2 Temporização de Circuitos Combinacionais

Neste semestre usaremos VHDL para fazer a modelagem e a simulação do comportamento temporal de circuitos digitais.

Objetivos: são dois os objetivos deste laboratório: (i) verificar o comportamento temporal do multiplexador; e (ii) verificar a corretude dos modelos do multiplexador, do demultiplexador e do decodificador através de **assert**s e com diagramas de tempo.

O trabalho pode ser efetuado em duplas. Na Seção 2.7 estão as questões que devem ser respondidas nesta aula e entregues ao professor.

Preparação: leia a seção 1.3 de [RH12], sobre circuitos combinacionais, mais a Seção 4.4 das notas sobre CMOS.

2.1 Modelo funcional *versus* modelo temporal

Na aula passada empregamos *modelos funcionais* para portas lógicas, e com aqueles construímos modelos para circuitos combinacionais compostos de várias portas lógicas. Aqueles modelos são chamados de *funcionais* porque representam somente o aspecto "função lógica" dos circuitos, e ignoram informações de tempo.

Nesta aula enriqueceremos os modelos funcionais com informação de tempo, criando modelos mais sofisticados para o comportamento daqueles circuitos.

Modelos temporais incorporam a informação de tempo de propagação dos componentes e permitem simulações mais detalhadas e realistas dos circuitos. Evidentemente, simulações mais detalhadas são mais custosas em termos de tempo e de memória.

Os modelos que usaremos nesta aula permitem a visualização do comportamento dos circuitos combinacionais ao longo do tempo, e a medição do tempo de propagação através de circuitos relativamente simples. Em breve estudaremos a propagação de sinais em somadores e nestes circuitos os problemas são um tanto mais severos do que o que veremos hoje. Cada coisa a seu tempo.

2.2 O Que É Um Circuito Combinacional?

Com a abstração para os sinais elétricos definida em termos dos *bits*, ou *sinais digitais* que podem assumir um dentre os valores discretos em $\mathbb{B}$, é possível definir o que se entende por um *dispositivo combinacional*, que é um dispositivo com as seguintes propriedades:

1. uma ou mais *entradas* digitais;
2. uma ou mais *saídas* digitais;
3. uma *especificação funcional* que determina o valor lógico de cada saída para cada uma das combinações das entradas; e

4. uma *especificação temporal* que determina, pelo menos, um limite superior para o tempo de propagação dos sinais que atravessam o dispositivo.

Uma *especificação funcional* é um contrato entre o projetista de um dispositivo e o usuário deste dispositivo, e garante que *entradas válidas produzem saídas válidas*.

A especificação funcional do dispositivo pode ser um conjunto de somas de produtos que relacionam todas as combinações de valores nas entradas com valores bem definidos nas saídas, ou ainda uma tabela verdade para cada saída que relaciona entradas com os valores de saída.

O comportamento temporal dos dispositivos pode ser mensurado nos terminais do dispositivo, ou computado através de simulações. O que nos interessa agora é o *tempo de propagação* dos sinais através do dispositivo, ou o *atraso* (*delay*) na propagação dos sinais introduzido pelo dispositivo. O tempo de propagação é o tempo necessário para que o circuito produza uma saída válida estável, a partir do instante em que suas entradas estabilizaram. Se necessário, reveja a Seção 4.4 das notas sobre CMOS.

Um *circuito combinacional* pode ser construído pela combinação de dispositivos combinacionais interligados e que atendem às seguintes restrições:

5. cada componente é um dispositivo combinacional;
6. cada uma das entradas está ligada a exatamente uma saída ou às constantes 0 ou 1;
7. o circuito não contém circularidade nas ligações ($\lambda \rightsquigarrow \sigma \rightsquigarrow \pi \not\rightsquigarrow \lambda$).

A restrição 6 proíbe a ligação de duas saídas – o que causa curto-circuitos – e mais ainda, proíbe entradas abertas. A restrição 7 impede laços que liguem as saídas do componente Λ à entradas do componente Π, que direta ou indiretamente alimentam as entradas de Λ.

