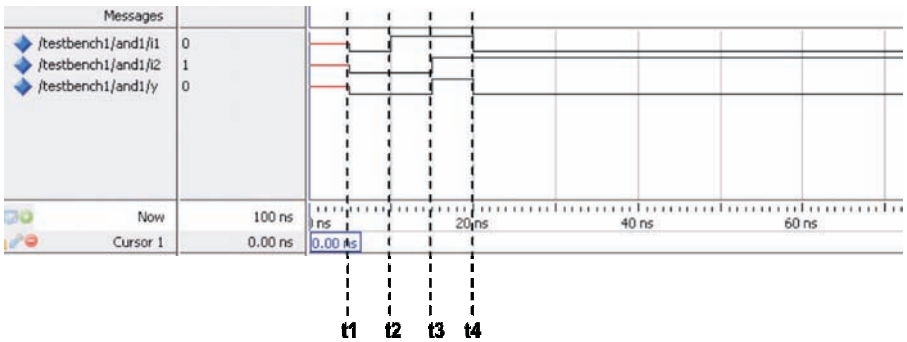


Figura 1.32 - Detalhamento maior da simulação da descrição

2.1 Introdução

Como mencionado na introdução, este livro apresenta uma abordagem orientada para o projeto, o

desenvolvimento e a implementação de dispositivos digitais aliados à tarefa de ensinar VHDL. A integração entre VHDL e projeto digital é explicada através de uma série de exemplos de concepção simples e prática.

Os exemplos de projetos apresentados neste capítulo evoluem gradativamente em complexidade e funcionalidade. Ao final, o estudante terá uma visão das principais características de um modelo completo em VHDL, envolvendo os conceitos necessários para unir todos os dispositivos em um único

2 Modelando e Simulando Circuitos

sistema digital que possa ser simulado e validado como um todo.

Inicialmente o conjunto de exemplos de dispositivos digitais que serão descritos em VHDL são circuitos digitais combinacionais. Gradativamente serão introduzidos os dispositivos de arquitetura mista, isto é, combinacionais e sequenciais.

O conjunto de exemplos de circuitos digitais é formado por:

1. Somador completo de 1 bit;
2. Somador genérico de n bits;
3. Multiplexador e Demultiplexador;
4. Decodificador e Codificador;
5. Unidade Lógica e Aritmética.

Na modelagem de um sistema digital em VHDL, uma entidade pode ser composta por diferentes dispositivos, denominados componentes, que previamente possuem uma entidade que descreve sua própria arquitetura. Desta forma, a entidade pode assumir diferentes composições de arquiteturas que, em conjunto, podem operar como um único sistema. A Figura 2.1 representa a composição uma entidade de forma geral.

Um sistema descrito em VHDL é composto por uma entidade (*entity*) e por uma ou mais arquiteturas (*architecture*). A entidade especifica os sinais de entrada e saída (I/O) de um componente, algumas vezes referidos como seus pinos.

A definição de entradas/saídas fornece as conexões para um dispositivo se comunicar externamente. Um "port" é uma declaração que define os nomes, os tipos, as direções e os possíveis valores-padrão para os sinais de entrada /saída da interface de um componente.

Uma arquitetura descreve o funcionamento de uma entidade e pode ser classificada em três tipos. O primeiro tipo de arquitetura, contendo apenas declarações simultâneas, é comumente referido como uma descrição *dataflow* (lado esquerdo da Figura 2.1). São declarações concorrentes executadas em

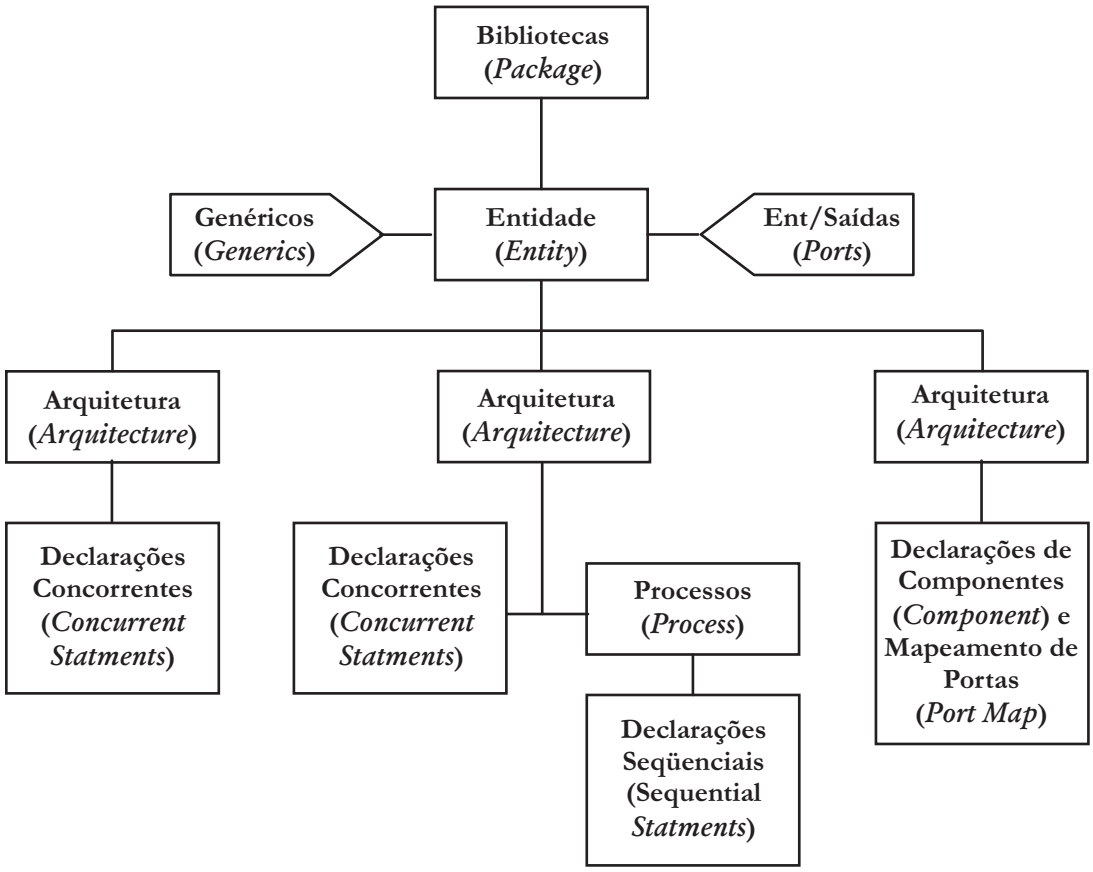


Figura 2.1 - Diversos dispositivos em um único sistema

paralelo (circuito puramente combinacional). Essas declarações podem ser posicionadas em qualquer ordem dentro da arquitetura.

O segundo tipo de arquitetura é a descrição comportamental em que as características funcionais e temporais (*timing*) são descritas utilizando VHDL concorrente, declarações e processos. O processo é uma afirmação de uma arquitetura concorrente. Todas as declarações contidas dentro de um processo são executadas em uma ordem sequencial, ficando suspensas até a execução da declaração anterior.

O terceiro tipo descreve a estrutura de uma entidade (lado direito da Figura 2.1) em termos de subcomponentes pré-definidos e suas interconexões. Este tipo se denomina descrição estrutural. Um elemento-chave de uma arquitetura VHDL estrutural é o componente pré-definido, que pode ser um módulo qualquer, podendo ser especificado de qualquer uma das formas de descrição, *dataflow*, comportamental ou estrutural. Todos os componentes usados na descrição estrutural de um modelo VHDL devem ter sido criados previamente.

As bibliotecas (*package*) são utilizadas para proporcionar uma coleção de declarações comuns, constantes, e/ou subprogramas com entidades e arquiteturas. Já a declaração para genéricos (*generics*) fornece um método estático para comunicar informações acerca de uma arquitetura para o ambiente externo. Elas são passadas através da construção da entidade.

2.2 Desenvolvendo dispositivos combinacionais

Neste capítulo serão detalhadas as construções utilizadas nos exemplos apresentados em código VHDL. Elas irão ajudar a ilustrar aspectos fundamentais quanto à estrutura global de código. Cada exemplo é seguido pelos comentários explicativos e pela simulação funcional com a análise dos resultados obtidos. Para a simulação funcional dos códigos, serão utilizados *testbenches*.

2.2.1 Somador completo de 1 Bit

Um circuito somador binário completo está representado na Figura 2.2. No exemplo, "a" e "b" representam os bits de entrada a serem adicionados, cin (*carry in*) é o bit de "vem-um" de entrada, "s" é o bit resultante da adição e *cout* (*carry out*) o bit de "vai-um" resultante da soma, caso haja. Como apresentado na tabela de verdade, s resulta da adição aritmética de "a", "b" e "cin" e, paralelamente, *cout* resulta o "vai um" (*carry out*).

Duas implementações possíveis para o circuito somador completo ilustrado na Figura 2.2 são apresentadas na Figura 2.3. Ambos os circuitos representam um somador completo e suas funções lógicas são equivalentes.

O primeiro circuito representa o bloco somador básico, obtido das equações "c_{out} = a.b+ a.c_{in}+b.c_{in}" e "s = a xor b xor c_{in}", advindas da tabela verdade e adequadamente minimizadas. Contudo o segundo circuito apresenta-se mais simplificado. Neste caso, é utilizada a técnica de modelagem do somador completo partindo composição de dois semi-somadores.

Neste exemplo, é apresentada a descrição de uma entidade (*entity*), com seus respectivos pinos (*port*), e

Entradas			Saídas	
a	b	c_{in}	s	c_{in}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Figura 2.2 - Somador completo: bloco diagrama e tabela verdade

de uma arquitetura (*architecture*), que descreve como o circuito implementa a função de um somador completo. O modelo completo para descrição de um somador completo pode ser:

Observe que, ao final da descrição, estão comentadas algumas linhas que descrevem uma segunda

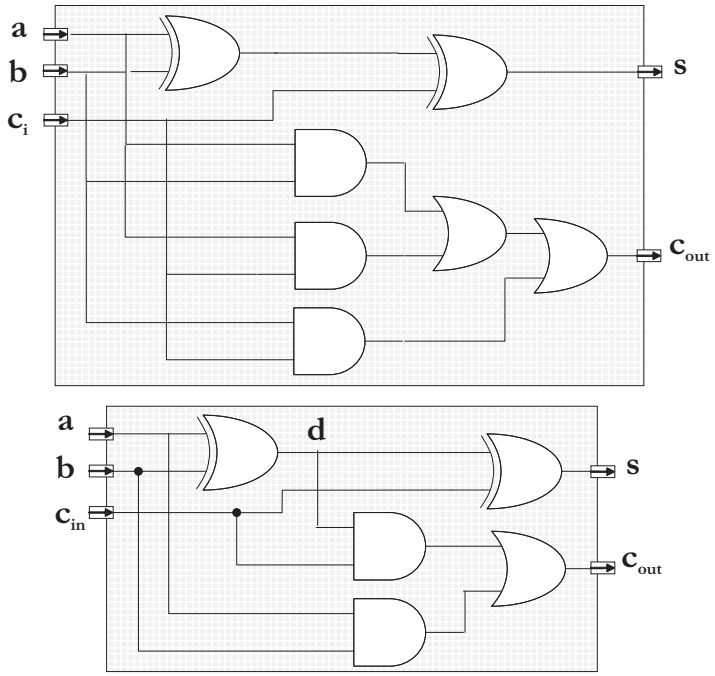


Figura 2.3 - Exemplos de circuitos possíveis para o somador completo

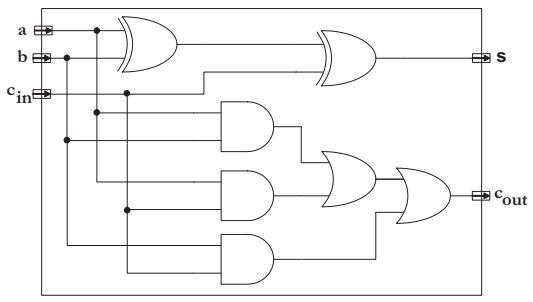
arquitetura para o somador completo, obtido das equações "c_{out} = a.b+d.c_{in}" e "s = d xor c_{in}".

O circuito final obtido a partir do código VHDL deve sempre ser verificado ainda no nível do projeto. Para testar, utiliza-se o simulador que executa a unidade do somador completo, utilizando-se da estratégia de *testbench*.

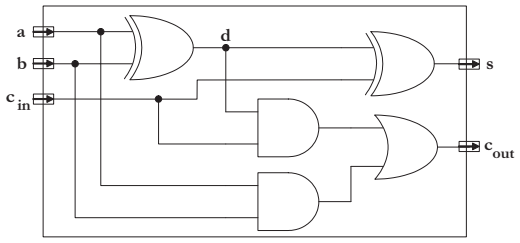
A seguinte descrição em VHDL, cuja entidade se denomina *testbench2*, é um exemplo de *testbench* para simulação funcional do somador completo.

Neste *testbench* observa-se a utilização de três sinais denominados "i_1", "i_2" e "i_3", que funcionam

```
-- ****************************************
--  Circuito: Somador completo:(soma_1bit.vhd)
--         a Entrada 1
--         b Entrada 2
--         s Saída = a + b
--         cin   vem um
--         cout  vai um
-- ****************************************
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY soma_1bit IS PORT (
              a    : in std_logic;
              b    : in std_logic;
              cin  : in std_logic;
              s    : out std_logic;
              cout : out std_logic);
END soma_1bit;
-------------------------------------
```



```
ARCHITECTURE dataflow OF soma_1bit IS
BEGIN
s <= a XOR b XOR cin;
cout <= (a AND b) OR (a AND cin) OR (b AND cin);
END dataflow.
-------------------------------------
--ARCHITECTURE dataflow OF soma_1bit IS
--
--Signal d: std.standard.bit;
--
--BEGIN
--d <= a XOR b;
--s <= d XOR cin;
--cout <= (a AND b) OR (d AND cin);
--
--END dataflow;
```



como fios que interligam o componente "soma_1bit" com o processo gerador de estímulos da entidade *testbench2*.

Um processo (*process*) é uma seção sequencial de código VHDL. É caracterizado pela presença de cláusulas tais como: IF; WAIT; CASE; LOOP e lista de sensibilidade (*sensitivity list*). Um processo deve estar no código principal e é executado sempre que um sinal da lista de sensibilidade mudar seu estado lógico.

Em VHDL, os sinais são utilizados parar interligar diferentes módulos de um circuito e também podem funcionar como estruturas internas de armazenamento de dados. Podem ser compartilhados por

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: Somador completo: (soma_1bit.vhd)
--            a Entrada 1
--            b Entrada 2
--            s Saida = a + b
--            cin   vem um
--            cout  vai um
-- ****************************************
ENTITY testbench2 IS END;
-----------------------------------------------
-- Testbench para soma_1bit.vhd
-- Validacao assincrona
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_soma_1bt OF testbench2 IS
-------------------------------------
-- Declaracao de componentes e sinais
-------------------------------------
component soma_1bit
         Port(a    : in std_logic;
              b    : in std_logic;
              cin  : in std_logic;
              s    : out std_logic;
              cout : out std_logic);
end component;
```

```
signal i_1 : std_logic;
signal i_2 : std_logic;
signal i_3 : std_logic;

Begin

soma1: soma_1bit PORT MAP (a => i_1, b => i_2, cin =>i_3,
       s => open, cout => open);

estimulo: PROCESS
   begin
   i_1  <= '0'; i_2  <= '0'; i_3  <= '0';
   wait for 5 ns; i_1  <= '1';
   wait for 5 ns; i_2  <= '1';
   wait for 5 ns; i_1  <= '0';
   wait for 5 ns; i_3  <= '1';
   wait for 5 ns; i_1  <= '1';
   wait for 5 ns; i_2  <= '1';
   wait for 5 ns; i_1  <= '0';
   wait;
end PROCESS estimulo;
end tb_soma_1bt.
```

vários processos e têm seu valor alterado apenas no final de um processo. A atribuição a sinais é feita através do operador "<=".

Como observado na Janela "Wave – default" (Figura 2.4) com os resultados da simulação, os sinais de estímulo (i_1, i_2 e i_3) gerados pelo processo do *testbench* são transferidos para as entradas "a", "b" e "c_{in}", respectivamente.

Na Figura 2.5, há um detalhamento da simulação do somador completo de 1 bit. Após 5 ns (linha 43 do código VHDL) do início da simulação (t1), a entrada a é colocada em nível lógico "1", resultando na saída (s) do circuito somador o nível lógico "1" (1 + 0 = 1). Em "t2" (linha 44 do código VHDL), a entrada "b" é colocada em nível lógico "1", logo a saída s apresenta o nível lógico "0" (1 + 1 = 0) e "vai um", observe que a saída "vai um" (*cout*) apresenta o nível lógico "1" caracterizando um *carry out*. No instante "t3", tem-se "a = 0" e "b = 1", resultando em "s = "1" (0 + 1 = 1)" (linha 45 do código VHDL) e observa-se que a saída "vai-um" (*cout*) apresenta o nível lógico "0".

Até o final de "t3", a entrada "vem-um" (c_{in}) do somador permaneceu com o nível lógico "0", não

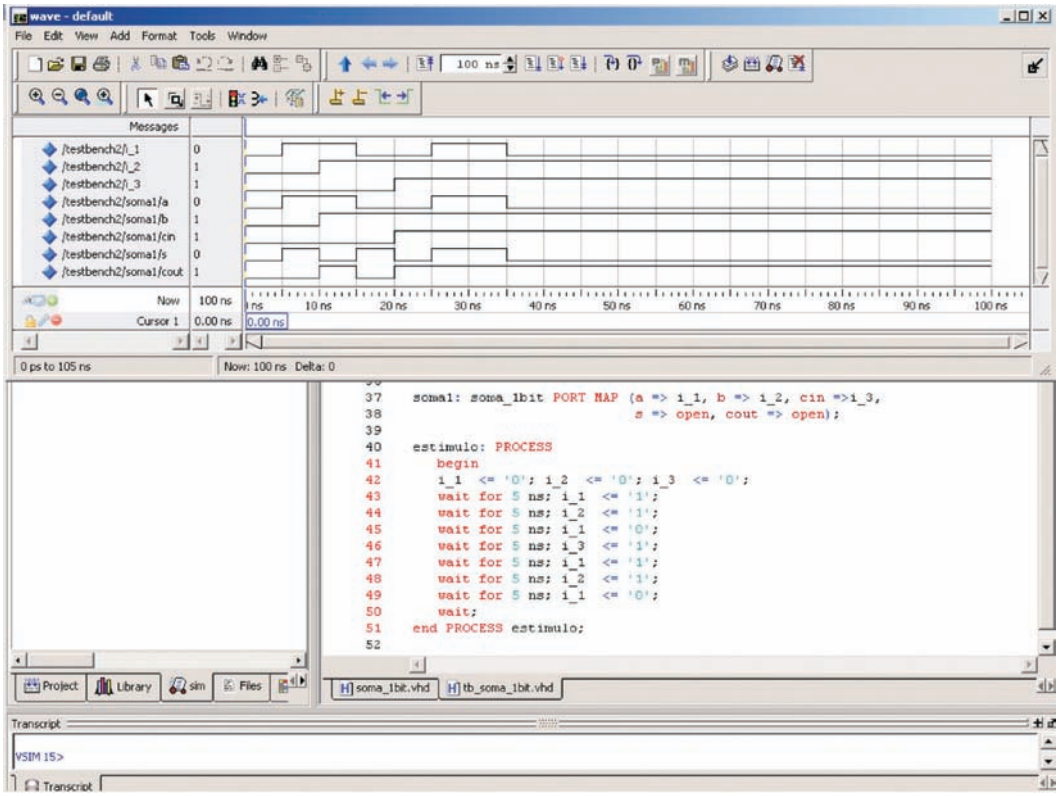


Figura 2.4 - Janela "Wave - default" com os resultados da simulação

contribuindo para a adição das entradas "$(a + b + c_{in})$". A partir de "t4" (linha 46 do código VHDL), a entrada "vem um" (c_{in}) é colocada em "1", resultando na saída (s) o nível "0", pois é a soma "$0 + 1 + 1 = 0$ $(a + b + c_{in})$".

A partir de 20 ns de simulação ($t > t4$), já se pode observar que o somador completo de 1 bit pode ser validado funcionalmente, isso significa que o comportamento do modelo em VHDL e de sua simulação deve ser equivalente à lógica funcional. Contudo essa validação não leva em conta as características

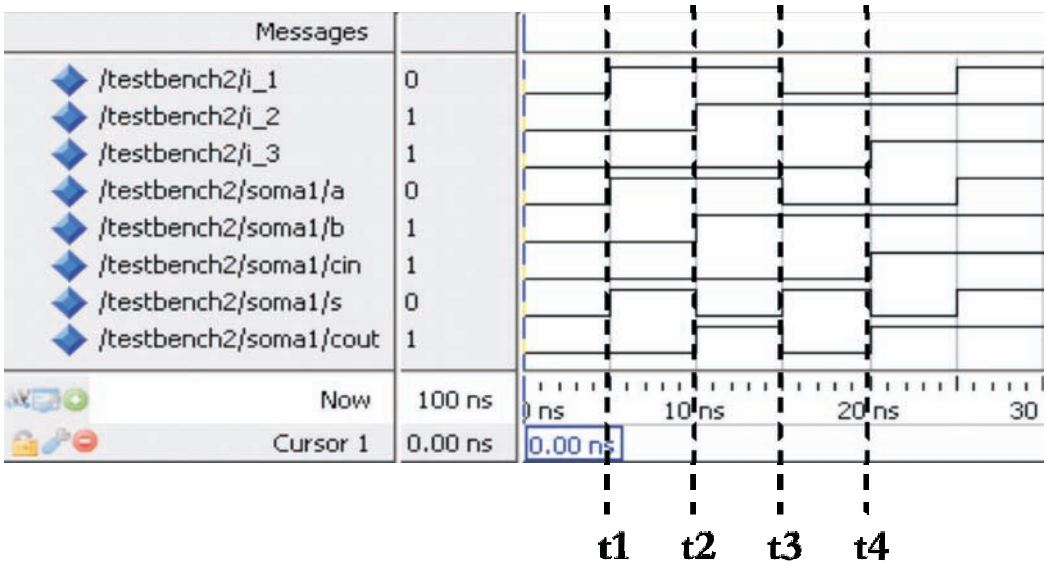


Figura 2.5 - Detalhamento dos resultados da simulação

temporais do modelo físico.

2.2.2 Somador genérico de n de bits

O método estático para comunicar informações de uma arquitetura para o ambiente externo é uma das ferramentas que permite a construção de circuitos de forma automática. Partindo-se de um componente básico, é possível gerar um componente mais complexo, desde que tenha uma estrutura repetitiva.

O próximo exemplo apresenta a descrição de um somador completo de 4 bits, construído de forma exaustiva (força bruta). A interconexão dos terminais dos somadores entre si, bem como com os pinos de entrada e saída da entidade, é realizada de forma manual, um a um. Desta forma, introduz-se o conceito de mapeamento das conexões de entrada e saída (*port map*) e barramentos (*bus*) internos de interconexão, ilustrados na Figura 2.6

Observando-se atentamente a Figura 2.6, verifica-se que as interconexões são realizadas com sinais auxiliares (*signal* Cry0, Cry1, Cry2, Cry3, Cry4) responsáveis pela transferência do *carry out* de um somador ao *carry in* do próximo e assim por diante. Os sinais "Cry0" e "Cry4" são os responsáveis pela interconexão dos *carry in* e *carry out* do primeiro e último somadores aos respectivos terminais (pinos) da entidade somador de 4 bits (*entity* soma4b), assim como os barramentos internos (*signal* busA, busB, busS) aos conjuntos de terminais de entrada e saída da entidade "soma_4b".

A descrição VHDL apresenta como o circuito implementa a função de um somador de 4 bits

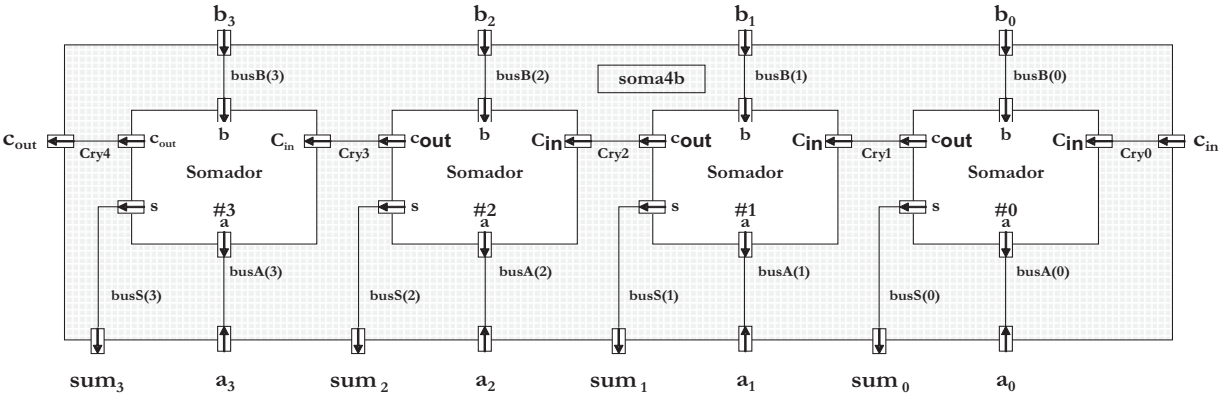


Figura 2.6 - Somador de 4 bits implementado a partir de 4 somadores de 1 bit

completo por intermédio de interconexões realizadas de forma exaustiva e introduz o tipo de arquitetura estrutural, que é descrita como um módulo lógico criado pela conexão de módulos mais simples, que se liga à entidade de um subcomponente de uma das várias arquiteturas alternativas para esse componente. O modelo completo para descrição da porta lógica somador de 4 bits é transcrito como segue.

Na descrição do somador de 4 bits, pode-se observar que é instanciado um somador completo de 1 bit quatro vezes por meio dos rótulos identificadores "som0:", "som1:", "som2:", e "som3:", de forma a representar os somadores "#0", "#1", "#2" e "#3", respectivamente, conforme ilustrado na Figura 2.6.

O circuito final obtido a partir do código VHDL é verificado ainda em nível de projeto. Durante o teste, utiliza-se o simulador para realizar a validação funcional da entidade do soma_4b. Os códigos do somador de 4 bits e do *testbench* (*testbench3*) em VHDL estão descritos como segue.

No processo de estímulo deste *testbench*, foi utilizada a cláusula LOOP. Esta é útil quando uma parte

```
-------------------------------------------
-- somador 4-bit Método forca bruta
--
-- Circuito: Somador de 4bit: soma_4b.vhd)
-- a4 Entrada a de 4 bits
-- b4 Entrada b de 4 bits
-- sum4 Saída de 4 bits  sum4 = a mais b
-- cin  vem um
-- cout vai um
-------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity soma4b is
    port (a4    : in std_logic_vector(3 downto 0);
          b4    : in std_logic_vector(3 downto 0);
          cin   : in std_logic;
          sum4  : out std_logic_vector(3 downto 0);
          cout  : out std_logic);
end soma4b;

--implementação estrutural do somador 4-bit

architecture structural of soma4b is
    component soma_1bit
        port (a    : in std_logic;
              b    : in std_logic;
              cin  : in std_logic;
              s    : out std_logic;
              cout : out std_logic);
    end component;

    signal Cry0, Cry1, Cry2, Cry3, Cry4 :std_logic;
    signal busA, busB, busS:std_logic_vector(3 downto 0);


begin
     busA <= a4;
     busB <= b4;
     cry0 <= cin;
     cout <= cry4;
     sum4 <= busS;

-- Instanciando um somador completo de 1 bit 4 vezes

som0: soma_1bit port map(
                   a => busA(0),
                   b => busB(0),
                   cin => cry0,
                   s => busS(0),
                   cout => cry1
               );

som1: soma_1bit port map(
                   a => busA(1),
                   b => busB(1),
                   cin => cry1,
                   s => busS(1),
                   cout => cry2
               );
```

```
som2: soma_1bit port map(
                    a => busA(2),
                    b => busB(2),
                    cin => cry2,
                    s => busS(2),
                    cout => cry3
                );
som3: soma_1bit port map(
                    a => busA(3),
                    b => busB(3),
                    cin => cry3,
                    s => busS(3),
                    cout => cry4
                );
end structural.
```

de código VHDL precisa ser instanciada repetidas vezes. O código a ser repetido inicia com LOOP e termina com END LOOP.

Na Figura 2.7, está ilustrado o processo de estímulo sendo repetido após 25 ns indicado pelo cursor ativo. Para inserir cursores na janela de visualização de sinais, basta clicar no ícone sinalizado com (+) na janela "Wave – default". De forma inversa, para remover um cursor ativo, basta clicar no ícone ao lado sinalizado com (-).

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: Somador de 4bit: soma_4b.vhd
--                 a4 Entrada a de 4 bits
--                 b4 Entrada b de 4 bits
--                 sum4 Saida de 4 bits  sum4 = a mais b
--                 cin  vem um
--                 cout vai um
-- ****************************************
ENTITY testbench3 IS END;
-------------------------------------------
-- Testbench para soma_4bit.vhd
-- Validacao assincrona
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_soma_4bt OF testbench3 IS
-----------------------------------
-- Declaracao do componente and2
-----------------------------------
component soma_4b
         port ( a4    : in std_logic_vector(3 downto 0);
                b4    : in std_logic_vector(3 downto 0);
                cin   : in std_logic;
                sum4  : out std_logic_vector(3 downto 0);
                cout  : out std_logic);
end component;

signal tb_a, tb_b  : std_logic_vector(3 downto 0);
signal tb_cin      : std_logic;
```

```
Begin

soma1: soma_4b PORT MAP (a4 => tb_a, b4 => tb_b, cin =>
tb_cin,sum4 => open, cout => open);

estimulo: PROCESS
   begin
   loop
      tb_a <= "0000"; tb_b <= "0000"; tb_cin <= '0';
      wait for 5 ns; tb_a <= "0101";
      wait for 5 ns; tb_b <= "0101";
      wait for 5 ns; tb_a <= "1010";
      wait for 5 ns; tb_cin <= '1';
      wait for 5 ns;
   end loop;
end PROCESS estimulo;

end tb_soma_4bt.
```

Os sinais de entrada no somador de 4 bits são determinados na sequência de 5 em 5 ns, de acordo com a cláusula (wait for 5 ns;), sendo responsável pela sequência dos valores binários apresentados à entrada no somador.

