UFPR – Bacharelado em Ciência da Computação 2 de setembro de 2019
CI210 – Projetos Digitais e Microprocessadores
Laboratório sobre Modelagem de Circuitos Combinacionais em VHDL — 2019-2

1 Modelagem de Circuitos Combinacionais em VHDL

VHDL é uma sigla que contém uma sigla: *VHSIC Hardware Description Language*, sendo VHSIC abreviatura para *Very High Speed Integrated Circuits*. Como o nome diz, VHDL é uma linguagem de **descrição** de *hardware* (*HDL*). Neste, e nos três próximos laboratórios, usaremos primariamente construções para descrever a *estrutura* dos circuitos. As construções para definir *comportamento* serão vistas adiante.

Diferentemente de C, um programa em VHDL *descreve* uma estrutura, ao invés de *definir* ou *prescrever* uma computação. Neste sentido, um programa em C é uma "receita de bolo" com ingredientes (variáveis) e modo de preparar (algoritmo), enquanto que em VHDL um programa descreve a interligação de componentes. A partir da estrutura, o compilador `ghdl` gera um simulador para o circuito descrito pelo código fonte. Adiante veremos como definir o comportamento de um circuito, sem que seja necessário definir a sua estrutura.

1.1 Objetivos

São dois os objetivos deste laboratório: (i) efetuar a modelagem estrutural em VHDL de multiplexadores, demultiplexadores e decodificadores; e (ii) verificar, através de simulação, a corretude dos modelos daqueles circuitos.

O trabalho pode ser efetuado em duplas.

1.2 Descrição gráfica de circuitos

Examine o diagrama da Figura 1 e liste, de forma sucinta, as convenções empregadas para que ele possa ser interpretado por discentes matriculados nesta disciplina. DeMorgan

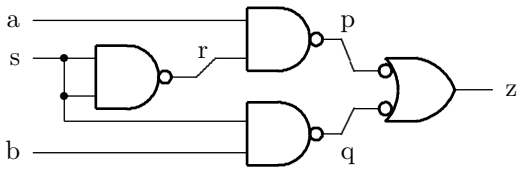


Figura 1: Circuito combinacional.

Espaço em branco proposital.

Sua lista deve conter, pelo menos:
(i) uma convenção para representar as portas lógicas;
(ii) uma convenção para representar as ligações;
(iii) uma convenção para representar as entradas;
(iv) uma convenção para representar as saídas; e
(v) uma convenção para nominar os valores lógicos transportados pelas ligações.

Até agora temos usado várias **convenções**:

* portas lógicas tem formatos distintos ($\neg \wedge \vee \oplus$);
* multiplexadores e decodificadores são trapézios;
* somadores são trapézios com um recorte/chanfro;
* registradores são retângulos com o triângulo do relógio;
* circuitos complexos são caixas retangulares;
* ligações (fios, sinais) são linhas;
* sinais 'fluem' da esquerda para a direita;
* entradas no lado esquerdo (ou acima);
* saídas no lado direito (ou abaixo);
* se não é óbvio, setas indicam o 'fluxo' dos dados; e
* um traço curto com um número indica largura do sinal, se maior do que 1.

O diagrama na Figura 2 ilustra estas convenções: são quatro os registradores, chamados de r1, r2, r3 e STAT, os sinais fluem da esquerda para a direita, o sinal clock tem um bit de largura, o sinal oper e a entrada do registrador STAT tem 4 bits de largura, e os demais sinais tem largura de 16 bits. O circuito no centro é uma unidade de lógica e aritmética.