Considere o dispositivo combinacional mostrado na Figura 1, que consiste de um circuito com três entradas (p, q, r) e dois componentes, cada um com sua especificação funcional e temporal. Para nos certificarmos de que o dispositivo na figura é combinacional basta observar que o sinal interno y torna-se válido e bem definido 2ns depois que as entradas p e q estabilizam. A saída s torna-se válida e bem definida $2 + 1$ns depois que as entradas p, q e r estabilizam. O sinal y está ligado a exatamente uma única saída, e não há circularidade porque não existe ligação entre as saídas y e s com as entradas p, q ou r.

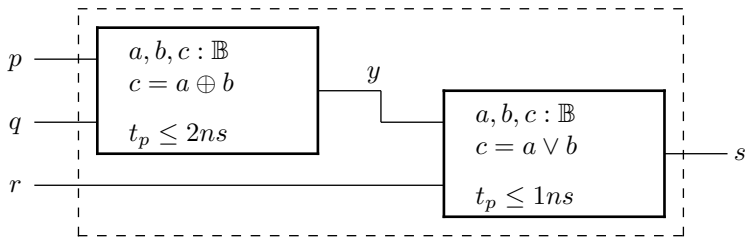


Figura 1: Dispositivo combinacional.

O valor da saída s, seja ele 0 ou 1, pode ser facilmente deduzido pela composição das funções dos dois componentes. Este dispositivo tem um tempo de propagação de 3ns, e sua saída é uma função dos sinais de entrada, sendo portanto um dispositivo combinacional. Esta forma de verificação pode ser aplicada a circuitos com um número arbitrário de componentes.

Como é computado o tempo de propagação através de um circuito combinacional? Para que se possa usar um determinado dispositivo com a confiança de que seu tempo de propagação é um limite superior para o intervalo entre o instante em que as entradas ficam estáveis e o instante no qual a saída fica estável, devemos computar o *tempo de propagação de pior caso*. Para cada caminho entre as entradas e a saída computa-se o atraso cumulativo naquele caminho, que é a soma dos atrasos impostos pelos componentes no caminho. O tempo de propagação do dispositivo é o mais longo dentre os tempos de propagação de todos os caminhos.

Ignorando sua função, qual é o tempo de propagação do circuito na Figura 2?

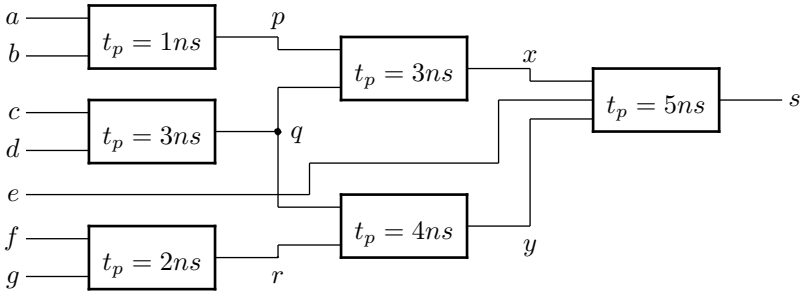


Figura 2: Circuito combinacional com atrasos.

1. O caminho mais curto é o da entrada e para a saída, que é de 5ns;
2. o caminho $\{a, b\} \rightsquigarrow p \rightsquigarrow x \rightsquigarrow s$ tem custo de $1 + 3 + 5 = 9$ns;
3. o caminho $\{c, d\} \rightsquigarrow q \rightsquigarrow x \rightsquigarrow s$ tem custo de $3 + 3 + 5 = 11$ns;
4. o caminho $\{c, d\} \rightsquigarrow q \rightsquigarrow y \rightsquigarrow s$ tem custo de $3 + 4 + 5 = 12$ns; e
5. o caminho $\{f, g\} \rightsquigarrow r \rightsquigarrow y \rightsquigarrow s$ tem custo de $2 + 4 + 5 = 11$ns.

