Como verificar um projeto
ou construindo *testbenches* em VHDL

Hans-Jorg Schneebeli (hans@ele.ufes.br)
Departamento de Engenharia Elétrica
Universidade Federal do Espirito Santo (www.ufes.br)

Uma vez descrito um dispositivo em VHDL, passa-se para a etapa de verificação, cujo objetivo é verificar se ele está funcionando corretamente. Funcionando corretamente significa que ele fornece o resultado esperado para todas as entradas válidas, isto é, que se esperam que aconteçam em funcionamento normal.

Para isto, analogamente aos dispositivos físicos, deve-se montar o dispositivo como um componente em uma bancada de testes (*testbench*), injetar-se os sinais adequados para fazer o dispositivo funcionar e verificar se as saídas estão corretas. Isto pode ser feito em VHDL usando um *testbench* virtual.

Definição do dispositivo de teste

O primeiro passo é definir o dispositivo de teste. Ele é descrito por uma entidade sem portas de entrada e saída. Por exemplo, para se testar um dispositivo filtro (mostrado no anexo) pode-se criar a entidade tb_filtro como abaixo.

```
library IEEE;
use IEEE.std_logic_1164.ALL;        -- tipos std_logic e std_logic_vector
use IEEE.numeric_std.ALL;           -- tipos unsigned e signed
use std.textio.ALL;                 -- deve ser incluido para uso de arquivos

entity TB_FILTRO is
end TB_FILTRO;
```

Na descrição da arquitetura, deve-se ter uma instância do dispositivo em teste e os vários sinais para fazê-lo funcionar. Para se criar uma instância, pode-se usar o esquema tradicional de se definir um componente como mostrado abaixo. Esta maneira é a única possível no VHDL-87. Não é necessário que a implementação seja sintetizável, pois não será gerado um circuito a partir dela.

```
architecture TESTBENCH1 of TB_FILTRO is

   component FILTRO
      port(CLK       : in  std_logic;
           RST       : in  std_logic;
           INPUT     : out std_logic;
           OUTPUT    : in  std_logic
```

```
        );
   end component FILTRO;

   constant PERIODO   : time := 10 ns;

   signal W_CLK       : std_logic := '0';      -- deve ser inicializado
   signal W_RST       : std_logic;
   signal W_INPUT     : std_logic;
   signal W_OUTPUT    : std_logic;

begin

   -- geracao do relogio com periodo PERIODO

   W_CLK <= not W_CLK after PERIODO/2;

   -- criacao de um FILTRO

   DUT : FILTRO port map( CLK     => W_CLK,
                          RST     => W_RST,
                          INPUT   => W_INPUT,
                          OUTPUT  => W_OUTPUT);

   --- falta a injecao de sinais

end TESTBENCH1;
```

No VHDL-93 é possível se instanciar sem o uso de componentes. Neste caso, a arquitetura ficaria como mostrado abaixo. Para isto, é necessário que o módulo FILTRO seja compilado(ou analisado no jargão ghdl) antes.

```
architecture TESTBENCH2 of TB_FILTRO is

   constant PERIODO   : time := 10 ns;

   signal W_CLK       : std_logic := '0';
   signal W_RST       : std_logic;
   signal W_INPUT     : std_logic;
   signal W_OUTPUT    : std_logic;

begin

   -- geracao do relogio com periodo PERIODO

   W_CLK <= not W_CLK after PERIODO/2;

   -- criacao de um FILTRO

   DUT : entity work.DXXX port map( CLK       => W_CLK,
                                    RST       => W_RST,
                                    D_INPUT   => W_MEMADDR,
                                    D_OUTPUT  => W_MEMDATA);
   --- falta a injecao de sinais

end TESTBENCH2;
```

Isto feito, tem-se um módulo da entidade a ser testada. Falta ainda se injetar os

sinais de entrada e verificar se as saídas estão corretas.

A geração dos sinais de entrada a partir de através de código VHDL pode ser feita de forma direta ou a partir de tabelas ou arquivos. Existem formas baseadas em interface gráfica (que implicita- ou explicitamente gera o código VHDL necessário) para esta geração, mas estão fora do escopo deste documento.