Na Figura 2.8 há um detalhamento da simulação do somador. Após 5 ns (linha 42 do código VHDL) do início da simulação (cursor 1), a entrada (a4) do somador é colocada em nível lógico "0101", resultando na saída (sum4) do circuito somador o nível lógico "0101", pois "(b4)" e "(c_{in})" em "t=0 ns" foram inicializados com os valores "0000" e "0", respectivamente.

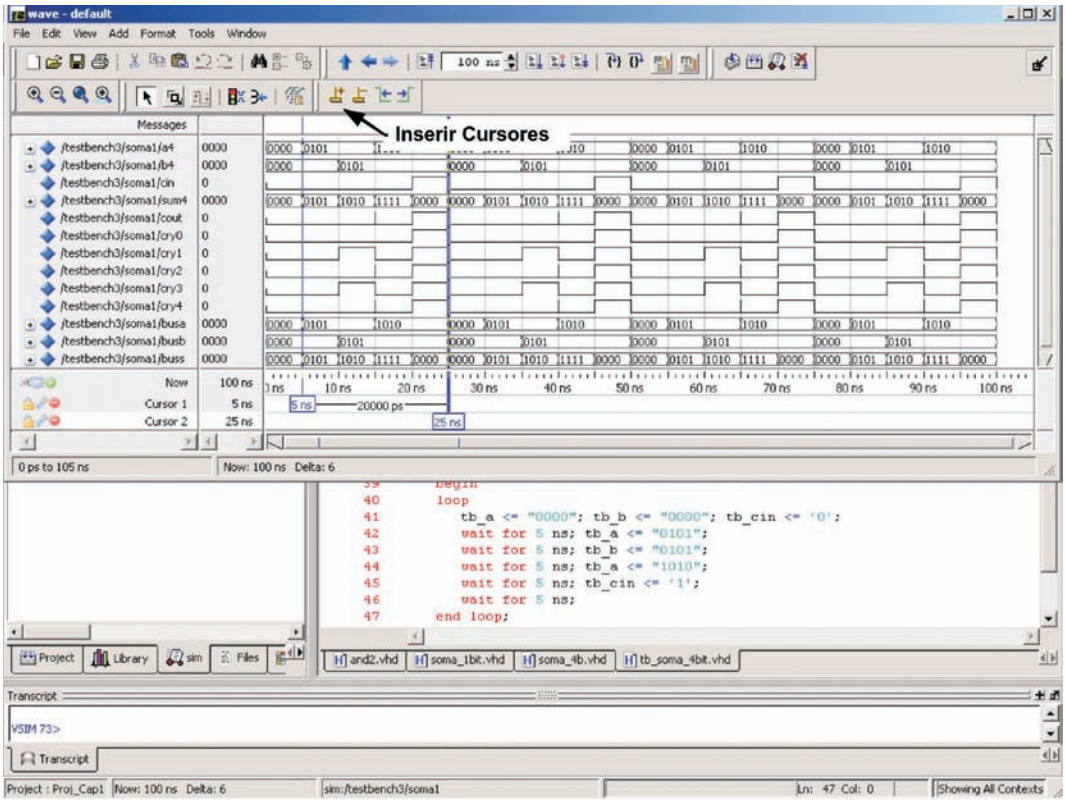


Figura 2.7 - Resultados da simulação do somador de 4 bits

Após 5 ns (linha 43 do código VHDL), a entrada "b4" do somador é colocada também em nível lógico "0101", resultando em "sum4 o valor "1010"" e o *cout* permanece em nível lógico "0". No instante de tempo 15 ns (linha 44 do código VHDL), à entrada "a4" do somador atribui-se o valor "1010", resultando em "sum4 o valor "1111"" com *cout* permanecendo em nível lógico "0". Nesse momento, observa-se que a saída *cout* apresenta um *glitch*[10] sinalizado pela elipse vertical na transição de 15 ns. Essa falha na saída *cout* do somador é resultante das comutações dos sinais de *carry* internos ao somador pelas interconexões dos sinais auxiliares (Cry0, Cry1, Cry2, Cry3, Cry4) responsáveis pela transferência do *carry*

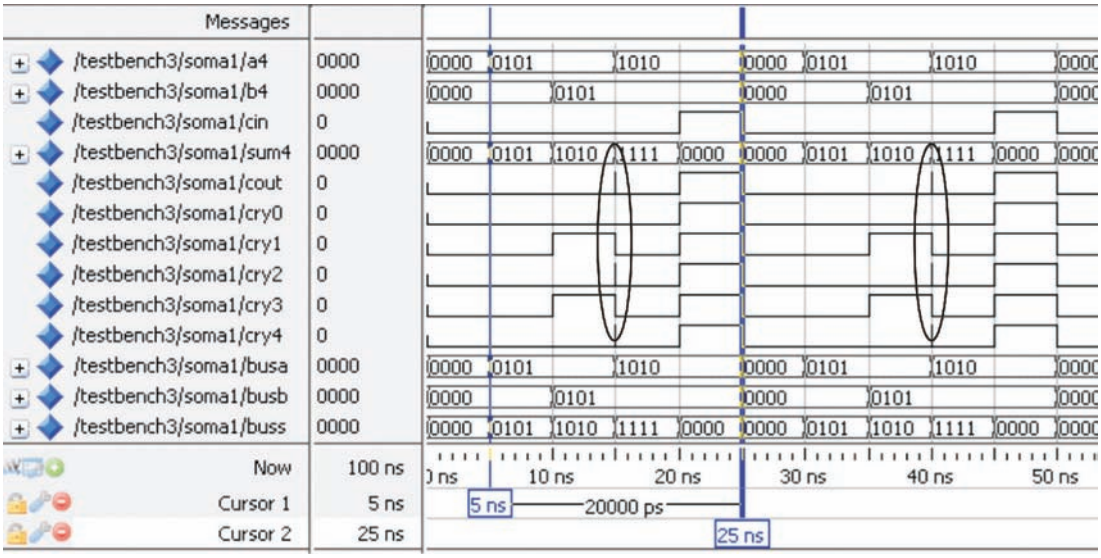


Figura 2.8 - Detalhamento dos resultados da simulação

out de um somador ao *carry in* do próximo e assim por diante, caracterizando o *ripple-carry*. Em "t=20 ns" (linha 45 do código VHDL), "c_{in}" é levado a nível lógico "1" resultando em sum4 o valor "0000" e o *cout* e todos os sinais de *carry* internos ao somador resultam o nível lógico "1" até t=25 ns, onde se encerra o laço (end loop;) voltando ao início (linha 42 do código VHDL) para se repetir indefinidamente.

Este somador de 4 bits é denominado somador com propagação do *carry* (*ripple-carry adder*), dado que os bits de transporte "propagam-se" de um somador para outro. Essa execução é vantajosa por sua simplicidade, mas inconveniente pelos problemas de velocidade (atraso).

Em um circuito real, as portas lógicas apresentam um pequeno retardo de tempo para mudarem seu estado lógico na saída, denominado de retardo de propagação (na ordem de nano segundos, porém, em computadores de alta velocidade, nano segundos são significativos). Assim, somadores com propagação do *carry* de 32 ou 64 bits podem levar de 100 a 200 ns para concluir sua adição devido à propagação do *carry* .

Por esse motivo, engenheiros criaram somadores mais avançados chamados somadores com *carry* antecipado (*carry-lookahead adders*). O número de portas necessárias para implementar o somador com carry antecipado é maior, mas seu tempo para executar uma adição é muito menor.

No próximo exemplo, encontra-se a descrição de um somador completo de n bits, construído pelo método *generate*. A interconexão dos terminais de cada um dos somadores entre si e com os pinos de entrada e saída da entidade é realizada de forma interativa. As conexões são realizadas por intermédio de um laço contador do tipo (*for I in 1 to N generate*) uma a uma, seguindo a ordem determinada pelo índice da contagem (I) que interconecta na sequência do mapeamento das conexões de entrada e saída da entidade com os terminais dos n somadores de 1 bit (component soma_1bit).

A descrição desse somador completo de n bits, construído pelo método *generate*, está no código

[10] *Glitch* é um pulso elétrico de curta duração, que normalmente é resultado de uma falha ou um erro de projeto, especialmente em um circuito digital.

VHDL transcrito como segue.

Dessa forma, a entidade *testbench* deve garantir a precisão da simulação, traduzindo fielmente o comportamento do *hardware* gerado, e modelar corretamente o comportamento das partes não-sintetizáveis. A descrição do *testbench* para validação do somador completo de n bits, construído pelo método *generate*, está no código VHDL que segue.

Para se obter um detalhamento dos sinais específicos do componente com o rótulo "soman: soma_nb", o qual identifica o somador genérico de n bits instanciado, que neste caso foi definido pelo valor constante igual a 4 (constant Nbit: integer := 4;), é necessário que, na janela "Start

```
-------------------------------------------------
-- somador nbit método generate
-- Circuito: Somadorde nbit: soma_nb.vhd
--                           an Entrada a de n bits
--                           bn Entrada b de n bits
--                           sn Saida de n bits  sn = a + b
--                           cin   vem um
--                           cout  vai um
-------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity soma_nb is
    generic(N : integer);
    port (an  : in std_logic_vector(N downto 1);
          bn  : in std_logic_vector(N downto 1);
          cin : in std_logic;
          sn  : out std_logic_vector(N downto 1);
          cout: out std_logic);
end soma_nb;

-- structural implementation of the N-bit adder

architecture structural of soma_nb is
    component soma_1bit
        PORT (a    : in std_logic;
              b    : in std_logic;
              cin  : in std_logic;
              s    : out std_logic;
              cout : out std_logic);
    end component;

    signal carry : std_logic_vector(0 to N);
begin
    carry(0) <= cin;
    cout <= carry(N);

    -- instantiate a single-bit adder N times
    gen: for I in 1 to N generate
som: soma_1bit port map(
      a => an(I),
      b => bn(I),
      cin => carry(I - 1),
      s => sn(I),
      cout => carry(I));
   end generate;
end structural.
```

Uma observação importante sobre o método *generate* é que ambos os limites do intervalo devem ser estáticos, o mesmo vale para a cláusula LOOP. Por exemplo, considerando o código (*gen: for I in 1 to N generate*), onde N é um parâmetro de entrada não-estático (*static*), resulta em um tipo de código que geralmente não é sintetizável.

Os modelos não-sintetizáveis não podem ser traduzidos para *hardware* e somente são utilizados para definir estímulos para simulações. Permitem, também, a monitoração de sinais, modelam o comportamento de outros circuitos para simulação do sistema e, em geral, são circuitos geradores de sinais de relógio, bem como memórias, processadores e circuitos de interface (por exemplo, conversores A/D ou D/A).

Simulation", seja feita a escolha da entidade *testbench* para iniciar essa simulação que, no caso, é o *testbench4*, conforme ilustrado pela Figura 2.9.

Para o início da simulação, seleciona-se a instância do componente a ser testado, neste caso, "soman". Após, é necessário apontar na tarja azul sobre a seleção e clicar com o botão direito do *mouse*

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: Somador de nbit: soma_nb.vhd
--           an Entrada a de n bits
--           bn Entrada b de n bits
--           sn Saida de n bits  sn = a mais b
--           cin   vem um
--           cout  vai um
-- ****************************************
ENTITY testbench4 IS END;
-----------------------------------------------
-- Testbench para soma_4bit.vhd
-- Validacao assincrona
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_soma_nb OF testbench4 IS
-----------------------------------
-- Declaracao do componente soma_nb
-----------------------------------
component soma_nb
        generic(N : integer);
        port (an  : in std_logic_vector(N downto 1);
              bn  : in std_logic_vector(N downto 1);
              cin : in std_logic;
              sn  : out std_logic_vector(N downto 1);
              cout: out std_logic);
end component;

constant Nbit: integer := 4;
signal tb_a, tb_b  : std_logic_vector(Nbit downto 1);
signal tb_cin      : std_logic;

Begin

soman: soma_nb
generic map(N=>Nbit)
PORT MAP (an => tb_a, bn => tb_b, cin => tb_cin,
          sn => open, cout => open);
```

```
estimulo: PROCESS
   begin
   loop
      tb_a <= "0000"; tb_b <= "0000"; tb_cin <= '0';
      wait for 5 ns; tb_a <= "0101";
      wait for 5 ns; tb_b <= "0101";
      wait for 5 ns; tb_a <= "1010";
      wait for 5 ns; tb_cin <= '1';
      wait for 5 ns;
   end loop;
end PROCESS estimulo;

end tb_soma_nb.
```

para obter o menu "<Add → Add to Wave>", e novamente o botão esquerdo do *mouse* para aceitar a opção e, desta forma, obter a janela "Wave – default", conforme ilustrado na Figura 2.10.

Na Figura 2.11, estão apresentados os sinais nas entradas e os resultados das saídas do somador completo de n bits implementado pelo método *generate*. Nesta janela, são apresentados somente os sinais do *port* do componente *soman*. Desta forma, obtém-se uma janela "Wave – default" com a

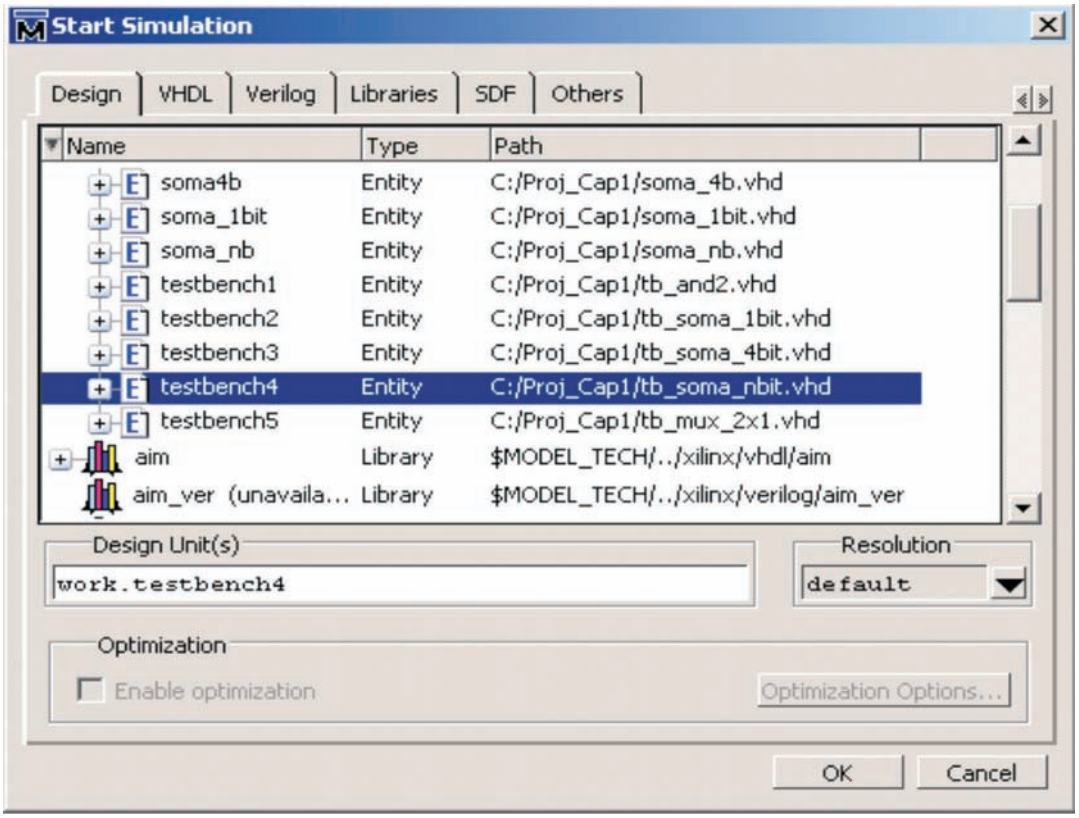


Figura 2.9 - Janela para escolha do *testbench* a ser simulado

representação dos sinais de entrada e saída específicos do somador n bits, apresentando um gráfico mais simples, sem os excessos de sinais inseridos pela opção "<Add → Add All Signals to Wave>", onde são inseridos todos os sinais da entidade para teste (*testbench4*), bem como os sinais auxiliares internos do componente *soman*.

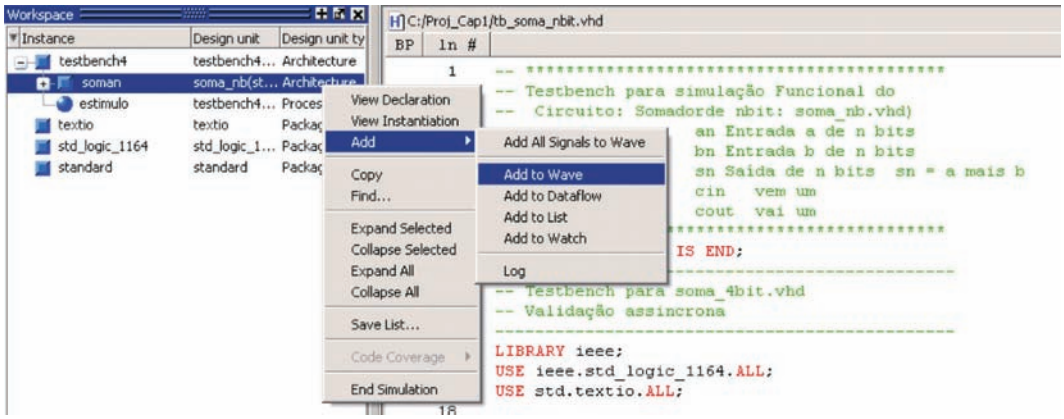


Figura 2.10 - Sequência para escolha dos sinais do componente *soman*

Na Figura 2.12, é possível verificar-se um detalhamento dos sinais, em tempo e em valores lógicos, das entradas e das saídas no somador de 4 bits, implementado pelo método *generate*. Observa-se que estes são idênticos aos do somador completo de 4 bits, construído de forma exaustiva (força bruta).

Na análise dos resultados da simulação, observa-se que, em 15 ns e 40 ns, surge um *glitch*, que também está presente no somador anterior devido à propagação do *carry* (*ripple-carry adder*) e que é inerente a esse modelo de somador.

Sobre a modelagem de somadores, é importante salientar que existem muitos outros modelos de

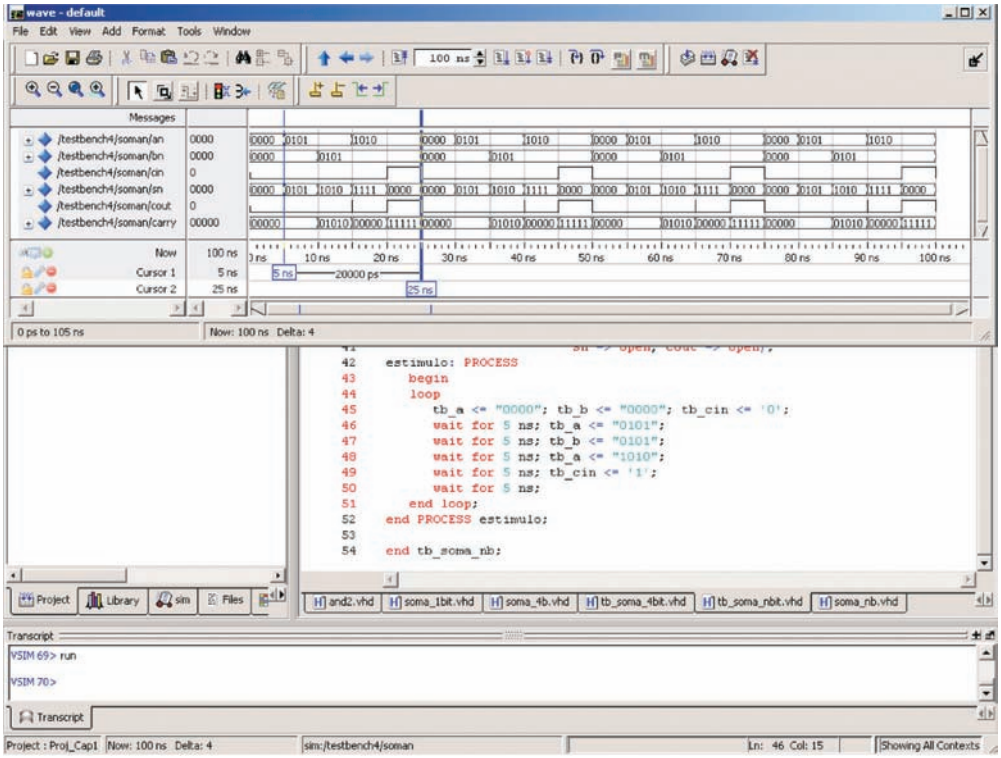


Figura 2.11 - Resultados da simulação do somador de n bits

somadores rápidos, tais como: *Simple Carry-Skip Adders; Multilevel Carry-Skip Adders; Carry-Select Adders*; dentre outros. Porém este livro tem como objetivo a prática de modelagem de circuitos simples em VHDL para fins didáticos, deixando de lado todo o rigorismo de uma análise necessária para uma

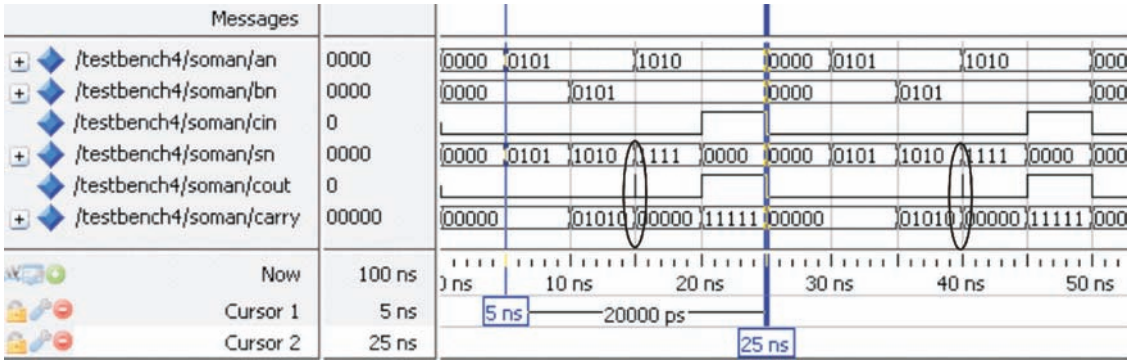


Figura 2.12 - Detalhamento dos resultados da simulação

implementação física (síntese física) com uma possível aplicação em um produto eletrônico. Contudo esta seção procura introduzir a modelagem de um somador mediante dois métodos e apresentar as técnicas para gerar os estímulos necessários na validação da lógica dos circuitos combinacionais de forma simples e prática.

2.2.3 Multiplexador

A forma mais básica de criação de códigos concorrentes é a utilização de operadores (AND, OR, +, -, *, sll, sra, dentre outros). Esses operadores podem ser utilizados para implementar diferentes combinações de circuitos. No entanto, como esses operadores mais tarde (fase de síntese) se tornam aparentes, os circuitos complexos normalmente são mais facilmente descritos, quando utilizam declarações sequenciais, mesmo que o circuito não contenha lógica sequencial. No exemplo que segue, um projeto utilizando apenas operadores lógicos é apresentado.

Um multiplexador de duas entradas para uma saída (mux_2x1) está apresentado na Figura 2.13. A saída "y" deve ser igual a uma das entradas (a, b) selecionadas pela entrada de seleção (sel).

A implementação do multiplexador 2x1 utiliza apenas operadores lógicos, conforme código VHDL transcrito como segue.

A simulação com *testbench*, para confirmar a funcionalidade do circuito do multiplexador 2x1, apresenta na saída do multiplexador dois sinais de relógio distintos, selecionados de forma assíncrona a título de exemplo e para introduzir a modelagem do sinal de *clock*.

Para gerar estímulos síncronos em uma simulação, podem ser utilizados diferentes métodos para descrever um sinal de relógio (*clock*), pois o código VHDL permite a descrição de circuitos não-

Entradas	Saídas
sel	y
0	a
1	b

$$y = \overline{sel} \cdot a + sel \cdot b$$

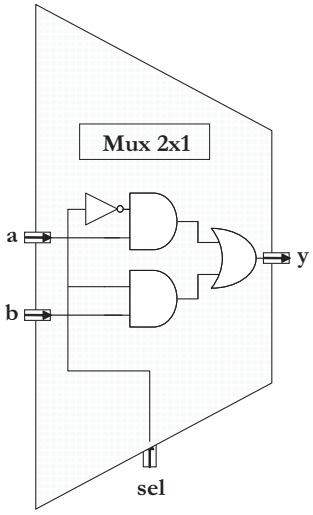


Figura 2.13 - Multiplexador 2x1, tabela verdade, equação booleana e bloco diagrama

sintetizáveis fisicamente.

Neste *testbench* são implementados dois métodos para a geração de *clock*. Na geração do sinal de

```
-----------------------------------------------
--  Circuito: multiplexador 2x1:(mux_2x1.vhd)
--          sel Selecao da entrada
--          a Entrada, sel = 0
--          b Entrada, sel = 1
--          y Saida y = nsel.a + sel.b
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mux2x1 is
    port (sel   : in STD_LOGIC;
          a,b   : in STD_LOGIC;
          y     : out STD_LOGIC);
end mux2x1;

architecture dataflow of mux2x1 is
begin
    y <= (a AND NOT sel) OR (b AND sel);
end dataflow.
```

relógio clk_1, é utilizado um tipo constante (constant T: time := 5 ns;) para determinar o período do primeiro processo descrito no corpo da arquitetura do *testbench*.

Essa forma simplifica a determinação do período do sinal de relógio "clk_1", bastando, para tal,