O tempo de propagação do circuito é o do caminho mais longo, que é 12ns.

Espaço em branco proposital.

2.3 Modelo para temporização

As arquiteturas das portas lógicas que usaremos empregam atribuições temporizadas para modelar o comportamento de portas lógicas reais. Por exemplo, na arquitetura do inversor no arquivo `aux.vhd`, o comportamento (temporal) é definido por

```
S <= reject t_cont inertial (not A) after t_inv;
```

O complemento de A será atribuído a S após o tempo de propagação t_inv. Pulsos na entrada do inversor com duração *menor ou igual* ao tempo de contaminação t_cont são ignorados – ou rejeitados – pelo simulador, mantendo a saída do inversor imutável durante t_cont.

As constantes t_inv e t_cont estão definidas no topo arquivo `packageWires.vhd`. O modelo para o sinal S é tal que pulsos em A com duração menor que t_cont não são transferidos para a saída. Este comportamento é similar ao de portas lógicas: um pulso na entrada é refletido para a saída caso este pulso dure tempo o suficiente para modificar o estado de todos os nós internos ao circuito. Posto de outra forma, os sinais tem 'inércia' e o qualificador **inertial** modela estes "atrasos inerciais".

atraso inercial

A Figura 3 mostra um diagrama de tempo para o modelo de temporização para atrasos inerciais com rejeição de pulsos estreitos. As entradas do modelo inicialmente estão na configuração α e sua saída é $f(\alpha)$. Ocorre uma mudança de α para β, possivelmente por causa de alguma corrida de sinais nos circuitos que alimentam o modelo, e esta alteração nas entradas não é refletida para a saída porque sua duração é menor do que o tempo de contaminação T_C do modelo. A nova configuração de entradas γ persiste por mais do que T_C, e decorrido o tempo de propagação T_P, a saída torna-se $f(\gamma)$.

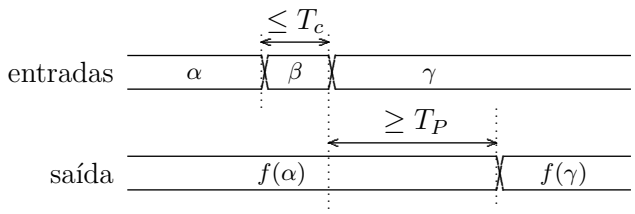


Figura 3: Modelagem da temporização em VHDL.

O modelo de temporização de VHDL é similar ao modelo definido em aula. O modelo para o tempo de propagação tem a mesma definição: uma vez que as entradas estejam válidas e estáveis, as saídas se tornam válidas e estáveis após T_P. A rejeição de pulsos estreitos tem uma interpretação similar àquela do tempo de contaminação, no sentido de que a saída não é alterada a menos que a alteração nas entradas tenha duração suficiente para provocar uma alteração na saída que perdure até que o nível na saída estabilize.

2.4 Material disponibilizado para sua tarefa

Etapa 1 Copie para sua área de trabalho o arquivo com o código VHDL:
(a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_combinacionais.tgz`
(b) expanda-o com: `tar xzf l_combinacionais.tgz`
o diretório `combinacionais` será criado;
(c) mude para aquele diretório: `cd combinacionais`

O arquivo `packageWires.vhd` contém definições dos tempos de propagação das portas lógicas e abreviaturas para nomes de sinais, uma vez que digitar `reg8` é mais econômico do que `bit_vector(7 downto 0)`. Este arquivo também contém as definições de temporização das portas lógicas. abreviaturas

O arquivo `aux.vhd` contém os modelos das portas lógicas *not*, *nand*, *nor* e *xor*, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `combin.vhd` contém um modelo para um multiplexador de duas entradas, *mux-2*. Este modelo serve de base para os modelos dos componentes *mux-4*, *mux-8*, *demux-2*, *demux-4*, *demux-8*, *decod-2*, *decod-4* e *decod-8*, da Seção 1.3 de [RH12]. Ao invés de reescrever estes modelos, utilize as arquiteturas dos modelos da aula passada – note que as declarações das portas lógicas devem incluir a especificação do tempo de propagação e de contaminação, conforme mostra o Programa 2, na pág. 6.