Para a verificação existem duas modalidades:
1 – verificação manual. Os sinais de saída são analisados, geralmente a cada pulso de relógio, bem como as entrada correspondentes.
2 – verificação automática. Se no *testbench* pode-se determinar qual a saída correta, pode-se comparar com a saída obtida na simulação com a saída correta. A saída correta pode ser fornecida pelo desenvolvedor ou gerada, ou ainda lida de um arquivo junto com o sinal de entrada.

A verificação manual exige que a cada ciclo de desenvolvimento (codificação, simulação e verificação) o desenvolvedor observe as saídas e analise se o funcionamento está correto ou não. Isto é adequado para módulos pequenos ou simples. A repetição desta leitura a cada ciclo de depuração pode levar o desenvolvedor a não observar uma regressão, isto é, o ocorrência de um erro que não existia e já tinha sido ocorrido. Isto pode acontecer mesmo que as modificações feitas no código estejam em outro trecho do código. Por isso, o uso de verificação automática será enfatizado neste documento.

Injeção direta dos sinais

Para se gerar os sinais e injetá-los no dispositivo em teste devem-se criar um ou mais processos que atribuam valores a eles em tempos determinados. Deve ser observado que o valor do sinal só muda quando se chega ao seu final ou quando é executado um comando wait, e também que tudo que acontece em um processo entre os comandos de espera acontece sem atraso, ou seja, duração zero. Assim, a temporização deve ser controlado com alguma das formas do comando wait.

As formas principais são

wait for esperar por um determinado tempo

wait until esperar por uma determinada condição

wait on esperar que um dos sinais mude de valor

wait espera incondicional, que encerra o processo.

Para a implementação do testbench (tb_filtro.vhd) mostrado acima foi gerado um sinal de relógio usando um comando concorrente.

W_CLK <= NOT W_CLK after PERIODO/2;

Observe que o tempo foi definido como uma constante do tipo time com o valor de 10 ns. Além disso, deve-se garantir que o valor inicial de W_CLK seja '0', pois caso contrário, o valor seria indefinido ('**U**') e a negação deste valor continuaria indefinido.

Em geral, é interessante acrescentar uma condição para cancelar a alternação do sinal depois de um tempo. Quando se garante que não há mais alteração de nenhum sinal e todos os processos estão parados, o simulador pára. Caso contrário, um limite de tempo deve ser explicitamente fornecido quando do chamado do simulador (por linha de comando ou ambiente gráfico). Para que o sinal de relógio não varie depois de um tempo é feito um E lógico entre o relógio e um sinal de habilitação W_CLK_ENABLE, que é inicialmente 1. O código correspondente é

signal W_CLK_ENABLE: std_logic := '1';

...

W_CLK <= W_CLK_ENABLE and not W_CLK after PERIODO/2;

W_CLK_ENABLE <= '1', '0' after 20*PERIODO;

Quando o sinal W_CLK passa a ser zero o relógio fica com o valor constante. No exemplo acima, isto acontece depois de 20 períodos de relógio, mas pode ser feito como mostrado adiante, ao final da geração dos sinais de entrada.

O sinal de reset (W_RST), inicialmente alto por 2 períodos de relógio, pode ser gerado usando um processo, como mostrado abaixo.

```
reset: process
begin
    W_RST <= '1';
    wait for 2*PERIODO;
    W_RST <= '0';
    wait;
end process reset;
```

Ou então, de forma semelhante ao usado para gerar o sinal de habilitação do relógio (W_CLK_ENABLE), como mostrado abaixo.

W_RST <= '1', '0' after 2*PERIODO;

Esta forma é absolutamente equivalente ao processo mostrado acima.

Os sinais de entrada podem ser especificados em um ou mais processos, mas pode-se usar um único processo para a geração do reset e dos sinais de entrada.

```
stimulus: process
begin
    W_INPUT <= '0';
```

```
      wait for 4*PERIODO;
      W_INPUT <= '1';
      wait for 7*PERIODO;
      W_INPUT <= '0';
      wait for 7*PERIODO;
      W_INPUT <= '1';
      W_CLK_ENABLE <= '0';
      wait;
  end process stimulus;
```

