5 Modelagem de Circuitos Sequenciais em VHDL

São quatro os objetivos deste laboratório: (i) apreender o conceito de *processo* em VHDL; (ii) efetuar a modelagem em VHDL de sistemas sequenciais síncronos; (iii) escrever um modelo para uma máquina de estados finita e (iv) verificar, através de simulação, a corretude do modelo. O trabalho pode ser efetuado em duplas.

A Seção 5.9 contém a(s) tarefa(s) por resolver nesta aula e que deve(m) ser enviada(s) ao professor.

5.1 Revisão: como funciona o mecanismo de simulação

VHDL emprega *simulação de eventos discretos*: os eventos ocorrem nos instantes determinados pela própria simulação. Por enquanto, consideremos que um *processo* é tão simples quanto uma atribuição; veremos em breve processos 'completos'. Uma simulação consiste de:

fase de inicialização todos os sinais são inicializados com os valores declarados, ou com os menores valores do tipo de cada sinal; tempo simulado $\leftarrow$ 0; todos os processos são executados exatamente uma vez;

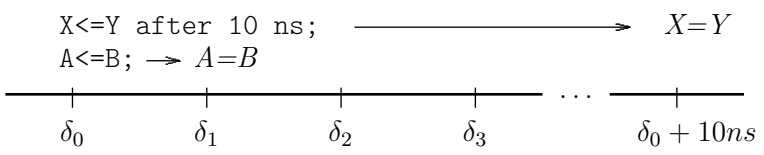
passo de simulação – primeira etapa o tempo simulado avança até o próximo instante em que uma transação está programada; todas as transações programadas para aquele instante são executadas; estas transações podem provocar eventos em sinais;

passo de simulação – segunda etapa processos que dependem dos eventos disparados na primeira etapa são então executados; depois que todos os processos executam, a simulação avança até o instante em que a próxima transação está programada.

Uma transação decorre da atribuição a um sinal; transação
– a atribuição será efetivada no próximo delta (Δt);
– se o sinal muda de estado, então este sinal sofre um evento, na primeira etapa; evento
– um evento no sinal S causa a execução dos processos que dependem de S, na segunda etapa.

Uma atribuição pode especificar o instante em que transação ocorrerá:

```
A <= B;                 -- transação no próximo delta
X <= Y after 10 ns;     -- transação depois de 10ns
```



Em nossas simulações, o femtosegundo (1fs = 10^{-15}s) é a menor unidade de tempo simulado, e ocorrem 1.000 Δt em cada fs.

Em resumo:
(1) O tempo simulado avança em função de eventos nos sinais;
(2) a cada mudança no estado de um sinal corresponde um evento;
(3) em função de um evento num sinal que aparece no lado direito de uma atribuição, a expressão é reavaliada, e se houver mudança no valor da expressão, o novo estado é atribuído ao sinal do lado esquerdo (primeira etapa);
(4) todos os comandos são avaliados e os eventos disparados passarão a ter efeito no próximo delta;
(5) enquanto houver eventos no delta corrente, estes são avaliados e eventos resultantes são programados para o próximo delta;
(6) processos que dependem dos eventos disparados neste delta são executados (segunda etapa);
(7) eventos causados pela execução dos processos são programados para o próximo delta; e
(8) eventos podem ser programados para ocorrer no futuro com **after**.

5.2 Comandos Sequenciais

Até agora empregamos somente os *comandos concorrentes* de VHDL. Estes comandos nos permitem escrever e verificar modelos para circuitos combinacionais pela interligação de componentes que são, eles próprios modelados como circuitos combinacionais.

Para que possamos escrever modelos para *flip-flops* e registradores temos que empregar os *comandos sequenciais* da linguagem. Estes comandos descrevem comportamento em termos de sequências de eventos e sua execução depende de ordem em que aparecem no código fonte. Nesse sentido, comandos sequenciais se assemelham àqueles da linguagem C.

Comandos sequenciais são 'executados' em tempo zero numa simulação, e *só podem ser usados em* ***processos***. Comandos sequenciais podem, também, ser usados para descrever lógica combinacional.

5.3 Processos

O *processo* é a construção de VHDL que encapsula um ou mais comandos sequenciais, que modelam uma determinada funcionalidade, como um registrador ou uma máquina de estados.

Processos são definidos dentro de arquiteturas, e numa arquitetura, todos os processos devem ter nomes distintos.

5.3.1 Processos *com* lista de sensibilidade

Processos podem ser usados com uma *lista de sensibilidade*, como mostrado no trecho de código abaixo.

```
nomeDoProcesso: process (listaDeSensibilidade)
  declarações;
begin
  comando_sequencial_1;
  comando_sequencial_2;
  comando_sequencial_3;
end process nomeDoProcesso;
```

Na inicialização de uma simulação, todos os processos são executados exatamente uma vez. Nos passos de simulação, quando ocorre um evento num sinal da *lista de sensibilidade* o processo é escalonado para execução na segunda etapa do delta corrente e os comandos sequenciais são executados na ordem do código fonte.

Se a execução dos comandos provoca eventos em sinais, estes eventos são escalonados para o próximo delta, ou para um instante futuro se a cláusula **after** for usada.

Quando a execução chega ao final do corpo do processo, ela é retomada do primeiro comando sequencial, no topo do processo. O corpo do processo se assemelha a um laço infinito, que só é executado quando ocorrer um evento em sinais da sua lista de sensibilidade. Esta é uma descrição inexata, mas ela será retificada em breve.

