

13.1 Modelagem de Circuitos Combinacionais em VHDL

VHDL é uma sigla que contém uma sigla: *VHSIC Hardware Description Language*, sendo VHSIC abreviatura para *Very High Speed Integrated Circuits*. Como o nome diz, VHDL é uma linguagem de **descrição** de *hardware* (*HDL*). Neste, e nos três próximos laboratórios, usaremos primariamente construções para descrever a *estrutura* dos circuitos. As construções para definir *comportamento* serão vistas adiante.

Diferentemente de C, um programa em VHDL *descreve* uma estrutura, ao invés de *definir* ou *prescrever* uma computação. Neste sentido, um programa em C é uma “receita de bolo” com ingredientes (variáveis) e modo de preparar (algoritmo), enquanto que em VHDL um programa descreve a interligação de componentes. A partir da estrutura, o compilador `ghdl` gera um simulador para o circuito descrito pelo código fonte. Adiante veremos como definir o comportamento de um circuito, sem que seja necessário definir a sua estrutura.

13.1.1 Objetivos

São três os objetivos deste laboratório: (i) efetuar a modelagem estrutural em VHDL de multiplexadores, demultiplexadores e decodificadores; (ii) verificar, através de simulação, a corretude dos modelos daqueles circuitos; e (iii) iniciar o contato com a linguagem VHDL.

O trabalho pode ser efetuado em duplas. Não há entrega(s) neste laboratório.

13.1.2 Descrição gráfica de circuitos

Veja a Seção 9.1 de `vhdl.pdf`.

13.1.3 VHDL

VHDL é uma linguagem extremamente versátil e é usada para

- * a modelagem e simulação de circuitos;
- * especificação, projeto e síntese de circuitos;
- * descrever lista de componentes e de ligações; e
- * verificação de corretude: se especificação \Rightarrow implementação.

Tipos de dados

Veja a Seção 9.2.1 de `vhdl.pdf`.

Como se dá a compilação de VHDL?

Veja a Seção 9.2.4 de `vhdl.pdf`.

Mecanismo de simulação

Veja a Seção 9.2.5 de `vhdl.pdf`.

13.1.4 Da tarefa

Etapa 1 Copie para sua área de trabalho o arquivo com o código VHDL, e extraia seu conteúdo com os seguinte comandos:

- (1) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_estrutural.tgz`
- (2) expanda-o com `tar xzf l_estrutural.tgz`
- (3) o diretório `estrutural` será criado na expansão da *tarball*;
- (4) mude para aquele diretório com `cd estrutural`

O arquivo `packageWires.vhd` contém definições ou apelidos para nomes de sinais para facilitar a digitação, porque digitar `reg8` é mais econômico do que `bit_vector(7 downto 0)`. **packages** são descritas na Seção 9.2.2.

O arquivo `aux.vhd` contém os modelos das portas lógicas *not*, *and*, *or*, *xor*, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `estrut.vhd` contém um modelo para um multiplexador de duas entradas, *mux-2*. Este modelo serve de base para que você escreva os modelos para os componentes *mux-4*, *mux-8*, *demux-2*, *demux-4*, *demux-8*, *decod-2*, *decod-4* e *decod-8*, que foram vistos em sala e estão definidos na Seção 4.3 de `combin.pdf`.

O *script* `run_mux.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run_mux.sh` somente (re)compila o simulador e executa a simulação. Eventuais mensagens de erro são mostradas na tela. Com qualquer argumento o *script* também dispara a execução de `gtkwave`: `run_mux.sh 1 &`.

Para compilar e verificar eventuais mensagens de erro, diga, no terminal
 prompt: `./run_mux.sh`
 para executar a simulação e ver o diagrama de tempos no `gtkwave`, diga
 prompt: `./run_mux.sh 1 &`

O arquivo `v_mux.sav` contém definições para o `gtkwave` tais como a escala de tempo e sinais a serem exibidos na tela para a verificação do modelo *mux-2*.

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, mova o arquivo `gtkwaverc` para o seu `$HOME`, como um arquivo escondido (`$HOME/.gtkwaverc`) e edite as duas últimas linhas – aquelas com `fontname_signals fixed 10` e `fontname_waves terminus 9`. Os números podem ser aumentados/reduzidos para melhorar a legibilidade. `gtkwave` deve ser reinicializado com a nova configuração.