Isto também poderia ser escrito como

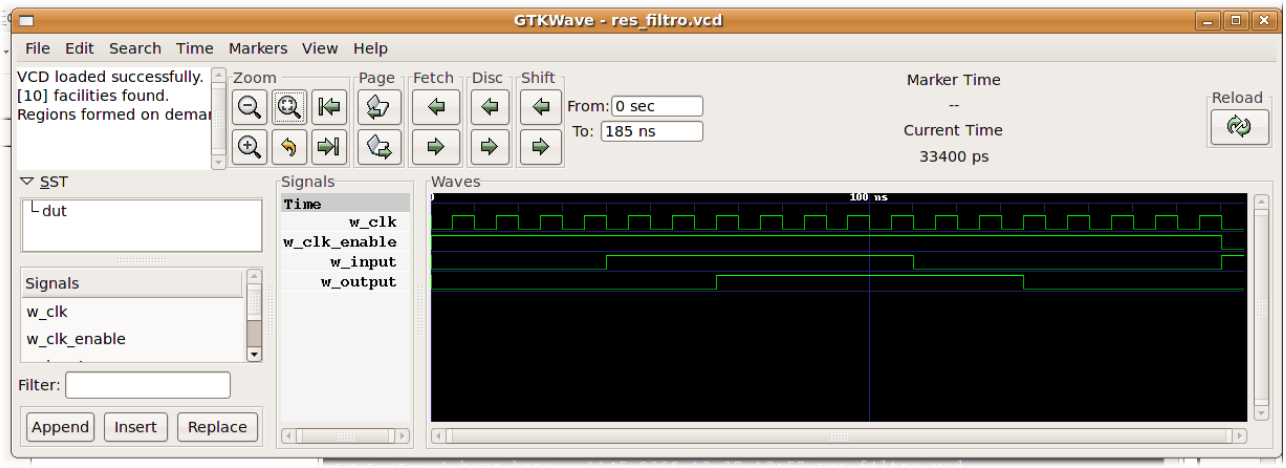
```
      W_INPUT <= '0',
                 '1' after 4*PERIODO,
                 '0' after 11*PERIODO,
                 '1' after 18*PERIODO;
```

Deve ser observado que agora se especifica o instante de tempo e não mais o intervalo de tempo. Além disso, o sinal de habilitação do relógio é desligado antes da espera incondicional, que encerra o processo.

Visualização

É possível se gerar um arquivo VCD, que permite se determinar a evolução no tempo de todos os sinais. No caso do ghdl, usa-se o parametro –vcd quando da chamada do simulador
ghdl -r tb_filtro –vcd=res_filtro.vcd

O arquivo pode ser visualizado com o gtkwave (ou programa equivalente).



O desenvolvedor deverá observar as formas de onda de entrada e saída e verificar se o funcionamento do dispositivo está correto. Isto só funciona para sistemas pequenos. Basta imaginar analisar o funcionamento de um Pentium em um PC nos primeiro segundo, que pode implicar na analise de 3.400.000.000 pulsos de relógio.

Impressão

Geralmente é mais interessante se poder imprimir os valores dos sinais, que geralmente são do tipo std_logic ou std_logic_vector. O mecanismo básico é o comando report. Originalmente, na versão 1987 do VHDL, este comando era subordinado a um comando assert (garanta que), que tem a seguinte estrutura.
assert <condição> report <texto> severityxto> seve <nivel>.

Na versão 1993, ele pode ser um comando independente, eliminando assim a necessidade do truque abaixo, que fazia com que fosse sempre impresso o texto.
assert false report <texto> severity <nivel>.

A condição é uma comparação (na verdade, um valor lógico falso ou verdadeiro), que caso seja falso, provocará a impressão do texto. O nível pode ser **NOTE**, **WARNING**, **ERROR**, ou **FAILURE.** Quando se executa o simulador, pode-se especificar qual nível de erro provocará o encerramento da simulação. Em geral, usam-se os níveis WARNING para indicar casos que não provocam o cancelamento da execução do simulador e FAILURE para indicar os casos de encerramento do simulador.

O texto é uma cadeia de caracteres (string), que em casos complexos, pode ser obtida concatenando-se várias cadeias menores e caracteres. Por exemplo, o comando abaixo, imprime a mensagem com um salto para uma nova linha ao final.
report "Mensagem impressa com salto para nova linha ao final" & CR & LF.

CR e LF são caracteres especiais que indicam o salto de linha.

Para se imprimir os valores de sinais, deve-se usar a função **image** definida para os tipos pré-definidos da linguagem como integer, bit e boolean.