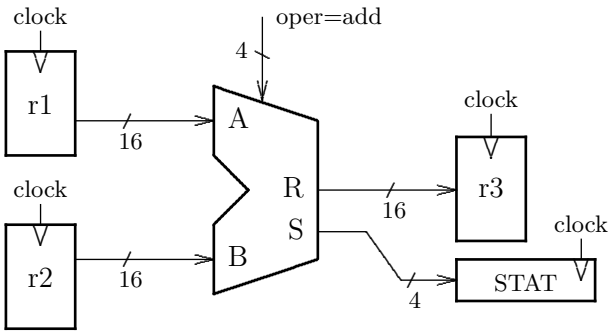


Figura 2: Exemplo das convenções usadas em diagramas de circuitos.

1.3 E se a descrição for textual?

A interpretação automática de desenhos é uma arte, mesmo que os desenhos sejam diagramas tão simples como nossos circuitos. Para facilitar a interpretação automática de representações para circuitos emprega-se texto.

A representação que usaremos nesta disciplina e adiante no curso é codificada na linguagem VHDL – V é uma sigla, HDL é a abreviatura para *Hardware Description Language* – grave este nome.

Circuitos são representados em VHDL por *sinais* (fios) e *componentes* (*design units*). Um componente é descrito por duas construções da linguagem, uma *entidade* – que descreve sua interface com o mundo externo – e uma *arquitetura* – que descreve seu comportamento através das construções da linguagem.

Uma entidade (**entity**) descreve as ligações externas do componente: entidade
(i) entradas são **in**
(ii) saídas são **out**
(iii) ligações são representadas por *sinais*, no nosso caso bits ou vetores de bits (bit_vectors).

Uma arquitetura (**architecture**) descreve as ligações internas do componente. A arquitetura declara quais componentes são usados na implementação, que sinais internos são necessários para interligar os componentes internos, e de que forma os componentes são interligados entre si e com os sinais da entidade. arquitetura

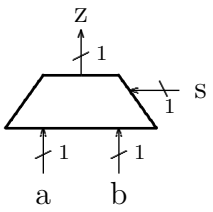
Uma certa entidade pode ser modelada em VHDL de mais de uma forma, como veremos em breve – para uma entidade (interface), podem ser projetadas mais de uma arquitetura (circuito).

Antes de passarmos a um exemplo, vejamos as convenções de tipografia para o código VHDL: palavras reservadas são grafadas em negrito (**entity**), nomes de sinais e componentes são grafados sem serifa (meuSinal), e comentários iniciam com '−−' e terminam no final da linha e são grafados com −−*caracteres inclinados*. Nesta aula vocês receberão todo o código necessário às tarefas.

1.3.1 MUX2 – entidade e arquitetura

O multiplexador da Figura 1 é descrito em VHDL pelas suas entidade e arquitetura. A entidade chamada mux2 descreve a interface deste componente, com três sinais de entrada e um de saída, todos com a largura de um bit.

```
entity mux2 is
  port(a,b : in  bit;
       s   : in  bit;
       z   : out bit);
end mux2;
```



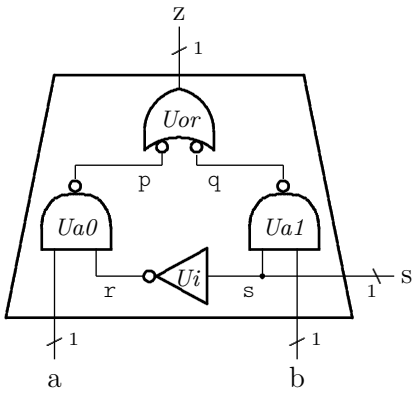
A arquitetura, chamada `estrutural` define a estrutura do circuito modelado – ela descreve como os componentes são interligados. Entre as palavras chave **architecture** e **begin** devem ser declarados os componentes usados na implementação do multiplexador – neste caso inversor, porta *and* e porta *or* – e os sinais internos necessários para ligar os componentes e a interface, que são os sinais `r`, `p` e `q`.

Entre o **begin** e o **end architecture** são definidas as ligações entre os componentes da implementação. Como mostra o diagrama ao lado do código, há uma correspondência direta entre o circuito e o código VHDL que o representa. Infelizmente, nas descrições textuais TODOS os sinais devem ser explicitamente nomeados.

