13.2 Temporização de Circuitos Combinacionais

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 13.2.10 estão as questões que devem ser respondidas nesta aula e entregues ao professor.

13.2.1 O que é um circuito combinacional?

Reveja a Seção 4.1.

13.2.2 Modelo do comportamento temporal de CCs

Reveja a Seção 5.4.

13.2.3 Modelo funcional *versus* modelo temporal

No laboratório sobre descrição estrutural empregamos *modelos funcionais* para portas lógicas, e com aqueles construímos modelos para circuitos combinacionais pela composição de várias portas lógicas. Aqueles modelos são chamados de *funcionais* porque representam somente o aspecto "função lógica" dos circuitos, e não contêm informações de tempo.

Neste laboratório enriqueceremos os modelos funcionais com informação de tempo, criando modelos mais sofisticados e realistas para o comportamento daqueles circuitos.

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

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.

13.2.4 Modelo para temporização em VHDL

Reveja a Seção 5.3.2. Reveja a Seção 9.3 das notas de aula (`vhdl.pdf`).

Espaço em branco proposital.

13.2.5 Material disponibilizado para sua tarefa

Etapa 5 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. Este arquivo também contém as definições de temporização das portas lógicas.

O arquivo `aux.vhd` contém os modelos das portas lógicas *not*, *and*, *or* 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, definidos na Seção 4.3, *Circuitos Combinacionais Básicos*, em `combin.pdf`.

13.2.6 Modelos de multiplexador

Reveja a Seção 5.3.2. Veja a Seção 9.3.4 de `vhdl.pdf` para a descrição do modelo temporizado do *mux*-2.

Ao invés de reescrever os modelos dos *mux*-N, utilize as arquiteturas dos modelos do laboratório anterior sem esquecer de que as declarações das portas lógicas devem incluir a especificação do tempo de propagação e de contaminação.

> Para simular com tempos de propagação diferentes de zero, edite o arquivo `packageWires.vhd` e troque as constantes simulate_time e simulate_rej para 1. As constantes estão no topo do arquivo. Se as constantes são zero, então a simulação é puramente funcional.

13.2.7 *Testbench* para os multiplexadores

Reveja a Seção–5.3.2. Examinaremos primeiro os quatro modelos do multiplexador.

O *script* `run_mux.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run_mux.sh` (re)compila o simulador e executa a simulação com **assert**s. Se invocado com qualquer argumento o *script* também dispara a execução de `gtkwave`: `./run_mux.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.
>
> 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 definições – os números podem ser alterados para melhorar a legibilidade.
O `gtkwave` deve ser reinicializado para que as definições tenham efeito.

O arquivo `v_mux.sav` contém definições para o `gtkwave` tais como a escala de tempo e sinais a serem exibidos na tela para a verificação dos modelos *mux*-2, *mux*-4 e *mux*-8.

O arquivo `tb_mux.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos seus modelos. A entidade tb_mux 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.

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 três 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 os oito elementos necessários para excitar e verificar o *mux*-2, na primeira tarefa deste laboratório.

No test_record_2, os campos s e mx são de tipo bit (`'0'`) e o campo a é um vetor de bits codificado em binário (`b"10"` – o prefixo `b` indica *binário*).

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.

Programa 13.51: Vetor de valores de entrada para testar modelos.

```
-- definição do vetor de testes para MUX-2
type test_record_2 is record
  a  : reg2;       -- entrada para multiplexadores
  s  : bit;        -- entrada de seleção
  mx : bit;        -- saída esperada do MUX
end record;
type test_array_2 is array(positive range <>) of test_record_2;

-- vetor de testes
constant test_vectors_2 : test_array_2 := (
  --s,   a,   mx
  ('0',b"00",'0'),  -- transcrição da tabela verdade do mux-2
  ('0',b"01",'1'),  -- s e a são entradas, mx é a saída
  ('0',b"10",'0'),
  ('0',b"11",'1'),
  ('1',b"00",'0'),
  ('1',b"01",'0'),
  ('1',b"10",'1'),
  ('1',b"11",'1'),
  --
  ('0',b"11",'1'),  -- não alterar estes três últimos
  ('0',b"11",'1'),
  ('0',b"11",'1')
  );

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

A sequência de valores de entrada para os testes dos modelos é gerada pelo processo U_testValues, com o 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 i-ésimo elemento do vetor é atribuído à variável v e 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.

O **assert** no Programa 13.52 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 13.52: 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 13.52 é emitida no terminal, indicando o erro.