Por exemplo, o arquivo VHDL abaixo quando executado imprime a tabela verdade das operações OR AND e XOR. O comando wait é necessário para se interromper a execução. No caso de não haver o comando, o processo é repetido indefinidamente.

```
entity reportout is
end reportout;

architecture beh of reportout is
begin
```

```
      process
      variable a,b: bit;
      variable x_or,x_and,x_xor: bit;
      begin
            report " A    B    OR    AND     XOR" & CR;
            a := '1';
            b := '0';
            L10:
            for a in bit loop
                for b in bit loop
                  x_or := a OR b;
                  x_and := a AND b;
                  x_xor := a XOR b;
                  report bit'image(a) & "  " & bit'image(b) & "   " &
                         bit'image(x_or) & "   " &
                         bit'image(x_and) & "     " &
                         bit'image(x_xor) & "    " & CR;
                  end loop;
            end loop L10;
            wait;
      end process;
end beh;
```

A saída neste caso é

```
reportout.vhd:13:16:@0ms:(report note):  A    B    OR    AND     XOR
reportout.vhd:22:24:@0ms:(report note): '0'  '0'   '0'   '0'     '0'
reportout.vhd:22:24:@0ms:(report note): '0'  '1'   '1'   '0'     '1'
reportout.vhd:22:24:@0ms:(report note): '1'  '0'   '1'   '0'     '1'
reportout.vhd:22:24:@0ms:(report note): '1'  '1'   '1'   '1'     '0'
```

Observe que são indicados o nome do arquivo, o número da linha, o tempo de simulação e o nível do comando.

Outro exemplo é o arquivo abaixo que imprime a seqüencia de fibonnacci, onde um elemento é a soma dos anteriores.

```
entity fibo is
end fibo;

architecture beh of fibo is
begin
      process
      variable a,b,c: integer;
      constant limit: integer := 100;
      begin
          a := 1;
          b := 1;
          report integer'image(b) & CR;
          while a < limit
          loop
              report integer'image(a) & CR;
              c := a + b;
              b := a;
              a := c;
          end loop;
          wait;
      end process;
```

```
end beh;
```

Para se imprimir sem as informações de nome de arquivo, etc. deve-se usar o comando **write** indicando o arquivo como **output** como alvo da operação, como mostrado na arquitetura abaixo. Observe a necessidade do comando
use std.textio.all;
antes da arquitetura.

```
use std.textio.all;

architecture beh2 of fibo is
begin

      process
      variable a,b,c: integer;
      constant limit: integer := 100;
      begin
          a := 1;
          b := 1;
          write(output,integer'image(b));
          while a < limit
          loop
              write(output,integer'image(a));
              c := a + b;
              b := a;
              a := c;
          end loop;
          wait;
      end process;

end beh2;
```

Para se imprimir vários valores na mesma linha, devem-se escrevê-los primeiro em uma linha (uma variável do tipo line) e então imprimir a linha como mostrado abaixo.

```
use std.textio.all;

architecture beh2 of reportout is
begin
      process
      variable a,b: bit;
      variable x_or,x_and,x_xor: bit;
      variable l: line;
      begin
            write(output," A    B    OR    AND     XOR");
            a := '1';
            b := '0';
            L10:
            for a in bit loop
                for b in bit loop
                  x_or := a OR b;
                  x_and := a AND b;
                  x_xor := a XOR b;
                  write(l,string'(" "));
```

```
                write(l,a);          write(l,string'("    "));
                write(l,b);          write(l,string'("     "));
                write(l,x_or);       write(l,string'("     "));
                write(l,x_and);      write(l,string'("       "));
                write(l,x_xor);
                writeline(output,l);
                end loop;
            end loop L10;
            wait;
        end process;

end beh2;
```

Deve ser observada a necessidade de se explicitar o parâmetro do write que escreve na linha, pois há uma ambiguidade entre bit_vector e string, ambos descritos com valores entre aspas duplos. Para isto, usa-se a notação **string'("texto")** com uso de um aspas simples.

