

1 Circuitos Combinacionais: mux, demux, sel

Objetivos São três os objetivos deste laboratório: (i) escrever modelos estruturais de circuitos combinacionais; (ii) verificar o comportamento temporal do multiplexador; e (iii) verificar a corretude dos modelos através de **asserts** e dos diagramas de tempo.

Preparação Ler seção 1.3 de [RH12], sobre circuitos combinacionais.

1.1 Material Disponibilizado Para Sua Tarefa

Da tarefa:

1. O trabalho pode ser efetuado em duplas;
2. copie para sua área de trabalho o arquivo com o código VHDL:

```
wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_combinacionais.tgz
```


Expandá-o com `tar xzf l_combinacionais.tgz`
o diretório combinacionais será criado. 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, pois digitar `reg8` é mais econômico do que `bit_vector(7 downto 0)`. abreviaturas

O arquivo `aux.vhd` contém os modelos das portas lógicas *inv*, *and*, *or*, *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, *mux2*. Este modelo serve de base para que você escreva os modelos para os componentes incompletos, quais sejam, *mux4*, *mux8*, *demux2*, *demux4*, *demux8*, *sel2*, *sel4*, *sel8*, que são definidos na Seção 1.3 de [RH12].

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 detectado pelo compilador VHDL, 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**. Não desperdice a ajuda que é oferecida.

O arquivo `v.vcd` contém definições para o gtkwave tais como a escala de tempo e sinais a serem exibidos na tela para a verificação do modelo *mux2*.

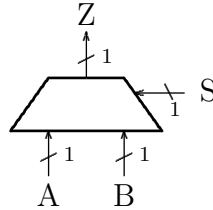
1.2 Modelo do Multiplexador

A entidade do multiplexador de duas entradas é mostrada abaixo. O circuito tem duas entradas A e B, uma entrada de seleção S e uma saída Z, todas com um bit de largura. Há uma correspondência exata entre os sinais no diagrama e as portas da entidade.

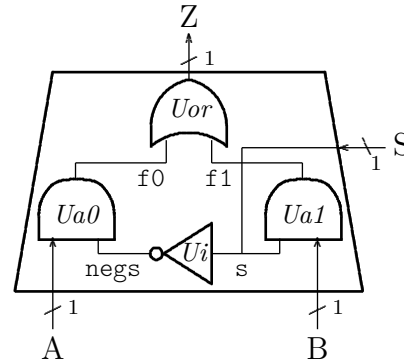
entity

Prog. 1: Entidade do mux2.

```
use work.p_wires.all;
entity mux2 is
  port(A,B : in bit;
        S : in bit;
        Z : out bit);
end mux2;
```



A implementação óbvia do mux2 é mostrada ao lado. São necessários três sinais internos, negs com o complemento de S, e f0,f1 para interligar as portas and e a porta or. Um modelo estrutural do mux2 é mostrado abaixo. Os componentes e os sinais internos são *declarados* entre o **architecture** e o **begin**. Os componentes são *instanciados* entre o **begin** e o **end architecture**.



Prog. 2: Arquitetura do mux2.

```
architecture estrutural of mux2 is

  — declaracao dos componentes
  component inv is          — inversor
    port(A : in bit; S : out bit);
  end component inv;

  component and2 is         — porta and-2
    port(A,B : in bit; S : out bit);
  end component and2;

  component or2 is          — porta or-2
    port(A,B : in bit; S : out bit);
  end component or2;

  — declaracao dos sinais internos ao mux2
  signal negs,f0,f1 : bit;

begin — inicio da area concorrente

  — instanciacao dos componentes
  Ui: inv generic map (t_inv) port map (s,negs);
  Ua0: and2 generic map (t_and2) port map (a,negs,f0);
  Ua1: and2 generic map (t_and2) port map (b,s,f1);
  Uor: or2 generic map (t_or2) port map (f0,f1,z);

end architecture estrutural; — fim da area concorrente
```

Cada componente tem um *label* (opcional) – Uxx: que significa “*design Unit xxx*” – e o mapeamento das portas do componente com os sinais da interface (declarados na entidade) e os sinais internos (declarados na arquitetura). Por exemplo, na instanciação `Ua0: and2 port map(a, negs, f0)` as portas *A*, *B* e *Z* do componente *and2* são ligados aos sinais *a*, *negs*, e *f0*, e este mapeamento é chamado de “mapeamento posicional” porque os sinais da arquitetura são associados às portas do componente na ordem em que as portas são declaradas.

port map

mapeamento
posicional

A região entre o **begin** e o **end architecture** é chamada de *área concorrente* e os comandos nesta área são executados concorrentemente. Isso significa que o código dos componentes instanciados é simulado (‘executado’) em paralelo, imitando o comportamento de um circuito real. Os quatro componentes do *mux2* reagem a eventos nas entradas e possivelmente provocam eventos na saída. Um *evento* é uma mudança num sinal ($1 \rightarrow 0$ ou $0 \rightarrow 1$).