Exemplo 5.1 O Programa 1 mostra um exemplo de processo com lista de sensibilidade. Sempre que ocorrer um evento no sinal clk, o corpo do processo é executado *em tempo zero*, de alto abaixo.

Se o evento em clk é tal que a função rising_edge() retorna TRUE, então o valor do sinal D é atribuído ao sinal Q. Se nenhuma outra atribuição é feita ao sinal Q, então ele preserva seu valor até o próximo evento em que rising_edge(clk)=TRUE. A função rising_edge() é definida na Seção 5.8 ◁

Programa 1: Primeira versão de um *flip-flop* D.

```
FF0: process (clk) -- somente clk na lista de sensibilidade
begin
  if rising_edge(clk) then
    Q <= D;
  end if;
end process FF0;
```

5.3.2 Processos *sem* lista de sensibilidade

Processos podem ser usados sem uma *lista de sensibilidade*. Estes processos devem conter um ou mais comandos que suspendem temporariamente a execução do processo, tal como o comando **wait**. Quando um destes comandos é executado, o processo fica suspenso até que a condição de espera seja resolvida. No código abaixo, quando o comando **wait on** é executado e a condição for falsa, a execução do processo é suspensa até que a condição de espera se torne verdadeira, quando então a execução é retomada no comando após o **wait**.

```
nomeDoProcesso: process  -- sem lista de sensibilidade
  declarações;
begin
  ...
  wait on (condição) ;   -- comando que suspende execução
  x <= ...;              -- reinicia neste ponto
end process nomeDoProcesso;
```

Processos podem ou conter uma lista de sensibilidade ou conter comandos **wait**, mas nunca os dois num mesmo processo.

Exemplo 5.2 Vejamos um exemplo simples de processo *sem* lista de sensibilidade, que é um *gerador de sinal de relógio*. Seja o período do relógio definido pela constante

```
constant CLOCK_PER : time := 50 ns;
```

podemos escrever um processo que executa periodicamente, e a cada execução muda o nível lógico do sinal de relógio, como mostra o Programa 2.

Programa 2: Gerador de relógio.

```
1 U_clock: process            -- sem lista de sensibilidade
2 begin
3   clk <= '0';               -- executa
4   wait for CLOCK_PER / 2;   -- espera meio ciclo
5   clk <= '1';               -- volta a executar
6   wait for CLOCK_PER / 2;   -- espera meio ciclo e volta ao topo
7 end process U_clock;
```

A linha 1 declara um processo chamado U_clock, sem lista de sensibilidade. Na linha 3 o sinal de relógio muda para 0, na linha 4 a execução fica suspensa até que decorra meio período do relógio, quando então a linha 5 é executada e clk muda para 1. Na linha 6 o processo é novamente suspenso. Decorrido meio período, a linha 3 é executada, e o ciclo se repete *ad aeternum*.

Quando a simulação é iniciada, o processo U_clock executa a linha 3 e se suspende na linha 4, inicializando o sinal clk em 0. ◁

Exemplo 5.3 O processo U_reset, no Programa 3, inicializa o sinal reset em 0 na linha 3 e se suspende por 3/4 de ciclo na linha 4. Decorrido este intervalo, reset muda para 1 e o processo se suspende "para sempre" na linha 6. ◁

Programa 3: Gerador de *reset*.

```
1 U_reset: process              -- sem lista de sensibilidade
2 begin
3   reset <= '0';               -- executa e
4   wait for CLOCK_PER * 0.75; -- espera por 37,5ns
5   reset <= '1';               -- volta a executar e
6   wait;                       -- se suspende para sempre
7 end process U_reset;
```

5.4 O que é um *flip-flop*?

Consultando as fontes em papel que estão disponíveis na minha estante, e tomado de alguma surpresa, passo a desvelar à dileta leitora que conceitos que emprego há umas tantas décadas são definidos em dicionários de alta qualidade como sendo:

flip-flop circuit *noum: an electronic circuit with two permanently stable conditions (as when one electron tube is conducting while the other is cut off) so that conduction is switched from one to the other by succcessive pulses*, Webster's Third New Internactional Dictionary;

flip-flop *verb: to change your opinion about something, especially when you then hold the opposite opinion*, Oxford Advanced Learner's Dictionary;

latch *noum: to catch or fasten by means of a latch*, Webster's Third New Internactional Dictionary;

basculador *(1)* ELETRÔN *circuito flip-flop*, Dic. Houaiss da Língua Portuguesa[1]; e

básculo *(1) ponte levadiça com mecanismo de contrapeso, (2) peça móvel de metal ou ferro, que gira apoiada num pino, destinada a abrir ou fechar ferrolhos de portas, janelas, etc*, Dic. Houaiss da Língua Portuguesa.

5.5 *Flip-flops* em VHDL

Em se tratando de sistemas digitais, um *flip-flop* é um circuito cuja saída, em condições normais de operação, pode estar em um de dois possíveis estados: ou a saída está no estado *flip*, ou está no estado *flop* – um *flip-flop* é um circuito *biestável* cujo comportamento é definido para somente uma de duas condições: ou sua saída é estável em 1, ou é estável em 0. Um *flip-flop* é um modelo para uma ponte levadiça ou um ferrolho: ou estão abertos, ou estão fechados.

Exemplo 5.4 ***Flip-flop*** **com wait** Vejamos o código que modela um *flip-flop*, empregando um processo sem lista de sensibilidade. A entidade FF0 descreve a interface de um *flip-flop* tipo D simples, com entrada D, relógio, e saída Q, e é mostrada no Programa 4.

