Aula 03 - Verificação de Hardware 2/3 - Test-bench

Prof. Sergio R. M. Canovas

PCS - Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

Agosto, 2020

Agenda

Ao final da aula você saberá:

- O que é o conceito de cobertura de um *test-bench*;
- Formas de descrever os testes em um *test-bench* VHDL;
- Como escrever código para simular um sinal de *clock* em VHDL;
- Como aguardar por um evento dentro de um *process*;

Cobertura de um test-bench

Cobertura de um test-bench

- Exercício: um *test-bench* para um somador de 64 bits que testa todas as combinações é viável? Explique o raciocínio.

Cobertura de um test-bench

- 2^{128} combinações de entradas;
- $2^{128} = (10^{\log_{10} 2})^{128} \approx 3,16 \times 10^{38}$
- Suponha 1 ns $= 10^{-9}$ s de tempo de processamento real do simulador VHDL para cada caso. A simulação demoraria $3,16 \times 10^{29}$ s $\approx 10^{22}$ anos.
- Se fosse 1 ps $= 10^{-12}$ s de tempo de processamento real para cada caso, ainda assim demoraria $3,16 \times 10^{26}$ s $\approx 10^{19}$ anos.

Cobertura de um test-bench

- **Conclusão**: mesmo em circuitos combinatórios, em que há um número finito de combinações de entradas, um *test-bench* completo pode ser inviável devido à explosão combinatória.
- Para circuitos sequenciais, em que as saídas dependem de todo o histórico de utilização, o número de sinais digitais de entrada possíveis, variando no tempo desde o instante inicial, é infinito.
- **Consequência**: Devemos considerar *test-benches* não completos, mas que ainda assim sejam bons.

Cobertura de um test-bench

- **Premissa:** É importante que um *test-bench* seja eficiente em capturar possíveis *bugs* no modelo do DUT;
- A parte de um DUT testada por um caso de teste é chamada de **cobertura** (*coverage*);
- Uma definição rigorosa de cobertura é difícil de se obter. Em geral, a cobertura refere-se à porcentagem do DUT que foi checada durante uma simulação [1].7;
- Quantificar um DUT também é difícil. Poderia ser, por exemplo, o número de linhas de código de seu modelo em VHDL;

Cobertura de um test-bench

- Por outro lado, se o DUT estiver modelado como uma máquina de estados finita com dados (FSMD), poderia ser a quantidade de estados e transições deste modelo;
- Infelizmente, essas representações são incompletas e não capturam o comportamento inteiro do modelo;

Cobertura de um test-bench

- Podemos, por exemplo, definir cobertura como o número de linhas de código que foram visitadas durante uma execução da simulação de determinado caso de teste;
- Se 100 de 1000 linhas de código foram visitadas nesta simulação, dizemos que a cobertura deste caso de teste foi de 10%;
- Mas isso não quer dizer que todos os possíveis cenários para essas 100 linhas foram verificados;
- Seja a linha de código:

$$a = b/c;$$

Cobertura de um test-bench

- Ela pode executar corretamente se $b = 4$ e $c = 2$, mas causará um erro se $c = 0$;
- Por isso, trata-se de uma métrica fraca para cobertura;
- Mesmo com as dificuldades para se quantificar um DUT e a cobertura de um caso de testes, é recomendável que se gere testes pensando em quantas métricas forem possíveis [1].7;
- **Em um cenário ideal, gostaríamos de executar um número mínimo de casos de teste para cobrir o máximo possível do DUT**;
- Para isso, é necessário dispor de algum método que estime a cobertura de casos de teste e gere um *test-bench* de uma maneira eficiente. Isso depende de cada projeto;

Cobertura de um test-bench

- Na falta de um método deste tipo, o autor do *test-bench* pode optar por gerar casos de teste aleatórios;
- Este método não é direcionado a encontrar *bugs* específicos, mas espera-se que os testes gerados aleatoriamente estejam distribuídos de forma equilibrada entre as entradas possíveis;
- Este método em geral provê casos de testes mais pobres em relação àqueles elaborados de forma específica, mas pode gerá-los mais rapidamente de forma automatizada.