A construção **port map** "faz a ligação" entre os sinais internos à arquitetura e os sinais declarados para cada componente individual usado na modelagem. O **port map** é quem liga sinais em componentes ao mapear os sinais internos declarados na arquitetura, aos sinais declarados nas entidades dos componentes. A instanciação dos componentes com o mapeamento das portas é similar a um operador aritmético prefixado; ao invés da forma usual, que é infixada ($c = a + b$), emprega-se a forma prefixada ($+(c, a, b)$).

O trecho de código abaixo é parte do material que lhe será fornecido nesta aula. O código está incompleto e serve apenas para mostrar como os componentes do modelo são interligados. As instruções de como compilar e simular este modelo estão na Seção 1.6.

```
architecture estrutural of mux2 is

  -- declaração de componentes
  component inv is
    port(A : in bit; S : out bit);
  end component inv;

  component nand2 is
    port(A,B : in bit; S : out bit);
  end component nand2;

  -- declaração de sinais internos
  signal r, p, q : bit;

begin

  -- instanciação e
  --   interligação dos componentes
  Ui:  inv  port map(s, r);

  Ua0: nand2 port map(a, r, p);

  Ua1: nand2 port map(b, s, q);

  Uor: nand2 port map(p, q, z);

end architecture estrutural;
```



Entre **architecture** e **begin**, os componentes e os sinais são *declarados*. Entre o **begin** e o **end architecture**, os sinais e componentes são *instanciados* e conectados através dos sinais internos e do mapeamento dos sinais nas portas dos entidades com o **port map**. O **port map** 'gruda' os sinais nos terminais do componente.

1.4 Projeto *top–down*

Uma forma de descrever o "ciclo de vida" do desenvolvimento de um sistema é com a chamada *espiral de projeto*, mostrada na Figura 3. Tipicamente, o processo se inicia com uma ideia, e a partir dela escreve-se uma *especificação* (abstrata) para o sistema. Esta especificação é *refinada* para um projeto, que essencialmente é uma versão mais detalhada e menos abstrata da especificação. A *implementação* do projeto resulta numa versão concreta da especificação, e esta pode então ser *verificada* quanto a sua corretude e avaliada quanto ao seu desempenho. Se a implementação não satisfaz à especificação, deve-se efetuar um novo ciclo da espiral, com repetições até que os requisitos de projeto estejam atendidos.

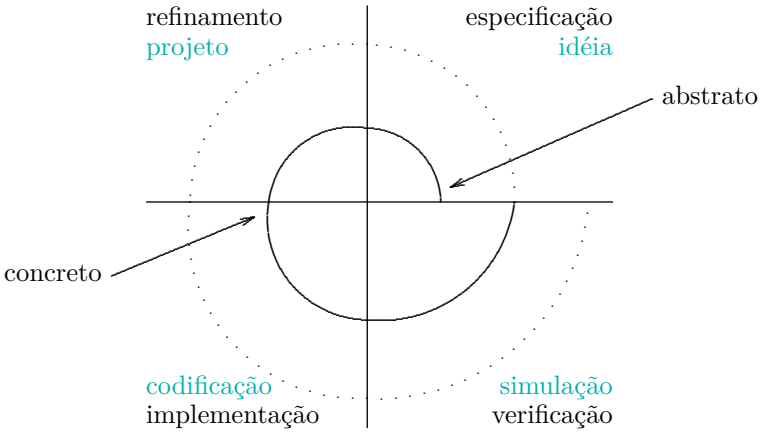


Figura 3: Espiral de projeto.

Da forma como VHDL será empregada nesta disciplina, (i) a *ideia* é a especificação em Português de um circuito, (ii) o *projeto* é o primeiro refinamento da especificação, que é obtido com trabalho intelectual, lápis e papel; (iii) a implementação é a *codificação* em VHDL, e (iv) a verificação é a confirmação de que a implementação satisfaz ao especificado, através de *simulação*.