A arquitetura waitSimples modela um *flip-flop* tipo D. O atributo 'event indica ocorrência de evento no sinal clk. O processo executa e fica bloqueado no **wait** a esperar por um evento no sinal clk, e além disso, que o evento faça clk='1'. Estas duas condições equivalem a uma borda ascendente no sinal clk: ocorreu uma mudança, e a mudança foi para 1. Após a detecção da borda, a entrada D é copiada para o sinal Q, que mantém seu estado até a próxima borda em clk.

Quando o processo é desbloqueado no **wait**, a execução prossegue até o fim do processo e é retomada no seu topo. Nesse exemplo, após o desbloqueio acontece a atribuição (Q<=D;) e a execução do processo é logo interrompida novamente no **wait**. ◁

[1]Este me parece um cacófato assaz deselegante. O autor, que é adepto do *uber*-dicionarista, admite que emprega, desavergonhadamente, a forma incorreta 'básculo' ao invés do horrendo 'basculador'. Atravessemos o Rubicão sem mais delongas.

Programa 4: *Flip-flop* com wait.

```
1  entity FF0 is
2    port(D:   in  bit;
3         clk: in  bit;
4         Q:   out bit);
5  end FF0;
6
7  architecture waitSimples of FF0 is
8  begin
9    FF: process
10   begin
11      wait for ( clk'event and clk = '1' ); -- borda em clk
12      Q <= D;
13   end process FF;
14 end architecture waitSimples;
```

Exemplo 5.5 ***Flip-flop*** **com lista de sensibilidade** A arquitetura listaSimples, no Programa 5, emprega uma lista de sensibilidade ao invés de um comando **wait** no corpo do processo. A lista de sensibilidade junto à palavra reservada **process** contém somente o sinal clk, e sempre que ocorrer um evento neste sinal, o processo FF é executado. O corpo de um processo com lista de sensibilidade é executado do início ao final sempre que ocorrer um evento num dos sinais da lista.

Programa 5: *Flip-flop* com lista de sensibilidade.

```
1  architecture listaSimples of FF0 is
2  begin
3    FF: process (clk)  -- eventos em clk causam execução
4    begin
5       if clk = '1' then       -- valor pós-evento = '1'
6          Q <= D;
7       end if;
8    end process FF;
9  end architecture listaSimples;
```

O corpo do processo, que é o comando **if**, só é executado se ocorrer um evento em clk. Se, após o evento clk='1', então ocorreu uma borda ascendente naquele sinal e a entrada deve ser registrada; se clk tem qualquer outro valor após o evento, não ocorreu uma borda ascendente, e o valor em Q deve permanecer como estava. O corpo do processo é 'executado' *em tempo zero*, somente após eventos no sinal *clk*. ◁

Exemplo 5.6 ***Flip-flop*** **com *reset* assíncrono** Vejamos como modelar um *flip-flop* com uma entrada de *reset* assíncrona. A entidade FF1, no Programa 6, declara duas entradas de controle, clk e rst. O processo na arquitetura rstAssincrono tem dois sinais na sua lista de sensibilidade, clk e rst. O **if** faz Q='0' sempre que ocorrer um evento em rst, e o evento fizer rst=0. O **elsif** testa a ocorrência de uma borda ascendente em clk, e se este for o caso, a entrada D é registrada. Em qualquer outra combinação nas entradas, o sinal Q mantém o seu valor.

A execução do processo pode ocorrer tanto por eventos em rst quanto em clk. É por isso que a cláusula **elsif** verifica se ocorreu um evento em clk, e mais ainda, se clk=1 após o evento. ◁

Programa 6: *Flip-flop* com *reset* assíncrono.

```
1  entity FF1 is
2    port(D:         in  bit;
3         clk, rst: in  bit;
4         Q:         out bit);
5  end FF1;
6
7  architecture rstAssincrono of FF1 is
8  begin
9    FF: process (clk, rst)  -- lista de sensibilidade
10   begin
11     if rst = '0' then
12       Q <= '0';
13     elsif clk'event and clk = '1' then  -- borda em clk
14       Q <= D;
15     end if;
16   end process FF;
17 end architecture rstAssincrono;
```

Exemplo 5.7 ***Flip-flop*** **com *set* e *reset* assíncronos** A entidade FF2, no Programa 7, declara três entradas de controle, clk, rst e set.

O processo na arquitetura setclr tem três sinais na sua lista de sensibilidade. O comando **if** coloca a saída Q em 0 caso rst='0'; o primeiro **elsif** a coloca em 1 caso set='0', ou registra a entrada D caso ocorra uma borda ascendente em clk. Em qualquer outra combinação de entradas, Q mantém o seu valor.

Este exemplo evidencia a distinção entre a lista de sensibilidade e o uso de **wait**s: com a lista de sensibilidade, caso ocorram eventos nos três sinais no mesmo delta, o código do processo decide a ordem de avaliação e as consequências dos eventos. Se **wait**s forem usados, a lógica das condições de espera pode ficar mais complexa, e o código confuso e de difícil compreensão. ◁

Programa 7: *Flip-flop* com *set* e *reset* assíncronos.

```
1  entity FF2 is
2    port(D:             in  bit;
3         clk, rst, set: in  bit;
4         Q:             out bit);
5  end FF2;
6
7  architecture setclr of FF2 is
8  begin
9    FF: process (clk, rst, set)  -- lista de sensibilidade
10   begin
11     if rst = '0' then          -- reset tem precedência
12       Q <= '0';
13     elsif set = '0' then
14       Q <= '1';
15     elsif clk'event and clk = '1' then -- borda em clk
16       Q <= D;
17     end if;
18   end process FF;
19 end architecture setclr;
```

Num *flip-flop* real, a ocorrência de rst='0' e set='0' é uma condição proibida e que viola a especificação temporal daquele circuito. Caso esta situação ocorra, o valor no sinal Q é indeterminado, e por causa da metaestabilidade, pode permanecer indeterminado por um tempo também indeterminado. O comportamento modelado na arquitetura setClr é uma versão idealizada do circuito e que abstrai a possibilidade de metaestabilidade para simplificar a modelagem e a simulação.