Formas de codificar estímulos

Formas de codificar estímulos

- Há 3 formas recomendadas para codificar o envio de estímulos e seu monitoramento em VHDL:

1. Programaticamente (já vimos);
2. Vetor de testes;
3. Leitura de arquivo;

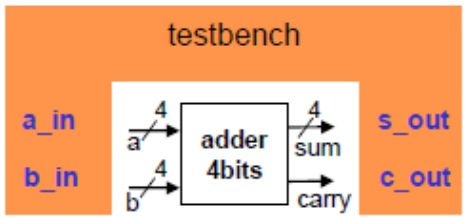
Forma programática


```
-- Testbench para somador
library IEEE;
use IEEE.numeric_bit.all;

entity testbench is
  -- Sempre vazio:
end testbench;

architecture testb of testbench is
-- DUT: component
component adder4bits is
port(
 a,b:   in   bit_vector  (3 downto 0);
 sum:   out  bit_vector  (3 downto 0);
 carry: out  bit );
end component;

signal a_in, b_in, s_out:    bit_vector   (3 downto 0);
signal c_out :  bit  ;
```



Forma programática


```
-- Conectar DUT
DUT: adder4bits port map(a_in, b_in, s_out, c_out);

process -- Circuito com sequência de testes
begin
  a_in <= "0000";
  b_in <= "0000";
  wait for 1 ns; -- espera estabilizar e verifica saída
  assert(c_out & s_out ="00000") report "Fail 0+0" severity error;

  a_in <= "0001";
  b_in <= "0001";
  wait for 1 ns; -- espera estabilizar e verifica saída
  assert(c_out & s_out ="00010") report "Fail 1+1" severity error;

  a_in <= "1000";
  b_in <= "1000";
  wait for 1 ns; -- espera estabilizar e verifica saída
  assert(c_out & s_out ="10000") report "Fail 8+8" severity error;
```

Forma programática


```
a_in <= "1111";
b_in <= "0001";
wait for 1 ns; -- espera estabilizar e verifica saída
assert(c_out & s_out ="10000") report "Fail F+1" severity error;

a_in <= "1110";
b_in <= "0111";
wait for 1 ns; -- espera estabilizar e verifica saída
assert(c_out & s_out ="10101") report "Fail E+7" severity error;

a_in <= "0111";
b_in <= "0011";
wait for 1 ns; -- espera estabilizar e verifica saída
assert(c_out & s_out ="01010") report "Fail 7+3" severity error;
```

Forma programática


```
        a_in <= "1100";
        b_in <= "0101";
        wait for 1 ns; -- espera estabilizar e verifica saída
        assert(c_out & s_out ="10001") report "Fail C+5" severity error;


        -- Limpa entradas (opcional)
        a_in <= "0000";
        b_in <= "0000";

        -- Informa fim do teste
        assert false report "Test done." severity note;
        wait; -- Interrompe execução
    end process;
end testb;
```

Forma programática

- Também é possível o uso de laços **for** dentro de *process*;
- Exemplo de um trecho de um *test-bench* de um contador;
- *saidai* é o sinal de saída do contador (DUT);

```
--! Testa se a contagem crescente está OK
clr<='1'; load<='0'; up<='1'; en<='1';
for i in 0 to modulo-1 loop
  --! Verifica a contagem
  assert saidai = i report
    "Contagem falhou. Esperado: " & integer'image(i) &
    " Obtido: " & integer'image(saidai)
    severity failure;
  wait until falling_edge(clk);
end loop;
```

Vetor de testes

- No estilo programático, os estímulos eram gerados e verificados um a um, diretamente no código;
- No caso de **vetor de testes**, o *process* gerador de estímulos percorre um vetor cujos elementos correspondem a casos de teste (estímulo/resposta esperada);
- Os valores no vetor de testes devem ter sido definidos previamente;

Vetor de testes

- No preâmbulo do *process*, declara-se o **tipo do elemento** do vetor de testes (*pattern_type*). Esse tipo deve prever campos para as entradas e as saídas, correspondendo a um teste auto-contido;
- Em seguida declara-se o **tipo correspondente ao vetor** em si (*pattern_array*), cujo tipo dos elementos é *pattern_type* descrito acima;
- Por fim, declara-se uma constante que é o **vetor de testes** propriamente dito, o qual conterá os casos de teste;