Modelo do multiplexador

Veja a Seção 9.2.3 de `vhdl.pdf`.

O modelo está pronto, e agora?

Veja a Seção 9.2.6 de `vhdl.pdf`.

Testbench para os multiplexadores

Veja a Seção 9.2.7 de `vhdl.pdf`.

Examinaremos primeiro os quatro modelos do multiplexador. O diagrama na Figura 13.15 mostra as ligações entre os componentes que você deve modelar e o processo que percorre o vetor de testes e gera as sequências de entradas-de-teste e saídas-de-teste. As entradas-de-teste (e , k , s) excitam seus modelos, que produzem saídas de acordo com suas especificações. As saídas-de-teste (mx , dm , dc) são comparadas com as saídas produzidas pelos modelos. O número de bits em e depende da largura do multiplexador, assim como o número de bits (n e m) dos sinais dm e dc .

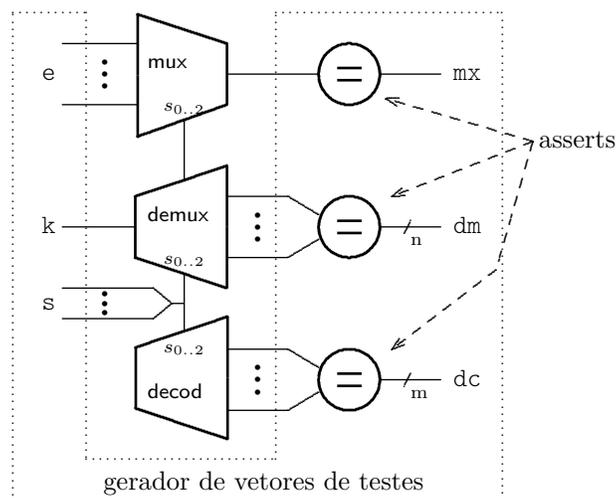


Figura 13.15: Ligações entre o *testbench* e os modelos.

O arquivo `tb_mux.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a correteza dos seus modelos dos multiplexadores (*mux-4*, *mux-8* com bits e *mux-8* com vetores de bits). Para simplificar a depuração de seu código, você deve verificar cada novo modelo assim que o código VHDL for completado.

Das mensagens de erro Em caso de erro de compilação ser detectado por `ghdl`, o `script run_mux.sh` aborta a compilação, e exibe as mensagens de erro emitidas pelo compilador. Estas mensagens são a melhor indicação que o compilador é capaz de emitir para ajudá-lo a encontrar o erro, e portanto **as mensagens de erro devem ser lidas**. Os programadores do `ghdl` dispenderam um esforço considerável para emitir mensagens de erro (relativamente) úteis. Não desperdice a preciosa ajuda que lhe é oferecida.

São usados três conjuntos de vetores de teste, um para cada largura de circuito. A seguir descrevemos os vetores de teste para circuitos de largura dois. Aqueles para largura quatro e oito são similares.

A arquitetura do TB declara os componentes que serão testados e um **record** que será usado para excitar os modelos. O registro `test_record_2` possui três campos e os valores destes campos devem ser atribuídos por você de forma a gerar todas (*todas?*) as combinações de entradas necessárias para garantir a corretude do seu modelo. O vetor de testes `test_array_2` contém os oito elementos necessários para excitar e verificar o `mux-2`, na primeira tarefa deste laboratório.

No `test_record_2`, os campos `s` e `mx` são de tipo bit ('0') e o campo `a` é um vetor de bits codificado em binário (`b"10"` – o prefixo `b` indica *binário*).

O campo `mx` é o bit com a saída esperada para um multiplexador quando os valores definidos em `s` e `a` são aplicados às entradas.