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 qualquer argumento o *script* dispara a execução de `gtkwave`: `./run.sh 1 &`

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

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

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, mova o arquivo `gtkwaverc` para o seu `$HOME`, como um arquivo escondido (`$HOME/.gtkwaverc`) e edite as duas últimas linhas – os números podem ser aumentados para melhorar a legibilidade.

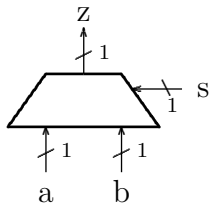
Espaço em branco proposital.

2.5 Modelo do multiplexador

A *entidade* e a arquitetura do multiplexador que estudamos na aula passada são mostradas abaixo. A diferença com relação à aula anterior é que a informação de tempo de propagação foi acrescida aos modelos das portas lógicas. A entidade do *mux-2* não foi alterada; na nova versão, os componentes na arquitetura são declarados e instanciados com a informação de tempo.

Programa 1: Entidade *mux-2*.

```
use work.p_wires.all;
entity mux2 is
  port(a,b : in  bit;
       s   : in  bit;
       z   : out bit);
end mux2;
```



Os tempos de propagação das portas lógicas (t_inv, t_nand2, t_nor2 e t_cont) são constantes declaradas no arquivo packageWires.vhd. O **generic map** permite redefinir uma constante – que foi originalmente definida na declaração da arquitetura – quando uma versão é instanciada. Na arquitetura do Programa 2, os tempos de propagação e de contaminação são 'redefinidos' para os (mesmos) valores das constantes declaradas em packageWires.vhd.

generic map

Programa 2: Arquitetura do *mux-2* **com temporização.**

```
architecture estrutural of mux2 is

  -- declaração dos componentes
  component inv is
    generic (prop : time; cont : time); -- t_prop, t_cont
    port(A : in bit; S : out bit);
  end component inv;

  component nand2 is
    generic (prop : time; cont : time); -- t_prop, t_cont
    port(A,B : in bit; S : out bit);
  end component nand2;

  component nor2 is
    generic (prop : time; cont : time); -- t_prop, t_cont
    port(A,B : in bit; S : out bit);
  end component nor2;

   -- declaração dos sinais internos ao mux2
  signal r, p, q : bit;

begin
  -- instanciação dos componentes
  Ui:  inv   generic map (t_inv,   t_cont) port map (s, r);
  Ua0: nand2 generic map (t_nand2, t_cont) port map (a, r, p);
  Ua1: nand2 generic map (t_nand2, t_cont) port map (b, s, q);
  Uor: nand2 generic map (t_nand2, t_cont) port map (p, q, z);
end architecture estrutural;
```

Esta forma de modelagem emprega a técnica de *information hiding*: toda a informação sobre tempo está escondida dentro da arquitetura do *mux-2*, e sua entidade não contém nenhuma informação sobre a temporização. Quando usarmos o *mux-2*

information hiding

para implementar um *mux-4*, nenhuma informação sobre tempo de propagação do *mux-2* é transferia explicitamente para o *mux-4* porque tal informação está escondida na arquitetura do *mux-2*, e fica portanto invisível ao usuário do modelo do mux4. O usuário dos modelos baseados no *mux-2* não mais se ocupa com definir a temporização dos circuitos.

2.6 *Testbench*

O arquivo `tb_combin.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos seus modelos. A entidade tb_combin é vazia porque o programa de testes é autocontido e não tem interfaces com nenhum outro circuito.