Vetor de testes


```
gerador_estimulos: process is

   type pattern_type is record
      -- Entradas
      op1: bit_vector(3 downto 0);
      op2: bit_vector(3 downto 0);
      -- Saídas
      soma_esperada: bit_vector(3 downto 0);
      carry_esperado: bit;
   end record;

   type pattern_array is array (natural range <>) of pattern_type;

   constant patterns: pattern_array :=
      --                                        op1  op2     carry_esperado  soma_esperada
      (("0000","0000","0000",'0'),    --   0 +  0  =                  0               0
       ("0001","0001","0010",'0'),    --   1 +  1  =                  0               2
       ("1000","1000","0000",'1'),    --   8 +  8  =                  1               0
       ("1111","0001","0000",'1'),    --   F +  1  =                  1               0
       ("1110","0111","0101",'1'),    --   E +  7  =                  1               5
       ("0111","0011","1010",'0'),    --   7 +  3  =                  0               A
       ("1100","0101","0001",'1'));   --   C +  5  =                  1               1
```

Vetor de testes

- A partir daí, itera-se sobre o vetor, injetando as entradas e verificando as saídas para cada um deles;

Vetor de testes


```
begin

   -- Para cada padrao de teste no vetor
   for i in patterns'range loop
      -- Injeta as entradas
      a_in <= patterns(i).op1;
      b_in <= patterns(i).op2;
      -- Aguarda que o modulo produza a saida
      wait for 10 ns;
      --  Verifica as saidas
      assert s_out = patterns(i).soma_esperada report "Erro na soma " &
      integer'image(to_integer(unsigned(patterns(i).op1))) & " + " &
      integer'image(to_integer(unsigned(patterns(i).op2))) severity error;
      assert c_out = patterns(i).carry_esperado report "Erro no carry para " &
      integer'image(to_integer(unsigned(patterns(i).op1))) & " + " &
      integer'image(to_integer(unsigned(patterns(i).op2))) severity error;
   end loop;

   -- Informa fim do teste
   assert false report "Teste concluido." severity note;
   wait;  -- pára a execução do simulador, caso contrário este process é
   reexecutado indefinidamente.
end process;
```

Leitura de arquivo

- Também é possível ler os casos de teste de um arquivo externo;
- É possível fazer isso em VHDL, instruindo o simulador a abrir um arquivo do disco;
- Inclui-se a biblioteca **textio**, cujas operações de leitura de arquivo não são sintetizáveis (salvo exceções para inicialização de memórias).

```
use std.textio.all;
```

Leitura de arquivo


```
gerador_estimulos: process is

    file tb_file : text open read_mode is "adder4bits_tb_arquivo.dat";
    variable tb_line: line;
    variable space: character;
    variable op1, op2, soma_esperada: bit_vector(3 downto 0);
    variable carry_esperado: bit;
```

Leitura de arquivo


```
begin
   while not endfile(tb file) loop  -- Enquanto não chegar no final do arquivo ...
      readline(tb file, tb line);  -- Lê a próxima linha
      read(tb line, op1);   -- Da linha que foi lida, lê o primeiro parâmetro (op1)
      read(tb line, space); -- Lê o espaço após o primeiro parâmetro (separador)
      read(tb line, op2);   -- Da linha que foi lida, lê o segundo parâmetro (op2)
      read(tb line, space); -- Lê o próximo espaço usado como separador
      read(tb line, soma esperada);  -- Da linha que foi lida, lê o terceiro
      parâmetro (soma_esperada)
      read(tb_line, space); -- Lê o próximo espaço usado como separador
      read(tb_line, carry_esperado); -- Da linha que foi lida, lê o quarto
      parâmetro (carry_esperado)
      -- Agora que já lemos o caso de teste (par estímulo/saída esperada), vamos
      aplicar os sinais.
      a_in <= op1;
      b_in <= op2;
      wait for 10 ns; -- Aguarda a produção das saídas
      --  Verifica as saidas
      assert s_out = soma_esperada report "Erro na soma " &
      integer'image(to_integer(unsigned(op1))) & " + " &
      integer'image(to_integer(unsigned(op2))) severity error;
      assert c_out = carry_esperado report "Erro no carry para " &
      integer'image(to_integer(unsigned(op1))) & " + " &
      integer'image(to_integer(unsigned(op2))) severity error;
   end loop;

   -- Informa fim do teste
   assert false report "Teste concluido." severity note;
   wait;  -- pára a execução do simulador, caso contrário este process é
   reexecutado indefinidamente.
end process;
```