Programa 13.48: Vetor de valores de entrada para testar o modelo do `mux-2`.

```
% label=code:le_test]
— definicao do vetor de testes para MUX-2
type test_record_2 is record
  a : reg2;      — entrada para multiplexadores
  s : bit;       — entrada de seleção
  mx : bit;      — saída esperada do MUX
end record;
type test_array_2 is array(positive range <>) of test_record_2;

— vetor de testes
constant test_vectors_2 : test_array_2 := (
  —s,   a,   mx
  ('0',b"00", '0'), — transcricao da tabela verdade do mux-2
  ('0',b"01", '1'),
  ('0',b"10", '0'),
  ('0',b"11", '1'),
  ('1',b"00", '0'),
  ('1',b"01", '0'),
  ('1',b"10", '1'),
  ('1',b"11", '1'),
  —
  ('0',b"11", '1'), — nao alterar estes três últimos
  ('0',b"11", '1'), — estes bits indicam o final da simulação
  ('0',b"11", '1')
);

— troque a constante para FALSE para testar seus modelos
constant TST_MUX_2 : boolean := true;
```

A sequência de valores de entrada para os testes dos modelos é gerada pelo processo `U_testValues`, com um laço **for ... loop**. A variável de iteração itera no espaço definido pelo número de elementos do vetor de testes (`test_vectors'range`) – o atributo **'range** representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações. O elemento do vetor é atribuído à variável `v` e os todos os campos do vetor são então atribuídos aos sinais que excitam os modelos. Lembre que o processo `U_testValues` executa concorrentemente com o seu(s) modelo(s) e quando os sinais de teste são atribuídos no laço,

estes provocam alterações nos sinais dos modelos.

O **assert** no Programa 13.49 verifica se a saída observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto COM RELAÇÃO AOS VETORES DE TESTE QUE VOCÊ ESCREVEU.

Se você escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no seu modelo.

Programa 13.49: Mensagem de verificação de comportamento.

```
assert TST_MUX_2 or (saidaMUX2 = esperadaMUX)
  report "mux2:␣saida␣errada␣sel=" & B2STR(s0) &
    "␣saiu=" & B2STR(saidaMUX2) & "␣esperada=" & B2STR(esperadaMUX)
  severity error;
```

Se os valores de `saidaMUX2` e `esperadaMUX` diferem, a mensagem no Programa 13.49 é emitida na tela do terminal, indicando o erro:

```
tb_teste.vhd:201:7:@2ns:(assertion error):
mux2: saída errada sel=1 saiu=0 esperada=1.
```

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

A condição de teste do **assert** é

```
TST_MUX_2 or (saidaMUX2 = esperadaMUX)
```

A constante `TST_MUX_2` está definida como `true` logo abaixo do vetor de testes para os modelos de dois bits:

```
constant TST_MUX_2 : boolean := true;
```

Por causa do `true or (...)`, o **assert** não efetua a comparação entre a saída e o valor esperado. A razão para isso é diminuir a poluição na tela durante os testes dos modelos.

Para testar seu(s) modelo(s), altere a constante respectiva (`TST_MUX_2`, `TST_MUX_4` ou `TST_MUX_8`) para `false` e então verifique os resultados.

Etapa 2 Para executar a simulação sem invocar `gtkwave`, diga

```
prompt: ./run_mux.sh
```

Verifique se há alguma mensagem de erro; se sim, leia a mensagem e use seus miolos antes de chamar o professor.

Para executar a simulação com o `gtkwave`, diga

```
prompt: ./run_mux.sh 1 &
```

Diagramas de tempo do gtkwave

O script `run_mux.sh` pode disparar a execução do `gtkwave` para mostrar o diagrama de tempo dos circuitos. O diagrama de tempo mostra, de cima para baixo:

- MUX-2** um agrupamento com os sinais para o `mux-2`, com
 - entr_2** os dois bits de entrada para `mux-2` (laranja);
 - s0** bit de seleção (azul);
 - saidamux2** a saída do `mux-2` (verde);
 - esperadamux** a saída esperada para os muxes (amarelo);
- MUX-4** um agrupamento com os sinais para o `mux-4`, com
 - entr_4** os quatro bits de entrada para `mux-2` (laranja);
 - s0,s1** bits de seleção (azul);
 - saidamux4** a saída do `mux-4` (verde);
 - esperadamux** a saída esperada para os muxes (amarelo);
- MUX-8** mesmo que para `mux-4`, com largura 8
- MUX-8-vet** um agrupamento com os sinais para o `mux-8`, com
 - entr_8** vetor com os bits de entrada para `mux-8` (laranja);
 - s** vetor com três bits de seleção (azul);
 - saidamux8vet** a saída do `mux-4` (verde);
 - esperadamux** a saída esperada para os muxes (amarelo);