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

Registro (**record**) test_record_2

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

Vetor (**array**) test_array_2

Copie seus vetores de teste da aula passada, e se for o caso, acrescente novos.

No test_record_2, os campos k, s e mx são de tipo bit (`'0'`), os campos a, dm e dc são vetores de bits codificados em binário (`b"10"`).

escalar: `'0'` vetor: `b"01"`

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

O campo dm é o vetor de bits com a saída esperada para um demultiplexador quando recebe as entradas definidas pelos valores em k,s.

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

Um único registro é usado para testar todos os circuitos ao mesmo tempo e portanto, dependendo do teste, alguns dos campos não são relevantes *naquele teste*.

Programa 3: Vetor de valores de entrada para testar modelos.

```
-- definicao do vetor de testes para MUX-2, DEMUX-2, DECOD-2
type test_record_2 is record
  k  : bit;       -- entrada de bit para demultiplexadores
  a  : reg2;      -- entrada para multiplexadores
  s  : bit;       -- entrada de seleção (para todos circuitos)
  mx : bit;       -- saída esperada do MUX
  dm : reg2;      -- saída esperada do DEMUX
  dc : reg2;      -- saída esperada do DECOD
end record;
type test_array_2 is array(positive range <>) of test_record_2;

-- vetor de testes
constant test_vectors_2 : test_array_2 := (
  --k,   a,   s,  mx,  dm,   dc
  ('0',b"00",'0','0',b"00",b"01"),
  ('0',b"00",'1','0',b"00",b"10"),
  ('1',b"01",'0','1',b"01",b"01"),
  ('1',b"10",'1','1',b"10",b"10"),
```

```
    -- não alterar estes três últimos --
    ('0',b"00",'0','0',b"00",b"01"),
    ('0',b"00",'0','0',b"00",b"01"),
    ('0',b"00",'0','0',b"00",b"01")
    );

  -- troque a constante para FALSE para testar seus modelos
  constant TST_MUX_2    : boolean := true;
  constant TST_DEMUX_2  : boolean := true;
  constant TST_DECOD_2  : boolean := true;
```

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

atributo '**range**

O **assert** no Programa 4 verifica se a saída observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto com relação aos vetores de teste que você escreveu.

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

Programa 4: Mensagem de verificação de comportamento.

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

Se os valores de saidaMUX2 e esperadaMUX diferem, a mensagem no Programa 4 é emitida no terminal, indicando o erro. A função B2STR converte um bit em uma *string* para que o valor do bit seja emitido; a função BV2STR converte um vetor de bits para uma *string*. O operador '&' concatena duas *strings*.

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

A condição de teste do **assert** é

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

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

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

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

Para testar seu modelo, altere a constante respectiva (TST_MUX_2, TST_DEMUX_2 ou TST_DECOD_2) para false e então verifique os resultados. Há uma tripla de constantes para cada largura de circuito, *viz* TST_MUX_2, TST_MUX_4 e TST_MUX_8.

2.7 Nem mesmo um circuito simples é bem-comportado?

Etapa 2 Verifique, cuidadosamente, as combinações de entradas e a saída do modelo mux2. A entradas são entr_2 (v_2.a(0) e v_2.a(1)) e o sinal de controle é s0 (v_2.s). Note que os **assert**s correspondentes ao mux2 não são impressos – o que indica que o comportamento é o esperado.

Contudo, no diagrama de tempos do `gtkwave`, os sinais s0, entr(1 **downto** 0) e saidamux2 indicam que há algo de podre no reino dos multiplexadores. Para gerar o diagrama de tempos para o `gtkwave` execute `./run.sh 1 &`. Observe os instantes nos quais as entradas se alteram. Uma vez identificado o problema, qual a sua solução?

Há um pulso transitório na saída do *mux-2* logo no início do primeiro intervalo de simulação, indicado pelo retângulo de cor ciano na Figura 4. Meça a duração deste pulso com os cursores do `gtkwave`, e procure em `packageWires.vhd` quais são os tempos de propagação similares à duração deste pulso, e similares ao intervalo entre o início da simulação e a borda de subida deste pulso. Por que ele ocorre?