5.5.1 Equivalência entre *lista de sensibilidade* e *wait*

O Programa 8 mostra dois processos com comportamento equivalente, o primeiro com lista de sensibilidade e o segundo com um comando **wait**.

Programa 8: Processos com lista de sensibilidade e com wait.

```
architecture doisProcessos of SisSeqSin is
begin

  FFlista: process (clk, rst)  -- COM lista de sensibilidade
    begin
      if rst = '0' then
          Q <= '0';
      elsif clk'event and clk = '1' then
          Q <= D;
      end if;
    end process FFlista;

  FFwait: process              -- SEM lista de sensibilidade
    begin
      if rst = '0' then
          Q <= '0';
      elsif clk'event and clk = '1' then
          Q <= D;
      end if;
      wait on (clk, rst);      -- espera eventos em clk, rst
    end process FFwait;

end architecture doisProcessos;
```

5.5.2 Sinais e Variáveis

Enquanto falávamos da modelagem de circuitos combinacionais, sinais eram os equivalentes aos fios de um circuito combinacional – a cada delta, o valor atribuído a um sinal é o resultado da avaliação do lado direito da expressão – RHS ou *Right Hand Side*, ou o *rvalue*, no jargão de compiladores – que lhe atribui valor.

RHS, rvalue

Quando modelamos circuitos sequenciais, alguns sinais "adquirem memória" e mantém seus valores entre as execuções dos processos que os manipulam. Os valores transportados por estes sinais são alterados pela combinação de eventos nas listas de sensibilidade dos processos, e do código sequencial que determina seus valores em função dos eventos apropriados. Sinais somente são atualizados no próximo delta, ou no evento "próxima borda do sinal de relógio" em circuitos sequenciais síncronos.

Frequentemente é necessário empregar 'fios' que transportam informação instantaneamente entre os comandos de um processo e estes 'fios' não podem ser sinais, porque estes somente são atualizados em um delta futuro, e não no delta corrente. Quando é necessária a comunicação instantânea de valores entre os comandos de um processo, deve-se empregar variáveis. Uma *variável* transporta valores entre comandos sequenciais em processos porque um novo valor é atribuído à variável imediatamente, e não num delta futuro.

variável

Variáveis só podem ser declaradas dentro de processos, e a atribuição a uma variável é denotada por ':= ', como `v := '1';`.

As portas de saída de uma entidade não podem ser lidas numa arquitetura ou processo. Para que seja possível ler valores de estado que são atribuídos às saídas, devem ser empregadas variáveis para manter o estado do modelo, e estas variáveis são atribuídas às portas de saída da entidade.

porta de saída não pode ser lida

Veremos exemplos em breve.

5.5.3 Atributos

Atributos fornecem informação adicional sobre vários tipos de objetos em VHDL. Dentre muitos, nos interessam os *atributos de sinais* e os *atributos de vetores*.

Para um sinal `S`, nos interessam os atributos:

`S'event` é *true* se ocorreu evento no sinal neste delta;
`S'active` é *true* se alguma transação ocorreu no sinal neste delta;
`S'last_value` retorna o valor de S, antes do último evento.

Já empregamos a atributo `'event` nos modelos dos *flip-flop*s. A condição para a detecção da borda pode ser ainda mais estrita, ao se verificar que o valor no delta anterior era mesmo '0' e não um valor inválido, por exemplo.

```
if( clk'event and clk='1' and clk'last_value='0' ) then
  ...
end if;
```

Para um vetor `V`, nos interessam os atributos:

`V'length` retorna o número de elementos de um vetor;
`V'range` retorna a faixa de índices de um vetor, ou se o vetor é um tipo restringido (**subtype**), retorna o conjunto de valores do subtipo.

Por exemplo, para os dois vetores de 12 elementos, as suas faixas são aquelas declaradas, *viz.*:

```
V: bit_vector(11 downto 0);
W: bit_vector(0 to 11);
...
... V'range ...  -- faixa de 11 a 0
...
... W'range ...  -- faixa de 0 a 11
...
```

5.5.4 Processos na modelagem de lógica combinacional

Processos podem ser usados para modelar lógica combinacional. O código de um "processo combinacional" deve evitar que sinais mantenham o mesmo estado entre duas execuções do processo – o processo não pode ter memória. Para tanto, o código do processo deve obedecer à três regras:

Regra 1 a lista de sensibilidade contém todos os sinais de entrada (sinais lidos no RHS, ou *rvalues*) usados pelo processo;

Regra 2 as atribuições às saídas do processo cobrem todas as combinações possíveis das entradas do processo;

Regra 3 a todas as variáveis no processo deve ser atribuído um valor antes que elas sejam usadas como entrada no RHS (*rvalue*).

