

3 *Flip-flops*, Contadores e Registradores

Objetivos São dois os objetivos deste laboratório: (i) verificar as diferenças de comportamento de contadores assíncronos e síncronos; e (ii) modelar e verificar o comportamento de um contador em anel e de um registrador de deslocamento com carga paralela.

Preparação Ler seções 2.2 a 2.5 de [RH12] sobre *flip-flops*, contadores e registradores.

3.1 Contadores Assíncronos e Síncronos

Da tarefa:

1. O trabalho pode ser efetuado em duplas;
2. copie para sua área de trabalho o arquivo
`wget http://www.inf.ufpr.br/roberto/ci210/vhdl/1_contadores.tgz`
Expand-a com `tar xzf 1_contadores.tgz`; um novo diretório `contadores` será criado. Mude para aquele diretório: `cd contadores`;
3. o *script* `run.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run.sh` somente (re)compila o simulador; com um argumento o *script* dispara a execução de `gtkwave`: `./run.sh a &`.

O arquivo `packageWires.vhd` contém definições de abreviaturas para nomes de sinais, e constantes para os tipos de propagação das portas lógicas.

O arquivo `aux.vhd` contém os modelos das portas lógicas *inv*, *and*, *or*, *xor* e de *flip-flops* (FFs) do tipo D e do tipo T, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `contador.vhd` contém quatro modelos de contadores de 4 bits. Os dois primeiros são contadores de 4 bits, um assíncrono (*contAssincrono4*) e um síncrono (*contSincrono4*). Estes modelos estão completos e serão usados para evidenciar a diferença nos comportamentos de circuitos síncronos e assíncronos. O terceiro modelo é um contador em anel (*contAnel4*), e o último um registrador de deslocamento com carga paralela (*regDesl4*). Estes contadores estão descritos na Seção 2.4 de [RH12].

O arquivo `tb_contador.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos modelos. A arquitetura do TB declara os componentes que serão testados e um **record** que será usado para excitar o modelo do registrador de deslocamento. O registro `test_record` possui três campos e os valores destes campos devem ser atribuídos de forma a gerar as combinações de entradas para garantir a corretude do modelo do registrador de deslocamento. O vetor de testes `test_array` contém alguns elementos para ilustrar as possibilidades.

No `test_record` do Prog. 1, os campos `e` e `s` são vetores de bits (prefixo `b`) e contém o valor de entrada do contador (usado somente no modelo `regDesl4`) e o valor esperado para a saída. O campo `c`, ativo em um, determina a carga de `e` no contador.

Prog. 1: Valores de entrada para testar o registrador de deslocamento.

```

type test_record is record
  e : reg4;          — entrada por carregar
  s : reg4;          — saída esperada
  c : std_logic;     — carrega registrador=0
end record;

type test_array is array(positive range <>) of test_record;

constant test_vectors : test_array := (
  — entr, saída, carga=0
  (b"0000", b"0000", '1'), — entradas são usadas somente
  (b"0000", b"0001", '1'), — com regDesl4
  (b"0000", b"0010", '1'), — e são ignoradas nos demais
  (b"0000", b"1001", '1'),
  (b"0000", b"1000", '0'), — carrega valores
  (b"0011", b"0000", '0'),
  (b"0000", b"0011", '0'),
  (b"0000", b"0001", '1'), — desloca
  (b"0000", b"0010", '1'),
  (b"0000", b"0011", '1')
);

```

A sequência de testes é implementada no 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`). Lembre que o processo `U_testValues` executa concorrentemente com os modelos dos somadores e quando os sinais de teste são atribuídos no laço, estes provocam alterações nos sinais dos modelos.

O comando **assert** verifica que a saída observada no contador é igual à esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto *correto, do ponto de vista dos vetores de teste que você escreveu*. Note que se você escolher valores de teste inadequados, ou errados, pode ser difícil identificar problemas no modelo.

```

assert rDesl = esperada
report "rDesl:␣saída=" & BV2STR(rDesl) & "␣esp=" & BV2STR(esperada)
severity error;

```

A função `BV2STR(v)` traduz um vetor de bits para uma *string* que pode ser impressa com o **assert**.