E finalmente a saída é gerado por um outro processo, disparado a cada transição do relógio, que mostra a entrada e a saída do dispositivo.

```
  verify: process(W_CLK)
  begin
      if rising_edge(W_CLK) then
            report " INPUT : " & std_logic'image(W_INPUT) &
                   " OUTPUT : " & std_logic'image(W_OUTPUT);
      end if;

  end process;
```

A saída da simulação é mostrada abaixo, devendo ser observados também os tempos. Neste caso, o valor a entrada é o anterior a transição e o da saída o

```
filter_tb.vhd:67:16:@5ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@15ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@25ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@35ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@45ns:(report note):  INPUT : '1' OUTPUT : '0'
filter_tb.vhd:67:16:@55ns:(report note):  INPUT : '1' OUTPUT : '0'
filter_tb.vhd:67:16:@65ns:(report note):  INPUT : '1' OUTPUT : '0'
filter_tb.vhd:67:16:@75ns:(report note):  INPUT : '1' OUTPUT : '1'
filter_tb.vhd:67:16:@85ns:(report note):  INPUT : '1' OUTPUT : '1'
filter_tb.vhd:67:16:@95ns:(report note):  INPUT : '1' OUTPUT : '1'
filter_tb.vhd:67:16:@105ns:(report note):  INPUT : '1' OUTPUT : '1'
filter_tb.vhd:67:16:@115ns:(report note):  INPUT : '0' OUTPUT : '1'
filter_tb.vhd:67:16:@125ns:(report note):  INPUT : '0' OUTPUT : '1'
filter_tb.vhd:67:16:@135ns:(report note):  INPUT : '0' OUTPUT : '1'
filter_tb.vhd:67:16:@145ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@155ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@165ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@175ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@185ns:(report note):  INPUT : '0' OUTPUT : '0'
filter_tb.vhd:67:16:@195ns:(report note):  INPUT : '0' OUTPUT : '0'
```

Um resultado bem interessante pode ser conseguido imprimindo-se estes valores a cada mudança do valor do relógio, o que pode ser conseguido comentando-se o teste de transição como mostrado abaixo.

```
  verify: process(W_CLK)
  begin
--    if rising_edge(W_CLK) then
            report " INPUT : " & std_logic'image(W_INPUT) &
                   " OUTPUT : " & std_logic'image(W_OUTPUT);
--    end if;
  end process verify;
```

A verificação pode ser automatizada com a inserção de comandos assert para comparar o valor obtido com o esperado como mostrado abaixo nos momentos críticos.

```
      assert W_OUTPUT = '0'
          report "Erro no tempo " & time'image(now) & CR & LF severity FAILURE;
```

Injeção de sinais com valores lidos de arquivo

Quando se tem muitos valores a testar pode ser interessante colocá-los em uma tabela, ou ainda mais fácil, lê-los de um arquivo texto. Pode-se ainda colocar neste arquivo o valor esperado. Geralmente a cada linha da tabela corresponde a um instante do tempo relacionado ao relógio.

No caso acima, poderia-se editar um arquivo teste.txt com o seguinte texto, onde a primeira linha documenta a ordem dos valores. Esta linha serve apenas como comentário e será ignorada quando da leitura.

```
# RST INPUT OUTPUT
100
100
000
000
000
000
000
000
010
010
010
011
011
001
001
001
000
000
000
000
```

```
000
```

Para se conseguir ler os valores, deve-se primeiro ler toda a linha e então acessar os valores como mostrado abaixo. Como não há leitura de dados do tipo std_logic, eles são lidos como caractere e convertidos para std_logic pela função to_stdlogic do pacote std_logic_extra no apêndice. Como este pacote não é padrão ele deve ser compilado (analisado no jargão ghdl) antes.

```
use std.textio.all;
use work.std_logic_extra.all;
architecture TESTBENCH3 of TB_FILTRO is

  signal W_CLK: std_logic := '0';
  signal W_RST: std_logic;
  signal W_INPUT: std_logic;
  signal W_OUTPUT: std_logic;
  signal W_CLK_ENABLE: std_logic := '1';

  constant PERIODO: time := 10 ns;

begin

  uut: entity work.FILTRO
      port map (  CLK => W_CLK,
                  RST => W_RST,
                  INPUT => W_INPUT,
                  OUTPUT => W_OUTPUT
             );

  W_CLK <= W_CLK_ENABLE and not W_CLK after PERIODO/2;

  stimulus: process
  variable ch: character;
  variable xrst,xin,xout: std_logic;
  file testcase: text;
  variable l: line;
  begin
      W_INPUT <= '0';
      W_RST <= '0';
      W_CLK_ENABLE <= '1';

      file_open(testcase,"teste.txt",read_mode);
      readline(testcase,l); -- le e ignora primeira linha
      while not endfile(testcase)
      loop
          readline(testcase,l);
          read(l,ch);
          xrst := to_stdlogic(ch);
          read(l,ch); xin := to_stdlogic(ch);
          read(l,ch); xout := to_stdlogic(ch);
          wait until rising_edge(W_CLK);
          W_RST <= xrst;
          W_INPUT <= xin;
          wait until falling_edge(W_CLK);
          report " INPUT : " & std_logic'image(W_INPUT) &
                 " OUTPUT : " & std_logic'image(W_OUTPUT);
```

```
          assert W_OUTPUT = xout
            report "Erro no tempo" & time'image(now) & CR & LF;
      end loop;
      file_close(testcase);
      wait for 4*PERIOD;
      W_CLK_ENABLE <= '0';
      wait;
  end process;

end architecture TESTBENCH3;
```