Exemplo 5.8 Um multiplexador de quatro entradas, que é um circuito combinacional, pode ser modelado com um processo, como mostra o Programa 9. O processo mux satisfaz à Regra 1 porque a lista de sensibilidade contém *todos* os sinais que são lidos pelo processo, tanto o sinal Sel que escolhe uma das entradas, quanto as quatro entradas (A,B,C,D). A Regra 2 é obedecida porque, nas quatro combinações da entrada Sel, ocorre uma atribuição ao sinal Y. O processo não emprega variáveis e portanto a Regra 3 é trivialmente satisfeita.

Esta não é a maneira mais eficiente de se codificar um multiplexador porque as entradas são avaliadas em sequência, e na síntese este código resulta num circuito mais complexo do que o necessário. Como está, o circuito resultante da síntese envolve uma cadeia de quatro *and*s. O comando **if** é descrito na Seção 5.6.1.

Em breve veremos um comando VHDL que modela um multiplexador que é sintetizado eficientemente. A *síntese* de um modelo é o processo de compilação que produz uma descrição de baixo nível um circuito – lista de conexões – ao invés de um simulador. ◁

Programa 9: Multiplexador modelado com um processo.

```
1  entity simpleMux is
2    port (Sel:         in  bit_vector(0 to 1);
3          A, B, C, D: in  bit;
4          Y:          out bit);
5  end simpleMux;
6
7  architecture ineficiente of simpleMux is
8  begin
9    mux: process (Sel,A,B,C,D)  -- lista de sensibilidade
10   begin
11     if    Sel = "00" then  Y <= A; -- todas as 4
12     elsif Sel = "01" then  Y <= B; --  combinações
13     elsif Sel = "10" then  Y <= C; --  para 2 bits
14     elsif Sel = "11" then  Y <= D;
15     end if;
16   end process mux;
17 end architecture ineficiente;
```

5.5.5 Processos na modelagem de lógica sequencial

Para descrever lógica sequencial, portanto com memória, processos devem obedecer à três regras:

Regra 1 a lista de sensibilidade não inclui todos os sinais no RHS (*rvalue*) das atribuições;

Regra 2 a lógica **if**−**then**−**elsif** é incompletamente especificada, indicando que um ou mais sinais/variáveis devem manter seus valores entre execuções do processo;

Regra 3 uma ou mais variáveis mantêm seus valores entre execuções do processo – variáveis podem ser lidas (RHS) antes que um valor lhes tenha sido atribuído.

Quando um processo atende às Regras 1 e 2, e opcionalmente à 3, durante a síntese o compilador gera um circuito com *flip-flop*s e registradores para manter os valores entre as execuções do processo, que tipicamente ocorrem na borda do sinal de relógio, ou quando sinais de controle assíncronos (reset) são ativados.

Programa 10: Modelo para um registrador de rotação.

```
1  entity rotate is
2    port (clk, rst, load: in  bit;
3          data:           in  bit_vector(0 to 7);
4          Q:              out bit_vector(0 to 7));
5  end rotate;
6
7  architecture correta of rotate is
8  begin
9    reg: process (rst, clk)
10     variable Qvar:bit_vector(0 to 7);
11   begin
12     if rst = '1' then          -- inicialização assíncrona
13       Qvar := "00000000";
14     elsif (clk'event and clk = '1') then
15       if load = '1' then                -- carga síncrona
16         Qvar := data;
17       else                             -- rotação síncrona
18         Qvar := Qvar(1 to 7) & Qvar(0);
19       end if;
20     end if;
21     Q <= Qvar;     -- atribui estado ao sinal da interface
22   end process;
23 end architecture correta;
```

Exemplo 5.9 A entidade rotate, no Programa 10 modela um registrador de deslocamento com 8 bits, que rotaciona seu conteúdo a cada ciclo do relógio.

Quando o sinal load é ativado, o registrador é carregado com o valor em sua entrada data. O processo reg declara a variável Qvar para manter o conteúdo do registrador entre os eventos no sinal de relógio.

Se o sinal rst é ativado, o registrador é carregado com zero. A cada borda do relógio, se o sinal load está ativo, um novo valor é atribuído à Qvar.

Do contrário, o conteúdo de Qvar é rotacionado de uma posição – o bit Q_0 é inserido na posição do bit Q_7, e os demais deslocam-se de uma posição.

Ao final da execução do processo, o conteúdo da variável Qvar é atribuído ao sinal da interface Q. O valor de Qvar, que mantém o estado do registrador, é lido na linha 18 – é usado no RHS – e portanto o sinal da interface Q não pode ser usado para manter o conteúdo do registrador.

A Regra 1 é atendida porque o sinal load não está na lista de sensibilidade. A Regra 2 é atendida porque os **if**s não cobrem todas as oito combinações possíveis dos três sinais de controle. A Regra 3 é atendida porque, dependendo das combinações das variáveis de controle (rst=1 e load=0), o valor em Qvar pode ser lido antes de que algum valor tenha sido atribuído àquela variável. ◁

Há uma diferença sutil porém importante entre variáveis e sinais: variáveis são atualizadas instantaneamente enquanto que sinais somente são atualizados no próximo delta, ou em momento futuro. No Programa 11, a arquitetura ERRADA da entidade rotate é praticamente a mesma que a arquitetura sequencial – a diferença é que, em ERRADA, o valor memorizado em Qsig é mantido num sinal e não numa variável. Suponha que não ocorram eventos no sinal rst, e portanto o processo reg só é executado nas bordas do relógio. O comportamento do modelo é aquele esperado pelo programador?