```
mux2: saída errada   sel=1   saiu=0   esperada=1
```

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

```
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. O código foi escrito assim para diminuir a poluição na tela durante os testes dos modelos.

Para testar seu modelo, altere a constante respectiva (TST_MUX_2, TST_MUX_4 ou TST_MUX_8) para false e então verifique os resultados.

Etapa 6 Para executar a simulação sem invocar `gtkwave`, diga
`prompt: ./run_mux.sh`
Verifique se há alguma mensagem de erro; se sim, leia a mensagem e use seus miolos antes de chamar o professor.

Para executar a simulação com o `gtkwave`, diga
`prompt: ./run_mux.sh 1 &`

Achtung: se o diagrama de tempo é idêntico ao da simulação funcional, sem mostrar atrasos de propagação, então edite `packageWires.vhd` e altere a constante simulate_time para 1. Esta constante está no topo do arquivo.

13.2.8 Nem mesmo um circuito simples é bem-comportado?

Etapa 7 Reveja a Seção–5.3.2. 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). Se os **assert**s correspondentes ao mux2 não são impressos, então o comportamento é o esperado. *Esperado* não é o mesmo que *correto*.

Contudo, no diagrama de tempos do `gtkwave`, os sinais s0, entr_2(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_mux.sh 1 &`
Observe os instantes nos quais as entradas se alteram. Uma vez identificado o problema, qual seria a solução? Reveja a Seção–5.3.2.

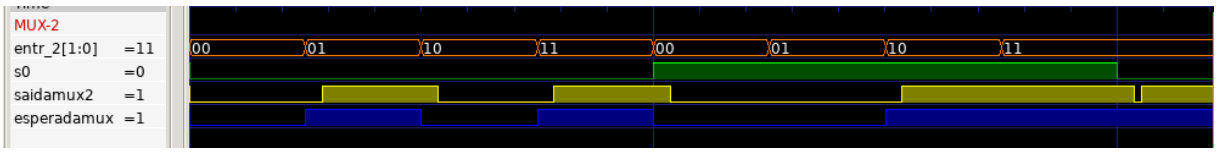

Figura 13.16: Pulsos transitórios na saída do multiplexador.

O pulso transitório – *indesejado porque viola a especificação funcional do mux-2* – ocorre no lado direito da Figura 13.16. 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 8 Conserte seu modelo do *mux*-2 para remover o pulso indesejado – o tal de *glitch* – e refaça os testes para garantir que a sua solução é mesmo correta.

Etapa 9 Uma vez que o funcionamento do *mux*-2 seja aquele especificado, você deve, lembrando das árvores projetadas em `combin.pdf`,

(1) escrever[5] o modelo para o *mux*-4. Para tanto edite o arquivo `combin.vhd`;

(2) repetir para o *mux*-8;

(3) repetir para o multiplexador de 8 entradas com um vetor de bits ao invés de oito escalares;

(4) acrescentar mais elementos aos respectivos vetores de teste em `tb_mux.vhd`; e

(5) verificar a corretude de seus modelos[6].

Quantos vetores são necessários para testar exaustivamente um *mux*-4? São duas entradas de seleção e quatro entradas de dados, logo são necessários $2^2 \times 2^4$ testes. Use o material biológico que se encontra entre suas orelhas e veja se é mesmo necessário executar *todos* os 64 testes, ou se algo pode ser aproveitado dos testes do *mux*-2. Reveja a Seção–5.3.2.