Uma outra alternativa seria ler os dados como bit aproveitando o fato de que estes apenas têm os valores 0 ou 1. Mas a atribuição de um valor bit a sinal std_logic ou para compara-lo com ele, deve-se converte-lo para um tipo std_ulogic usando a função to_stdulogic (observe que não é to_std_logic), que é identico ao tipo std_logic, mas sem ter a possibilidade de resolução de conflitos de atribuição, que acontece quando usamos o estado de alta impedância ('Z') para construir barramentos.

```
use std.textio.all;

architecture TESTBENCH4 of TB_FILTRO is

  signal W_CLK: std_logic := '0';
  signal W_RST: std_logic;
  signal W_INPUT: std_logic;
  signal W_OUTPUT: std_logic;
  signal W_CLK_ENABLE: std_logic;
  constant PERIODO: time := 10 ns;
begin

  uut: entity work.FILTRO
      port map (  CLK => W_CLK,
                  RST => W_RST,
                  INPUT => W_INPUT,
                  OUTPUT => W_OUTPUT
                  );

  W_CLK <= W_CLK_ENABLE and not W_CLK after PERIODO/2;

  stimulus: process
  variable xrst,xin,xout: bit;
  file testcase: text;
  variable l: line;
  begin
      W_INPUT <= '0';
      W_RST <= '0';
      W_CLK_ENABLE <= '1';

      file_open(testcase,"teste.txt",read_mode);
      readline(testcase,l); -- le e ignora primeira linha
      while not endfile(testcase)
      loop
          readline(testcase,l);
```

```
          read(l,xrst);
          read(l,xin);
          read(l,xout);
          wait until rising_edge(W_CLK);

          W_RST <= to_stdulogic(xrst);
          W_INPUT <= to_stdulogic(xin);
          wait until falling_edge(W_CLK);
          report " INPUT : " & std_logic'image(W_INPUT) &
                 " OUTPUT : " & std_logic'image(W_OUTPUT);

          assert W_OUTPUT = to_stdulogic(xout)
              report "Erro " & CR & LF;

      end loop;

      file_close(testcase);
      wait for 4*PERIODO;
      W_CLK_ENABLE <= '0';
      wait;
  end process;

end architecture TESTBENCH4;
```

Bibliografia

[1] http://www.vhdl-online.de/TB-GEN/ent2tb1.htm

[2] http://www.doulos.com/knowhow/perl/testbench_creation/

[3] http://www.nialstewartdevelopments.co.uk/downloads.htm.

[4] Peakfpga. Welcome to the VHDL Language Guide em http://www.peakfpga.com/vhdlref/index.html.

[5] Peter Ashenden. The Designer's Guide to VHDL. Morgan Kaufmann. 1996.

[6] Dan Biederman. An Overview on Writing a VHDL Testbench. Integrated Systems Design,. p. 23-38, 1998

[7] Mujtaba Hamid. Xilinx Application Note XAPP199: Writing Effective Testbenches em http://www.xilinx.com/support/documentation/application_notes/xapp199.pdf

[8] Donna Mitchell. Test Bench Generation from Timing Diagrams (Appendix D to VHDL Made Easy by David Pellerin) 1996. Em http://www.syncad.com/paper_p_vme.htm

[9] Donna Mitchell. Manual and Automatic VHDL/Verilog Test Bench Coding Techniques. Dedicated Systems Magazine. 2001. Em http://www.dedicatedsystems.com/magazine/01q2/2001q2_p027.pdf.