```
-- ***************************************
-- Testbench para simulacao Funcional do
-- circuito: multiplexador 2x1:(mux_2x1.vhd)
--           sel Seleção da entrada
--           a Entrada, sel = 0
--           b Entrada, sel = 1
--           y Saida y = nsel.a + sel.b
-- ***************************************
ENTITY testbench5 IS END;
-----------------------------------------------
-- Testbench para mux_2x1.vhd
-- Validacao sincrona (clk1 e clk2)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_mux_2x1 OF testbench5 IS
-----------------------------------
-- Declaracao do componente mux2x1
-----------------------------------
component mux2x1
    port (sel   : in STD_LOGIC;
          a,b   : in STD_LOGIC;
          y     : out STD_LOGIC);
end component;

constant T         : time := 5 ns; -- período para o clk_1
signal clk_1, clk_2: std_logic;
signal tb_sel      : std_logic;

Begin

mux: mux2x1 PORT MAP (sel => tb_sel, a => clk_1, b => clk_2,
                y => open);

clk1: process                       -- clk_1 Generator
begin
    clk_1 <= '0', '1' after T/2, '0' after T;
    wait for T;
end process;

estimulo: PROCESS
   begin
      tb_sel <= '0'; clk_2 <= '0';
      wait for 22 ns; tb_sel <= '1';
      loop                          -- clk_2 Generator
     clk_2 <= '1';
       WAIT FOR 2 ns;
     clk_2 <= '1';
     WAIT FOR 8 ns;
 end loop;
    end PROCESS estimulo;
end tb_mux_2x1.
```

Implementação do processo de estímulo para o clk_1;

Implementação do processo de estímulo para o clk_2;

escolher outro valor adequado de tempo para constante "T". Neste método, o sinal de relógio possui um ciclo de trabalho, do inglês *duty cycle*[11], padrão de 50%.

No segundo processo (estímulo) do *testbench*, é utilizado um laço com a cláusula LOOP (terminado por END LOOP), com o objetivo de repetir a execução de linhas de código, gerando um sinal de relógio. Nesta parte do código, o gerador do segundo sinal de relógio "clk_2" é modelado, com a cláusula WAIT FOR (espera por), de forma a possuir um ciclo de trabalho ajustável (diferente de 50%).

O ciclo de trabalho, dado em termos percentuais, é a duração do sinal de relógio em nível lógico "1" durante um período de relógio. Pode ser calculado a partir da equação

```
duty cycle = τ / T
```

onde

"τ" é a duração do sinal de relógio é nível lógico "1";
"T" é o período do sinal de relógio.

O segundo sinal de relógio é modelado com um ciclo de trabalho definido pelas linhas 49 e 51 do código VHDL (Figura 2.15), gerando um sinal de relógio com período (T) de 10 ns, 2 ns em nível lógico 1 e 8 ns em nível "0", configurando um ciclo de trabalho de 20% (2/10).

A Figura 2.15 ilustra os dois sinais de relógio gerados para estímulos das entradas (a, b) do multiplexador, bem como o sinal "sel" que comuta os sinais de relógio para saída "y" do multiplexador.

Na Figura 2.16, está apresentado um detalhamento da simulação do multiplexador. Como pode ser observado, após 10 ns do início da simulação (cursor 1), a entrada "a" do multiplexador é transferida para a saída "y" do circuito, dado que "sel" foi inicializado com "0". Após 10 ns (cursor 2 em 15 ns), observa-se que o primeiro sinal de relógio (clk_1) do processo gerador possui um período de 5 ns (5000 ps) e um ciclo de trabalho de 50%.

Nessa simulação, são apresentados somente os sinais do "port" do componente "mux" (multiplexador 2x1). Dessa forma, obtém-se uma janela "Wave – default" com a representação dos sinais de entrada e saída específicos do multiplexador, apresentando os sinais da coluna "Message" reduzidos, sem a indicação de todo o caminho (*path*). Essa configuração pode ser alternada de caminho completo para caminho curto simplesmente apontando e clicando no ícone na parte inferior à esquerda da coluna "Message" da janela "Wave – default", conforme ilustra a Figura 2.14.

[11] *Duty Cycle* é utilizado para descrever a fração de tempo em que um sinal de relógio está em um nível lógico "1" (estado ativo).

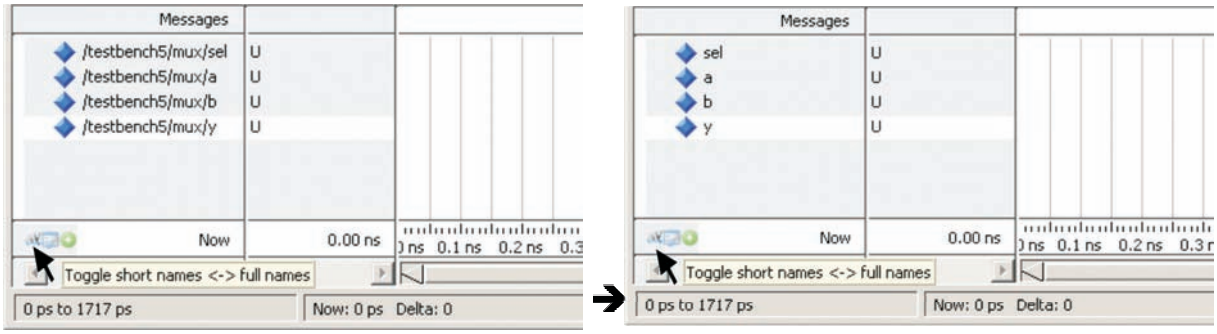


Figura 2.14 - Seleção para caminho curto da coluna "Message"

Em 22 ns de simulação (cursor 3), o sinal seletor sel foi levado a nível lógico "1", transferindo a entrada "b" do multiplexador, segundo sinal de relógio (clk_2), para a saída "y".

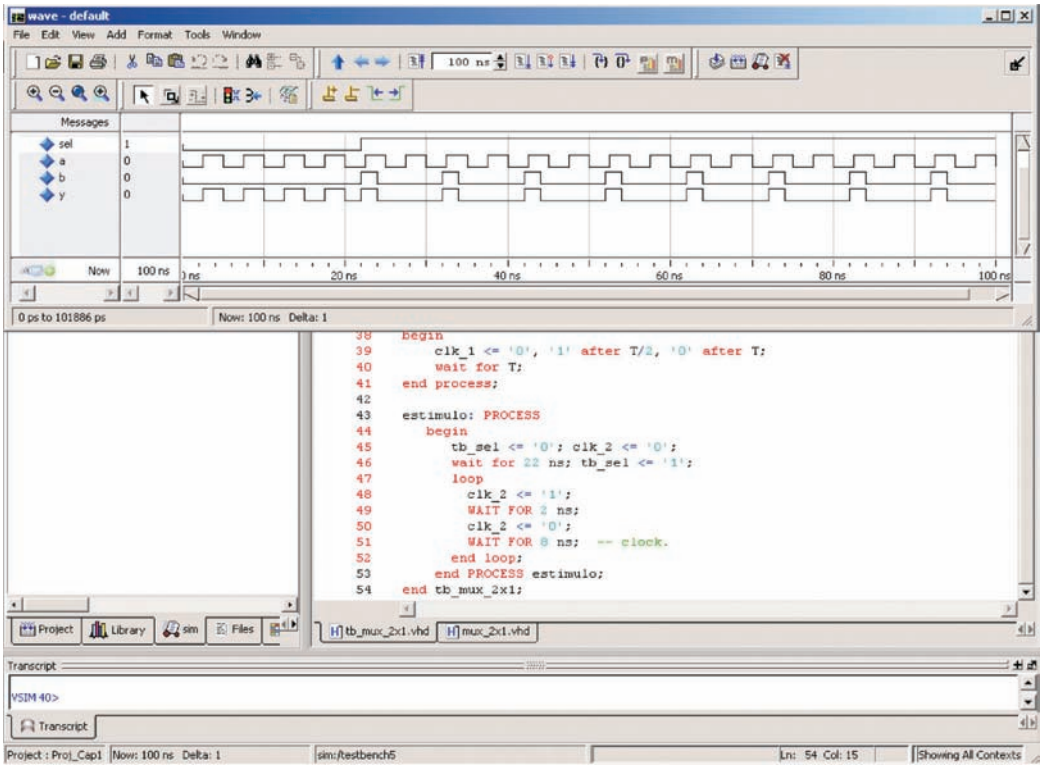


Figura 2.15 - Resultados da simulação do multiplexador 2x1

O sinal de relógio clk_2 está sendo gerado no laço LOOP/END LOOP (linhas 47 a 52 do código VHDL -) com período de 10 ns (intervalo de 10000 ps entre os cursores 3 e 4) e um *duty cycle* de 20%.

Duas outras descrições são apresentadas, no anexo B, com uso da declaração WHEN/ELSE. O primeiro exemplo apresenta um multiplexador 2x1 para barramentos de 8 bits, e o segundo exemplo de um *buffer tri-state* (alta impedância).

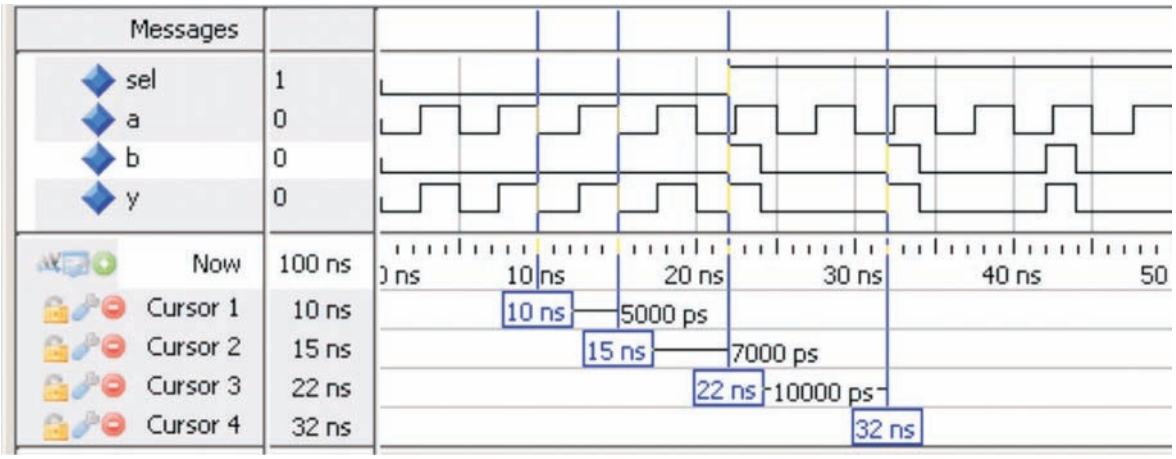


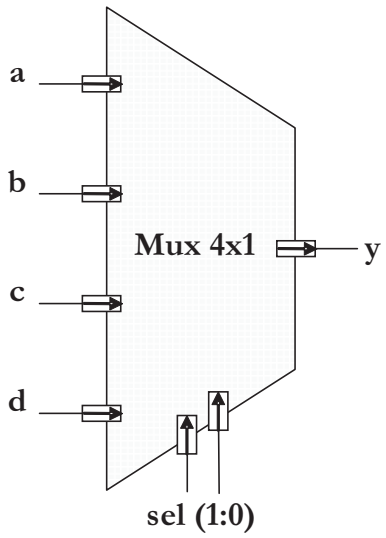
Figura 2.16 - Detalhamento dos resultados da simulação

A implementação do multiplexador 4x1 exemplifica o uso das declarações WHEN/ELSE (*simple* WHEN) e de outras, tais como WITH/SELECT/WHEN (*selected* WHEN). Neste exemplo, é apresentada uma solução com as declarações WHEN/ELSE, conforme código VHDL transcrito como segue.

Uma proposta de *testbench* para validação do multiplexador 4x1 com as declarações WHEN/ELSE é apresentada conforme código VHDL transcrito como segue.

Nesta descrição, um novo processo de estímulo é proposto baseado no *testbench* anterior (circuito do

A partir do multiplexador anteriormente descrito, a Figura 2.17 ilustra o bloco diagrama e a tabela verdade da próxima estrutura a ser descrita em VHDL – a de um multiplexador 4x1. A estrutura do multiplexador em nível de portas lógicas está apresentada na Figura 2.18



Entradas		Saídas
sel_1	sel_0	y
0	0	a
0	1	b
1	0	c

Figura 2.17 - Bloco diagrama e tabela verdade de um multiplexador 4x1

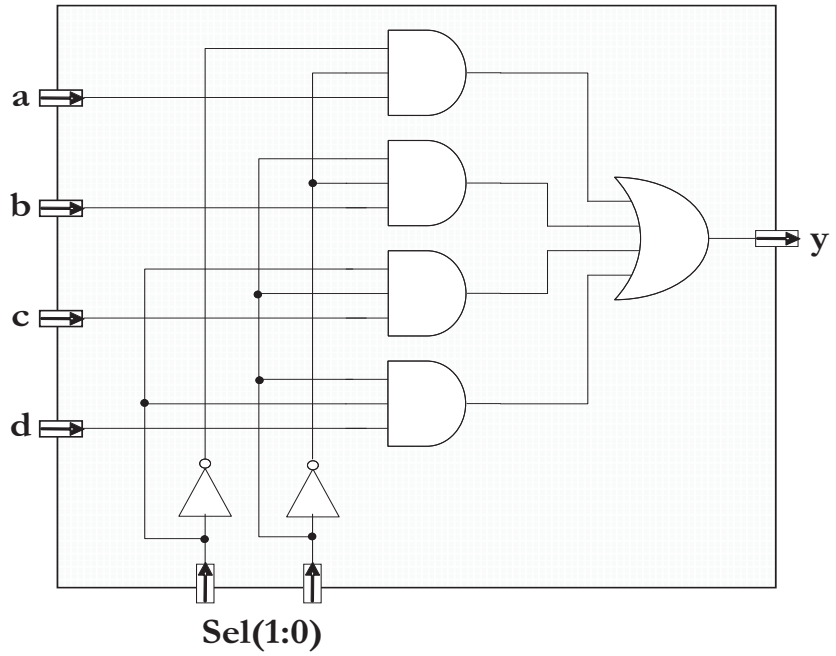


Figura 2.18 - Circuito do multiplexador 4x1 em nível de portas lógicas

multiplexador 2x1), de forma a testar e validar o multiplexador 4x1. As modificações em relação ao anterior consistem em utilizar o LOOP gerador do segundo sinal de relógio (clk_2) para também gerar os estímulos das entradas seletoras (sel) do multiplexador 4x1. Este LOOP é também responsável pela geração dos sinais de estímulo para selecionar, bem como determinar os valores das entradas (c, d) do multiplexador.

```
-----------------------------------------------
--  Circuito: multiplexador 4x1:(mux1_4x1.vhd)
--             sel (1:2) Selecao da entrada
--             a Entrada, sel = 00
--             b Entrada, sel = 01
--             c Entrada, sel = 10
--             d Entrada, sel = 11
--             y Saida (WHEN/ELSE)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------

ENTITY mux4x1 IS
    PORT ( sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           a, b, c, d: IN STD_LOGIC;
           y         : OUT STD_LOGIC);
    END mux4x1;
-----------------------------------------------
ARCHITECTURE mux1 OF mux4x1 IS
   BEGIN
 y <=  a WHEN sel="00" ELSE
       b WHEN sel="01" ELSE
       c WHEN sel="10" ELSE
       d;
   END mux1.
-----------------------------------------------
```

A entrada (sel), à medida que tem seu valor incrementado de "00" a "11", determina qual das entradas (a, b, c ou d) será direcionada para a saída "y" (Figura 2.19).

```
-- ***************************************
-- Testbench para simulacao Funcional do
-- Circuito: multiplexador 4x1:(mux1_4x1.vhd)
--                  sel (1:2) Selecao da entrada
--                  a Entrada, sel = 00
--                  b Entrada, sel = 01
--                  c Entrada, sel = 10
--                  d Entrada, sel = 11
--                  y Saida (WHEN/ELSE)
-- ***************************************
ENTITY testbench6 IS END;
-----------------------------------------------
-- Testbench para mux1_4x1.vhd
-- Validacao sincrona (clk1 e clk2)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_mux1_4x1 OF testbench6 IS
-----------------------------------
-- Declaracao do componente mux2x1
-----------------------------------
component mux4x1
    PORT ( sel     : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
       a, b, c, d  : IN STD_LOGIC;
       y           : OUT STD_LOGIC);
end component;
constant T: time := 5 ns; -- período para o clk_1
signal clk_1, clk_2   : std_logic;
signal tb_c, tb_d     : std_logic;
signal tb_sel         : STD_LOGIC_VECTOR (1 DOWNTO 0);

Begin

mux1: mux4x1 PORT MAP ( sel => tb_sel, a => clk_1, b => clk_2,
             c => tb_c, d => tb_d, y => open);

clk1: process             -- gerador do clk_1
begin
    clk_1 <= '0', '1' after T/2, '0' after T;
    wait for T;
end process;
estimulo: PROCESS
   begin
      tb_sel <= "00"; clk_2 <= '0';
      tb_c<= '0'; tb_d <= '1';
      wait for 11 ns; tb_sel <= tb_sel + '1';
      loop                            -- gerador do clk_2
```

```
                clk_2 <= '1';
                WAIT FOR 2 ns;
                clk_2 <= '0';
                WAIT FOR 8 ns;
                tb_sel <= tb_sel + '1';
                if tb_sel = "01" then tb_c <= '1'; end if;
                if tb_sel = "10" then tb_d <= '0'; end if;
              end loop;
    end PROCESS estimulo;
end tb_mux1_4x1.
-------------------------------------------------
```

A Figura 2.20 traz um detalhamento da simulação do multiplexador 4x1, onde pode ser observado que, após a inicialização de todos os estímulos de entrada (linhas 48 e 49 do código VHDL - figura 2.19), é realizado o incremento de "tb_sel" (tb_sel <= tb_sel + '1';) (linha 50 - Figura 2.19), selecionando as diferentes entradas da componente mux4x1.

Passados 11 ns (cursor 1) do início da simulação, o processo entra no LOOP, que irá se repetir enquanto a simulação estiver ativa. Este LOOP gerador do segundo sinal de relógio (clk_2) é também responsável pela geração dos sinais de estímulo das outras duas entradas (c, d) do multiplexador, que

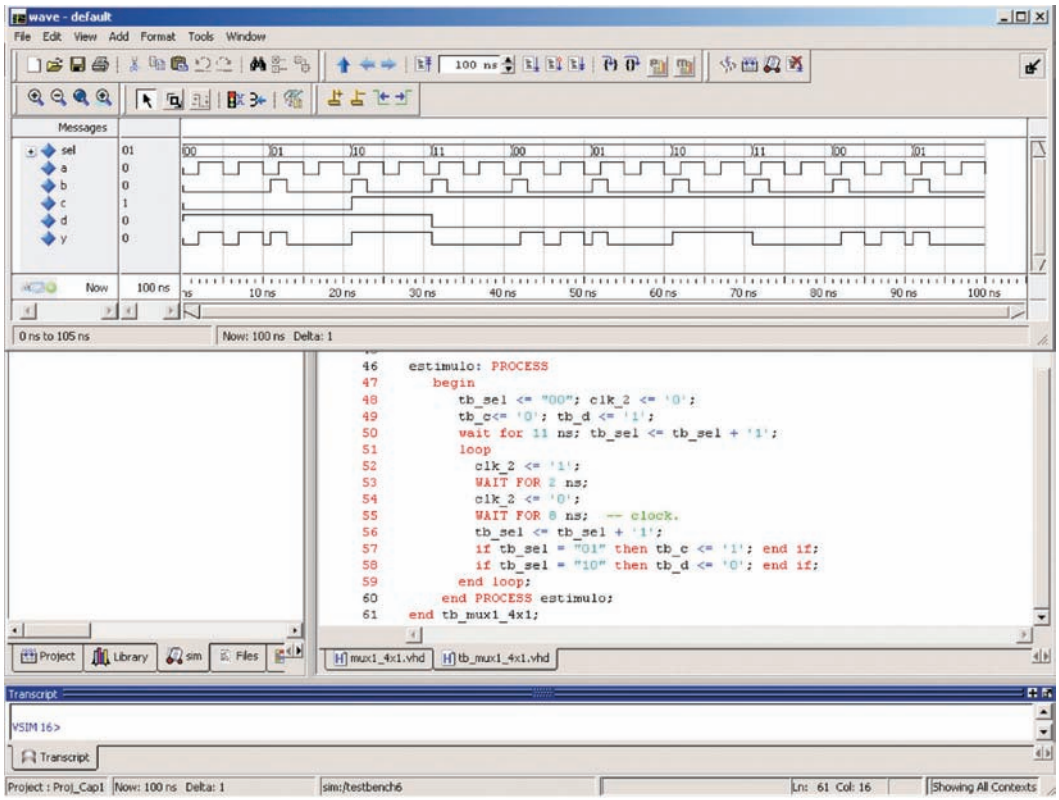


Figura 2.19 - Resultados da simulação do multiplexador 4x1

serão atualizadas com novos valores lógicos aos 21 e 31 ns (cursores 2 e 3), respectivamente (Figura 2.20).

Os estímulos para as entradas seletoras são gerados (linhas 50 e 56 do código VHDL - Figura 2.19) por meio de um contador de dois bits (tb_sel). Este contador testa, de forma incremental, todas

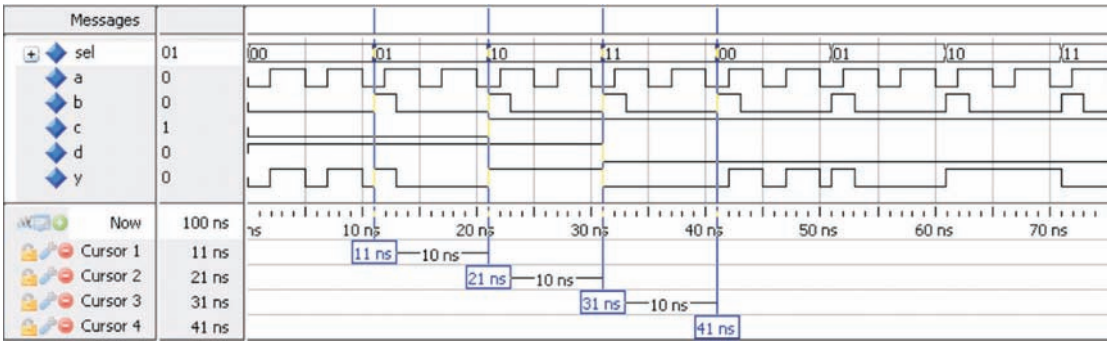


Figura 2.20 - Detalhamento dos resultados da simulação

as quatro entradas do multiplexador, conforme pode ser visto na sequência identificada pelos cursores ilustrados na Figura 2.20. Aos 41 ns (cursor 4), o contador atinge o valor lógico "00", iniciando uma nova contagem na entrada seletora. Os valores lógicos, as entradas do multiplexador (c, d) permanecem inalteradas com os valores lógicos "1" e "0", respectivamente.

Neste processo, são utilizadas cláusulas que indicam decisão, cuja sintaxe básica é "IF condição THEN atribuição", para simultaneamente selecionar, em sequência crescente, todas as entradas (a, b, c, d), bem como modificar os valores iniciais dos sinais de estímulo para as entradas (c, d) do multiplexador.

Duas outras descrições possíveis são apresentadas no anexo C, com uso da declaração WITH/SELECT/WHEN (*selected* WHEN). Os resultados simulados são evidentemente idênticos aos obtidos do componente multiplexador (mux4x1) anteriormente descrito em código concorrente.

As declarações em VHDL da classe combinacional são WHEN e GENERATE. Além destas, atribuições usando apenas os operadores lógicos (AND, OR, +, -, *, sll, sra, etc.) também podem ser usadas para construir códigos concorrentes dentro de processos (*process*). No anexo D, a cláusula CASE utilizada em seções de código sequenciais é combinada com WHEN (classe combinacional) para descrever um multiplexdor 2x1.

A lógica combinacional, por definição, é aquela na qual a saída do circuito depende unicamente do nível atual dos sinais aplicados na entrada do circuito, como ilustra a Figura 2.21(a). Fica evidente que, a princípio, o sistema não requer memória e pode ser implementado usando portas lógica básicas.

Em contrapartida, a lógica sequencial é definida como aquela em que a saída depende não só do estado atual das entradas, mas também do estado anterior da saída, conforme ilustrado na Figura 21 (b). Portanto, elementos de armazenamento do tipo *flip-flop* (FF) são utilizados como memória do estado anterior e são ligados ao bloco de lógica combinacional através de um laço de realimentação (*feedback loop*), de tal modo que também afetam a saída futura do circuito.

Um erro comum é pensar que qualquer circuito que possua elementos de armazenamento (*flip-flops*) é sequencial. A Random Access Memory (RAM) é um exemplo. A memória RAM pode ser

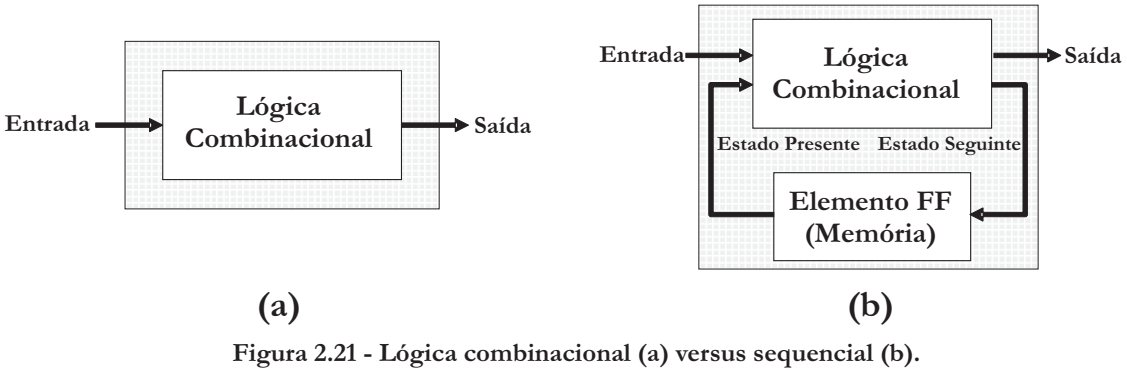


Figura 2.21 - Lógica combinacional (a) versus sequencial (b).

modelada como ilustra a Figura 2.22.

Observe que os elementos que aparecem na matriz de armazenamento entre o caminho da entrada até a saída não utilizam um laço de realimentação (*feedback loop*). Por exemplo, a operação de leitura da memória depende apenas do vetor endereço, que é o estado presente aplicado à entrada da RAM, e os resultados obtidos na saída não são afetados por acessos anteriores à RAM.

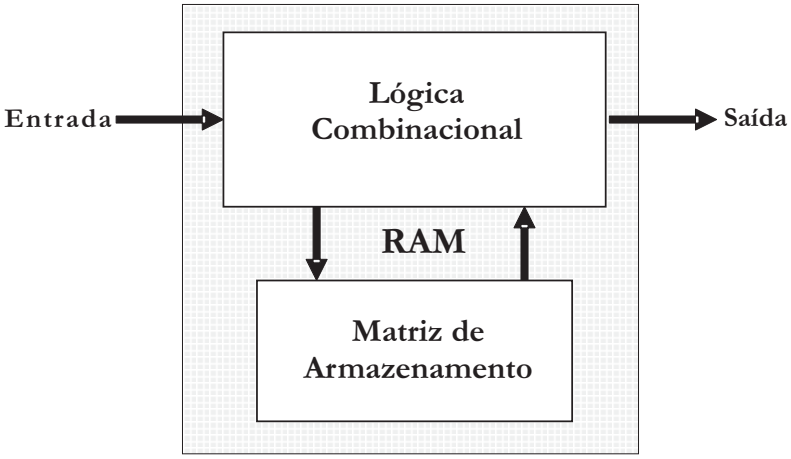


Figura 2.22 - Modelo de memória RAM.

2.2.4 Demultiplexador

O inverso de um multiplexador (MUX) é um demultiplexador (DEMUX). Neste circuito combinacional o sinal presente em sua única linha de entrada é comutado para uma de suas 2s saídas, de acordo com s linhas de controle (variáveis de seleção). Se o valor binário nas linhas de controle for "x", a saída "x" é selecionada.

Conforme ilustra a Figura 2.23, o MUX e o DEMUX selecionam o caminho dos dados. Uma determinada saída do demultiplexador corresponderá a uma determinada entrada do multiplexador.

No multiplexador, define-se apenas qual entrada passará seu valor lógico a sua única saída. Um multiplexador pode ser usado para selecionar uma de n fontes de dados, que deve ser transmitida através de um único fio. Na outra ponta, um demultiplexador pode ser usado para selecionar um único fio para um de n destinos. A função de um demultiplexador é o inverso de um multiplexador e vice-versa.

Um demultiplexador de 1xn saídas possui uma única entrada de dados e s entradas de seleção para

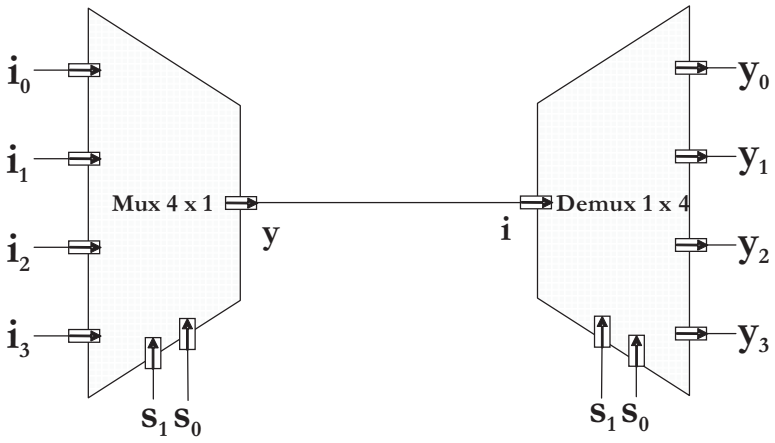
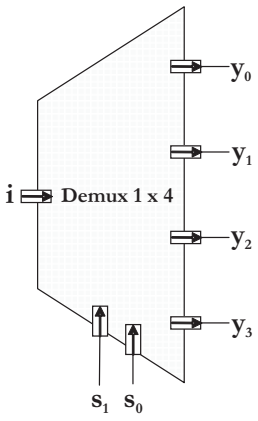


Figura 2.23 - Bloco diagrama de um multiplexador 4x1 conectado a um demultiplexador 1x4

selecionar uma das "(n = 2s)" saídas de dados (Figura 2.24).

O demultiplexador 1x4 (Figura 2.24 e Figura 2.25) é também chamado decodificador, contudo, nesta configuração, o circuito possui a função de um distribuidor de dados e pode ser visto como um dispositivo similar a uma chave rotativa unidirecional.

Uma possível descrição do demultiplexador 1x4 para implementar o circuito da Figura 2.25 é desenvolvida com as declarações WHEN/ELSE. Na sequência, é apresentado código VHDL correspondente, transcrito como segue.



Entradas			Saídas			
i	s1	s0	y3	y2	y1	y0
x	0	0	0	0	0	x
x	0	1	0	0	x	0
x	1	0	0	x	0	0
x	1	1	x	0	0	0

Figura 2.24 - Bloco diagrama e tabela verdade de um demultiplexador 1x4

A descrição do *testbench* (*testbench6a*) para validação do demultiplexador 1x4 é desenvolvida a partir do exemplo anterior, para validação do multiplexador 4x1 (*testbench6*). No código VHDL, foi acrescentado o componente demultiplexador (demux1x4), que é interconectado à saída do multiplexador que servirá de

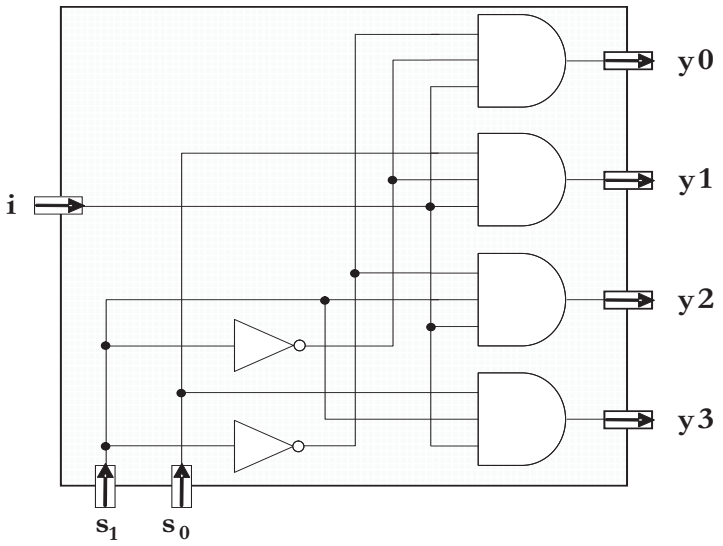


Figura 2.25 - Circuito do demultiplexador 1x4 em nível de portas lógica

estímulo na sequência selecionada pelo sinal seletor (tb_sel). O sinal seletor gerado pelo processo estímulo do *testbench* é conectado simultaneamente às entradas (sel) do multiplexador (mux4x1) e (s) do demultiplexador (demux1x4), conforme código VHDL transcrito como segue.