Note que **é você quem deve ajustar a saída esperada** para cada um dos circuitos nos vetores de teste. O projetista dos modelos é responsável por escrever os vetores de teste e portanto sua tarefa é ajustar os campos `mx` (saída esperada dos `mux-N`). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros.

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, veja a observação na página 340.

Teste dos multiplexadores

A coluna `mx` de `test_vectors` é a saída esperada para o `mux-2` e o

```
assert TST_MUX_2 or (saidaMUX2 = esperadaMUX)
```

emitirá mensagem de erro somente se o modelo produzir saída diferente de `mx`. Se a saída esperada é a produzida, então o **assert** fica silente porque o circuito está correto, segundo o vetor de testes *que você, projetista, escreveu*.

Uma vez que você esteja certo de que o `mux-2` está correto, teste o `mux-4`. Este circuito é composto de três `mux-2`, e você garante que o `mux-2` é um circuito que atende a sua especificação. O `mux-4` tem seis entradas e sua tabela verdade tem $2^6 = 64$ linhas, sendo portanto necessários 64 testes. Certo?

Sim, é certo. Podemos usar de inteligência e descobrir qual é o número *mínimo* de testes para garantir a corretude do `mux-4`. Este número é bem menor do que 64.

Gere o vetor de testes (reduzido) para o `mux-4` e verifique sua corretude. Isso feito, troque ‘TST_MUX_4 para false e execute a simulação para testar seu modelo.

O `mux-8` é composto de dois `mux-4` – que você garante que é correto – e de um `mux-2` – que você também garante que é correto. O `mux-8` tem onze entradas e sua tabela verdade tem $2^{11} = 2048$ linhas. São necessários 2048 testes. Certo? Hmm...

Gere o vetor de testes (reduzido) para o *mux-8* e verifique sua corretude. Isso feito, troque ‘TST_MUX_8 para false e execute a simulação para testar seu modelo.

Vetor de bits na entrada do *mux8vet* Há duas entidades distintas para o *mux8*. A primeira, *mux8*, emprega oito sinais de tipo bit nas entradas, e três bits para seleção. A segunda, *mux8vet* mostrada no Programa 13.50, tem como entradas um vetor de 8 bits *entr:reg8* e outro vetor *sel:reg3* para a seleção. As duas arquiteturas são idênticas, exceto que as ligações dos sinais da interface aos componentes devem usar seleção de campos de bits.

Programa 13.50: Entidade do *mux8* com vetores de bits.

```
entity mux8vet is
  port(entr : in  reg8; — vetor com 8 bits
        sel  : in  reg3; — vetor com 3 bits
        z    : out bit); — saída de 1 bit
end mux8vet;
```

Para a entidade *mux8*, as entradas são a,b,c,d,e,f,g,h, enquanto que para a entidade *mux8vet*, as entradas são *entr(0)*, *entr(1)*, ..., *entr(6)*, *entr(7)*.

13.1.5 *Testbench* para os demultiplexadores e decodificadores

O *script* *run_estrut.sh* compila o código VHDL e gera um simulador para os modelos dos demultiplexadores e decodificadores/seletores. Este *script* é similar a *run_mux.sh*.

Etapa 3 Para esta etapa você deve:

- (1) escrever os modelos, segundo a Seção 3.4 de *combin.pdf*, para o *demux-2*. Para tanto edite o arquivo *estrut.vhd*;
- (2) repetir para o *demux-4* e *demux-8*;
- (3) acrescentar mais elementos aos respectivos vetores de teste em *tb_estrut.vhd*; e
- (4) verificar a corretude de seus modelos. Basta dizer `./run_estrut.sh` ler eventuais mensagens de erro, resolver os problemas, e então dizer `./run_estrut.sh 1 &`.