Um projeto pode ser refinado de mais de uma forma, na sua fase inicial, e a cada refinamento, o resultado é mais concreto e próximo da implementação. Por exemplo, partindo-se de uma especificação (abstrata) do comportamento desejado, esta sofre um refinamento que resulta num diagramas de blocos, que ao ser refinado produz o projeto dos blocos, o refinamento dos blocos resulta na formalização da especificação, que é a modelagem concreta dos componentes do projeto em VHDL – código VHDL para os *componentes* do projeto.

1.5 VHDL

VHDL é uma linguagem extremamente versátil e é usada para

- a modelagem e simulação de circuitos;
- especificação, projeto e síntese de circuitos;
- descrever lista de componentes e de ligações; e
- verificação de corretude: se especificação $\Rightarrow$ implementação.

A linguagem é padronizada pelo IEEE, no padrão IEEE 1076-1987 V[HSIC] H D L, ou *Very High Speed Integrated Circuit Hardware Description Language*. O padrão

IEEE Std 1164 define pacote std_logic, e o padrão IEEE Std 1076.3 VHDL Numeric Standard define uma biblioteca de funções numéricas. Há mais, muito mais, nos bons textos do ramo.

1.5.1 Como se dá a compilação de VHDL?

A compilação de código VHDL se dá em três fases: análise, elaboração e simulação. No nosso caso, o resultado da compilação é um programa em C gerado por `ghdl`, que simula o comportamento do modelo escrito em VHDL. Usaremos `gtkwave` para verificar o comportamento dos modelos dos circuitos. Detalhes em seguida.

Análise Na fase de análise, o compilador VHDL verifica a corretude da sintaxe e a semântica dos comandos. Na análise sintática, são sinalizados erros de grafia das palavras reservadas, sinais que não foram declarados, etc. Na análise da semântica, são verificados os tipos dos sinais, e se os operadores são aplicados a operandos com os tipos apropriados – somar dois números faz sentido, somar dois **if**s não.

A análise pode ser efetuada somente em partes de um modelo. Por exemplo, um par entidade-arquitetura (uma *design unit*) pode ser analisado em separado, e se nenhum erro é encontrado, o resultado da análise é armazenado na biblioteca `work-obj93.cf`, para uso futuro. Esta biblioteca é mantida no diretório em que ocorre a compilação.

Elaboração Na fase de elaboração, o compilador constrói um modelo completo de toda a hierarquia do projeto. Reveja o código da arquitetura do mux2 na página 4. A hierarquia consiste da arquitetura do mux2 e mais as declarações dos componentes inv e nand2.

O compilador examina a arquitetura de mais alto nível da hierarquia e expande todas as declarações dos componentes instanciados naquela arquitetura. Para cada componente, o compilador examina sua arquitetura e expande todas as declarações lá instanciadas. A elaboração termina quando todas as declarações forem substituídas por atribuições – para ser mais preciso, a elaboração termina quando só restam processos e os sinais que os interligam. De forma muito simplificada, um *processo* é uma generalização para uma atribuição. Voltaremos a falar de processos em outro laboratório.

Simulação Ao final da elaboração, o `ghdl` traduz a estrutura de processos e os sinais que os interligam para um programa em C, que depois de compilado, permite simular a execução do modelo completo. Veja, na Seção 1.5.4, como ocorre a simulação.

A saída da simulação pode ser mostrada na tela, pode ser gravada em arquivos, ou pode gerar um conjunto de dados que representa um diagrama de tempo para ser exibido com o programa `gtkwave`.

Neste laboratório e nos próximos, o processo de compilação está escondido num *script* que lhes será fornecido juntamente com o código VHDL para verificar os modelos por estudar.