Você quer mesmo escrever $2^3 \times 2^8 = 2^{11}$ vetores de teste para o *mux*-8? Como você convenceria o advogado do seu cliente de que um conjunto menor de testes é o *suficiente* para garantir a corretude do seu modelo?

Copie seus vetores de teste do laboratório passado (Seção 13.1), e se for o caso, acrescente novos elementos ao vetor.

[5] Você **pode** se lembrar do que fez no laboratório passado. Eu sei, o filme é uma b∗st∗.

[6] Para aqueles que tem memória de invertebrado: basta dizer "`./run_mux.sh`", ler eventuais mensagens de erro, resolver os problemas e então dizer "`./run_mux.sh 1 &`".

13.2.9 *Testbench* para os demultiplexadores e decodificadores

Reveja a Seção–5.3.2. O *script* `run_combin.sh` compila o código VHDL e gera um simulador para os modelos dos demultiplexadores e decodificadores. Este *script* é similar a `run_mux.sh`.

Etapa 10 Lembrando das árvores mostradas em `combin.pdf`, para esta etapa você deve:

(1) escrever os modelos para o *demux*-2. Para tanto edite o arquivo `combin.vhd`;

(2) repetir para o *demux*-4 e *demux*-8;

(3) acrescentar mais elementos aos respectivos vetores de teste em `tb_combin.vhd`; e

(4) verificar a corretude de seus modelos[7].

Achtung: no TB dos demultiplexadores e decodificadores, em `tb_combin.vhd`, um único registro é usado para testar os dois circuitos de largura N (*demux*-N e *decod*-N) ao mesmo tempo e portanto, dependendo do teste, alguns dos campos não são relevantes naquele teste.

No registro com os vetores de teste, 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 (somente) pelos valores em `s`.

Os **assert**s que verificam a corretude dos demultiplexadores e decodificadores estão comentados para reduzir a poluição na tela. Para testar seus modelos, altere a constante respectiva (TST_DEMUX_2 ou TST_DECOD_2) para false e então verifique os resultados. Há uma tripla de constantes para cada circuito, *viz* TST_DEMUX_2, TST_DEMUX_4 e TST_DEMUX_8.

> Você é quem escreve os vetores de teste e portanto sua tarefa é ajustar os campos `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.

Etapa 11 Caso necessário, aplique sua solução para o problema hamletiano do *mux*-2 aos modelos do *demux*-2 e *decod*-2. Reveja a Seção–5.3.2.

Responda às questões na última página e as entregue ao professor até as 23:30 da terça de carnaval – envie fotos das respostas, ou PDFs.

Histórico das Revisões:
26fev21: Não esqueça da minha caloi: *Reveja a Seção–5.3.2.*
22jun20: remoção do material que está em `vhdl.pdf`, separação dos TBs para mux e demux;
10out16: acréscimo de **`with-select`**
03set16: vetores de testes separados por tamanho de circuito;
25ago16: sugestões de Zanata, troca seletor para decodificador;
22sep15: modelo de temporização com T_c;
14sep15: definição de circuito combinacional;
30jul15: remoção de modelagem estrutural, mais abstração;
13ago13: inserção de lstinline, revisão no texto;
20mar13: segunda versão;
01nov12: primeira versão.

EOF

[7] Reveja a nota de rodapé número 6.

13.2.10 Questionário

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 12 Leia o código VHDL dos modelos e estime os tempos de propagação dos 9 circuitos, tomando como base os tempos de propagação das portas lógicas em `packageWires.vhd`. Mostre suas contas.

Etapa 13 Repita para o tempo de contaminação dos nove circuitos. Mostre suas contas.

Etapa 14 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 15 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 13.3.4 de `vhdl.pdf` para responder às próximas perguntas. Copie o 'miolo' dos seus modelos na folha de respostas.

Etapa 16 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 17 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.