```
-------------------------------------------------
-- Circuito: demultiplexador 1x4:(demux1_1x4.vhd)
--                   s  Selecao da entrada
--                   i  Entrada
--                   y  Saídas,  y(3:0)
-- Utilização das declaracoes de (WHEN/ELSE)
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY demux1x4 IS
      PORT ( s   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
             i   : IN STD_LOGIC;
             y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END demux1x4;
-----------------------------------------------
 ARCHITECTURE demux1_1x4 OF demux1x4 IS
 BEGIN
      y <= "0001" WHEN s ="00" And i = '1' ELSE
           "0010" WHEN s ="01" And i = '1' ELSE
           "0100" WHEN s ="10" And i = '1' ELSE
           "1000" WHEN s ="11" And i = '1' ELSE
           "0000";
END demux1_1x4
```

Os resultados obtidos na saída do demultiplexador (demux1x4) devem ser evidentemente idênticos aos da entrada do multiplexador (mux4x1) anteriormente descrito. A Figura 2.26 ilustra os dois sinais de relógio (clk_1, clk_2) gerados para estímulos das entradas (a, b), os sinais (tb_c, tb_d) de estímulo das entradas (c, d) do MUX. O sinal (tb_yi) utilizado para interconectar a saída "y" do MUX na entrada "i" do DEMUX, também estão apresentados, bem como o sinal "s", que comuta os sinais de entrada do MUX, e simultaneamente os de saída do DEMUX.

```
-- ****************************************
-- Testbench para simulacao Funcional dos
-- Circuito: multiplexador 4x1:(mux1_4x1.vhd)
--                  sel (1:0) Seleção da entrada
--                  a Entrada, sel = 00
--                  b Entrada, sel = 01
--                  c Entrada, sel = 10
--                  d Entrada, sel = 11
--                  y Saida (WHEN/ELSE)
-- Circuito: demultiplexador 1x4:(demux1_1x4.vhd)
--                  s  Seleção da entrada
--                  i  Entrada
--                  y  Saidas,  y(3:0)
-- Utilizacao das declaracoes de (WHEN/ELSE)
-- ****************************************
ENTITY testbench6a IS END;
-----------------------------------------------
-- Testbench para mux1_4x1.vhd
-- Validacao sincrona (clk1 e clk2)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_mux1_4x1 OF testbench6a IS
-----------------------------------
-- Declaracao do componente mux2x1
-----------------------------------
component mux4x1
    PORT ( sel         : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           a, b, c, d  : IN STD_LOGIC;
           y           : OUT STD_LOGIC);
end component;

component demux1x4
    PORT (s   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          i   : IN STD_LOGIC;
          y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
end component;

constant T: time := 5 ns; -- período para o clk_1
signal clk_1, clk_2         : std_logic;
signal tb_c, tb_d, tb_yi    : std_logic;
signal tb_sel               : STD_LOGIC_VECTOR (1 DOWNTO 0);

Begin

mux1: mux4x1 PORT MAP ( sel => tb_sel, a => clk_1, b => clk_2,
      c => tb_c, d => tb_d, y => tb_yi);
```

```
demux1: demux1x4  PORT MAP ( s => tb_sel, i => tb_yi, y =>
open);

clk1: process   -- clk_1 Generator
begin
   clk_1 <= '0', '1' after T/2, '0' after T;
   wait for T;
end process;

estimulo: PROCESS
   begin
      tb_sel <= "00"; clk_2 <= '0';
      tb_c<= '0'; tb_d <= '1';
      wait for 11 ns; tb_sel <= tb_sel + '1';
      loop
            clk_2 <= '1';
            WAIT FOR 2 ns;
            clk_2 <= '0';
            WAIT FOR 8 ns;-- clock.
            tb_sel <= tb_sel + '1';
            if tb_sel = "01" then tb_c <= '1'; end if;
            if tb_sel = "10" then tb_d <= '0'; end if;
          end loop;
    end PROCESS estimulo;
end tb_mux1_4x1.
```

Na Figura 2.27, apresenta-se um detalhamento da simulação do multiplexador 4x1 conectado ao demultiplexador 1x4. As entradas "sel" (do MUX) e "s" (do DEMUX) recebem os mesmos estímulos, para que haja um sincronismo das entradas com as saídas, isto é, "i0" seja reproduzido em "y0", "i1" seja reproduzido em "y1", e assim sucessivamente (Figura 2.26). Para que fosse possível utilizar descrições anteriores, adotaram-se a, b, c e d como entradas do MUX (correspondentes a y0, y1, y2 e y3, respectivamente). As saídas do DEMUX permanecem conforme as definidas em sua *entity*, que são

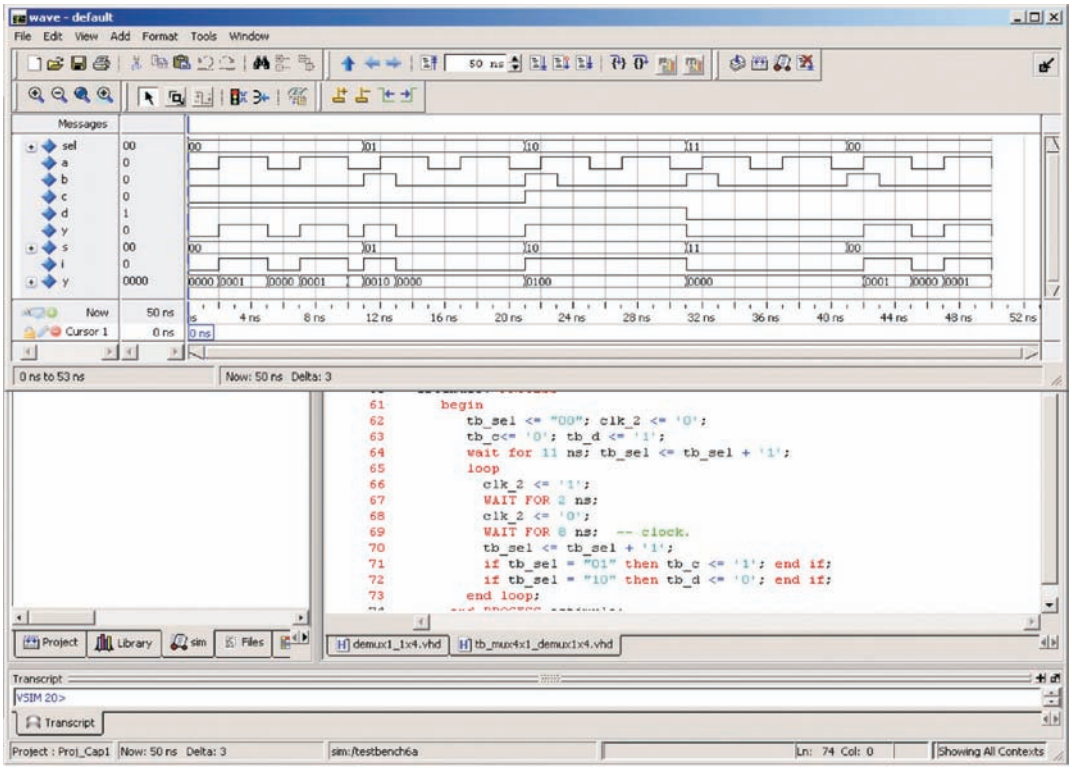


Figura 2.26 - Resultados da simulação do multiplexador 4x1 conectado ao demultiplexador 1x4

"y0", "y1", "y2" e "y3". À medida que sel é incrementada e, consequentemente, s a cada 11 ns (wait for 11 ns; tb_sel <= tb_sel + '1';), uma das quatro entradas do MUX é selecionada e é reproduzida na saída correspondente do DEMUX.

Observa-se que, nesta mesma Figura, a saída "y" do DEMUX está expandida, detalhando cada uma das saídas (y(3), y(2), y(1) e y(0)). Com esse detalhamento, foi possível observar que em 31 ns (Figura 2.28) há um *glitch*. Para expandir um sinal composto de n bits (vetor), basta utilizar o botão esquerdo do *mouse* apontando no ícone "+" ao lado do sinal.

2.2.5 Decodificadores

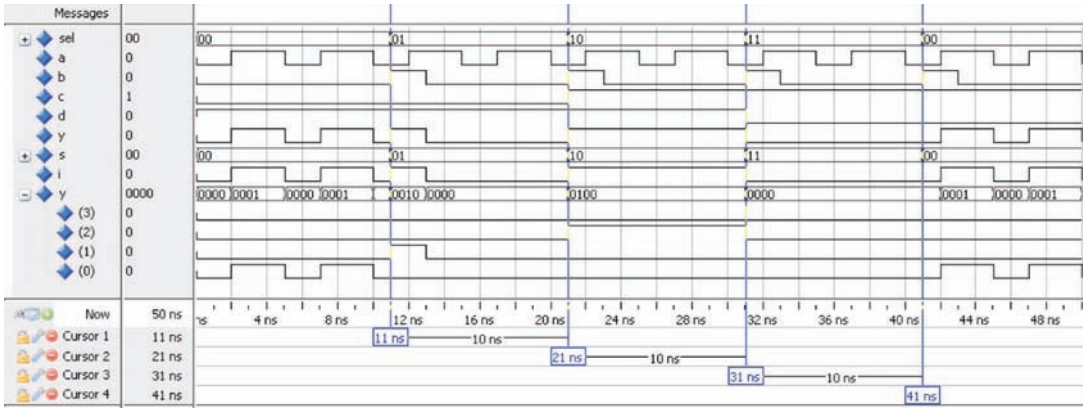


Figura 2.27 - Detalhamento de um ciclo de seleção completo MUX/DEMUX

Um decodificador é um circuito que converte códigos presentes nas entradas em um outro código nas saídas. Geralmente a entrada codificada tem menos bits do que a saída codificada. O decodificador mais usado é o decodificador n para 2n, ou decodificador binário.

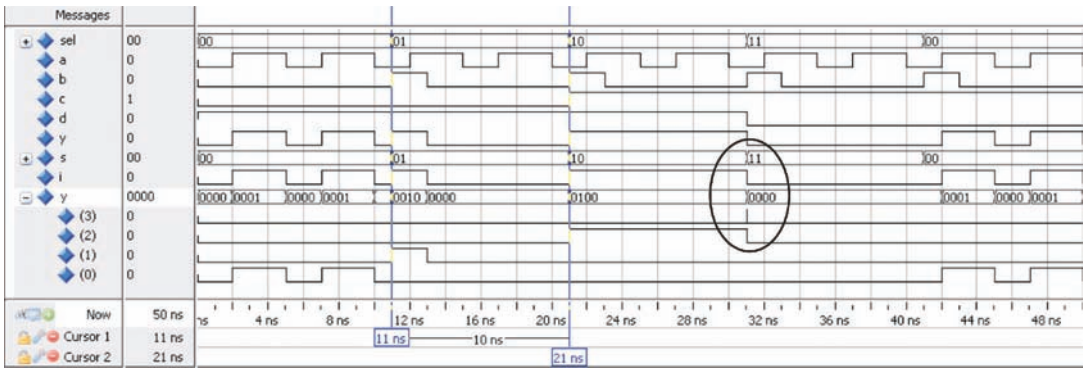


Figura 2.28 - *Glitch* observado na saída "y(3)" a partir da expansão do vetor de saída "y" na tela de simulação.

Os conteúdos abordados até aqui permitem uma boa noção do que define as bases do VHDL. Pode-se agora concentrar os projetos no código em si. O código VHDL pode ser combinacional (paralelo) ou sequencial (concorrente). A classe combinacional está sendo estudada em detalhes neste volume (volume 1), no entanto a sequencial será vista mais profundamente na continuação deste livro (volume 2).

Essa divisão é muito importante para permitir uma melhor compreensão das declarações que estão destinadas a cada classe de código, bem como as consequências de usar-se uma ou outra. Uma comparação da classe combinacional versus sequencial já está sendo gradativamente introduzida nesta Seção, para demonstrar as fundamentais diferenças entre lógica combinacional e lógica sequencial em nível de descrição de *hardware*.

O tipo especial de declaração, denominada IF, já vem sendo utilizada nas descrições de *testbenches*. Esta é empregada para especificar condições (sinais de estímulo) desejadas para desenvolver a sequência dos testes necessários para a validação funcional dos componentes sob teste.

Para concluir esta seção, discutem-se alguns códigos concorrentes, ou seja, estudam-se as declarações que só podem ser utilizadas fora de PROCESS, FUNCTION, ou PROCEDURES. Elas são as declarações da classe combinacional WHEN e GENERATE. Além destas, outras atribuições que utilizam apenas operadores (lógica, aritmética, etc.) podem naturalmente ser empregadas para criar circuitos combinacionais.

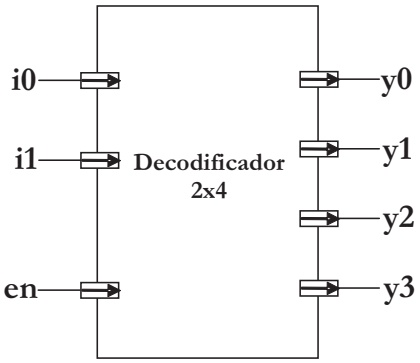
No Anexo E, está descrito um decodificador que tem como entrada código BCD[12] e como saída o código para um *display* de sete segmentos.

Um decodificador binário completo é um módulo que tem n entradas, 2n saídas e uma entrada especial (*enable*) para habilitar as saídas correspondentes ao código binário nas entradas.

A cada instante, uma única saída será ativada baseada no valor da entrada de n bits. Por exemplo, o código binário das entradas (i0, i1) determina qual saída é ativada, caso a entrada de habilitação esteja ativa (en <= '1'), conforme representado pelo bloco diagrama e respectiva tabela verdade da Figura 2.29.

A Figura 2.30 ilustra o decodificador binário completo 2x4 em nível de portas lógicas obtido da tabela verdade da Figura 2.29.

Um decodificador binário é comumente utilizado para identificar e habilitar um elemento dentre um conjunto de dispositivos, com base num código ou endereço. Por exemplo, um banco de memórias RAM utilizadas por um processador, que são selecionadas individualmente conforme o barramento de



Entradas			Saídas			
i1	i0	en	y3	y2	y1	y0
x	x	0	0	0	0	0
0	0	1	0	0	0	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0

Figura 2.29 - Decodificador binário, bloco diagrama e tabela verdade.

endereços, é decodificado para acessar e habilitar o dispositivo de memória correspondente a sua faixa de endereços, conforme ilustra a Figura 2.31.

[12]BCD – *Binary Coded Decimal*, representação binária dos dez símbolos decimais (0 a 9).

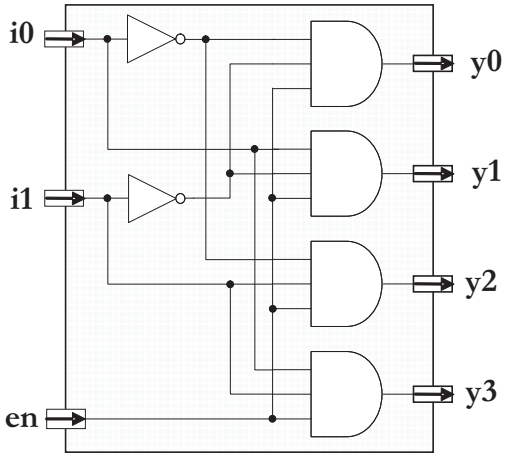


Figura 2.30 - Decodificador binário 2x4 em nível de portas lógicas

Além de inúmeras aplicações, este tipo de decodificador é utilizado para sintetizar circuitos combinados. O circuito resultante, em geral, não é uma boa alternativa do ponto de vista de custos, porém seu projeto é imediato e suas alterações são fáceis de implementar.

A implementação do decodificador 2x4 é desenvolvida com as declarações WHEN/ELSE, para implementar o circuito da Figura 2.30. Na sequência, é apresentado código VHDL correspondente,

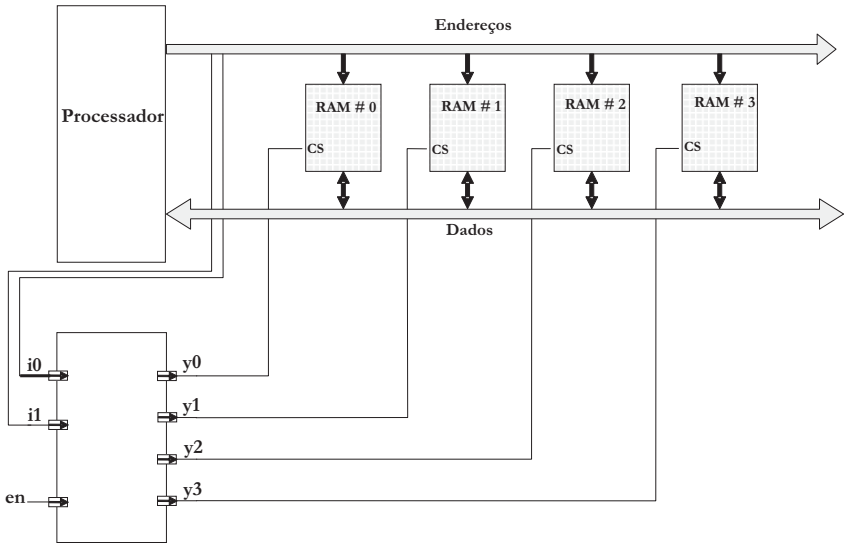


Figura 2.31 - Decodificador binário completo 2x4 utilizado como seletor de dispositivos

transcrito como segue.

Uma descrição de *testbench* (*testbench7*) para validação do decodificador 2x4 é desenvolvida utilizando-se um sinal contador (tb_i) para selecionar a saída ativa. O sinal de habilitação inicia desabilitado e, após 10 ns, é habilitado. No código VHDL, foi acrescentado o componente decodificador (deco2x4), que é instanciado (deco1: deco2x4 PORT MAP (en => tb_en, i => tb_i, y => open);) e interconectado ao processo de estímulo, conforme código VHDL transcrito como segue.

```
-----------------------------------------------
-- Circuito: decodificador 2x4:(deco1_2x4.vhd)
--      en habilita saída
--      i Entrada, i = (00:11)
--      y Saida (WHEN/ELSE)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY deco2x4 IS
      PORT (en  : IN STD_LOGIC;
            I   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
            Y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END deco2x4 ;
-----------------------------------------------
 ARCHITECTURE deco1_2x4 OF deco2x4 IS
 BEGIN
     y <=  "0001" WHEN i ="00" And en = '1' ELSE
           "0010" WHEN i ="01" And en = '1' ELSE
           "0100" WHEN i ="10" And en = '1' ELSE
           "1000" WHEN i ="11" And en = '1' ELSE
           "0000";
END deco1_2x4.
```

Os resultados obtidos na saída do decodificador 2x4 são apresentados na . Neste *testbench*, não são utilizados sinais de relógio para estímulo das entradas, apenas os estímulos gerados pelo sinal "tb_i", que, após inicializado em "00", é incrementado em uma unidade (tb_i <= tb_i + '1';) a cada 5 ns (linhas 40, 41, 42 e 43 do código VHDL - Figura 2.32).

Na Figura 2.33 está apresentado um detalhamento da simulação do decodificador. Como pode

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: decodificador 2x4:(deco1_2x4.vhd)
--                   en habilita saida
--                   i Entrada, i = (00:11)
--                   y Saida (WHEN/ELSE)
-- ****************************************
ENTITY testbench7 IS END;
-----------------------------------------------
-- Testbench para deco1_2x4.vhd.vhd
-- Validacao assincrona
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_deco1_2x4 OF testbench7 IS
-----------------------------------
-- Declaracao do componente deco2x4
-----------------------------------
component  deco2x4
      PORT (  en  : IN STD_LOGIC;
              I   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
              y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
end component;
```

```
signal tb_en      : std_logic;
signal tb_i       : STD_LOGIC_VECTOR (1 DOWNTO 0);

Begin

deco1: deco2x4 PORT MAP (en => tb_en, i => tb_i, y => open);

estimulo: PROCESS
   begin
      tb_i <= "00";
      tb_en<= '0';
      wait for 10 ns; tb_en <= '1';
      loop
          wait for 5 ns;
          tb_i <= tb_i + '1';
          end loop;
    end PROCESS estimulo;
end tb_deco1_2x4.
```

ser observado, até 10 ns do início da simulação (cursor 1), todas as saídas (y(3), y(2), y(1), y(0)) permanecem desabilitadas, pois a entrada (en) de habilitação apresenta o valor lógico '0'. Após 10 ns (wait for 10 ns; tb_en <= '1';), observa-se que cada uma das saídas é ativada na sequência correspondente ao valor lógico binário das entradas (i(1), i(0)).

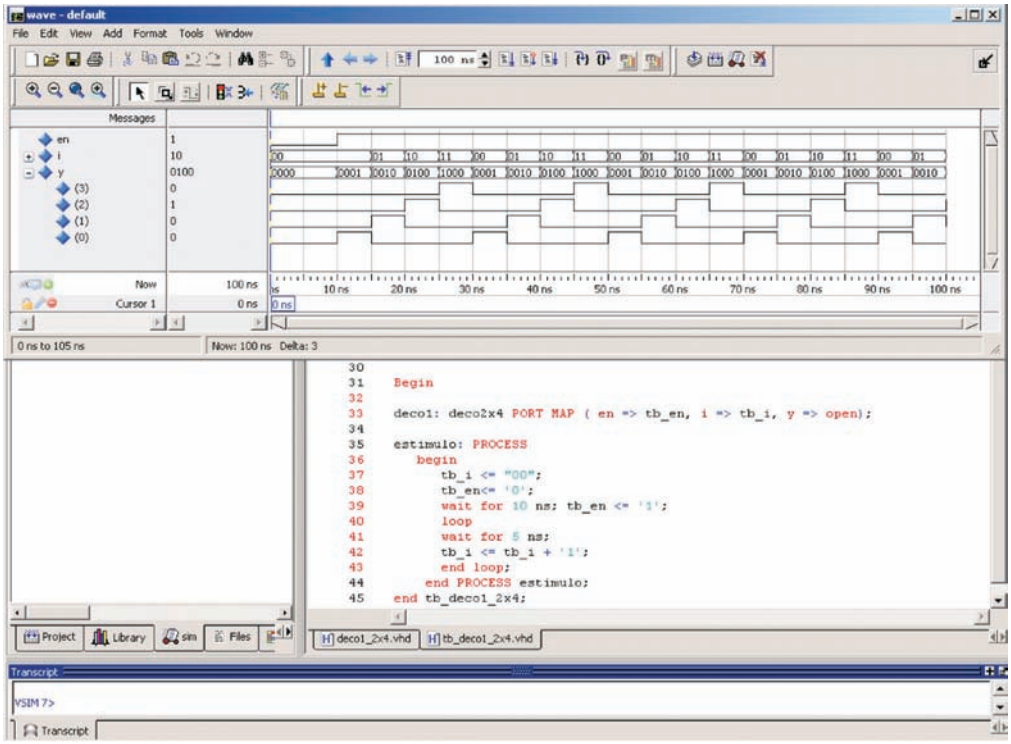


Figura 2.32 - Resultados da simulação do decodificador 2x4

Durante a seleção das saídas, o sinal de habilitação (en) permanece em estado lógico '1', permitindo que as saídas sejam ativadas na sequência. Em 30 ns (cursor 2), o processo de estímulo irá se repetir deste ponto em diante a cada 20 ns.