[10] Kevin Skahill. VHDL for Programmable Logic. Addison-Wesley. 1996.

[11] Stephan Doll. VHDL verification course. Em http://www.stefanvhdl.com/

Anexo

O arquivo abaixo (filter.vhd) mostra uma implementação de um circuito que gera uma saída um sempre que a entrada permanecer em um por três períodos de relógio e continuando a gerar um até que a entrada seja zero por três períodos de relógio.

```
library ieee;
use ieee.std_logic_1164.all;

entity FILTER is
      port(
            CLK:        in    std_logic;
            RST:        in    std_logic;
            INPUT:      in    std_logic;
            OUTPUT:     out   std_logic
      );
end;

architecture beh of FILTER is
type STATETYPE is (Q0,Q1);
signal STATE : STATETYPE;
signal LAST : STD_LOGIC_VECTOR(1 downto 0);
begin

      -- geração da saída (Moore)
      OUTPUT <= '1' when STATE = Q1 else '0';

      -- descrição das transições
      process(CLK,RST)
      begin
            if RST = '1' then
                  STATE <= Q0;
            elsif rising_edge(CLK) then
                  case STATE is
                  when Q0 =>
                        if INPUT = '1' and LAST = "11" then
                              STATE <= Q1;
                        end if;
                  when Q1 =>
                        if INPUT = '0' and LAST = "00" then
                              STATE <= Q0;
                        end if;
                  end case;
                  LAST <= INPUT & LAST(1);
            end if;
      end process;

end architecture beh;
```

O arquivo abaixo (std_logic_extra.vhd) mostra a implementação de funções para conversão dos tipos std__logic e std_logic_vector para um cadeia de caracteres (String) e o contrário. As funções implementadas são:

to_character(std_logic)	Converte um escalar do tipo std_logic em um caracter correspondente.
to_stdlogic(character)	Converte um caracter em um valor do tipo std_logic. Caso não haja correpondencia é retorna 'U'
to_string(std_logic_vector)	Converte um vetor do tipo std_logic_vector em uma string
to_stdlogicvector(string)	Converte uma cadeia de caracteres em um vetor do tipo std_logic_vector.

```
--
-- functions to convert std_logic to/from character and
--           to convert std_logic_vector to/from string
--
-- author:         hans (hans@ele.ufes.br)
--
-- v1.0   09/07 : only to_string functions
-- v2.0   09/08 : include to_stdlogic functions
--



library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package std_logic_extra is
    function to_character(arg: std_logic) return character;
    function to_string(arg: std_logic_vector) return string;
    function to_stdlogic(arg: character) return std_logic;
    function to_stdlogicvector(arg: string) return std_logic_vector;
end package;

package body std_logic_extra is

    function to_character(arg: std_logic) return character is
    begin
          case arg is
            when '0' => return '0';
            when '1' => return '1';
            when 'L' => return 'L';
            when 'H' => return 'H';
            when 'X' => return 'X';
            when 'U' => return 'U';
            when '-' => return '-';
            when 'W' => return 'W';
            when 'Z' => return 'Z';
            when others => return '*';
            end case;
    end function to_character;

    function to_stdlogic(arg: character) return std_logic is
    begin
          case arg is
            when '0' => return '0';
            when '1' => return '1';
```

```
            when 'L' => return 'L';
            when 'H' => return 'H';
            when 'X' => return 'X';
            when 'U' => return 'U';
            when '-' => return '-';
            when 'W' => return 'W';
            when 'Z' => return 'Z';
            when others => return 'U';
            end case;
    end function to_stdlogic;

    function to_string(arg: std_logic_vector) return string is
    variable S: string(1 to arg'length) := (others=>'*');
    variable J: natural;
    begin
          J := 1;
          for I in arg'range  loop
                S(J) := to_character(arg(I));
                J := J + 1;
          end loop;
          return S;
    end function to_string;

    function to_stdlogicvector(arg: string) return std_logic_vector is
    variable V: std_logic_vector(arg'length-1 downto 0) := (others=>'0');
    variable J: integer;
    variable II : natural;
    begin
          J := V'left ;
          for I in arg'range  loop
                V(J) := to_stdlogic(arg(I));
                J := J - 1;
          end loop;
          return V;
    end function to_stdlogicvector;

end package body std_logic_extra;
```