Programa 11: Modelo *errado* para o registrador de rotação.

```
1  architecture ERRADA of rotate is
2  begin
3    reg: process (rst, clk)
4      signal Qsig:bit_vector(0 to 7);          -- SIGNAL
5    begin
6      if rst = '1' then
7        Qsig <= "00000000";
8      elsif (clk = '1' and clk'event) then
9        if load = '1' then
10         Qsig <= data;
11       else
12         Qsig <= Qsig(1 to 7) & Qsig(0);
13       end if;
14     end if;
15     Q <= Qsig;     -- atribui estado ao sinal da interface
16   end process;
17 end architecture ERRADA;
```

O comportamento do modelo não é o esperado pelo programador porque a atualização do sinal da interface Q só terá efeito visível na próxima execução do processo reg, embora os dois **if**s computem o novo valor de Qsig no delta corrente. Este sinal, Qsig, é atualizado no delta corrente mas a atribuição à Q somente acontecerá na próxima execução do processo, no delta em que ocorrer um evento em clk, portanto com um ciclo de atraso em relação ao que se deseja.

5.6 Comandos Sequenciais

Vejamos quais comandos podem ser empregados em processos. Estes são chamados de *comandos sequenciais* porque são executados na ordem do código fonte, de forma similar aos comandos da linguagem C.

5.6.1 Comando Sequencial IF-THEN-ELSE

Já vimos vários usos do comando **if** nos modelos dos *flip-flop*s. O trecho abaixo mostra o comando **if** em toda sua glória. Se condição1 é verdadeira, então os comandos em comandos1 são executados; do contrário, condição2 é avaliada, e se verdadeira, comandos2 são executados; do contrário as duas condições são falsas e comandos3 são executados.

```
if condição1 then      -- condição deve ter tipo Boolean
  comandos1;
elsif condição2 then   -- cláusula opcional
  comandos2;
else                   -- cláusula opcional
  comandos3;
end if;
```

As cláusulas **elsif** e **else** são opcionais, e **if**s aninham da forma óbvia.

```
if cond_externa then  -- if's aninhados
  comandos_externos;
else
   if cond_interna then
      comandos_internos;
   end if;
end if;
```

5.6.2 Comando Sequencial CASE

O comando sequencial **case** é similar ao *switch* da linguagem C. A expressão de escolha é avaliada e a cláusula que 'casar' com o resultado da avaliação é executada.

No exemplo abaixo a expressão controle é avaliada e se uma das cláusulas teste_i for selecionada, o respectivo comando_i é executado. A palavra reservada **others** 'casa' todos os valores que não estão explicitamente listados e deve ser a última cláusula do comando. Uma cláusula pode conter mais de um valor, sendo estes separados por uma barra vertical.

```
case controle is
  when teste1 =>
    comando1;
  when teste2 | teste3 =>  -- dois casos por testar
    comando2;
  when others =>           -- todos os casos não cobertos acima
    comandoOthers;
end case;
```

Os testes devem ser mutuamente exclusivos, e todas os possíveis valores de controle devem ser cobertos, nem que seja por **others**.

Ao contrario do **if**−**then**−**else**, no **case** não há prioridade na avaliação das cláusulas e portanto este comando é mais adequado para a modelagem de circuitos tais como o multiplexador, como mostra a arquitetura eficiente para o multiplexador de quatro entradas no Programa 12. Este comando é sintetizado como um multiplexador de quatro entradas, e não como uma sequência de **if**s encadeados.

Programa 12: Modelo eficiente para processo do multiplexador.

```
1  architecture eficiente of simpleMux is
2  begin
3    mux: process (Sel,A,B,C,D)  -- lista de sensibilidade
4    begin
5      case Sel is
6        when "00" => Y <= A;  -- todas as quatro
7        when "01" => Y <= B;  --   combinações
8        when "10" => Y <= C;  --   estão cobertas
9        when "11" => Y <= D;
10     end case;
11   end process mux;
12 end architecture eficiente;
```

Verifique se este processo obedece às três regras para processos que modelam circuitos combinacionais.

5.6.3 Comando Sequencial FOR

O comando **for** de VHDL é muito mais expressivo do que seu insípido primo da linguagem C. Infelizmente, este comando não é sintetizável e é usado apenas em *testbenches* para gerar sequências de valores de teste.

A variável de indução i tem escopo somente no corpo do laço, e encobre outra variável com mesmo nome que esteja declarada em escopo externo ao **for**. O tipo da variável de indução é inferido pelo compilador.

```
for i in faixa loop
  comando;
end loop;
```

O valor de i cobre todos os valores em faixa e portanto i pode iterar sobre um subconjunto dos inteiros, ou sobre elementos de um tipo, ou sobre o conjunto de índices de um vetor.

Vejamos um exemplo em que a faixa de valores de laços pode ser determinada de três formas diferentes: as duas últimas empregam atributos de vetores de bits para determinar a faixa de valores das variáveis de indução j e k – o tamanho do vetor pode ser alterado pelo programador e o compilador recomputará a nova faixa de valores para indexar o vetor. Os três laços determinam a paridade[2] do vetor de bits D e são mostrados no Programa 13.

Espaço em branco proposital.

[2]Número de bits em 1.