1.5.2 Tipos de dados

Alguns dos tipos básicos de dados disponíveis em VHDL são mostrados na Tabela 1. O tipo bit tem dois valores, '0' e '1' – note as aspas simples. Um vetor de bits, ou um bit_vector, é representado por mais de um bit, enfeitado com aspas duplas: "1001". As regras para caracteres (aspas simples) e cadeias (aspas duplas) são similares àquelas para bits e vetores de bits. Constantes booleanas, números inteiros, números reais e constantes de tempo são representadas sem aspas.

TIPO	VALOR	EXEMPLO DE USO
Bit	'1', '0'	D <= '1';
Bit_vector	vetor de bits	Dout <= "0011";
Boolean	TRUE, FALSE	start <= TRUE;
Integer	-2,-1,0,1,1	Cnt <= Cnt + 1;
Real	2.3, 2.3E-4	med <= sum/16.0;
Time	100 ps, 3 ns	sig <= '1' after 12 ns;
Character	'a','1','@'	c <= 'k';
String	vetor de char	msg <= "error: " & "badAddr";

Tabela 1: Subconjunto pequeno dos tipos de dados em VHDL.

A atribuição a um sinal é representada pela "flecha dupla" <= e a concatenação de vetores de bits, ou de caracteres, é representada pelo &, como mostra o exemplo da concatenação das cadeias de caracteres. Ah sim: VHDL não é *case sensitive*.

1.5.3 Sinais e Vetores de Sinais

Os *sinais* em VHDL correspondem aos fios que transportam os bits no circuito e a linguagem define sinais de um bit, e vetores de sinais com muitos bits.

O código abaixo declara um sinal x com um bit de largura, e este é inicializado em '1'. O sinal vb é um vetor com 4 bits de largura e a posição de cada um dos bits é aquela em que o bit mais significativo está à esquerda, e o menos significativo à direita. Esta é a declaração adequada para vetores que representam números.

```
-- declaração de sinais do tipo bit
signal x: bit := '1'; -- inicializado em '1'
signal vb: bit_vector(3 downto 0) := "1100"; -- inicializa em 12
```

O sinal bv é declarado com a direção contrária àquela do sinal vb, e o bit mais significativo deste vetor é aquele na direita.

```
signal bv : bit_vector(0 to 3); -- bit 0 é o Mais Significativo
```

As atribuições abaixo produzem resultados que talvez não sejam intuitivos a uma primeira vista.

```
signal bH : bit_vector(7 downto 0); -- bit 7 é Mais Sign.
signal bL : bit_vector(0 to 7);     -- bit 0 é Mais Sign.

bH <= b"11000000" -- bH se torna 0xc0 = 192
bL <= b"11000000" -- bL se torna 0x03 = 3
```

É possível selecionar um subconjunto dos bits de um sinal. Os comandos abaixo declaram 4 sinais: um com 1 bit, um com 8 bits, e dois sinais com 4 bits de largura.

A primeira atribuição seleciona o bit mais significativo do sinal `v8` e o atribui à `x`. As atribuições à `v4` e `t4` selecionam o quarteto central de `v8`, e armazenam estes quartetos na "ordem numérica" em `v4`, e na "ordem invertida" em `t4`.

```
signal x:  bit; -- um bit
signal v8: bit_vector(7 downto 0); -- vetor de 8 bits
signal v4: bit_vector(3 downto 0); -- vetor de 4 bits
signal t4: bit_vector(3 downto 0); -- vetor de 4 bits
  ...
x  <= v8(7);           -- atribuição do bit mais significativo
v4 <= v8(5 downto 2); -- atribuição do quarteto central de v8
t4 <= v8(2 to 5);      -- mesmo quarteto, na ordem inversa
```

O operador que seleciona um ou mais bits de um vetor é um par de parenteses (`v8(7)`), ao invés de um par de colchetes como em C. Em VHDL é possível selecionar um subconjunto com vários elementos de um vetor (`v8(2 to 5)`). seleção de bits