Ao final do laço a simulação termina no comando **assert** `FALSE`, que faz com que a execução do simulador se encerre.

O arquivo `s.vcd` contém definições para o `gtkwave` tais como a escala de tempo e sinais a serem exibidos na tela para a verificação dos modelos.

Da tarefa: (continuação)

1. verifique se os contadores `contAssincrono4` e `contSincrono4` operam corretamente;

Note que a saída do contador assíncrono é o complemento da saída do contador síncrono, e portanto este contador decrementa. Para que o contador assíncrono incrementasse, seria necessário alterar uma ligação em cada um dos FFs. Qual é esta ligação?

3.2 Temporização

No topo do arquivo `packageWires.vhd` estão as declarações das constantes com o tempo de propagação das portas lógicas definidas em `aux.vhd`. Edite `packageWires.vhd` e altere a definição da constante `simulate_time` (linha 17) de 0 para 1. Recompile os modelos e verifique seu funcionamento com `gtkwave`. Talvez seja necessário expandir a escala de tempo no `gtkwave` para que as diferenças entre os dois contadores fiquem mais evidentes.

Da tarefa: (continuação)

1. verifique a corretude dos contadores com um modelo para a temporização que é mais próximo da realidade;

3.3 Contador em Anel

O arquivo `contador.vhd` contém uma arquitetura incompleta para o modelo de um contador em anel. Complete o modelo e verifique sua corretude.

Da tarefa: (continuação)

1. complete o modelo do contador em anel;
2. verifique a corretude funcional do seu modelo com `gtkwave`, usando tempo de propagação zero para os FFs – edite `packageWires.vhd` e altere a definição da constante `simulate_time` na linha 17 de 1 para 0;
3. verifique a corretude temporal do seu modelo com `gtkwave`, usando tempo de propagação maior que zero para os FFs – edite `packageWires.vhd` e altere a definição da constante `simulate_time` na linha 17 de 0 para 1;

3.4 Registrador de Deslocamento com Carga Paralela

O arquivo `contador.vhd` contém uma arquitetura incompleta para o modelo de um contador com carga paralela. Complete o modelo e verifique sua corretude. O `assert` no TB deve ser des-comentado para esta parte do trabalho.

Da tarefa: (continuação)

1. complete o modelo do contador com carga paralela;
2. acrescente os vetores de teste que achar conveniente ao TB, e verifique a corretude do seu modelo;
3. verifique a corretude funcional do seu modelo com `gtkwave`, usando tempo de propagação zero para os FFs – edite `packageWires.vhd` e altere a definição da constante `simulate_time` na linha 17 de 1 para 0;
4. verifique a corretude temporal do seu modelo com `gtkwave`, usando tempo de propagação maior que zero para os FFs – edite `packageWires.vhd` e altere a definição da constante `simulate_time` na linha 17 de 0 para 1;

3.5 Mecanismo de Simulação em VHDL

Os simuladores gerados a partir de modelos VHDL empregam *simulação de eventos discretos* (*discrete event simulation*). Com esta técnica, as transições nos sinais ocorrem em instantes determinados pelo comportamento dos modelos da própria simulação. Uma simulação consiste de uma *fase de inicialização* e de uma série de *passos de simulação*, descritos no que se segue.