Programa 13: Exemplos de indexação de laço for.

```
1  signal D: bit_vector(0 to 9);
2
3  paridade3: process(D)
4    variable otmp: Boolean := FALSE;
5  begin
6    ...
7    for i in 0 to 9 loop  -- faixa definida explicitamente
8      if D(i) /= '1' then otmp := not otmp; end if;
9    end loop;
10   ...
11   for j in 0 to (D'length - 1) loop  -- comprimento do vetor
12     if D(j) /= '1' then otmp := not otmp; end if;
13   end loop;
14   ...
15   for k in D'range loop  -- faixa de valores do índice
16     if D(k) /= '1' then otmp := not otmp; end if;
17   end loop;
18   ...
19 end process paridade3;
```

5.6.4 Comando Sequencial WHILE

O comando **while** é similar ao da linguagem C: enquanto a condição for verdadeira, os comandos no corpo do laço são executados. Este comando não é sintetizável e é usado para gerar sequências de valores para testes, ou para ler os caracteres de um arquivo de texto, por exemplo.

```
while condição loop
  comandos;
end loop;
```

5.6.5 Comando Sequencial LOOP

O comando sequencial **loop** é usado para efetuar tarefas repetitivas e tampouco é sintetizável. O laço pode ser terminado com uma cláusula **exit when**; se a condição avaliar como verdadeira, o laço termina.

```
nomeDoLoop: loop
  ...
  exit when condição;
  ...
end loop nomeDoLoop;
```

Este laço tem um *label* que pode ser usado para interromper a execução de laços aninhados; e o *label* do laço que se deseja interromper é usado no **exit**. No exemplo abaixo, quando a condição é verdadeira, a execução dos dois laços é interrompida.

```
Externo: loop
   ...
   Interno: loop
      ...
      if condição then
         exit Externo; -- salta para fora dos dois laços
      end if;
      ...
   end loop Interno;
   ...
end loop Externo;
```

5.7 Modelagem de máquinas de estado

Empregaremos o estilo de modelagem de Máquinas de Estado (MEs) descrito em [PJA08]: são necessários dois processos, um para o registrador de estado, e um processo combinacional para as funções de próximo estado e de saída. Dependendo da complexidade da ME, pode ser conveniente separar a função de saída da função de próximo estado.

O Programa 14 é um protótipo para as MEs que empregaremos. Um tipo (states) é definido para os estados, com um elemento do tipo para identificar cada um dos estados. Dois sinais do tipo states são declarados para armazenar o estado atual e o próximo estado.

Programa 14: Modelo para MEs.

```
1  type states is (s0, s1, s2, s3);   -- novo tipo para estados
2  signal curr_st, next_st : states;  -- sinais do novo tipo
3
4  U_state_reg: process(reset,clk)    -- registrador de estado
5  begin                              -- lógica sequencial
6     if reset = '0' then
7        curr_st <= s0;               -- estado inicial
8     elsif rising_edge(clk) then
9        curr_st <= next_st;          -- troca de estado na borda
10    end if;
11 end process U_state_reg;
12
13 U_st_transitions: process(curr_st) -- função de próx estado
14 begin                              -- lógica combinacional
15    case curr_st is
16        when s0 => next_st <= s1;
17        when s1 => next_st <= s2;
18        when s2 => next_st <= s3;
19        when s3 => next_st <= s0;
20    end case;
21 end process U_state_transitions;
```

O processo U_state_reg mantém o estado atual, e nas bordas do relógio, faz com que o próximo estado torne-se o estado atual. Quando reset=0, a ME é colocada em seu estado inicial.

O processo U_st_transitions computa o próximo estado em função do estado atual. A cada evento no sinal curr_st, disparado por uma borda no relógio no processo

U_state_reg, o novo estado é computado em função do estado atual, e na próxima borda o estado atual progredirá para o próximo estado. Este processo modela lógica combinacional.

Espaço em branco proposital.

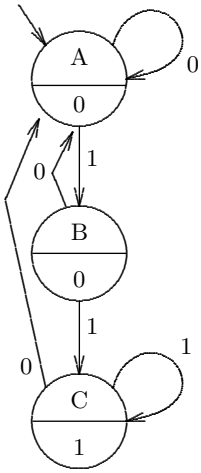
Exemplo 5.10 Máquina de Moore Vejamos um exemplo completo: uma *Máquina de Moore* que detecta sequências de dois ou mais **1**s em sua entrada. O diagrama de estados da ME é mostrado ao lado do código no Programa 15.

A linha 1 declara o tipo para os estados e a linha 2 declara os sinais para o estado atual e o próximo estado. O sinal found é a saída da ME. O processo U_state_reg mantém o estado atual e o atualiza nas bordas do relógio. O estado inicial é o estado A. O processo U_st_trans computa o próximo estado em função do estado atual e das entradas, e define a saída da ME estado a estado.

Verifique se o código do processo U_st_trans corresponde à função de próximo estado codificada no diagrama de estados. Verifique também se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◁

Programa 15: Máquina de Moore para 11^+.

```
1  type states is (A, B, C);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7    if reset = '0' then
8       curr_st <= A;
9    elsif rising_edge(clk) then
10      curr_st <= next_st;
11   end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin              -- Máquina de Moore
16   case curr_st is
17     when A =>
18       if entr = '0' then
19         next_st <= A;
20       else
21         next_st <= B;
22       end if;
23       found <= '0';
24     when B =>
25       if entr = '0' then
26         next_st <= A;
27       else
28         next_st <= C;
29       end if;
30       found <= '0';
31     when C =>
32       if entr = '0' then
33         next_st <= A;
34       else
35         next_st <= C;
36       end if;
37       found <= '1';
38   end case;
39 end process U_st_trans;
```



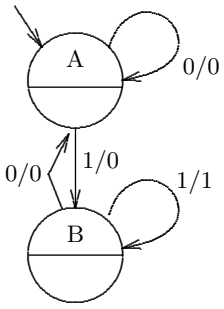
Exemplo 5.11 Máquina de Mealy O Programa 16 mostra a implementação para uma *Máquina de Mealy* que detecta sequências de dois ou mais **1**s em sua entrada.