Leitura de arquivo

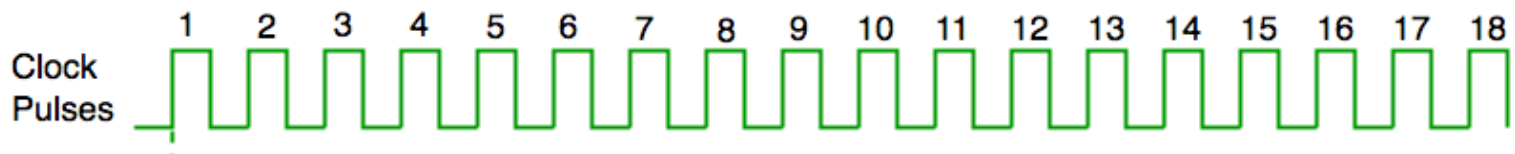
- Conteúdo de **adder4bits_tb_arquivo.dat**:

```
0000 0000 0000 0
0001 0001 0010 0
1000 1000 0000 1
1111 0001 0000 1
1110 0111 0101 1
0111 0011 1010 0
1100 0101 0001 1
```

Simulação do clock

Simulação do clock

- Em circuitos sequenciais síncronos, usa-se um sinal de ***clock*** como base tempo;
- O ***duty cycle*** é a porcentagem de tempo de um ciclo em que esse sinal fica em alto, em geral 50%;
- Em geral, usa-se a borda de subida (*rising edge*) como gatilho para transferir dados entre elementos do circuito, fazendo com que ele execute passo-a-passo as funções a que se propõe;



Simulação do clock

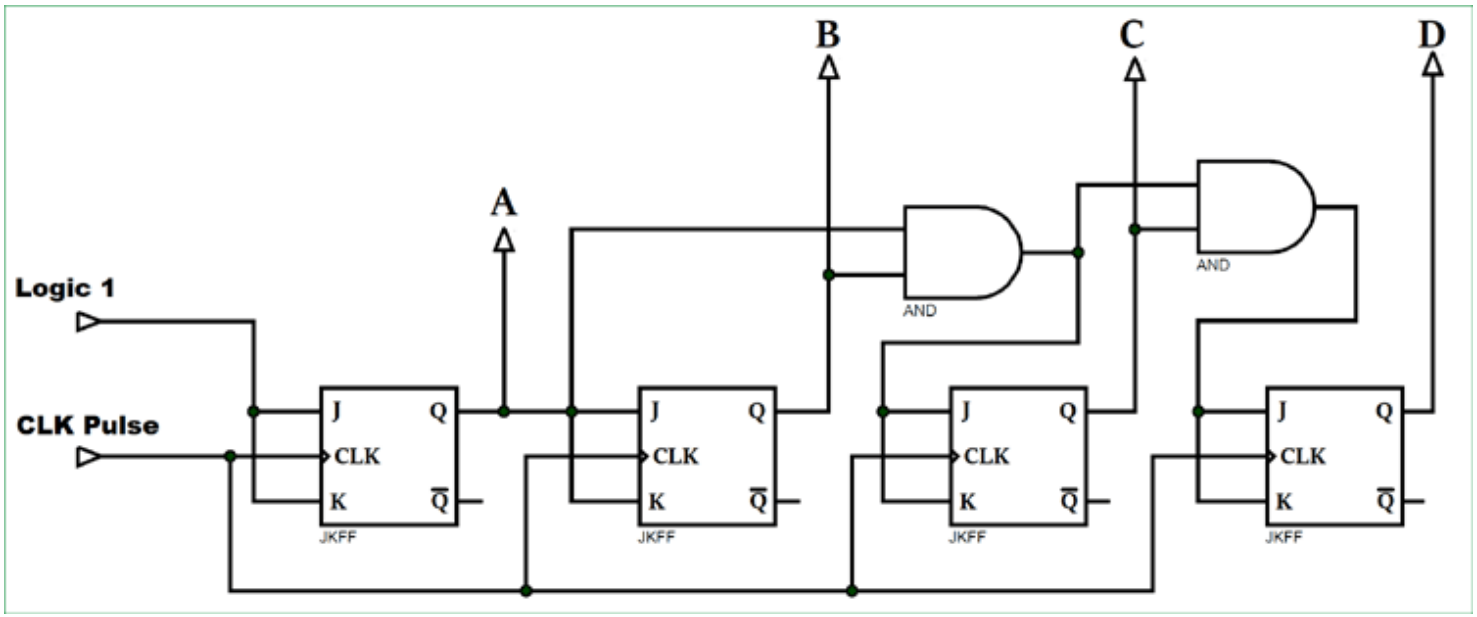


Figura: Contador síncrono - Fonte: `https://circuitdigest.com/tutorial/synchronous-counter`