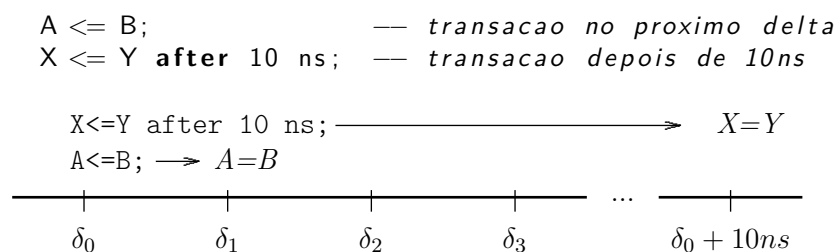
Quando a expressão do lado direito de uma atribuição é avaliada, uma *transação* é escalonada para um instante futuro no tempo simulado. Se a transação provoca uma mudança no sinal – de 0 para 1, por exemplo – então a transação provoca um *evento* naquele sinal.

Uma *transação* decorre da atribuição a um sinal, e a atribuição será efetivada no próximo *delta* (Δt), que é o próximo instante no tempo simulado.

Fase de inicialização: (i) todos os sinais são inicializados com os valores declarados, ou com os menores valores para cada tipo de sinal; (ii) o tempo simulado é inicializado em 0; e (iii) todos os processos são executados exatamente uma vez – processos são discutidos adiante;

passo de simulação: o tempo simulado avança até o próximo instante (*delta*) em que uma *transação* está programada; todas as transações programadas para aquele delta são executadas; estas transações podem provocar *eventos* em sinais; processos que dependem destes eventos são então executados; depois que todos os processos executaram, a simulação avança até o delta em que está programada a próxima transação.

Se o sinal muda de estado, então este sinal sofre um *evento*. Um evento no sinal S causa a execução dos processos que dependem de S no delta corrente. Um momento arbitrariamente distante no futuro pode de ser especificado para a ocorrência de uma transação. Por exemplo, a atribuição à A ocorrerá no próximo delta, enquanto que a atribuição à X ocorrerá após 10 ns, conforme indica o diagrama abaixo.



Na simulação de eventos discretos o tempo simulado avança em função de eventos nos sinais. Por causa de um evento ocorrido num sinal do lado direito de uma atribuição, aquela expressão é reavaliada, e se houver mudança no estado do sinal, o novo estado será atribuído ao sinal do lado esquerdo da atribuição no próximo delta, ou em instante determinado pelo programador.

Todas as expressões com sinais são avaliadas e os eventos terão efeito no próximo delta. Enquanto houver eventos programados para o delta corrente, estes são avaliados e eventos resultantes são escalonados para um próximo delta.

3.6 Processos em VHDL

O *processo* é a construção em VHDL que permite escrever *código sequencial*, que é executado na ordem do código fonte. Note que “código sequencial” não é o mesmo que “circuito sequencial” porque código sequencial pode ser usado para descrever circuitos combinacionais.

A declaração de um processo é mostrada no Prog. 2. O NOME é opcional, a lista de sensibilidade contém os sinais cujos eventos causam a execução do processo. Sinais que são locais ao processo são declarados entre o **process** e o **begin**. Estes sinais somente são visíveis no corpo do processo. Os *comandos sequenciais* são similares aos comandos de Pascal, e são executados na ordem do código fonte.

process

Prog. 2: Esqueleto de um processo.

```
NOME: process (lista de sensibilidade)
      declaracoes de sinais locais ao processo
begin
      comandos sequenciais
end
```

É possível escrever processos de tal forma que valores atribuídos a sinais num processo retenham seus valores até a próxima execução daquele processo, modelando assim circuitos sequenciais. A execução do código no corpo de um processo pode ser encarada como executando em tempo zero. Os processos executam concorrentemente com outros processos e com comandos concorrentes tais como atribuições e **when else**.

O processo no Prog. 3 é um modelo para uma *báscula (latch)*. Este processo somente será executado se ocorrer um evento nos sinais *set* e/ou *reset*. Ocorrendo um evento em qualquer dos sinais da lista de sensibilidade, o código do processo é executado, e o valor do estado da *báscula* é computado em função do estado atual dos sinais e das entradas. Note que *set* tem precedência sobre *reset* neste modelo. O sinal *estado* retém o seu valor entre execuções do processo.

lista de
sensibilidade

Prog. 3: Modelo para uma *báscula*.

```
U_bascula: process (set , reset)
      signal estado: std_logic;
begin

      if set = '1' then
          estado <= '1';
      elsif reset = '1' then
          estado <= '0';
      end if;

end process;
```

Os comandos são executados na ordem do código, e quando o último comando do corpo do processo é executado, a execução prossegue do primeiro comando.