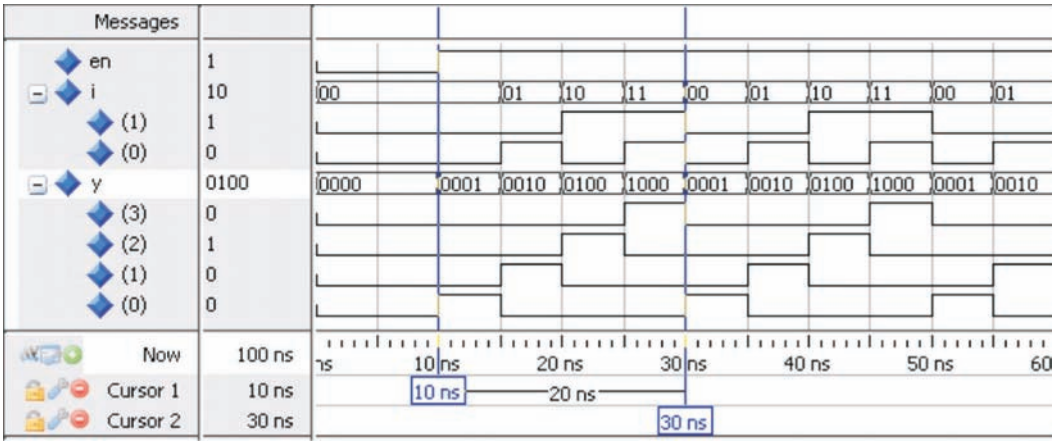


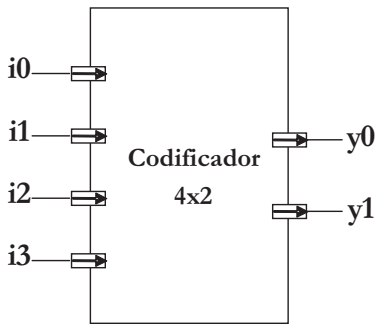
Figura 2.33 - Detalhamento dos resultados da simulação na saída do decodificador 2x4

2.2.6 Codificadores

Como exemplo da descrição do decodificador binário completo, um codificador realiza a função inversa de decodificador. O codificador é um circuito cujo código de saída tem normalmente menos bits do que o código de entrada. O codificador mais simples é o 2n para n ou codificador binário. Ele tem função oposta ao decodificador binário, também denominado de codificador de prioridade.

Por exemplo, um codificador de prioridade 4x2 é um dispositivo combinacional com 4 entradas (i3, i2, i1, i0) e duas saídas (y1, y0). As saídas "y1" e "y0", na forma de número binário, indicam qual a entrada de maior prioridade está ativa em "1", conforme ilustra a Figura 2.34 no bloco diagrama e a tabela verdade de um codificador de prioridade (4x2).

A Figura 2.35 ilustra o codificador de prioridade 4x2 em nível de portas lógicas obtido da tabela verdade da Figura 2.34. Neste codificador, foi arbitrado que, quando nenhuma das entradas estiver ativa, a saída assume alta impedância "ZZ". Esta condição e implementada pelo circuito representado pela porta NOR de quatro entradas, que é responsável pela habilitação dos *buffers tri-state* (anexo C) para as saídas (y0, y1) do codificador.



Entradas				Saídas	
i3	i2	i1	i0	y1	y0
0	0	0	0	Z	Z
0	0	0	1	0	0
0	0	1	0	0	1

Figura 2.34 - Codificador de prioridade 4x2, bloco diagrama e tabela verdade

Este circuito atribui à entrada "i0" a maior prioridade e à entrada "i3" a menor prioridade. Neste tipo de codificador, apenas uma entrada é ativada de cada vez. Se mais de uma entrada for ativada no codificador de prioridade, ele assume que apenas uma das entradas é responsável por ativar a saída correspondente à entrada de mais alta prioridade. Se na entrada o bit menos significativo for "1", a saída é "00", se o bit seguinte for "1", a saída é dependentemente do anterior e, neste caso, a saída permanece em "00", pois o bit anterior (menos significativo) possui maior prioridade. Caso o anterior não estivesse em "1", então a saída seria "01" e assim sucessivamente.

Os codificadores são utilizados como conversores de código. Os mais comuns utilizados são os códigos: Binário; BCD; Octal e Hexadecimal.

A descrição em VHDL do codificador de prioridade 4x2 é desenvolvida com as declarações WHEN/ELSE, de forma a implementar o circuito da Figura 2.35. Na sequência, é apresentado código VHDL correspondente, transcrito como segue.

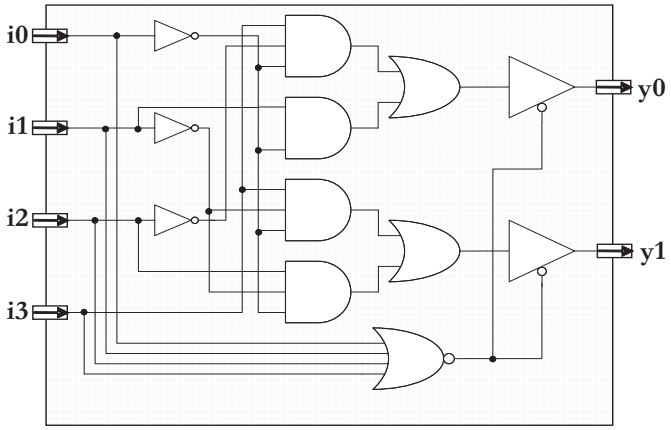


Figura 2.35 - Codificador de prioridade 4x2 em nível de portas lógicas

A descrição do *testbench* (*testbench8*) para validação do codificador de prioridade 4x2 é desenvolvida utilizando-se de um sinal contador (tb_i <= tb_i + '1';) de 4 bits para estimular as entradas "i(3:0)". O sinal inicia desabilitando todas as entradas (tb_i <= "0000";) e, após 5 ns, é incrementado em uma unidade a cada 5 ns (em *loop*) sucessivamente, para configurar todas as combinações possíveis nas entradas do codificador, conforme código VHDL transcrito como segue.

```
-----------------------------------------------
-- Circuito: codificador 4x2:(code1_4x2.vhd)
--                    i Entradas i = (3:0)
--                    y Saidas   y = (00:11)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY code4x2 IS
      PORT (i :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            y :OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END code4x2;
-----------------------------------------------
 ARCHITECTURE code1_4x2 OF code4x2 IS
 BEGIN
     y <=  "00" WHEN i(0) ='1' ELSE
           "01" WHEN i(1) ='1' ELSE
           "10" WHEN i(2) ='1' ELSE
           "11" WHEN i(3) ='1' ELSE
           "ZZ";
END code1_4x2.
```

Os resultados simulados na saída do codificador 4x2 são apresentados na Figura 2.36. No *testbench8*, também não são utilizados sinais de relógio para estímulo das entradas, apenas os estímulos gerados pelo sinal tb_i que, após ser inicializado em "0000", é incrementado em uma unidade (tb_i <= tb_i + '1';) a cada 5 ns (linhas 34, 35, 36 e 37 do código VHDL - Figura 2.36).

Na Figura 2.37, está apresentado um detalhamento da simulação do codificador. Como pode ser

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: codificador 4x2:(code1_4x2.vhd)
--                            i Entradas i = (3:0)
--                            y Saídas   y = (00:11)
-- ****************************************
ENTITY testbench8 IS END;
-----------------------------------------------
-- Testbench para code1_4x2.vhd
-- Validacao assincrona
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_code1_4x2 OF testbench8 IS
-----------------------------------
-- Declaracao do componente deco2x4
-----------------------------------
component  code4x2
      PORT (i: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
end component;

signal tb_i       : STD_LOGIC_VECTOR (3 DOWNTO 0);

Begin

code1: code4x2 PORT MAP (i => tb_i, y => open);

estimulo: PROCESS
   begin
      tb_i <= "0000";
      loop
          wait for 5 ns;
          tb_i <= tb_i + '1';
          end loop;
     end PROCESS estimulo;
end tb_code1_4x2.
-----------------------------------
```

observado, até 5 ns (cursor 1) a partir do início da simulação, todas as entradas (i(3),i(2),i(1),i(0)) permanecem desabilitadas. Desta forma, a saída do codificador apresenta alta impedância "ZZ". Após 5 ns (wait for 5 ns;), observa-se que os bits da saída "y" são ativados com o valor "00" correspondente à prioridade da entrada, pois, para este período de tempo, a entrada (i(0)) de maior prioridade encontra-

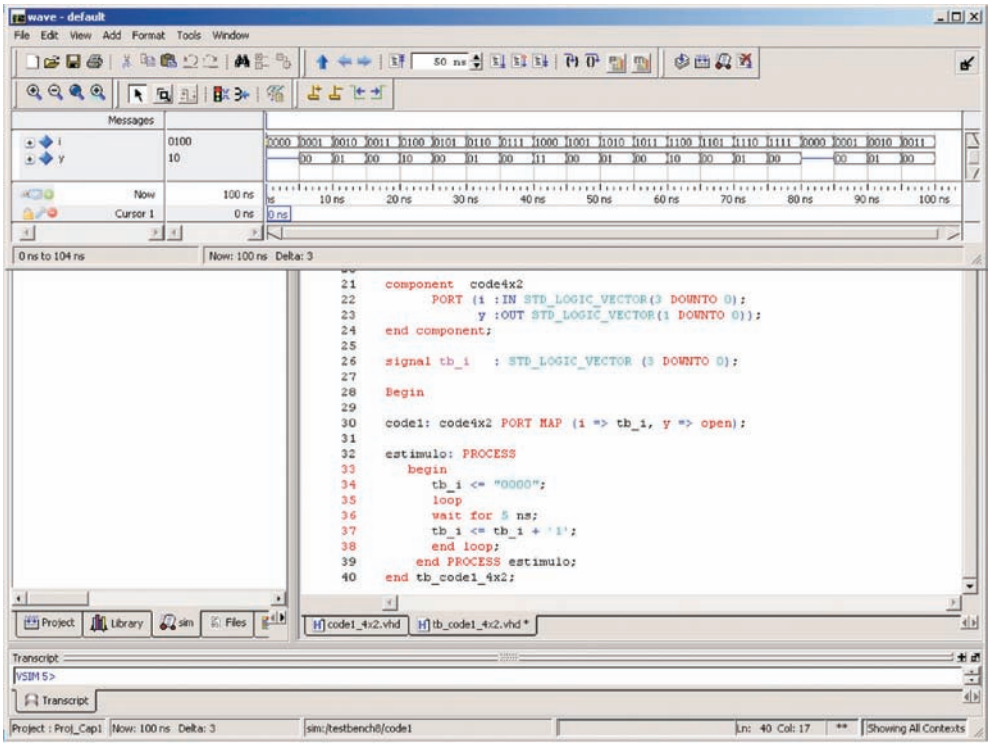


Figura 2.36 - Resultados da simulação do codificador 4x2

se em estado lógico "1".

De acordo com os estímulos do *testbench8*, as condições das entradas são incrementadas de um a cada 5 ns, desta forma impondo todas as condições possíveis paras as entradas do codificador.

Observa-se na simulação, entre os tempos de 10 ns até 20 ns (cursores 1 e 2), que as duas primeiras entradas (i(1) e i(0)) de mais alta prioridade são estimuladas com o valor lógico "0011" e a saída do codificador apresenta o valor "00" correspondente à entrada (i(0)) de prioridade "0" (mais alta).

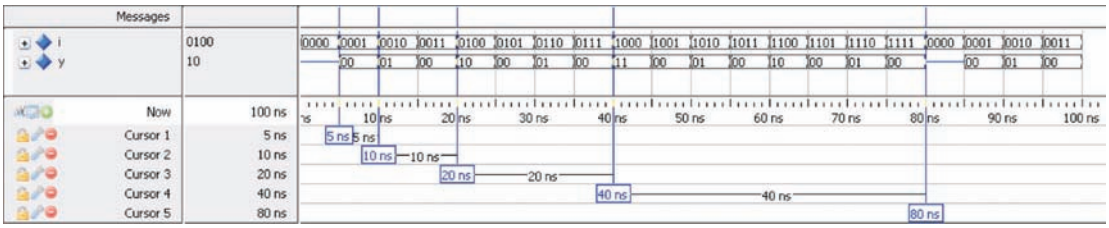


Figura 2.37 - Detalhamento dos resultados da simulação na saída do codificador 4x2.

Na simulação de 20 ns até 40 ns (cursores 3 e 4), semelhante situação ocorre para as três primeiras entradas (i(2),i(1),i(0)) e o decodificador de prioridade determina o valor de saída de acordo com a entrada de maior prioridade. Na sequência da simulação, de 40 ns até 80 ns (cursores 4 e 5), as quatro entradas (i(3),i(2),i(1),i(0)) são combinadas de forma binária crescente até que todas quatro tenham o valor lógico "1". Desta forma, são validados aos resultados na saída "y" do codificador, que apresenta o valor "00" correspondente à entrada (i(0)) de prioridade "00" (mais alta).

2.2.7 Unidade Lógica e Aritmética

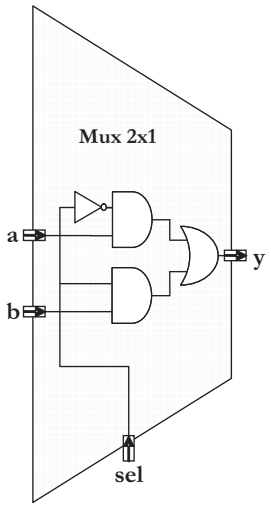
Uma Unidade Lógica Aritmética (ULA) é um circuito combinacional que realiza operações lógicas e aritméticas em um par de operandos. As funções realizadas por uma ULA são controladas por

um conjunto de entradas para a seleção das operações.

Para obter a descrição VHDL de uma ULA, é necessário desenvolver um projeto modular, que consiste em dividir todo sistema em blocos menores e mais fáceis de serem entendidos. Alguns desses blocos podem ser reutilizados de exemplos anteriores deste volume.

Nesse exemplo, a ULA é divida em dois blocos: um Lógico e outro Aritmético. O bloco Lógico, conforme ilustra a Figura 2.38, consiste de dois módulos: MUX 2x1 e duas operações lógicas (AND/OR).

Um bloco lógico pode ser composto de diferentes tipos de operações lógicas, desde simples AND/OR até operações mais complexas com: XOR; XNOR; SHIFT; ROTATE; COMPLEMENT, entre outras. Conforme ilustrado na Figura 2.38, o MUX 2x1 seleciona a operação AND ("oper = 0") ou seleciona OR ("oper = 1"), de acordo com a tabela da Figura 2.38.



oper	FUNÇÃO	OPERAÇÃO (bit a bit)
0	a . b	AND
1	a + b	OR

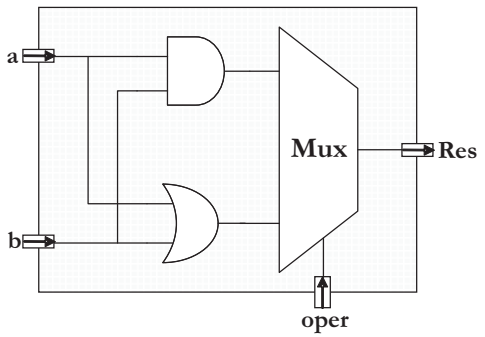


Figura 2.38 - Multiplexador seleciona duas (AND/OR) operações lógicas

Ao desenvolver-se cada módulo separadamente, é possível aplicar-se a técnica de *bit-slice*[13]. Quando um pequeno circuito lógico implementa uma função elementar, é denominado de *bit-slice*, conforme exemplificado no início do capítulo 1, com o somador completo de 1 bit na Seção 2.2.1 e o bloco lógico AND/OR/MUX 2x1 ilustrado na Figura 2.38.

Interconectando-se adequadamente um conjunto de *bit-slices*, conforme ilustrado pela Figura 2.39, é possível obter-se circuitos mais complexos em funcionalidade bem como número de bits operando simultaneamente (em paralelo).

Acrescentando-se um bloco aritmético ao multiplexador do bloco lógico, obteremos uma unidade lógica e aritmética. O bloco lógico é formado por módulos projetados anteriormente, tal como o somador completo (*Full Adder*). A unidade aritmética realiza basicamente adição do conjunto de entradas (a, b), completando o conjunto de funções realizadas pela ULA, conforme ilustra a Tabela 2.

[13]O conceito *bit-slice* (bits em fatias) ocorreu em minicomputadores e equipamentos de grande porte, que consiste em dividir o todo em pequenas unidades de poucos bits. Atualmente a nanoeletrônica permite colocar todos esses pequenos elementos em uma só pastilha e fabricar, por exemplo, um microprocessador.

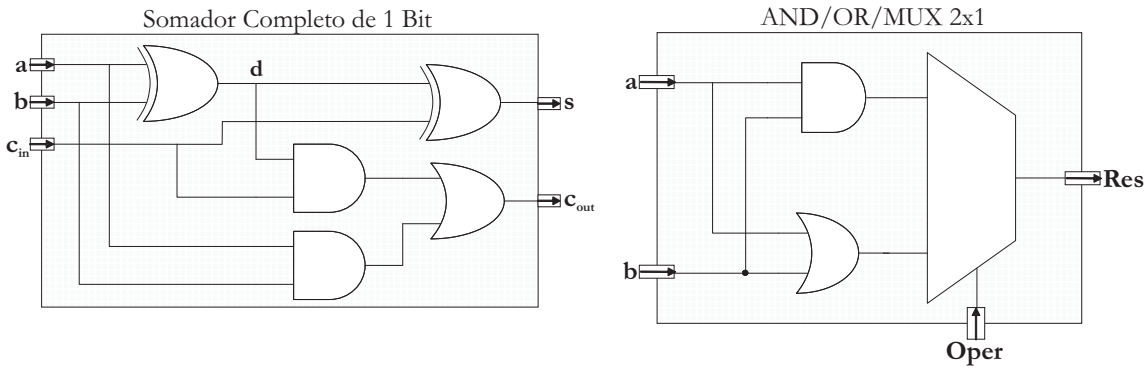


Figura 2.39 - Somador Completo de 1 bit e bloco lógico AND/OR/MUX 2x1

A partir da seleção (*oper*) das operações disponíveis, pode-se realizar o conjunto de operações lógicas e aritméticas listadas na Tabela 1 e ilustrado na Figura 2.39.

O diagrama de blocos da Figura 2.40 apresenta uma ULA de 1 bit desenvolvida a partir de *bit-slices* que são interconectados para constituírem um novo *bit-slice*, mais complexo (ULA), contudo ainda

Tabela 2 - Funções da ULA

Oper		FUNÇÃO	OPERAÇÃO (bit a bit)
0	0	a . b	AND
0	1	a + b	OR
1	0	a mais b	Adição (a+b)
1	1	reset	Zera o resultado

básico, pois realiza operações com operandos de apenas 1 bit.

No próximo exemplo, a idéia é projetar uma ULA de 4 bits, com dois operandos (a,b) e com duas

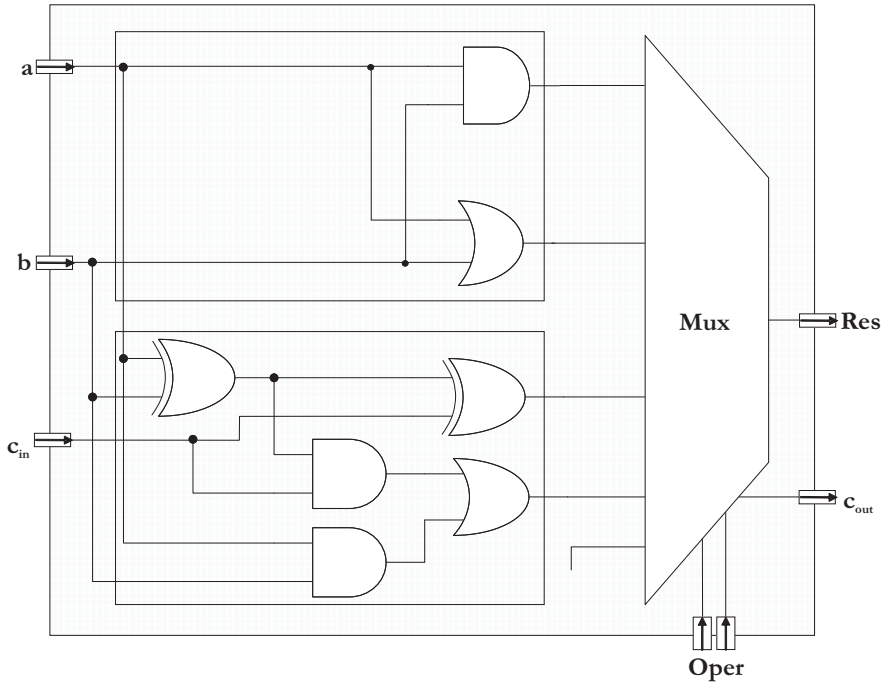


Figura 2.40 - Unidade Lógica Aritmética de 1 bit

entradas de seleção para as operações (oper) pré-definidas. A entrada *oper* seleciona uma das duas operações lógicas (oper <= "00" ou oper <= "01") ou uma operação aritmética (oper <= "10"). A operação selecionada por "11" representa uma operação de *reset*, ou seja, a saída da ULA recebe "0000". As quatro possíveis funções realizadas pela ULA foram especificadas de acordo com a Tabela 1.

A Figura 2.41 apresenta o bloco diagrama de uma unidade lógica e aritmética de n bits para quatro operações que será utilizada como base do projeto da ULA de 4 bits (n=4).

Na descrição da ULA de 4 bits para quatro operações utiliza-se o método de geração para estruturas lógicas desenvolvidas pelo método *generate* apresentado na Seção 2.2.2 (somador genérico de n bits). A interconexão dos terminais de cada um dos blocos constituintes da ULA é realizado de forma iterativa e é determinada pelo instanciamento de quatro multiplexadores 4x1 conectados em paralelo e

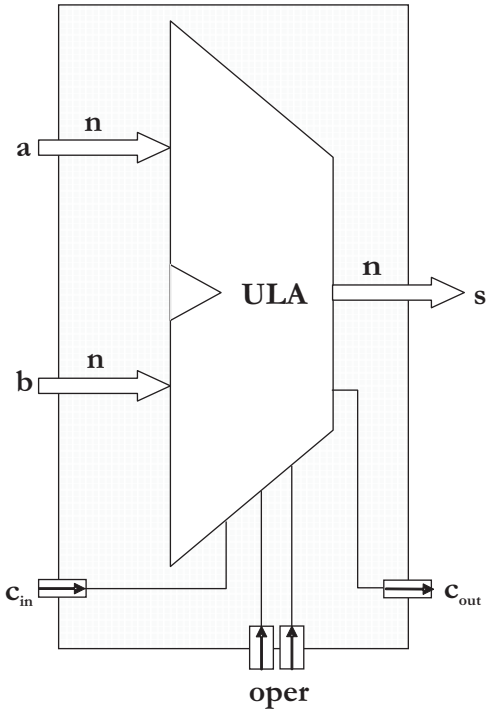


Figura 2.41 - Unidade Lógica Aritmética de n bits para 4 operações

um somador completo de 4 bits gerado por "transferência de parâmetro" para a entidade correspondente, conforme o código VHDL transcrito como segue.

As conexões entre os blocos são implementadas por intermédio de um laço contador típico (for I in 0 to Nbit-1 generate), onde uma constante inteira (constant Nbit : integer := 4;) determina o número de bits com o qual será constituída a ULA como um todo.

Para conformar um multiplexador de quatro bits, neste exemplo, é instanciado quatro vezes

```
-------------------------------------------------
-- Circuito: ula 4 bits:(ula1_4bits.vhd)
--                   oper  Selecao da entrada
--                   a  Entrada a(3:0)
--                   b  Entrada b(3:0)
--                   s  Saida   s(3:0)
--                   cin Entrada carry
--                   cout Saida   carry
-- Utilizacao mux1_4x1, soma_nb, AND/OR
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY ula4bits IS
      PORT (oper: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
            A   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
            b   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
            s   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
            cin : IN STD_LOGIC;
            cout:OUT STD_LOGIC);
END ula4bits;

architecture structural of ula4bits is
---------------------------------
```

```
-- Declaracao do componente mux4x1
----------------------------------
component mux4x1
    PORT ( sel          : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           a, b, c, d   : IN STD_LOGIC;
           y            : OUT STD_LOGIC);
end component;
----------------------------------
-- Declaracao do componente soma_nb
----------------------------------
component soma_nb
        generic(N  : integer);
        port (an   : in std_logic_vector(N downto 1);
              bn   : in std_logic_vector(N downto 1);
              cin  : in std_logic;
              sn   : out std_logic_vector(N downto 1);
              cout : out std_logic);
end component;

constant Nbit              : integer := 4;
signal t_and, t_or, t_out  : std_logic_vector(Nbit-1 downto 0);

Begin
-- instanciando o mux4x1 Nbit vezes
   mux: for I in 0 to Nbit-1 generate
   mux: mux4x1 port map( sel => oper, a =>  t_and(I), b => t_or(I),
               c =>  t_out(I), d => '0', y => s(I));
   end generate;

-- instanciando o soma_1bit N vezes, onde N = Nbits
   soman: soma_nb  generic map(N=>Nbit)
          PORT MAP ( an => a, bn => b, cin => cin,
               sn => t_out, cout => cout);
-- bloco logico AND/OR
   t_and <= a and b;
   t_or  <= a or  b;
end structural.
```

(Nbit) um multiplexador 4x1 simples, seguindo a ordem determinada pelo índice de contagem (I), que interconecta entradas e saídas na sequência do mapeamento das conexões do multiplexador de 1 bit (*bit-slice*) que, neste caso, é o componente mux4x1 desenvolvido na Seção 1.3 (multiplexador 4x1).

A instância do somador completo de n bits também é implementada pelo método *generate*. Porém, nesta instância, o somador n bits tem sua dimensão determinada pela passagem da constante "Nbit", definida inicialmente no corpo da arquitetura da ULA de 4 bits. Observa-se que o *port map* do somador de n bits (soma_nb generic map(N=>Nbit)) não utiliza o laço contador para gerar um somador de 4 bits no corpo da arquitetura da ULA. Nesta declaração, para o somador de n bits o parâmetro N, cujo valor é igual a quatro, é transferido pela declaração (generic map(N=>Nbit)) que instancia o somador completo de n bits. Este somador é gerado pela entidade (soma_nb) descrita na Seção 2.2.2 (somador genérico de n bits), que localmente executa o método *generate* por meio de um laço contador implementando o somador de 4 bits a partir de um único somador de um bit (soma_1bit).

Neste *testbench*, não são utilizados sinais de relógio para estimular as entradas, pois o circuito da ULA é puramente combinacional. O processo gerador de estímulos inicia zerando todos os sinais (linhas 49 e 50 do código VHDL), conforme ilustrado na Figura 2.42. O sinal seletor de operação (tb_oper <= "00") é inicializado em zero, configurando a ULA para a operação lógica AND, o sinal de

O bloco lógico AND/OR é descrito ao final utilizando-se de dois sinais temporários, um para a operação AND (t_and <= a and b;) e outro para a operação OR (t_or <= a or b;). Essas operações lógicas estão ocorrendo em paralelo, entre si, com o somador e o multiplexador. Para dois valores de 4 bits presentes nas entradas dos operandos (a,b), as três operações ocorrem simultaneamente, contudo somente uma delas é selecionada para a saída do multiplexador, de acordo com o valor de 2 bits presente na entrada seletora (oper).

Importante observar que o valor da quarta entrada do multiplexador 4x1 está sempre zerado (d => '0') no *port map* do multiplexador (mux4x1) e, desta forma, são gerados quatro multiplexadores 4x1 conectados em paralelo, cuja seleção da quarta entrada configura uma operação de *reset* na saída (s) da ULA, resultando o valor "0000".

Para a validação funcional da descrição da ULA de 4 bits para quatro operações, é desenvolvido o *testbench9*, para gerar todas as combinações possíveis nas entradas da ULA, conforme código VHDL transcrito como segue.