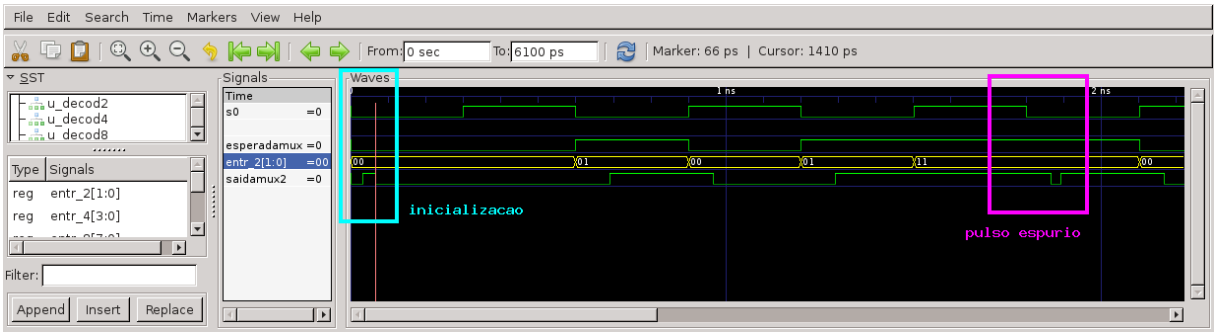


Figura 4: Pulsos transitórios na saída do multiplexador.

É por causa destes pulsos transitórios – que ocorrem assim que o circuito é inicializado – que empregamos sinais de *reset* nos circuitos. A duração do *reset* deve ser mais demorada do que o transitório provocado pelo caminho mais longo no circuito. O projetista deve usar um intervalo que garanta que, depois que *reset* fica inativo, todos os sinais estejam estáveis e definidos.

O segundo pulso transitório na saída do *mux-2*, indicado pelo retângulo magenta na Figura 4, tem natureza distinta daquele provocado pela inicialização do circuito. Meça a duração deste pulso com os cursores do `gtkwave`, e procure em `packageWires.vhd` os tempos de propagação similares à duração do pulso. Por que ele ocorre?

Etapa 3 Aplique sua solução para o problema hamletiano do *mux-2* aos modelos do *demux-2* e *decod-2*, caso necessário.

> *Você é quem escreve os vetores de teste e portanto sua tarefa é ajustar os campos mx (saída esperada dos muxN), dm (saída esperada dos demuxN), e dc (saída esperada dos decodN). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **assert**s indicarão falso-positivos para erros inexistentes.*

Os **assert**s que verificam a corretude dos demultiplexadores e decodificadores estão comentados para reduzir a poluição na tela. Após completar a verificação do *mux-2*, altere a constante do *demux-2* para false e verifique seu modelo. Repita para *decod-2*.

Responda às questões na última página e as entregue ao professor até as 13:30 de amanhã – deixe a folha de respostas no escaninho de seu professor, na recepção do Departamento.

2.8 Mais sobre VHDL

VHDL provê várias construções interessantes para 'abstração'. Dentre elas veremos (i) pode-se definir mais de uma arquitetura para uma entidade, e (ii) comandos concorrentes de seleção.

2.8.1 Uma entidade, muitas arquiteturas

Lembre da espiral de projeto: a cada avanço na espiral, a realização da especificação se torna mais concreta e mais próxima da implementação final.

Uma *entidade* define a interface de um componente com o mundo exterior àquele componente. Uma *arquitetura* é refinamento, ou uma implementação, para a especificação daquele componente.

Esta forma de trabalhar é capturada de uma maneira simples por VHDL: uma entidade pode ser implementada por mais de uma arquitetura. A primeira arquitetura é a mais abstrata e pode ser uma especificação executável; a próxima é mais detalhada e contém mais detalhes da implementação; a última é um modelo concreto que pode ser sintetizado como um circuito.