Em *aux.vhd* há dois modelos de *flip-flops*, FFT e FFD. Naqueles modelos o estado do sinal interno ao processo, que mantém o estado do FF, se altera na borda do relógio caso as entradas *set* e *reset* estejam inativas. A função

`rising_edge(s)` retorna TRUE caso o evento mais recente no sinal `s` tenha sido uma mudança de 0 para 1.

Um processo *sem* lista de sensibilidade deve conter ao menos um comando **wait**. Na inicialização do simulador, todos os processos são executados uma vez. Processos com **waits** executam até que um **wait** seja encontrado, e então a execução fica suspensa até que alguma condição seja satisfeita. Por exemplo, o processo no Prog. 4 é o gerador de sinal de relógio do *testbench*.

wait

Prog. 4: Processo sem lista de sensibilidade, execução periódica.

```
U_relogio: process
begin
    rel <= '0';
    wait for CLOCK_PERIOD / 2;
    rel <= '1';
    wait for CLOCK_PERIOD / 2;
end
```

Na fase de inicialização o processo é executado e faz *rel=0* e então é suspenso até que decorra o tempo especificado no **wait for**, que no caso é meio período do relógio. Decorrido o intervalo, o processo volta a executar do ponto em que parou e faz *rel=1*, e então é suspenso novamente por meio período. Decorrido o tempo, a execução é retomada na primeira linha do processo, repetindo o ciclo, gerando portanto uma onda quadrada no sinal *rel*.

O processo no Prog. 5 é usado para inicializar os contadores. O processo é executado na fase de inicialização da simulação, quando ocorre a atribuição *tb_rst=0*, e a execução é paralisada até que decorra o intervalo especificado por *t_reset*, quando então ocorre a atribuição *tb_rst=1*. O último comando **wait**, sem nenhuma condição, causa a paralisação do processo, que não será executado novamente.

Prog. 5: Processo sem lista de sensibilidade que gera o sinal de *reset*.

```
U_reset: process
begin
    tb_rst <= '0';      — ativado na fase de inicializacao
    wait for t_reset;
    tb_rst <= '1';      — sinal fica inativo, para sempre
    wait;
end process;
```

Processos com lista de sensibilidade não podem conter comandos **wait**, e processos com comandos **wait** não tem uma lista de sensibilidade.

3.7 Atributos de sinais

Atributos fornecem informação adicional sobre vários tipos de objetos em VHDL. Alguns dos *atributos de sinais* são listados abaixo. Note que o atributo é separado do, ou ligado ao, nome do sinal pelo apóstrofo (`'`).

atributos

event S'event = TRUE se ocorreu um evento no sinal neste delta;

active S'active = TRUE se uma transação ocorreu no sinal neste delta;

last_value S'last_value retorna o valor de S, antes do último evento.

O trecho de Programa 6 permite determinar se ocorreu uma borda ascendente no sinal clk: o valor corrente do sinal é 1, ocorreu um evento neste delta (mudança de estado), e o valor de clk no delta anterior era 0, indicando uma transição de 0 para 1. A condição testada é equivalente ao corpo da função `rising_edge`, mencionada anteriormente.

Prog. 6: Detecção de borda ascendente com atributos.

```
if( clk='1' and clk'event and clk'last_value='0' ) then
    estado <= entrada;
end if
```

Referências

- [RH12] *Sistemas Digitais e Microprocessadores*, R.A.Hexsel, 2012, Editora da UFPR.
- [PJA04] *VHDL Tutorial*, <http://www.inf.ufpr.br/roberto/ci210/vhdl/vhdl-tutorial.pdf>

EOF