área
concorrente

evento

Os tempos de propagação das portas lógicas (*t_inv*, *t_and2*, *t_or2*) são constantes declaradas no arquivo `packageWires.vhd`. Um **generic map** permite redefinir uma constante – originalmente definida na declaração da arquitetura – quando aquela arquitetura é instanciada. Na arquitetura do Prog. 2, o tempo de propagação é ‘redefinido’ para os valores das constantes declaradas em `packageWires.vhd`.

generic map

1.3 Testbench

O arquivo `tb_combin.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos seus modelos. Para simplificar a depuração do seu código VHDL, você deve verificar cada novo modelo assim que for completado.

A entidade `tb_combin` é vazia porque o programa de testes é autocontido e não tem interfaces com nenhum outro circuito. A arquitetura do TB declara os componentes que serão testados e um **record** que será usado para excitar os modelos. O registro `test_record` 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` contém onze elementos para ilustrar as possibilidades, e excitar o *mux2* na primeira tarefa deste laboratório.

Registro
(**record**)
`test_record`Vetor (**array**)
`test_array`

No `test_record`, o campo *k* é um bit (`'0'`), os campos *a* e *s* são vetores de bits¹ codificados em binário (`b"01001100"`). 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 *sl* é o vetor de bits com a saída esperada para um seletor cujas entradas são definidas pelos valores em *s*.

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

Note que um único registro é usado para testar todos os circuitos e portanto, dependendo do teste, alguns dos campos não são relevantes *naquele teste*. Por exemplo, para testar o *mux2*, somente o bit 0 do sinal *s* (*s*(0)) é relevante; para testar *mux4*, *s*(1 **downto** 0) são relevantes, e para o *mux8* todos os três bits são relevantes, *s*(2 **downto** 0).

seleção de
componentes de
um vetor de
bits: *s*(0)
s(M **downto** N)

¹Aspas simples para escalares e aspas duplas para vetores.

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

```

type test_record is record
  k  : bit;      — entrada de bit para demultiplexadores
  a  : reg8;     — entrada para multiplexadores
  s  : reg3;     — entrada de selecao
  mx : bit;     — saida esperada do MUX
  dm : reg8;     — saida esperada do DEMUX
  sl : reg8;     — saida esperada do SEL
end record;

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

constant test_vectors : test_array := (
  —k,   a,       s,   mx,   dm,       sl
  ('0',b"00000011",b"000", '0',b"00010000",b"00000000"),
  ('0',b"00000011",b"001", '1',b"00010000",b"00000000"),
  ...
  ('0',b"00000001",b"000", '1',b"00000000",b"00000000")
);

```

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 comando **assert** é similar a um printf() em C e pode ser usado para exibir o valor de sinais ao longo de uma simulação. Este comando tem três cláusulas: **assert** condicao **report** string **severity** nivel. A *condição* deve ser falsa para que o simulador imprima a *string*. A severidade pode ter quatro níveis: note, warning, error, failure, e a última (failure) aborta a simulação.

assert

O **assert** abaixo 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. Note que se você escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no seu modelo.

Prog. 4: Mensagem de verificação de comportamento.

```

assert saidaMUX2 = esperadaMUX
report "mux2:␣saida␣errada␣sel=" & B2STR(s(0)) &
"␣saiu=" & B2STR(saidaMUX2) & "␣esperada=" & B2STR(esperadaMUX)
severity error;

```

Se os valores de saidaMUX2 e esperadaMUX diferem, a mensagem "tb_combin.vhd:125:9:200ps:(assertion error): mux2: saida errada sel=1 saiu=1 esperada=0" 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*.

&
concatenação

A seleção de um subcampo de bits é obtida especificando-se quais bits deseja-se selecionar. Considere os quatro sinais declarados no Prog. 5, respectivamente com 1, 8, 4 e 4 bits de largura, e as atribuições com as larguras de campo apropriadas.

seleção de campo

Prog. 5: Atribuições de campos de bits.

```

signal k: bit;           — um bit
signal a: reg8;          — vetor de oito bits
signal r: reg4;          — vetor de quatro bits
signal t: reg4;          — vetor de quatro bits
...
k <= a(7);               — atribuicao do bit mais significativo
r <= a(5 downto 2);      — atribuicao do quarteto central de a
t <= a(2 to 5);          — mesma atrib, ordem dos bits invertida

```

A ordem definida com **downto** corresponde ao vetor de bits com a posição mais significativa à esquerda, que é a ordem natural de representação de números, enquanto que a definição com **to** coloca o bit mais significativo à direita, com o ‘peso’ dos bits em ordem crescente.

downto, to

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

1.4 Nem mesmo um circuito simples é bem-comportado?

Da tarefa: (continuação)

1. verifique, cuidadosamente, as combinações de entradas e a saída do modelo mux2. As entradas são entr(0) e entr(1) (v.a(0) e v.a(1)) e o sinal de controle é s0 (v.s(0)). Note que os **asserts** correspondentes ao mux2 não são impressos – o que indicaria que o comportamento é o esperado. Contudo, no diagrama de tempos 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 diga `./run.sh 1 &`;
2. observe os instantes 440ps, 840ps e 1240ps;
3. uma vez identificado o problema, qual a solução?