```
-- ****************************************
-- Circuito: ula 4 bits:(ula1_4bits.vhd)
--            oper    Seleção da entrada
--            a       Entrada a=
--            b       Entrada
--            s       Saida
--            cin     Entrada Carry
--            cout    Saida Carry
-- Utilizacao mux1_4x1, soma_nb, AND/OR
-- ****************************************
ENTITY testbench9 IS END;
-----------------------------------------------
-- Testbench para ula1_4bits.vhd
-- Validacao assincrona
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_ula1_4bits OF testbench9 IS
-----------------------------------_
-- Declaracao do componente ula4bits
-----------------------------------
component  ula4bits
    PORT (oper     : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          a        : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
          b        : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
          s        : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
          cin      : IN STD_LOGIC;
          cout     : OUT STD_LOGIC);
end component;

signal tb_oper     : STD_LOGIC_VECTOR (1 DOWNTO 0);
signal tb_a, tb_b  : STD_LOGIC_VECTOR (3 DOWNTO 0);
signal tb_cin      : STD_LOGIC;

Begin

ula1: ula4bits PORT MAP (oper => tb_oper,
                         a => tb_a,
                         b => tb_b,
                         s => open,
                         cin => tb_cin,
                         cout => open);
estimulo: PROCESS
```

```
begin
      tb_oper <= "00";tb_cin <= '0';
      tb_a <= "0000";tb_b <= "0000";
      wait for 5 ns;
      tb_oper <= "10";tb_cin <= '1';
      tb_a <= "1111";
      loop
         wait for 5 ns;
         tb_cin <= '0';
         tb_a <= tb_a + "0001";
         tb_b <= tb_b + "0010";
         tb_oper <= tb_oper + '1';
         end loop;
   end PROCESS estimulo;
end tb_ula1_4bits.
```

estímulo da entrada do *carry* (tb_cin <= '0';) também é zerado, bem com os estímulos para as entradas dos operandos (tb_a <= "0000"; tb_b <= "0000";).

Após manter inicializados todos os sinais em zero por 5 ns (linha 51 do código VHDL), é selecionada a operação de adição (tb_oper <= "10"), o sinal de estímulo da entrada do *carry* (tb_cin <= '1';) é levado a '1', simulando um sinal do *carry* de saída (*cout*) proveniente de uma outra ULA de 4 bits idêntica. Por exemplo, duas ULAs ligadas em cascata podem configurar uma ULA de 8 bits. Nesse momento, também o sinal de estímulo para a entrada do operando a é configurado com o máximo valor lógico de 4 bits (tb_a <= "1111";). Desta forma, é possível simular a propagação do sinal de *carry* através da ULA em teste, bem como validar o *carry* de saída (*cout*), conforme ilustrado na Figura 2.42 (linhas 52 e 53 do código VHDL).

Após o teste de propagação do sinal de *carry* através da ULA, o processo de estímulos do *testbench* entra em *loop* (linha 54 até 60 do código VHDL - Figura 2.42). No início do *loop*, há um tempo de espera de 5 ns e após este o sinal de estímulo da entrada do *carry* (tb_cin <= '0';) é sempre zerado. O estímulo para a entrada do operando a é incrementado de uma unidade (tb_a <= tb_a + "0001";) e o da entrada

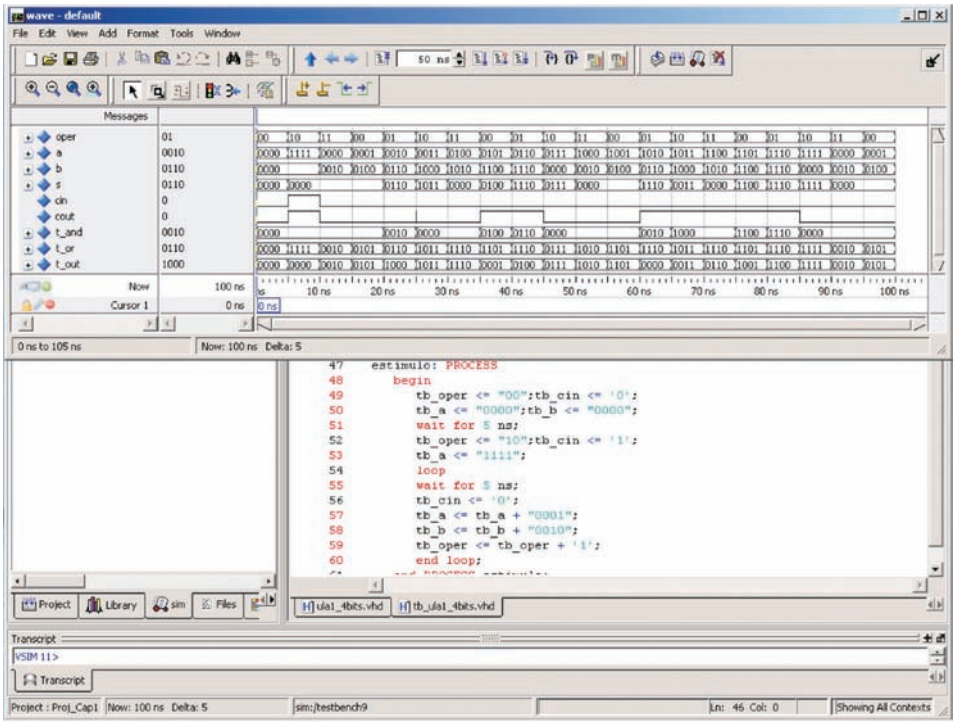


Figura 2.42 - Resultados da simulação da ULA de 4 bits para quatro operações

do operando "b" é incrementado de duas unidades (tb_b <= tb_b + "0010";), bem como o sinal seletor de operação é incrementado de uma (tb_oper <= tb_oper + '1';) e assim voltando ao início do *loop* para aguardar mais 5 ns, repetindo os respectivos incrementos dos sinais de estímulos, para configurar todas as combinações possíveis nas entradas da ULA.

Os resultados dos sinais simulados na ULA de 4 bits são apresentados em detalhes na Figura 2.43, onde pode ser observado que, a partir do início da simulação (cursor 1) até 5 ns (cursor 2), todas as entradas permanecem zeradas. Após os 5 ns iniciais (wait for 5 ns;) até 10 ns (cursor 3), é observado o teste de propagação do sinal de *carry* através da ULA, onde o máximo valor lógico para 4 bits (tb_a <= "1111";) é adicionado com os zeros do segundo operando (tb_b <= "0000";) mais a entrada do *carry* (tb_cin <= '1';), resultando zero na saída (s) da ULA e "1" é propagado para o *carry* de saída (*cout*).

Considerações Finais

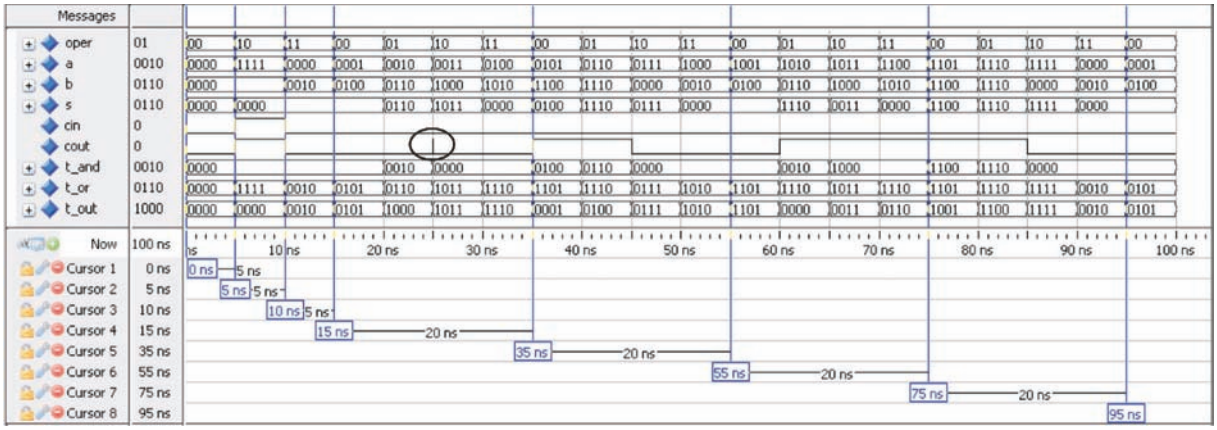


Figura 2.43 - Detalhamento dos resultados da simulação da ULA de 4 bits para quatro operações

Desta forma, é observada a propagação do sinal de *carry* através da ULA. O estímulo (tb_cin <= '1';) conectado ao *carry* de entrada (cin) é transferido do primeiro somador de 1 bit até o quarto e deste para a saída (*cout*) da ULA.

Na simulação de 10 ns até 15 ns (cursores 3 e 4), observa-se a operação de *reset* na saída (s) da ULA. Nota-se que internamente os resultados das três operações estão presentes nos sinais auxiliares (t_and, t_or, t_out) e ocorrem paralelamente. Esses resultados das operações (AND/OR/soma) são representados no detalhamento dos três últimos sinais da coluna "Messages" da Figura 2.43.

A partir de 15 ns (cursor 4) até 95 ns (cursor 8), as três operações (AND/OR/soma) e o *reset* repetem-se em sequência de 20 em 20 ns, mas os operandos têm seus respectivos incrementos configurando diferentes combinações nas entradas da ULA e, desta forma, observa-se que as operações lógicas e aritméticas ocorreram em paralelo, porém somente um resultado da operação que é determinado pelo sinal de estímulo (tb_oper) é apresentado na saída (s) da ULA, conforme é ilustrado na Figura 2.43.

Em 25 ns de simulação, observa-se que o *carry* de saída (*cout*) apresenta um típico *glitch* (sinalizado pela elipse na Figura 2.43). Esta falha no *carry* de saída da ULA é resultante das comutações dos sinais de *carry* internos ao somador de 4 bits, que são responsáveis pela transferência do *carry out* de um somador ao *carry in* do próximo e assim por diante, conforme exemplificado na Seção 2.2.2 (somador genérico de n bits).

Não é possível projetar sistemas digitais sem entender alguns blocos básicos, tais como portas lógicas e *flip-flops*. A maioria dos circuitos digitais baseados em portas lógicas e *flip-flops* são normalmente projetados a partir de equações booleanas. Neste volume, várias técnicas são desenvolvidas utilizando-se a linguagem VHDL para descrição de dispositivos digitais. A maior dificuldade nos métodos tradicionais de projeto é a conversão manual da descrição do projeto em um conjunto de equações booleanas. Essa dificuldade é eliminada com o uso de linguagens de descrição de *hardware*.

Uma abordagem do fluxo de projeto, síntese funcional, bancada de testes virtual e simulação de circuitos combinacionais básicos como aprendizagem prática é o principal objetivo deste volume. No que tange aos circuitos desenvolvidos nos exemplos práticos, também servem de embasamento no estudo da eletrônica digital básica, tal como no estudo das portas lógicas em circuitos combinacionais.

Neste volume, não foi levada em conta uma visão das características temporais dos modelos físicos referentes às entidades desenvolvidas, pois o foco é a prática no uso da ferramenta computacional de modelagem e simulação ModelSim e, por meio desta, o desenvolvimento de circuitos básicos através do fluxo de projeto em VHDL na metodologia *Top-Down*, bem como simular os circuitos projetados utilizando-se da estratégia de validação funcional baseada no uso de *testbenches*.

Este volume encerra uma importante etapa no estudo de circuitos digitais construídos a partir de blocos básicos combinacionais. Os próximos dois volumes destinam-se a uma abordagem mais aprofundada, incluindo comparadores, geradores de paridade, memórias (ROM/RAM), multiplicadores/divisores, registradores (deslocamento, carga paralela, acumulador, serial-paralelo), contadores (binário, BCD, Johnson, Gray/*up*, *down*, *up-down*), máquina de estados, geradores de *clock* e sequenciadores.

No segundo volume, segue-se com a prática em VHDL com circuitos combinacionais de maior complexidade, introduzindo os princípios da lógica sequencial, bem como visa à síntese física em um dispositivo eletrônico reconfigurável (FPGA). Para tal, o ModelSim é utilizado integrado a um novo ambiente para desenvolvimento denominado *ISE* Project Navigator[14] (navegador de projetos), que se apresenta como um novo método de aprendizado, aplicado na área de eletrônica digital.

O ISE é de concepção da empresa Xilinx®. Constitui-se de um conjunto de ferramentas de *software* integradas, que permite o desenvolvimento de um projeto, desde o projeto da lógica de funcionamento, da modelagem, simulação e *testbench*, até a implementação física em um FPGA.

O terceiro volume estimula o aprendizado fundamentado no desenvolvimento de soluções baseadas em problemas reais, tais como o desenvolvimento e a implementação física de sistemas processadores digitais de sinais, filtros digitais, análise espectral e aplicações para áudio e vídeo. Neste volume, faz-se o estudo da eletrônica digital considerando-se a possibilidade de uma aplicação em outras áreas da ciência, caracterizando-se por ser um método interdisciplinar.

Anexo A

[14]ISE - Integrated Software Environment - é um ambiente de *software* integrado para o desenvolvimento de sistemas digitais da empresa Xilinx®.

Instruções para o download e a instalação do ModelSim®: Acesse o site da Xilinx digitando www.xilinx.com. Após a abertura da página inicial, com o botão esquerdo do *mouse* pressione sobre "download center" (em destaque na Figura 1).

Antes de iniciar o *download*, leia atentamente a licença de uso do *software*, denominada ModelSim Xilinx Edition III License Agreement (Figura 3). Após a leitura dos termos da licença e havendo concordância em relação a eles, com o botão esquerdo do *mouse* pressione sobre "I Agree" (em

Figura 1 - Imagem da tela inicial da Xilinx