Achtung: no TB dos demultiplexadores e decodificadores, em *tb_estrut.vhd*, um único registro é usado para testar os dois circuitos de largura *N* (*demux-N* e *decod-N*) ao mesmo tempo e portanto, dependendo do teste, alguns dos campos não são relevantes *naquele teste*.

No registro com os vetores de teste, o campo *dm* é o vetor de bits com a saída esperada para um demultiplexador quando recebe as entradas definidas pelos valores em *k,s*.

O campo *dc* é o vetor de bits com a saída esperada para um decodificador cujas entradas são definidas pelos valores em *s*.

Usaremos o mesmo procedimento dos *mux-n* para testar os demultiplexadores. Em *tb_estrut.vhd*, mude TST_DEMUX_2 para false.

A tabela verdade das saídas *z* e *w* do *demux-2* é mostrada ao lado, e o vetor de testes que lhe corresponde está em *tb_estrut.vhd*.

s	k	z	w
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Da mesma forma que com os multiplexadores, os *demux-4* e *demux-8* são composições de *demux-2*. Uma vez que o componente ‘pequeno’ tenha sido verificado, a verificação do com-

ponente ‘grande’ necessita de um número relativamente pequeno de testes para também ser verificada.

Os **asserts** que verificam a corretude dos demultiplexadores e decodificadores estão desabilitados para reduzir a poluição na tela. Para testar seus modelos, altere a constante respectiva (TST_DEMUX_2 ou TST_DECOD_2) para `false` e então verifique os resultados. Há uma tripla de constantes para cada circuito, *viz* TST_DEMUX_2, TST_DEMUX_4 e TST_DEMUX_8.

Você é quem escreve os vetores de teste e portanto sua tarefa é ajustar os campos `dm` (saída esperada dos *demux-N*) e `dc` (saída esperada dos *decod-N*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros inexistentes.

Diagramas de tempo do gtkwave

O script `run_estrut.sh` pode disparar a execução do `gtkwave` para mostrar o diagrama de tempo dos circuitos. O diagrama de tempo mostra, de cima para baixo:

- seleção** bits de seleção, agrupados num vetor de três bits `s [2:0]` e os três bits isolados `s0,s1,s2`, todos em azul;
- DEMUX-2** um agrupamento com os sinais para o *demux-2*, com
 - inp** entrada para os demuxes (laranja);
 - sdemux2** dois bits de saída do *demux-2* (amarelo);
 - esperadademux_2** a saída esperada para o *demux-2* (verde);
- DEMUX-4** mesmo que para o *demux-2*, com largura 4;
- DEMUX-8** mesmo que para o *demux-2*, com largura 8;
- DECOD-2** um agrupamento com os sinais para *decod-2*, similar ao do *mux-2*
- DECOD-4** mesmo que para o *decod-2*, com largura 4;
- DECOD-8** mesmo que para o *decod-2*, com largura 8.

Teste dos decodificadores

Você já entendeu o processo.

Etapa 4 Acrescente ao arquivo `estrut.vhd` o código VHDL para os modelos dos decodificadores de 2,4,8 saídas. Acrescente e/ou altere os elementos do vetor de testes para verificar a corretude dos seus modelos.

Você é quem deve ajustar a *saída esperada* para cada um dos circuitos nos vetores de teste. O projetista dos modelos é responsável por escrever os vetores de teste e portanto sua tarefa é ajustar os campos `dm` (saída esperada dos *demux-N*), e `dc` (saída esperada dos *decod-N*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros.

Histórico das Revisões:

24jun2020: remoção do material que está em vhd1.pdf, separação dos TBs para mux e demux;
02set2019: ajustes cosméticos, diagramas do gtkwaverc;
15ago2018: texto introdutório, exemplos com deltas, mux8vet, gtkwave, (true or) nos asserts;
03set2016: vetores de testes separados por tamanho de circuito;
26ago2016: exemplos de vetores de testes;
25ago2016: incluídas sugestões de Zanata, remoção de FPGA, troca seletor para decodificador;
22ago2016: compilação VHDL;
12set2015: ajustes cosméticos, diagrama com simulador;
29jul2015: primeira versão.

EOF