Do ponto de vista do código VHDL, caso haja mais do que uma arquitetura, geralmente[1] o compilador emprega aquelsa última 'vista', que é a que está "mais longe" da definição da entidade no código fonte, e ignora a(s) arquitetura(s) definida(s) 'antes'. O Programa 5 mostra um exemplo.

Programa 5: Duas arquiteturas para uma entidade.

```
entity meuProjeto is
  port( ... )
end meuProjeto;

architecture especificacao of meuProjeto is
begin
   ...
end architecture especificacao;

architecture concreta of meuProjeto is
begin
   ...
end architecture concreta;
```

A entidade meuProjeto define a interface do componente. A arquitetura concreta é a versão detalhada e que pode ser sintetizada diretamente pelo compilador VHDL. A arquitetura especificacao define a função e o comportamento desejado de meuProjeto. Na medida em que o projeto é refinado, cada arquitetura mais concreta é inserida no código abaixo da menos concreta, para que esta última seja usada pelo compilador. Dessa forma, é fácil comparar as diferentes versões de um mesmo modelo.

[1] Seu compilador pode fazer de outra forma.

2.8.2 Comandos concorrentes de seleção

Vejamos o que VHDL nos oferece de mais abstrato do que projetos com portas lógicas. Como uma linguagem assaz versátil, VHDL nos permite trabalhar com baixo nível de abstração – interligação de portas lógicas – e também com código um tanto mais abstrato. Vejamos as duas maneiras de modelar seleção.

A *área concorrente* de uma arquitetura é a região de código entre o **begin** e o **end architecture**. O trecho de código no Programa 6 mostra a arquitetura chamada exemplo da entidade área_concorrente. Na área concorrente estão dois comandos, ou dois processos, que são executados concorrentemente: a disjunção e a conjunção de dois sinais. Num circuito, esta arquitetura seria implementada com duas portas lógicas que constantemente amostram suas entradas e produzem suas saídas em função daquelas. O modelo VHDL deve ter o mesmo comportamento e portanto os dois processos executam concorrentemente: qualquer alteração em a e/ou em b provocará a avaliação do lado direito das atribuições, e se for o caso, eventos nos sinais r e s.

Programa 6: Área concorrente num arquitetura.

```
architecture exemplo of area_concorrente is
begin
  -- inicio da área concorrente

  U_or:  r <= a or  b;

  U_and: s <= a and b;

  -- final da área concorrente
end architecture exemplo;
```

A linguagem nos oferece dois *comandos concorrentes* de seleção que podem ser usados na área concorrente: o comando **when–else** e o comando **with–select**.

A forma mais simples do comando de atribuição condicional **when–else** é **when–else**

```
a when cond else b;
```

Se a condição cond é verdadeira, então o comando tem o valor de a, do contrário o comando tem o valor de b. Quando usado numa atribuição, este comando se comporta como um comando de seleção de C: `s = (cond ? a : b);`

```
s <= a when cond else b;
```

O comando pode ser encadeado com várias cláusulas **else**:

```
s <= a when cond_1 else
     b when cond_2 else
     c when cond_3 else
     d;
```

As condições deste comando são avaliadas na ordem do texto: primeiro cond_1, se esta é falsa então cond_2 é avaliada, e se esta é falsa então cond_3 é avaliada. O comando acima pode ser usado para implementar um *mux-4*:

```
z <= a when (s = "00") else
     b when (s = "01") else
     c when (s = "10") else
     d;
```

Um modelo completo é mostrado no Programa 7, para entradas e saídas que são vetores de 8 bits – o multiplexador tem 4 entradas de 8 bits cada. Este modelo é um tanto mais abstrato do que um modelo estrutural com a mesma função.