Os modelos Moore e Mealy são semelhantes, exceto que na Máquina de Mealy são necessários apenas dois estados e a saída depende tanto do estado atual quanto da entrada: o sinal found é atualizado nas quatro possíveis combinações de estado e entrada.

Verifique se o código do processo U_st_trans corresponde à função de próximo estado do diagrama de estados, e se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◁

Programa 16: Máquina de Mealy para 11^+.

```
1  type states is (A, B);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7    if reset = '0' then
8       curr_st <= A;
9    elsif rising_edge(clk) then
10      curr_st <= next_st;
11   end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin              -- Máquina de Mealy
16   case curr_st is
17     when A =>
18       if entr = '0' then
19         next_st <= A;
20         found   <= '0';
21       else         -- entr='1'
22         next_st <= B;
23         found   <= '0';
24       end if;
25     when B =>
26       if entr = '0' then
27         next_st <= A;
28         found   <= '0';
29       else         -- entr='1'
30         next_st <= B;
31         found   <= '1';
32       end if;
33   end case;
34 end process U_st_trans;
```



5.8 Funções

VHDL nos permite definir *funções* que computam valores, e que podem ser usadas no lado direito de expressões. Funções têm argumentos de tipo entrada (**in**) e um valor de retorno, e podem ser usadas, por exemplo, na conversão de tipos, tais como a conversão de inteiros para vetores de bits, e a conversão de vetores de bits para cadeias de caracteres.

Exemplo 5.12 Função para detecção de borda A função rising_edge(), no Programa 17, pode ser usada para testar a ocorrência de uma borda de subida num sinal. Seu argumento é um sinal do tipo bit e seu valor é um Booleano. Os registradores de estado nas MEs da Seção 5.7 empregam esta função para detectar bordas no sinal de relógio.

Se esta função estiver declarada na entidade de um *flip-flop*, ela pode ser empregada em todas as arquiteturas para aquela entidade. Se a função estiver declarada numa biblioteca, então ela pode ser empregada em todas as entidades que façam uso da biblioteca. Veja `packageWires.vhd` para outros exemplos de função. ◁

Programa 17: Função que detecta bordas ascendentes.

```
function rising_edge(signal S: bit)
  return boolean is
begin
  if (S'event) and          -- ocorreu evento em S
     (S = '1') and          --   e valor atual é '1'
     (S'last_value = '0')   --   e valor anterior era '0'
  then
    return TRUE;
  else
    return FALSE;
  end if;
end rising_edge;
```

Exemplo 5.13 Conversões entre Boolean e Bit O Programa 18 mostra duas funções de conversão de tipos: (i) BOOL2BIT() converte um valor do tipo boolean para o tipo bit; (ii) BIT2BOOL() faz a conversão na direção oposta. Lembre que VHDL é uma linguagem fortemente tipada e este tipo de conversão, aparentemente inútil, é um preço módico a se pagar, comparando-se aos benefícios advindos dos tipos fortes. ◁

Programa 18: Funções de conversão entre boolean e bit.

```
function BOOL2BIT(b: in boolean) return bit is
  variable s : bit;
begin
  case b is
    when TRUE   => s := '1';
    when others => s := '0';
  end case;
  return s;
end;

function BIT2BOOL(s: in bit) return boolean is
  variable b : boolean;
begin
  case s is
    when '1'    => b := TRUE;
    when others => b := FALSE;
  end case;
  return b;
end;
```

5.9 Da tarefa:

Etapa 1 Projete uma Máquina de Moore, com uma entrada de dados D, uma saída C, uma entrada de relógio *rel*, e uma entrada de *reset*. Após a inicialização, sempre que o circuito detectar a sequência $\cdots$01111110$\cdots$ (0, seis 1, 0) a saída C produz um pulso com duração de um ciclo.

```
D:  00111011111100111111110001010
C:  00000000000001000000100000
```

A sequência $\cdots$011111101111110$\cdots$ produz um único pulso na saída C.

Etapa 2 Copie para sua área de trabalho o arquivo com o código VHDL:
(a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_processos.tgz`
(b) expanda-o com: `tar xzf l_processos.tgz`
o diretório `processos` será criado;
(c) mude para aquele diretório: `cd processos`

Ao contrário dos outros laboratórios, neste você deve escrever seu código VHDL para a máquina de estados diretamente no arquivo `tb_me.vhd`. Você deve consertar e completar o esqueleto que está no TB, acrescentando mais estados ao tipo states e cláusulas ao **case** do processo U_st_transitions. Uma vez completado seu modelo, e gerados os casos de teste que **comprovem** sua corretude, envie o código do `tb_me.vhd` para o seu professor, com assunto `ci210-ME`, e os nomes dos dois autores, até as 12:59 no dia seguinte ao deste laboratório.

seu código deve ser escrito no TB

Referências

[RH12] *Sistemas Digitais e Microprocessadores*, R A Hexsel, 2012, Editora da UFPR.

[PJA08] *The Designer's Guide to VHDL*, Peter J Ashenden, 3a Ed, 2008, Morgan Kaufmann.

EOF

Histórico das Revisões:
02set2019: revisão do texto;
06set2018: revisão do texto;
19set2017: revisão do texto, formatação de exemplos de código;
07set2017: revisão do texto, acréscimo de exemplos;
16set2016: revisão do texto por Marco Zanata;
12set2016: acréscimo de exemplos, melhorias no esqueleto do código VHDL;
01out2015: revisão do texto;
19set2015: primeira versão.