Simulação do clock

- Fisicamente, um **gerador de *clock*** pode ser implementado com um cristal oscilador, usualmente de quartzo, associado a um circuito;

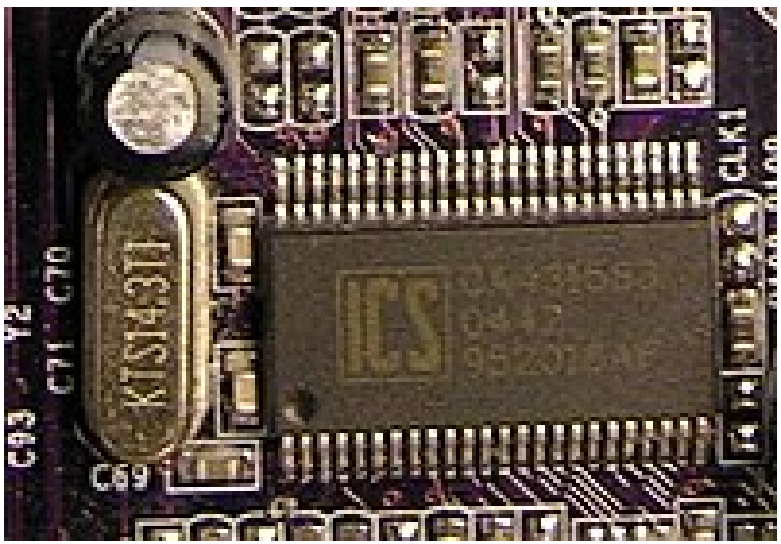


Figura: Gerador de clock - Fonte: `https://en.wikipedia.org/wiki/Clock_generator`

Simulação do clock

- Em VHDL, para gerarmos um sinal de *clock* para simulação, escrevemos um *process* com instruções **wait for**, que não são sintetizáveis;
- Lembrar que, na simulação, os *process* de uma *architecture* executam em paralelo, e que cada um executa indefinidamente (em *loop*) se não houver a instrução **wait;** no final.

Simulação do clock


```
architecture meu_tb_arch of meu_tb is

    signal clk: bit;

begin

    -- Clock
    clk_process: process begin
        clk <= '1';
        wait for 10 ns;
        clk <= '0';
        wait for 10 ns;
    end process;

    -- ...
```

Simulação do clock


```
architecture meu_tb_arch of meu_tb is

    signal clk: bit := '0';

begin

    -- Clock (process "resumido")
    clk <= not clk after 10 ns;

    -- ...
```

Simulação do clock

- **Problema:** Desse jeito, o sinal de *clock* é gerado indefinidamente, e a simulação nunca pára.
- **Solução:** Criar um sinal que habilita ou não o *clock*.

Simulação do clock


```
architecture meu_tb_arch of meu_tb is

    signal clk: bit := '0';
    signal simulando: bit;

begin

    -- Clock
    clk <= (simulando and (not clk)) after 10 ns;

    -- Estimulos
    stim_proc: process
    begin
        simulando <= '1';

        -- Faça o que precisa fazer...

        simulando <= '0';
        wait;
    end process;
```

Esperando por um evento em um process

Esperando por um evento em um process

- Certos circuitos sequenciais demoram vários ciclos de clock para executar uma operação (ex.: multiplicador sequencial);
- O criador do *test-bench* pode não saber de antemão quanto tempo esperar para coletar e comparar o resultado, nem saber precisamene quantos ciclos de *clock* aguardar;
- Alguns circuitos digitais sequenciais fornecem sinais de eventos. Ex.: um transmissor serial pode ter um sinal DONE que fica ativo quando a transmissão de um quadro de dados se encerra;

Esperando por um evento em um process

- Nesse caso, podemos usar a instrução **wait until** do VHDL;
- Exemplo:

```
wait until rising_edge(done);
```

Obrigado!


Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
PCS

Referências

D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*.
Springer US, 2009.