Programa 7: *mux-4x8* **implementado com when−else.**

```
entity mux4x8 is
  port(a,b,c,d: in  bit_vector(7 downto 0);
       s:       in  bit_vector(1 downto 0);
       z:       out bit_vector(7 downto 0));
end mux4x8;

architecture when_else of mux4x8 is
begin
   z <= a when (s = "00") else   -- atribuição condicional
        b when (s = "01") else
        c when (s = "10") else
        d;  -- todos os demais casos
end architecture when_else;
```

O comando concorrente de atribuição selecionada **with−select** é mais parecido com um `switch` do C do que com uma cadeia de `if`s. Para um sinal de seleção `s` que pode assumir um dentre dois valores (`v1`,`v2`), o comando **with−select** atribui à sua 'saída' o sinal correspondente ao valor de seleção. Vejamos um exemplo. **with−select**

```
  with s select
    r <= a when v1,
         b when v2;
```

Uma cláusula **others** pode ser usada para capturar todos os demais casos:

```
  with s select
    r <= a when v1,
         b when others;  -- todos os demais casos
```

Vejamos como fica o nosso *mux-4*, com um comando de atribuição selecionada.

```
  with s select
    z <= a when "00",
         b when "01",
         c when "10",
         d when others;  -- todos os demais casos
```

Ao contrário do **when−else**, todas as cláusulas do **with−select** tem a mesma precedência e geralmente este comando resulta em circuitos menores e mais rápidos. O Programa 8 mostra a arquitetura do *mux-4x8* com **with−select**.

Programa 8: *mux-4x8* **implementado com with−select.**

```
architecture with_select of mux4x8 is
begin
  with s select           -- atribuição selecionada
    r <= a when "00",
         b when "01",
         c when "10",
         d when others;  -- todos os demais casos
end architecture with_select;
```

Espaço em branco proposital.

Referências

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

[PT96] *VHDL Made Easy!*, D Pellerin and D Taylor, 1996, Prentice Hall.

[PJA90] *The VHDL Cookbook*, `http://freecomputerbooks.com/The-VHDL-Cookbook.html`

EOF

Histórico das Revisões:
02set19: ajustes cosméticos no texto;
16ago18: ajustes cosméticos no texto, (true or (...)) nos asserts;
10out16: **with-select**
03set16: vetores de testes separados por tamanho de circuito;
15ago16: ajustes nos exercícios;
22sep15: modelo de temporização com T_c;
13ago13: inserção de lstinline, revisão no texto;
01nov12: primeira versão.

Nome:

Nome:

As questões abaixo devem ser respondidas e entregues ao professor. Os problemas podem ser respondidos em dupla; os dois nomes devem estar na folha de respostas.

Etapa 4 Leia o código VHDL dos modelos e estime os tempos de propagação dos nove circuitos, tomando como base os tempos de propagação das portas lógicas em `packageWires.vhd`.

Etapa 5 Repita para o tempo de contaminação dos nove circuitos.

Etapa 6 Compile os modelos e verifique seu funcionamento com `gtkwave`. Verifique as diferenças nos tempos de propagação das versões de 2, 4 e 8 entradas dos multiplexadores, demultiplexadores e decodificadores. Use os cursores do `gtkwave` para medir os tempos.

Etapa 7 Confirme, com base nos diagramas de tempo, se os tempos medidos são similares aos que você estimou. Justifique as diferenças nas suas estimativas.

Veja a Seção 2.8.2 deste documento para responder às próximas perguntas. Copie o 'miolo' dos seus modelos na folha de respostas.

Etapa 8 Implemente um *mux-8* com o comando **when**−**else** e verifique seu modelo. Acrescente o novo modelo ao arquivo `combin.vhd` *após* a definição da arquitetura do *mux-8* – VHDL usa a última arquitetura que encontra após a definição da entidade.

Etapa 9 Implemente um *mux-8* com o comando **with**−**select** e verifique seu modelo. Acrescente o novo modelo ao arquivo `combin.vhd` *após* a definição da última arquitetura do *mux-8*.