Na sequência, a navegação será dirigida para a página de *downloads*, ilustrada na Figura 2. Na base desta página, encontra-se o link para o ModelSim Xilinx Edition. Com o botão esquerdo do *mouse*, selecione Download ModelSim XE (em destaque na Figura 2).

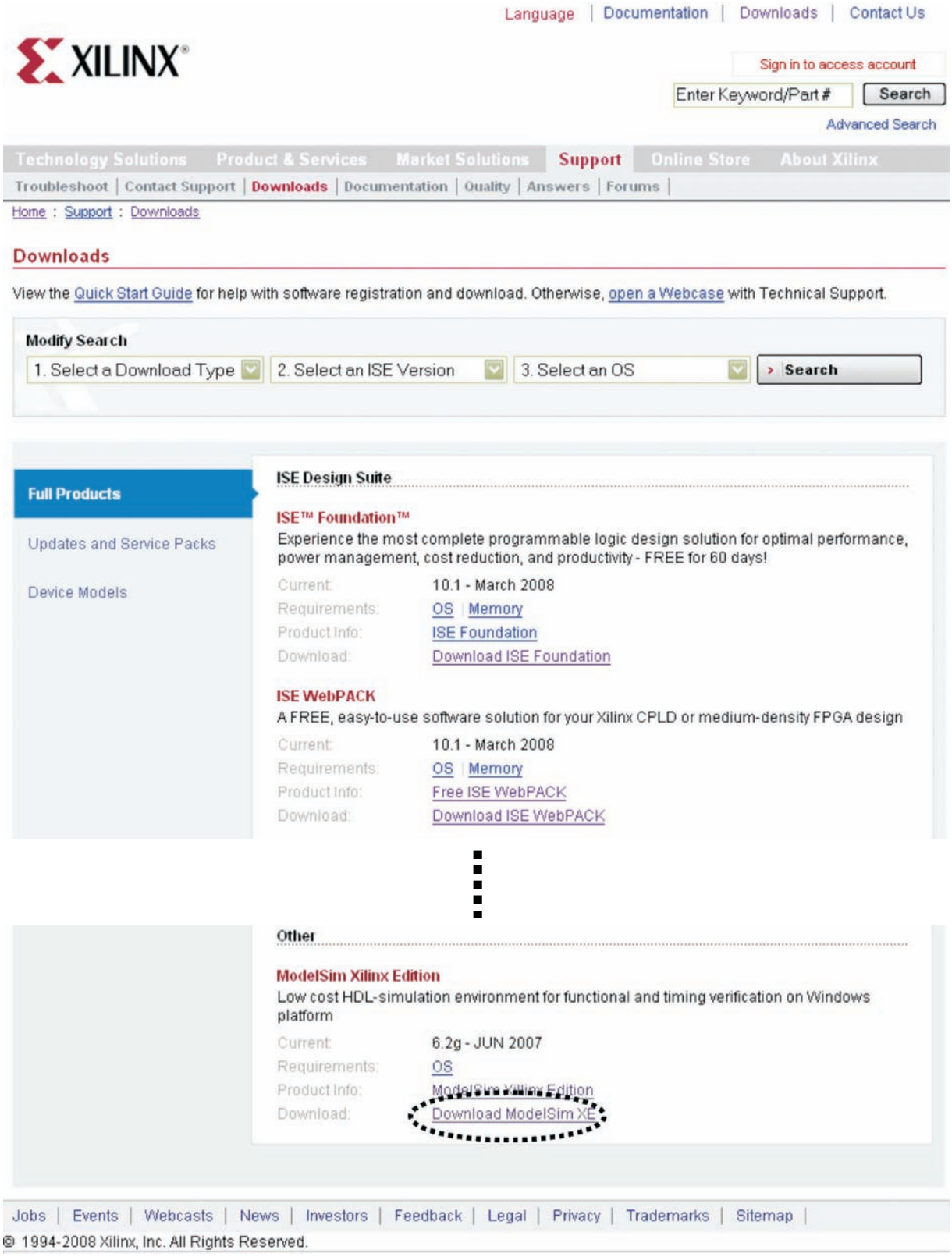


Figura 2 - Imagem parcial da tela de downloads da Xilinx

destaque na Figura 3).

Após a concordância com os termos da licença, a seguinte janela, ilustrada na Figura 4, permitirá o acesso ao download do ModelSim Xilinx Edition III. Para tanto, selecione, com o botão esquerdo do *mouse*, Version 6.3 (em destaque na Figura 4).

Figura 3 - Imagem parcial da tela com o acordo de licença ModelSim Xilinx Edition III License Agreement

Utilizando uma ferramenta para *download* disponível em seu computador, selecione o local (pasta) para salvar o arquivo compactado mxe_3_6.3c.zip, o qual contém a instalação da Versão 6.3. A Figura 5 ilustra o andamento do *download* do arquivo em questão.

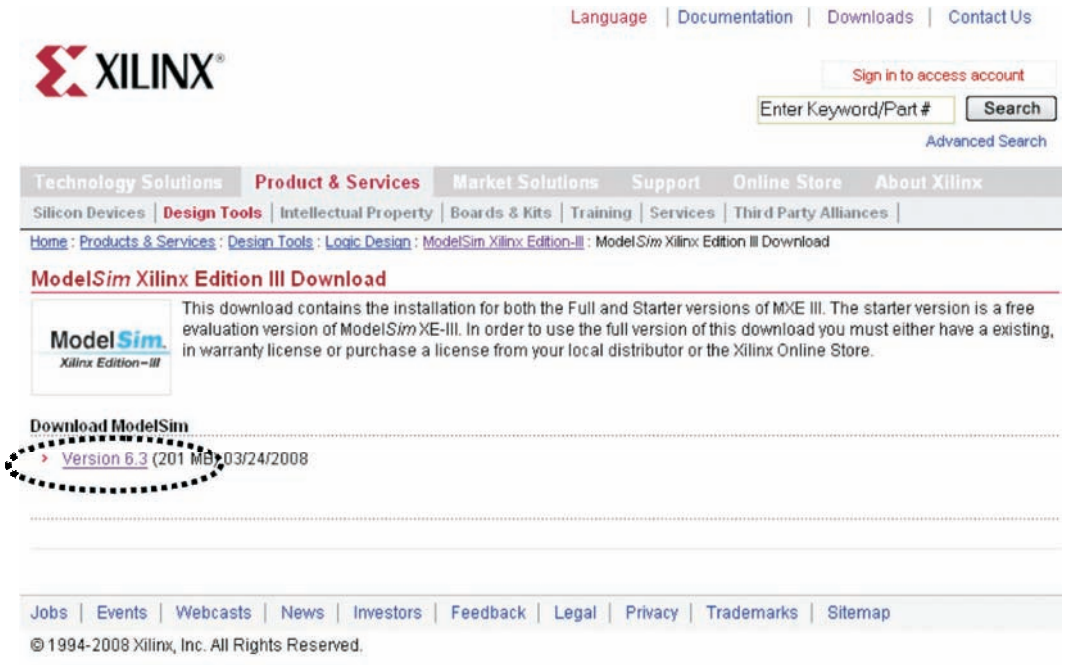


Figura 4 - Imagem da tela de seleção de *download* do ModelSim Xilinx Edition III Version 6.3

O próximo passo é iniciar a instalação do ModelSim. Para tanto, pressione o botão esquerdo do *mouse* sobre o ícone , presente no local (pasta) no qual o arquivo está gravado.

Dado que o arquivo de instalação está compactado, um programa descompactador de arquivo

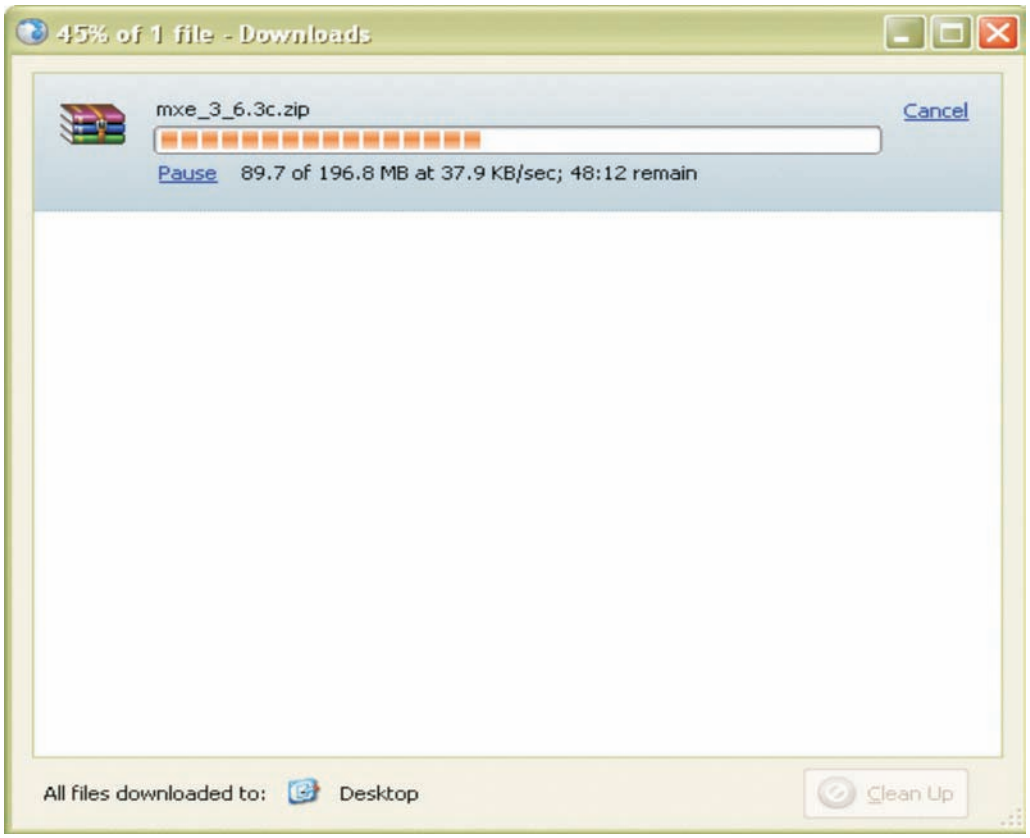


Figura 5 - Imagem da tela de *download* do "mxe_3_6.3c.zip"

disponível em seu computador iniciará sua execução. A Figura 6 ilustra esse procedimento, executado pelo programa Win.

Uma nova janela do descompactador abrirá, solicitando um local para gravar os arquivos do ModelSim após a descompactação (Figura 8). Após definido o local, pressione o botão esquerdo do *mouse* em "OK".

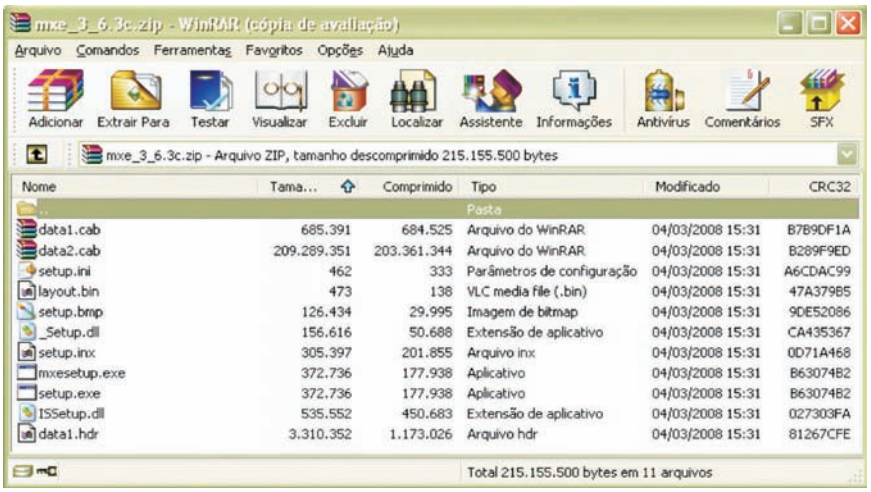


Figura 6 - Imagem da tela inicial do WinRAR

Figura 7 - Imagem parcial da tela principal do WinRAR

Após ter sido concluída a descompactação dos arquivos, abra a pasta destino dos arquivos e execute a instalação utilizando o arquivo "mxesetup.exe" (utilize o botão esquerdo do *mouse*).

A tela ilustrada na Figura 9 aparecerá. Em seguida, pressione o botão esquerdo do *mouse* sobre

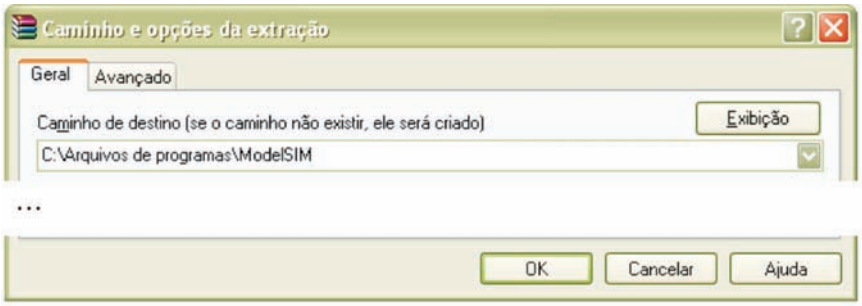


Figura 8 - Imagem parcial da tela principal do WinRAR. Caminho para instalação: "C: \Arquivos de programas\ModelSim"

"Next>".

Uma nova janela (Figura 10) abrirá solicitando que seja escolhida a versão do ModelSim a ser instalada.

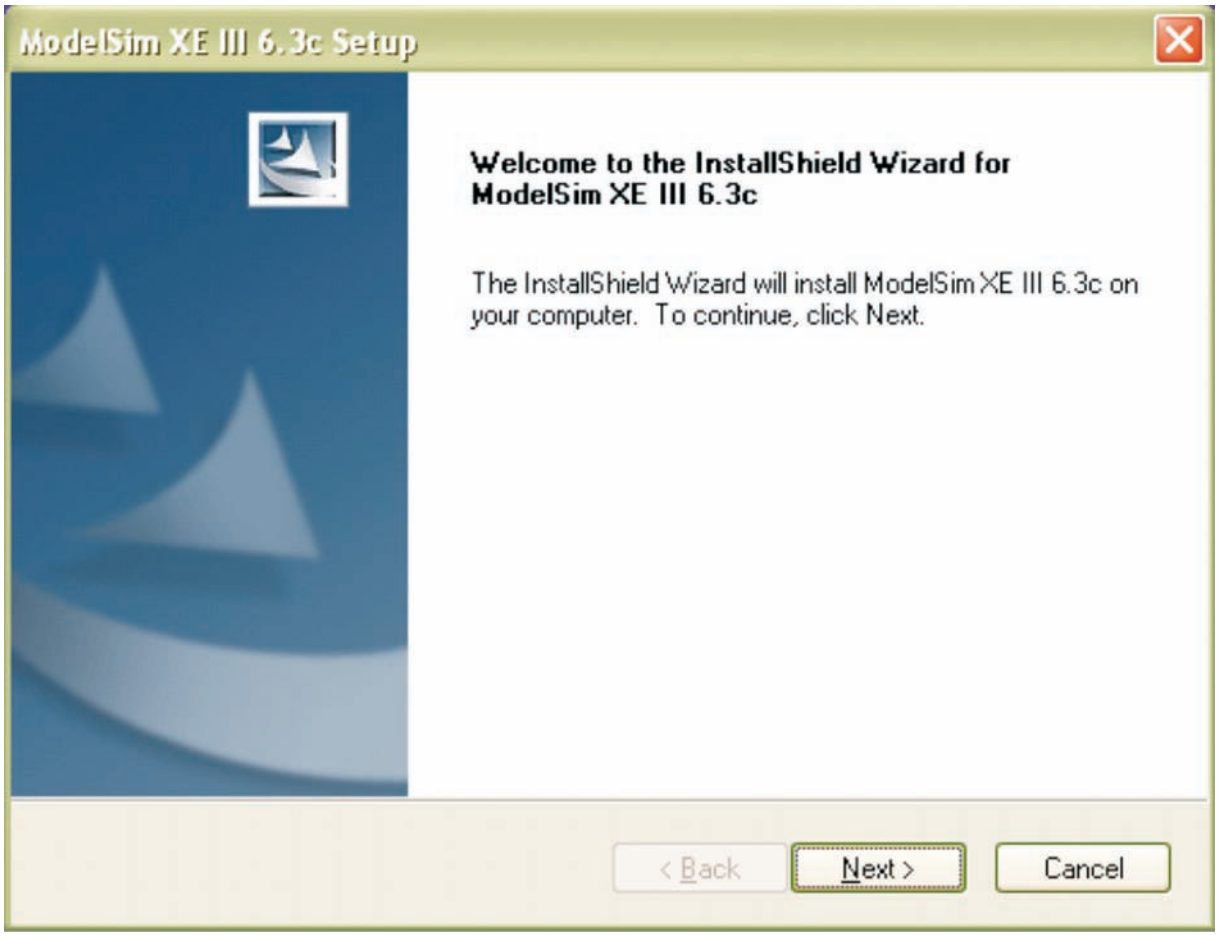


Figura 9 - Janela inicial da instalação do ModelSim XE 6.3c (*InstallShield Wizard*)

destino é "C: \Modeltech_xe_starter". Caso a instalação deva ser feita em outro local, clique sobre "Browse..." e defina outro local. Se estiver de acordo com o local padrão, pressione o botão esquerdo

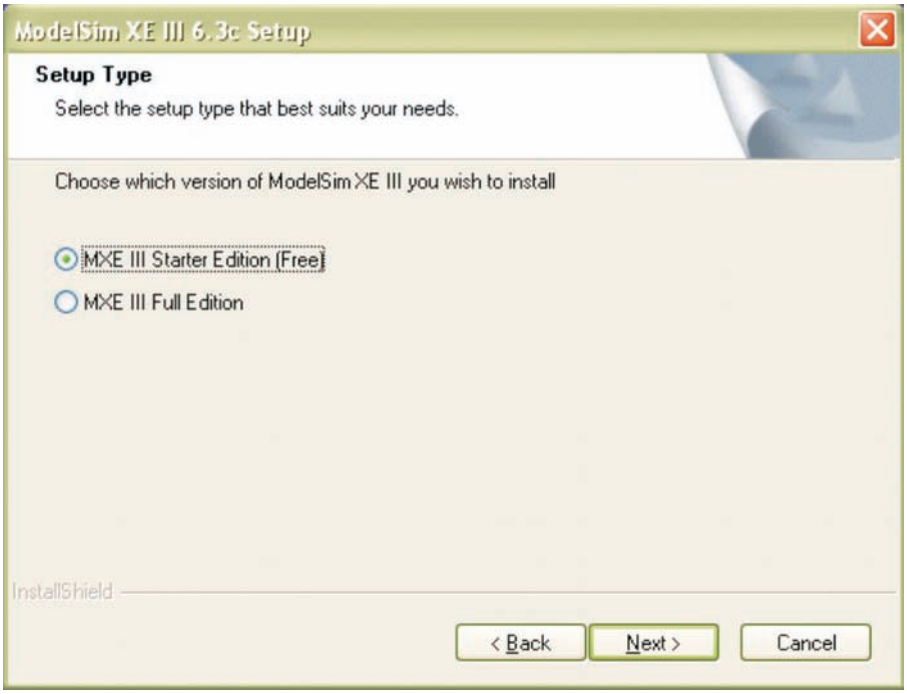


Figura 10 - Janela de seleção da versão a ser instalada do ModelSim XE 6.3c

do *mouse* sobre "Next>" (Figura 12).

Após selecione as bibliotecas a serem instaladas: "VHDL Custom". Para prosseguir, pressione o botão esquerdo do *mouse* sobre "Next>" (Figura 13).

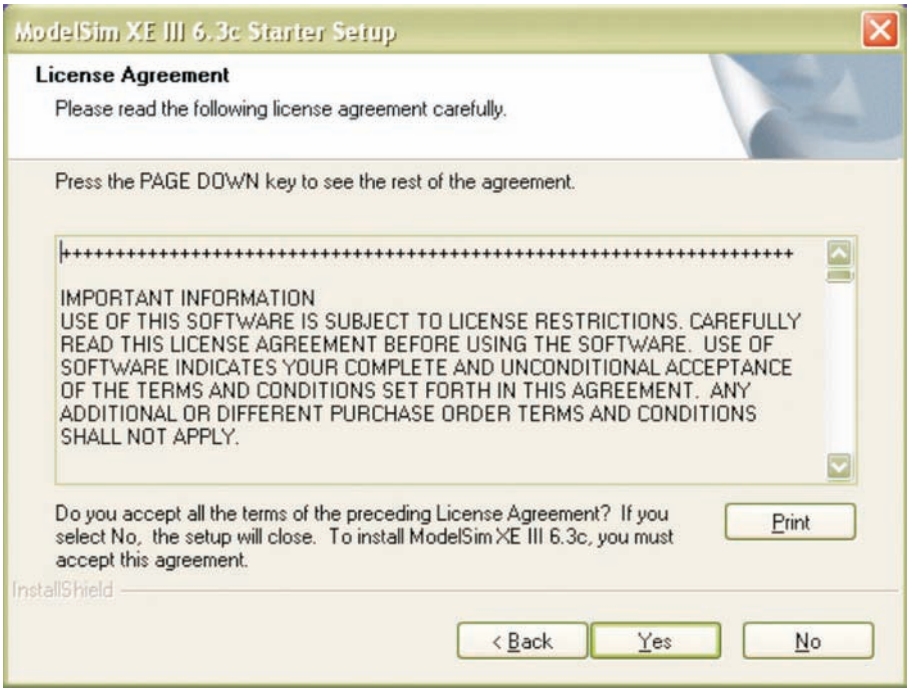


Figura 11 - Imagem parcial do acordo de licença ModelSim XE III 6.3c

Na próxima janela da instalação do ModelSim XE III 6.3c (Figura 14), mantenha selecionadas todas as bibliotecas para simulações no ModelSim e pressione o botão esquerdo do *mouse* sobre "Next>".

Na sequência, é sugerido uma pasta *(folder)* para o programa em instalação (Figura 15). Caso

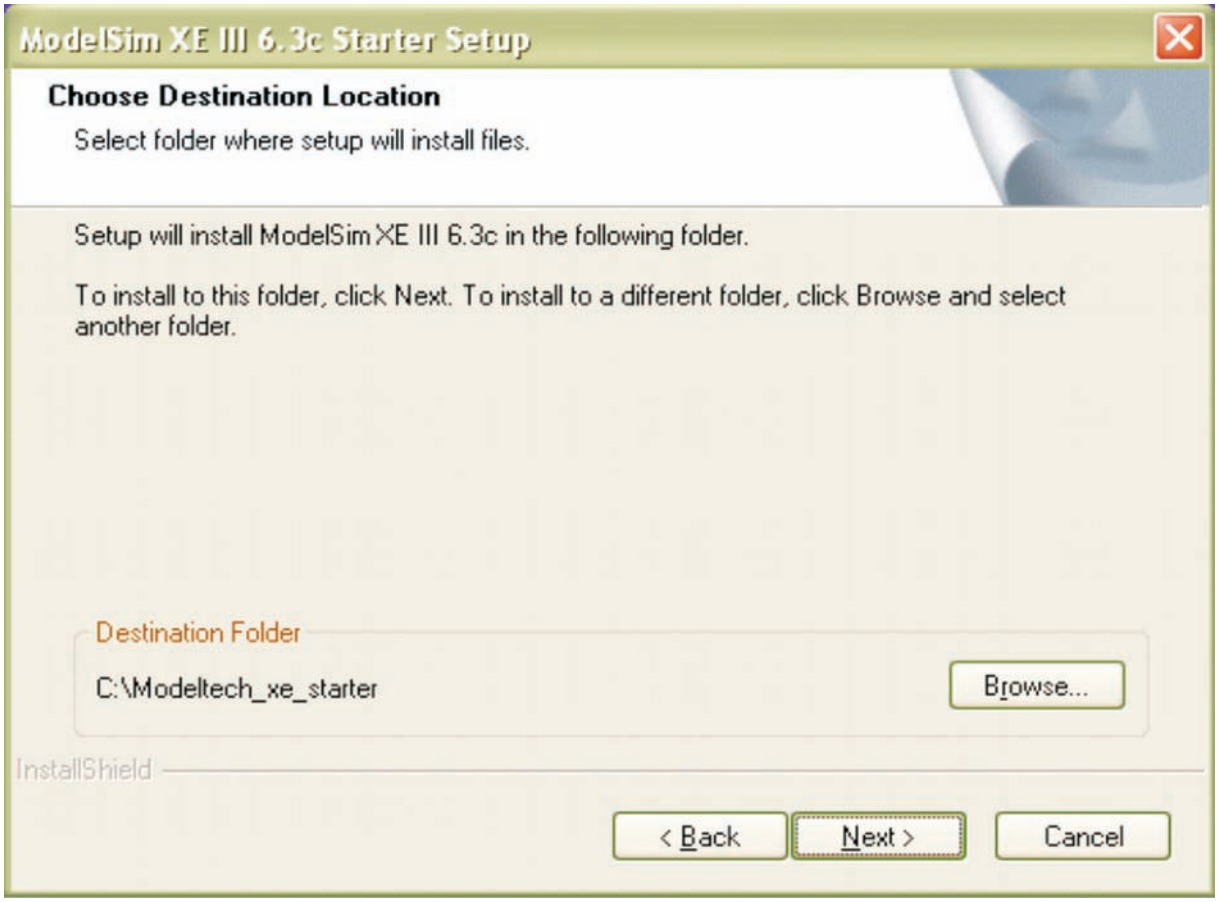


Figura 12 - Janela para definição para a pasta de instalação do ModelSim XE III 6.3c

concorde, mantenha como "Program Folder" a sugestão ModelSim XE III 6.3c. Em seguida, pressione o botão esquerdo do *mouse* sobre "Next>". O procedimento inicia a instalação (Figura 16).

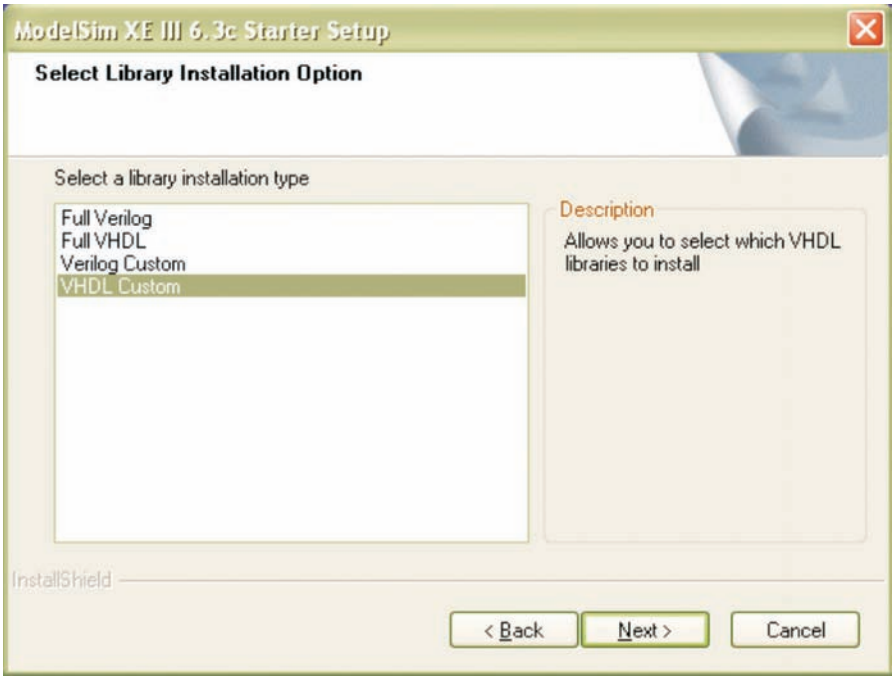


Figura 13 - Janela para seleção das bibliotecas do ModelSim XE III 6.3c a serem instaladas

Após gravar todos os arquivos do ModelSim na pasta destino, o programa instalador sugere a criação de um atalho para o programa que ficará disponível no *desktop* de seu computador. Caso concorde, pressione o botão esquerdo do *mouse* sobre "Sim" (Figura 17).

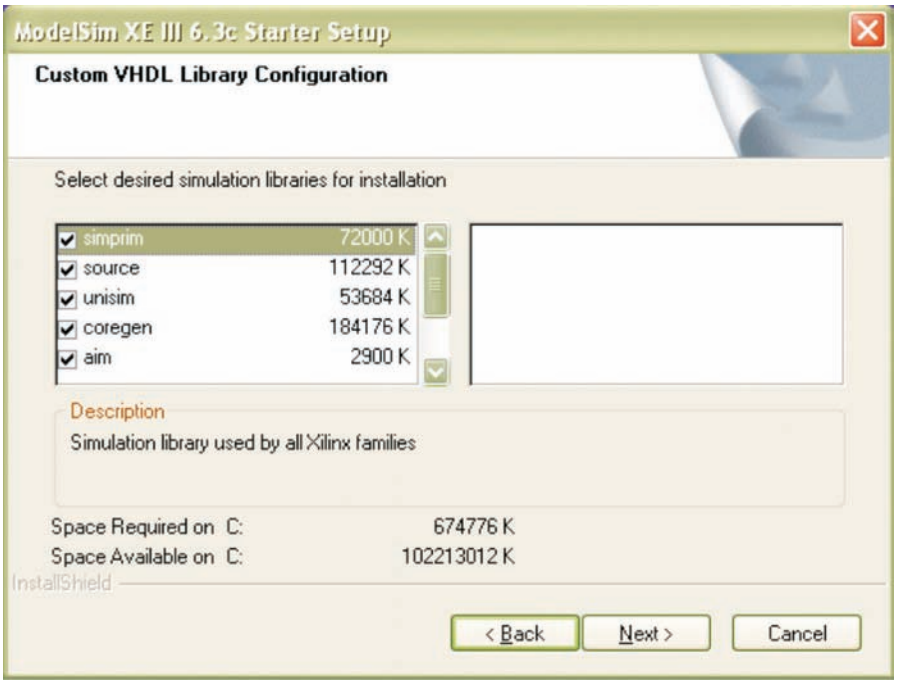


Figura 14 - Janela para seleção das bibliotecas para simulações no ModelSim XE III 6.3c a serem instaladas

Na sequência, um questionamento sobre a adição do diretório de executáveis do ModelSim no *path* de seu computador. Esse procedimento é útil no caso de execução de simulações a partir de arquivos em lote (*batch files*). Caso concorde, pressione o botão esquerdo do *mouse* sobre "Sim"

Figura 15 - Janela para seleção do "Program Folder" para o ModelSim XE III 6.3c

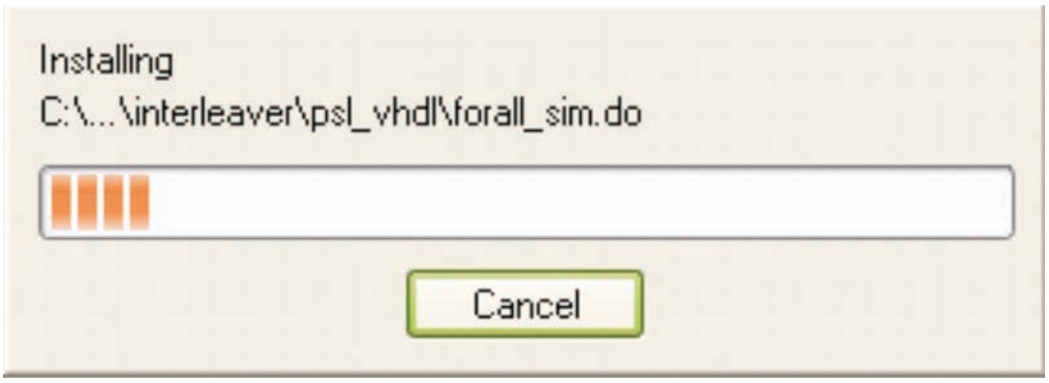


Figura 16 - Janela de andamento da instalação do ModelSim XE III 6.3c

(Figura 18).

O próximo passo, ilustrado pela Figura 19, é a solicitação de licença para o uso do ModelSim. Para avançar, pressione o botão esquerdo do *mouse* sobre "Sim".

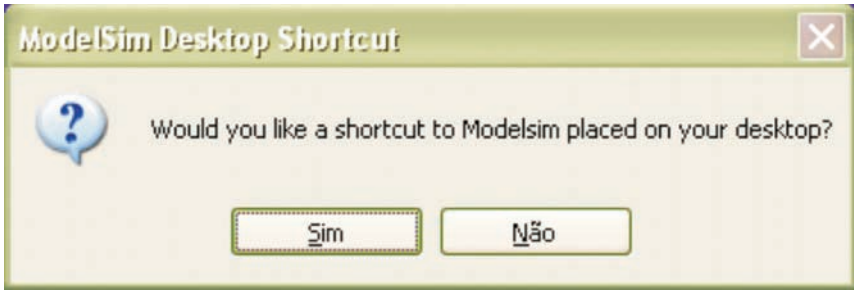


Figura 17 - Criação de um atalho para acesso ao ModelSim XE III 6.3c

A tela final da instalação é apresentada (Figura 20). Pressione o botão esquerdo do *mouse* sobre "Finish" e siga as instruções para solicitação da licença.

Para solicitar uma licença para uso do ModelSim XE III 6.3c, é necessário um cadastro na Xilinx. Esse cadastro é feito a partir da janela que abre no *browser* de seu computador (Figura 21). Faça seu

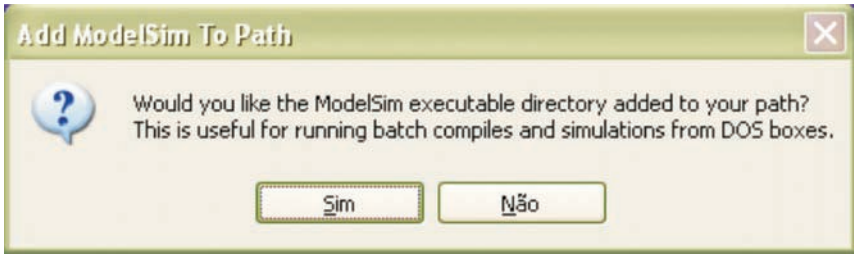


Figura 18 - Adição do diretório com executáveis do ModelSim XE III 6.3c ao *path*

registro pessoal na Xilinx selecionando, com o botão esquerdo do *mouse*, a opção "Register" (em destaque na Figura 21).

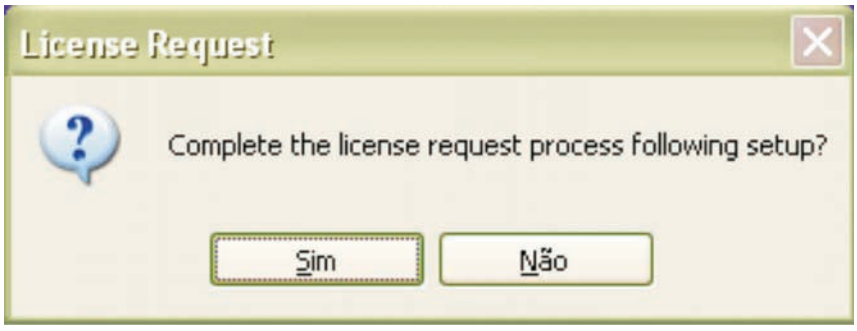


Figura 19 - Janela para solicitação de licença para o uso do ModelSim XE III 6.3

Após o preenchimento do cadastro (registro), você receberá via e-mail um arquivo denominado "license.dat". Grave este arquivo para a pasta "win32xoem".

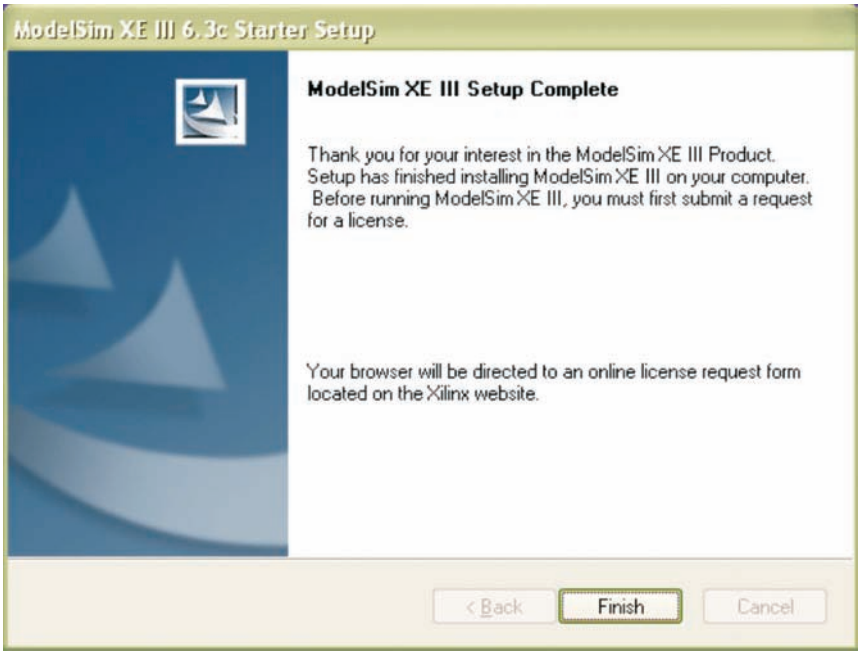


Figura 20 - Janela final da instalação do ModelSim XE III 6.3

Para que seja possível utilizar o ModelSim, é necessário validar a licença. Para tanto, siga a sequência de ações ilustrada na Figura 22 para executar o "Licensing Wizard", utilizando sempre o botão esquerdo do *mouse*.

No início da execução do Licensing Wizard, a janela ilustrada na Figura 23 é aberta com algumas

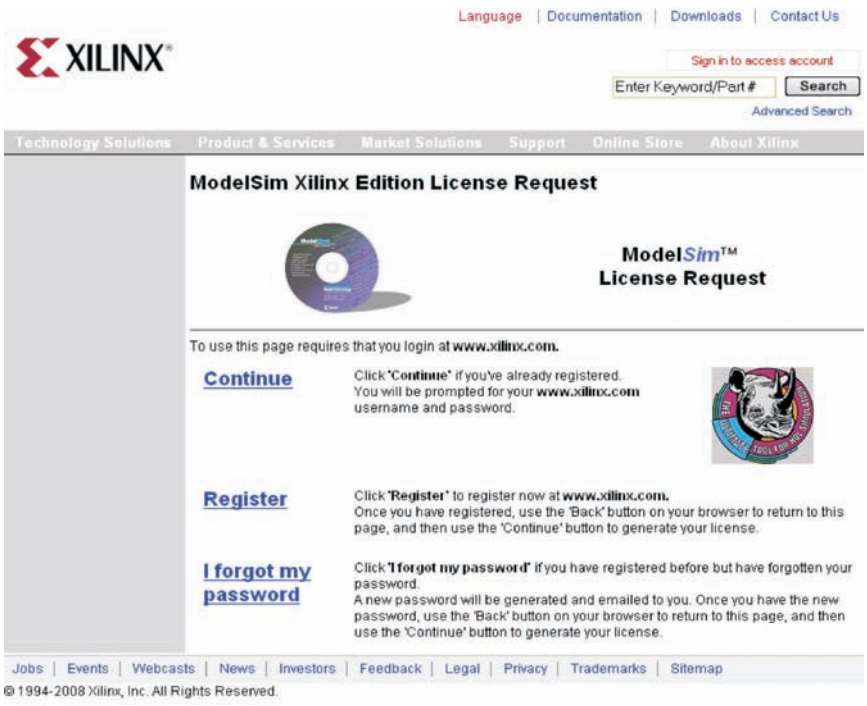


Figura 21 - Site da Xilinx para solicitação de licença de uso para o ModelSim XE III 6.3

informações sobre a função deste programa. Com o botão esquerdo do *mouse*, pressione sobre "Continue" (em destaque na Figura 23).

No primeiro passo, deve ser informado o local onde está gravado o arquivo de licença "license.dat" enviado pela Xilinx e recebido por e-mail. Na janela "License File Location" (Figura 24), digite o caminho para acesso à licença, ou utilize o botão "Browse" para montar esse caminho.

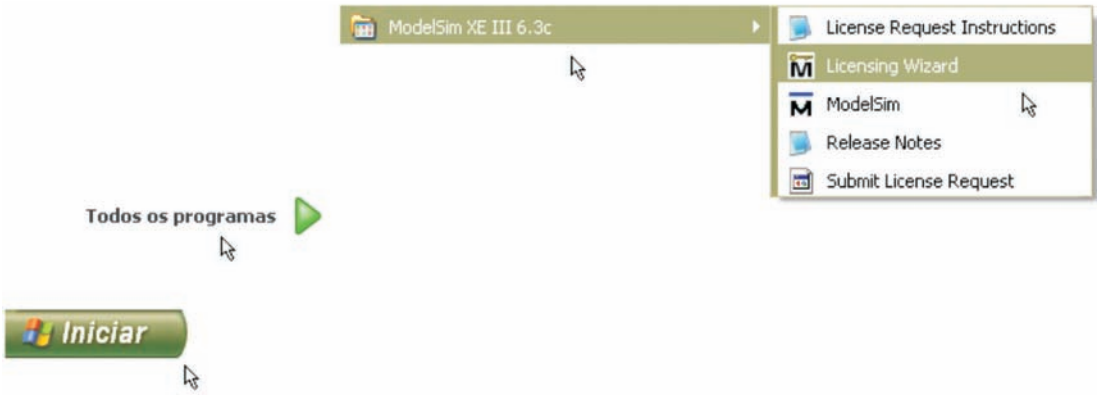


Figura 22 - Sequência de ações para acessar o "Licensing Wizard" (da esquerda para a direita)

Após a correta indicação do local onde o arquivo de licença está gravado, a licença é validada (segundo passo). O terceiro passo é a verificação das variáveis de ambiente do Windows®. Pressione o botão esquerdo do *mouse* sobre "Yes" (em destaque na Figura 25), caso deseje que as modificações

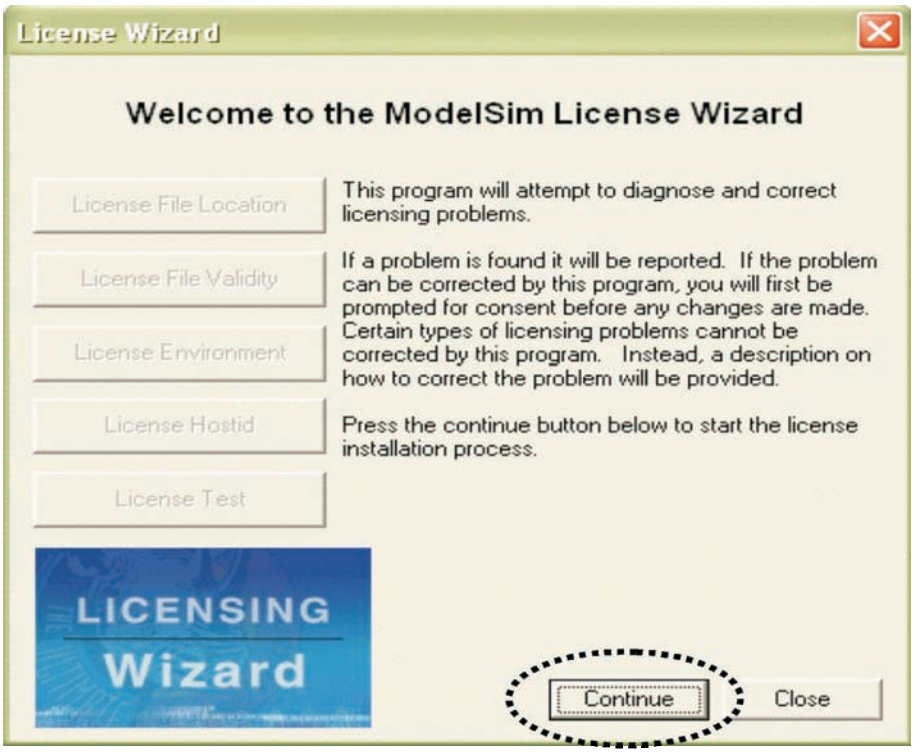


Figura 23 - Janela inicial do "Licensing Wizard"

sejam automáticas. Se as alterações serão feitas manualmente, escolha "No".

Feitas as modificações nas variáveis de ambiente de forma automática, a seguinte janela será aberta (Figura 26), indicando que a atualização ocorreu com sucesso. Pressione com o botão esquerdo

Figura 24 - Indicação da localização do arquivo "license.dat" (*Step 1: Locarting license file*)

do *mouse* sobre "OK" (em destaque na Figura 26).

Após o "OK" (Figura 26), o *License Wizard* indica que deve ser executado novamente para realizar um diagnóstico da licença. Para tanto, siga a sequência de ações ilustrada na Figura 27 para executar o *Licensing Wizard* novamente.

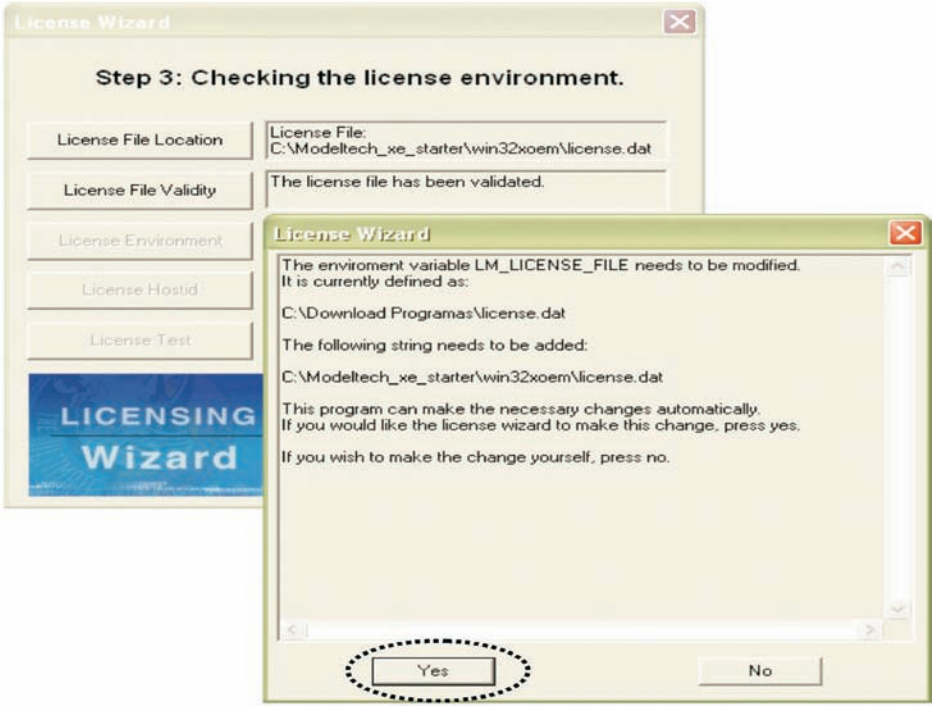


Figura 25 - Janelas indicando a validação da licença (*License File Validity*) e janela solicitando permissão para modificar as variáveis de ambiente do Windows®

Como resposta ao diagnóstico realizado, o *License Wizard* abre uma janela informando que o teste de licença ocorreu com sucesso (Figura 28). Pressione com o botão esquerdo do *mouse* sobre "OK" (em destaque na Figura 28).

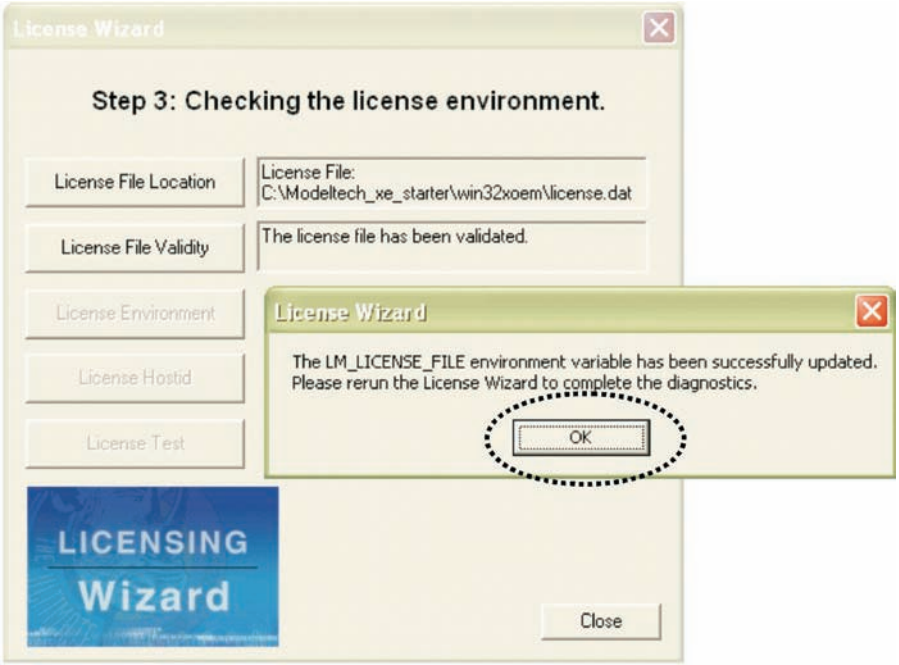


Figura 26 - Janela indicando que as variáveis de ambiente do Windows® foram atualizadas com sucesso

A Figura 29 ilustra a finalização do *License Wizard*.

Para iniciar o ModelSim, utilize o atalho representado pelo ícone , na área de trabalho de seu computador.

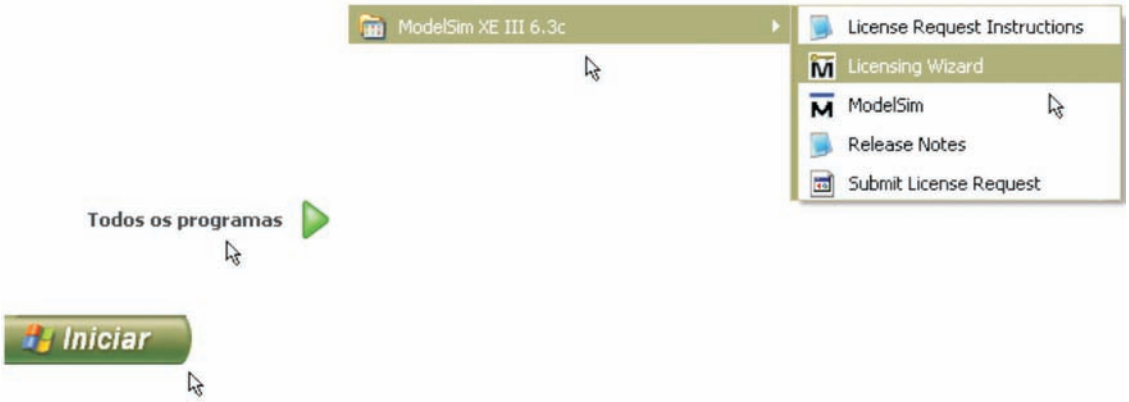


Figura 27 - Sequência de ações para acessar o "Licensing Wizard" (da esquerda para a direita)

Sugere-se como *testbench* o código VHDL transcrito a seguir.

2. Neste exemplo, é apresentado um *buffer tri-state*, cuja função é desacoplar eletricamente

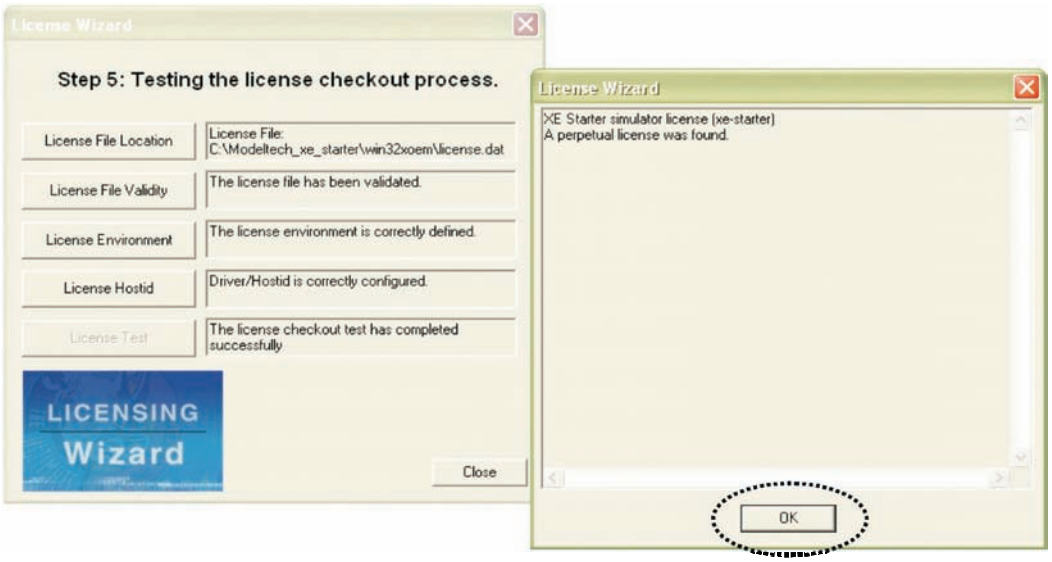


Figura 28 - Janela de finalização do teste da licença para uso do ModelSim XE III 6.3c

conexões de duas ou mais saídas simultâneas.

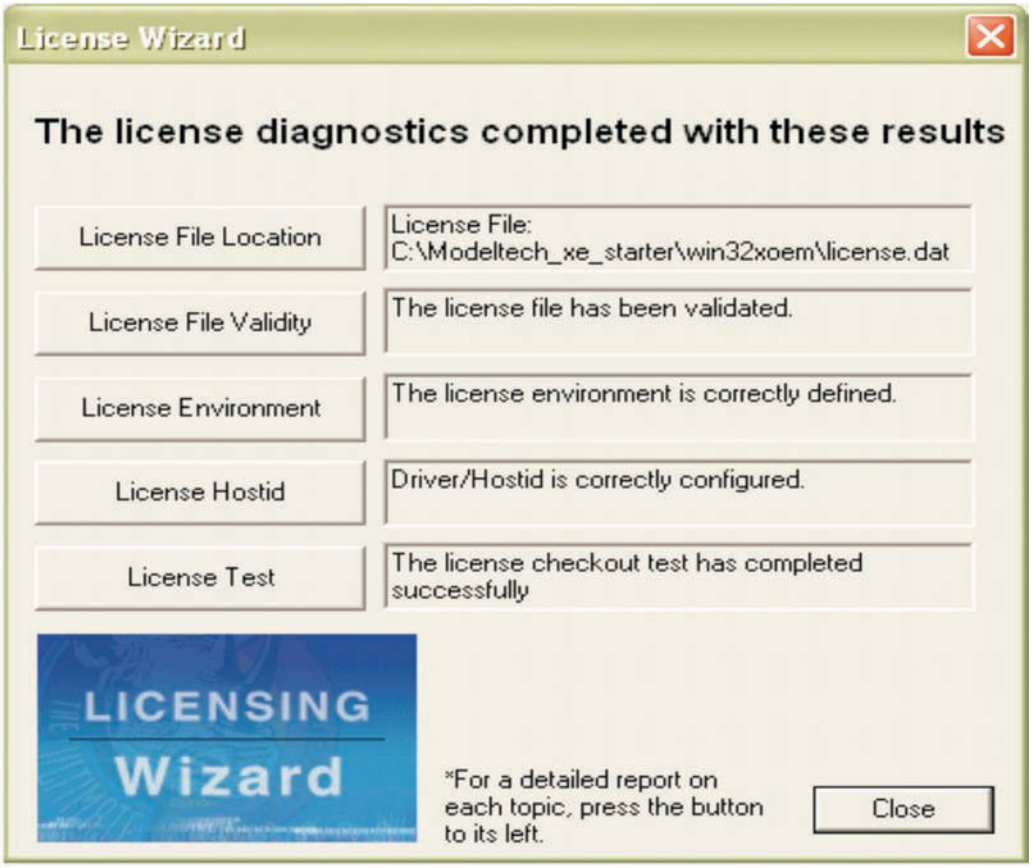


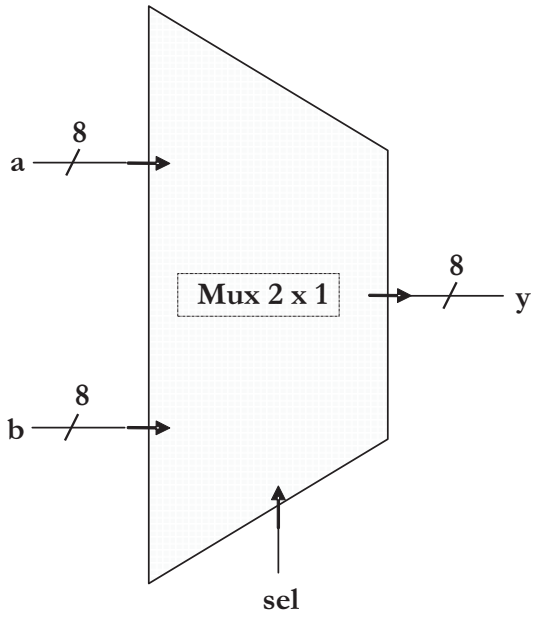
Figura 29 - Resumo das operações concluídas pelo *License Wizard*

Sugere-se como *testbench* para *buffer tri-state* acoplado à saída do  . 2x1 o código transcrito a seguir.

Anexo B

Duas descrições são apresentadas com uso da declaração WHEN/ELSE, conforme os códigos VHDL transcritos como segue.

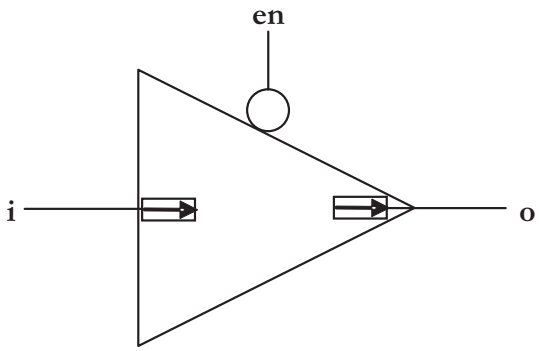
1. Este exemplo apresenta um multiplexador 2x1 (dois barramentos de 8 bits x um barramentos de 8 bits)