1.5 Modelagem Estrutural de Circuitos Mais Complexos

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, na linha 11, de 1 para 0. Recompile os modelos e verifique seu funcionamento com gtkwave.

Da tarefa: (continuação)

1. acrescente ao arquivo combin.vhd o código VHDL para os modelos dos multiplexadores de 4 e 8 entradas, dos demultiplexadores de 2,4,8 saídas, e dos seletores de 2,4,8 saídas;
2. acrescente e/ou altere os elementos do vetor de testes para verificar a corretude do seu modelo.

Há duas entidades distintas para o mux8. A primeira, mux8, emprega oito sinais de tipo bit nas entradas, e três bits para seleção. A segunda, mux8vet

mostrada no Prog. 6, tem como entradas um vetor de 8 bits `entr: reg8` e outro vetor `sel: reg3` para a seleção. As duas arquiteturas são idênticas, exceto que as ligações dos sinais da interface aos componentes internos devem usar seleção de campos de bits, como discutido acima.

Prog. 6: Entidade do `mux8` com vetores de bits.

```
entity mux8vet is
  port(entr : in  reg8; — vetor com 8 bits
        sel  : in  reg3; — vetor com 3 bits
        Z    : out bit); — saída de um bit
end mux8vet;
```

Note que é você quem deve ajustar a saída esperada dos circuitos nos vetores de teste. O projetista dos modelos é responsável por escrever os vetores de teste e portanto sua tarefa é ajustar os campos `mx` (saída esperada dos `muxN`), `dm` (saída esperada dos `demuxN`), e `sl` (saída esperada dos `selN`). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros.

a responsabilidade é do projetista!

Os **asserts** que verificam a corretude dos demultiplexadores e seletores estão comentados para reduzir a poluição na tela. Após completar a verificação dos três multiplexadores, comente seus **asserts** e descomente aqueles dos demultiplexadores. Repita para os seletores.

1.6 Temporização

Os modelos das portas lógicas contêm o comando **after** para imitar o tempo de propagação de portas lógicas reais. Por exemplo, no modelo do inversor em `aux.vhd`, o comportamento (temporal) é definido por

after

```
S <= (not A) after prop;
```

O complemento de A será atribuído a S após o tempo de propagação definido no **generic** da entidade. A constante `t_inv` está definida no arquivo `packageWires.vhd`, e seu valor é 15 ps (15×10^{-12} s). O comportamento do sinal S é tal que pulsos em A com duração menor que `t_inv` são rejeitados, e 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 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 comando **after** modela “atrasos inerciais”.

atraso inercial

O modelo da temporização das portas com duas ou mais entradas é um pouco mais sofisticado e usa a cláusula **reject**. Por exemplo, o comportamento da porta `and2` é definido por

reject

```
S <= reject t_rej inertial (A and B) after prop;
```

O tempo de propagação também é aquele definido no **generic**, exceto que pulsos com duração maior que `t_rej` são refletidos na saída da porta. Quando se usa a cláusula **reject**, é necessário incluir o qualificador **inertial**. Quando se usa apenas o **after**, não é necessário qualificar o atraso como sendo inercial. A especificação completa de um atraso sem rejeição é

```
A <= inertial B after prop;
```

Da tarefa: (continuação)

1. leia o código VHDL dos modelos e estime os tempos de propagação dos circuitos, tomando como base os tempos de propagação das portas lógicas. Note que o *tempo de propagação* de um circuito é o pior caso dentre os tempos de propagação para as várias combinações de entradas; tempo de propagação
2. edite `packageWires.vhd` e altere a definição da constante `simulate_time`, na linha 11, de 0 para 1. Recompile os modelos e verifique seu funcionamento com `gtkwave`;
3. verifique as diferenças nos tempos de propagação das versões de 2, 4 e 8 entradas dos multiplexadores, demultiplexadores e seletores. Use os cursores do `gtkwave` para medir os tempos;
4. confirme, com base nos diagramas de tempo, se os tempos medidos são similares aos que você estimou.

Referências

- [RH12] *Sistemas Digitais e Microprocessadores*, R.A.Hexsel, 2012, Editora da UFPR.
- [PJA04] *VHDL Tutorial*,
/home/html/inf/roberto/ci210/vhdl/vhdl-tutorial.pdf