```
-----------------------------------------------
--  Circuito: multiplexador 2x1:(mux8b_2x1.vhd)
--                 sel8b Seleção da entrada
--                 a8b Entrada, sel = 0
--                 b8b Entrada, sel = 1
--                 y8b Saida (WHEN/ELSE)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
 -----------------------------------------------
ENTITY mux8b2x1 IS
    PORT ( sel8b   : IN STD_LOGIC;
           a8b     :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           b8b     :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           y8b     : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
    END mux8b2x1;
-------------------------------------------------
ARCHITECTURE mux8b OF mux8b2x1 IS
   BEGIN
      y8b <= a8b WHEN sel8b='0' ELSE b8b;
   END mux8b
-------------------------------------------------
```

```
-- ****************************************
-- Testbench para simulacao Funcional do
-- Circuito: multiplexador 2x1:(mux8b_2x1.vhd)
--            sel8b Selecao da entrada
--            a8b Entrada, sel = 0
--            b8b Entrada, sel = 1
--            y8b Saida (WHEN/ELSE)
-- ****************************************
ENTITY testbench6a IS END;
-------------------------------------------
-- Testbench para mux8b_2x1.vhd
-- Validacao assincrona
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_sel8b_2x1 OF testbench6a IS
-----------------------------------
-- Declaracao do componente mux2x1
-----------------------------------
component mux8b2x1
    PORT ( sel8b   :IN STD_LOGIC;
           a8b     :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           b8b     :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           y8b     :OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
end component;

signal tb_sel8b         :std_logic;
signal tb_a8b, tb_b8b   :STD_LOGIC_VECTOR (7 DOWNTO 0);
Begin

mux8b: mux8b2x1 PORT MAP (sel8b => tb_sel8b, a8b => tb_a8b,
 b8b => tb_b8b, y8b => open);
estimulo: PROCESS
   begin
      tb_sel8b <= '0';
      tb_a8b <= (OTHERS => '0');
      tb_b8b <= (OTHERS => '0');
      wait for 10 ns;
      loop
            tb_a8b <= tb_a8b + "00000110";
            WAIT FOR 10 ns;
            tb_sel8b <= '1';
            WAIT FOR 10 ns;
            tb_b8b <= tb_b8b + "00000111";
            WAIT FOR 10 ns;
            tb_sel8b <= '0';
            WAIT FOR 10 ns;
         end loop;
    end PROCESS estimulo;
end tb_sel8b_2x1.
```

Anexo C




```
-------------------------------------------
-- Circuito: Buffer 3 State: (buff_3stat.vhd)
--                  en saida
--                  i  Entrada
--                  o  Saida
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY buff3stat IS
      PORT (en  : IN STD_LOGIC;
            i   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            o   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
      END buff3stat;
-------------------------------------------
ARCHITECTURE buff_3stat OF buff3stat IS
    BEGIN
       o <= i WHEN (en='1') ELSE
              (OTHERS => 'Z');
    END buff_3stat;
-------------------------------------------
```

```
-- ****************************************
-- Testbench para simulacao Funcional dos
-- Circuito: multiplexador 2x1:(mux8b_2x1.vhd)
--                 sel8b Selecao da entrada
--                 a8b Entrada, sel = 0
--                 b8b Entrada, sel = 1
--                 y8b Saida (WHEN/ELSE)
-- Circuito: Buffer tri-state: (buff_3stat.vhd)
--                  en saida
--                  i  Entrada
--                  o  Saída
-- ****************************************
ENTITY testbench6a IS END;
-------------------------------------------
-- Testbench para mux8b_2x1.vhd
-- Validacao assincrona
-------------------------------------------
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_sel8b_2x1 OF testbench6a IS
-----------------------------------
-- Declaracao do componente mux2x1
-----------------------------------
component mux8b2x1
    PORT ( sel8b      :IN STD_LOGIC;
           a8b        :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           b8b        :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           y8b        :OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
end component;

-----------------------------------
-- Declaracao do buffer tri-state
-----------------------------------

component buff3stat
      PORT (en  : IN STD_LOGIC;
            i   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            o   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
end component;

signal tb_sel8b         :std_logic;
signal tb_a8b, tb_b8b   :STD_LOGIC_VECTOR (7 DOWNTO 0);
signal tb_i             :STD_LOGIC_VECTOR (7 DOWNTO 0);

Begin

mux8b: mux8b2x1 PORT MAP (sel8b => tb_sel8b, a8b => tb_a8b,
             b8b => tb_b8b, y8b => tb_i);

buff3: buff3stat PORT MAP (en => tb_sel8b, i => tb_i, o => open);

estimulo: PROCESS
   begin
      tb_sel8b <= '0';
      tb_a8b <= (OTHERS => '0');
      tb_b8b <= (OTHERS => '0');
      wait for 10 ns;
      loop
          tb_a8b <= tb_a8b + "00000110";
          WAIT FOR 10 ns;
          tb_sel8b <= '1';
          WAIT FOR 10 ns;
          tb_b8b <= tb_b8b + "00000111";
          WAIT FOR 10 ns;
          tb_sel8b <= '0';
          WAIT FOR 10 ns;
        end loop;
    end PROCESS estimulo;
end tb_sel8b_2x1.
```

Duas soluções alternativas para a implementação do multiplexador 4x1, exemplificado na seção, são apresentadas com utilização de WITH/SELECT/WHEN (*selected* WHEN), conforme os códigos VHDL transcritos como segue.

Na descrição acima, a entrada para seleção (sel) poderia ter sido declarada como um inteiro e, neste caso, o código VHDL seria o seguinte:

```
-----------------------------------------------
--  Circuito: multiplexador 4x1: (mux2_4x1.vhd)
--        sel (1:2) Selecao da entrada
--        a Entrada, sel = 00
--        b Entrada, sel = 01
--        c Entrada, sel = 10
--        d Entrada, sel = 11
--        y Saida (WITH/SELECT/WHEN)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY mux4x1 IS
PORT (sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
     a, b, c, d: IN STD_LOGIC;
     y: OUT STD_LOGIC);
END mux4x1;
-----------------------------------------------
 ARCHITECTURE mux2 OF mux4x1 IS
 BEGIN
     WITH sel SELECT
          y <= a WHEN "00",    -- nota: ";" troca por ","
               b WHEN "01",
               c WHEN "10",
               d WHEN OTHERS;         -- não é permitido d
 WHEN "11"
 END mux2.
 -----------------------------------------------
```

Anexo D

```
-----------------------------------------------
--  Circuito: multiplexador 4x1:(mux3_4x1.vhd)
--               sel (1:2) Selecao da entrada
--               a Entrada, sel = 00
--               b Entrada, sel = 01
--               c Entrada, sel = 10
--               d Entrada, sel = 11
--               y Saida (WITH/SELECT/WHEN)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY mux4x1 IS
PORT (sel: IN INTEGER RANGE 0 TO 3;
      a, b, c, d: IN STD_LOGIC;
      y: OUT STD_LOGIC);
END mux4x1;
-----------------------------------------------
ARCHITECTURE mux3 OF mux4x1 IS
BEGIN
    WITH sel SELECT
         y <= a WHEN 0,
              b WHEN 1,
              c WHEN 2,
              d WHEN 3;-- aqui, 3 ou OTHERS são equivalentes
END mux3;               -- e são testados para todas as opções.

-----------------------------------------------
```

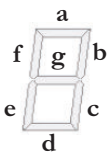
A título de exemplo, uma descrição alternativa para o multiplexador 2x1 é apresentada com uso da declaração CASE, contendo também a classe combinacional WHEN utilizada dentro de um processo.

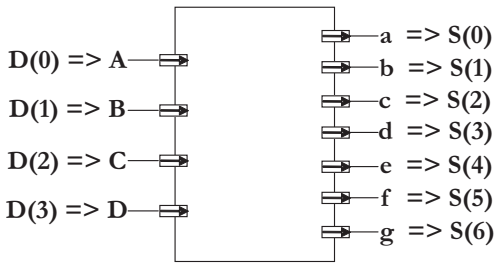
Anexo E

```
-------------------------------------------------
--  Circuito: multiplexador 2x1s:(mux_2x1.vhd)
--            sel Selecao da entrada
--            a Entrada, s = 0
--            b Entrada, s = 1
--            y Saida y = nsel.a + sel.b
-------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mux2x1s is
    port (sel   : in STD_LOGIC;
          a,b   : in STD_LOGIC;
          y     : out STD_LOGIC);
end mux2x1s;

architecture comport of mux2x1s is
begin
    mux: process (sel,a,b)
     begin
        case sel is
            when '0' => y <= a;
            when '1' => y <= a;
            when others=> y <= '0';
        end case;
    end process; --fim mux
end comport.
```

Um *display* de 7 segmentos é utilizado em relógios, calculadoras e outros dispositivos para mostrar números decimais (dígitos). Um dígito é mostrado iluminando um subconjunto dos 7 segmentos.

Um decodificador de 7 segmentos tem como entrada 4 bits (BCD) e o código 7 segmentos como saída de 7 bits.





Entradas D(3-0)				Saídas S(6-0)						
D	C	B	A	g	f	e	d	c	b	a
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0
0	1	0	1	1	1	0	1	1	0	1
0	1	1	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

```
-----------------------------------------------
-- Circuito: decodificador 7seg:(deco2_7seg.vhd)
-- D  Entrada BCD (A,B,C,D)
-- S  Saida & segmentos (a,b,c,d,e,f,g)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY seg7 IS
PORT (D       : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
      S       : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
END seg7;

ARCHITECTURE display OF SEG7 IS
BEGIN--gfedcba------------DCBD--------------------
s <=  "0111111" WHEN d = "0000" ELSE --0      a
      "0000110" WHEN d = "0001" ELSE --1    _ _ _
      "1011011" WHEN d = "0010" ELSE --2   |     |
      "1001111" WHEN d = "0011" ELSE --3  f|  g  |b
      "1100110" WHEN d = "0100" ELSE --4   |_ _ _|
      "1101101" WHEN d = "0101" ELSE --5   |     |
      "1111101" WHEN d = "0110" ELSE --6  e|     |c
      "0000111" WHEN d = "0111" ELSE --7   |_ _ _|
      "1111111" WHEN d = "1000" ELSE --8      d
      "1101111" WHEN d = "1001" ELSE --9
      "0000000"; --para entradas não BCD
END display.
-------------------------------------------------
```

REFERÊNCIAS

AMORE, Roberto d', VHDL: **Descrição e Síntese de Circuitos Digitais.** Rio de Janeiro: Livros Técnicos e Científicos Editora S.A. LTC, 2005. 259 p.

MAZOR, Stanley; LANGSTRAAT, Patricia. **A Guide to VHDL.** 2. ed. Massachusetts, USA: Kluwer Academic Publisher, 1995. 336 p.

PEDRONI, Volnei A. **Circuit desing with VHDL.** Massachusetts, USA: MIT Press, 2004. 363 p.

PERRY, Douglas L. **VHDL:** programming by examples. 4. ed. USA: McGraw-Hill, 2002. 476 p.

A diagramação deste livro foi realizada pela editora Feevale.
Fonte utilizada nos textos: Garamond