Vetores de bits podem ser representados nas bases hexadecimal (`x`), octal (`o`) e binária (`b`). A representação binária implícita é a normal: se nada for dito em contrário, o vetor é um vetor de bits.

```
hexadecimal  <= x"C0";        -- 1100 0000    OITO bits
octal        <= o"300";       -- 011 000 000  NOVE bits
binaria_expl <= b"11000000"; -- 11000000 base-2 explícita
binaria_impl <= "11000000";  -- 11000000 base-2 implícita
```

1.5.4 Mecanismo de simulação

A simulação de modelos escritos em VHDL emprega a *Simulação de Eventos Discretos* – os eventos ocorrem em instantes determinados pela própria simulação.

Um *processo* é uma construção básica da linguagem e que estudaremos adiante. No trecho de código acima, cada uma das atribuições é um processo, embora o uso do nome 'processo' seja reservado para um comando específico, que veremos em breve. processo

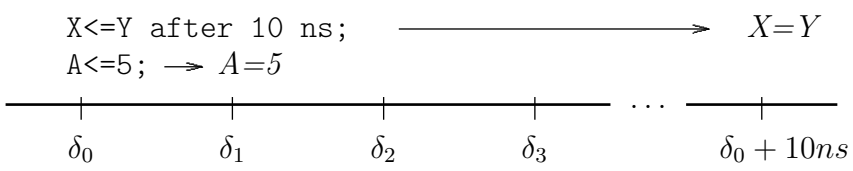
A simulação consiste de duas fases: uma fase de inicialização e passos de simulação.

Na *fase de inicialização* (i) todos os sinais são inicializados com os valores declarados, ou com os menores valores possíveis para os respectivos tipos dos sinais; (ii) o tempo simulado é inicializado em zero; e (iii) todos os processos (atribuições) são executados exatamente uma vez.

A atribuição a um sinal causa uma transação, e a atribuição será efetivada no próximo delta (Δt). Se o sinal muda de estado, então este sinal sofre um evento. Um evento no sinal S causa a execução de processos que dependem de S. delta

Uma atribuição pode especificar o instante em que transação ocorrerá:

```
A <= 5;              -- atribuição válida no próximo delta
X <= Y after 10 ns; -- atribuição depois de 10ns
```



No δ_0 ocorre a atribuição de 5 para A, e em δ_1 o sinal A já carrega o valor 5. Em δ_0 a atribuição de X em Y é enfileirada e escalonada para ocorrer no instante $\delta_0 + 10ns$.

Durante um *passo de simulação* (i) o tempo simulado avança até o próximo instante (δ) em que uma transação está programada; (ii) todas as transações programadas para aquele δ são executadas; (iii) estas transações podem provocar eventos em sinais; (iv) processos que dependem destes eventos são então executados; e (v) depois que todos os processos executam, a simulação avança até o δ em que a próxima transação está programada para ocorrer. Então repete-se o passo de simulação.

transação

evento

O tempo simulado avança em função de eventos nos sinais; a cada mudança no estado de um sinal corresponde um evento. Se ocorre um evento num sinal no lado direito de uma atribuição, a expressão do lado direito é reavaliada, e se houver mudança no estado do sinal representado pela expressão, o novo estado é atribuído ao sinal do lado esquerdo – causando um evento neste sinal. Vejamos um exemplo.

```
A <= 5;
B <= 4;
C <= A + B;
D <= C * 2;
```

As atribuições a A e a B ocorrem em δ_0 e provocam eventos em A e B. Em δ_1, o lado direito de C <= A + B; é avaliado e a alteração de valor em C provoca um evento naquele sinal. Em δ_2, o lado direito de D <= C * 2; é avaliado e a alteração provoca um evento em D. Em δ_3, D valerá 18.

Note que, ao contrário de C, todas as atribuições que podem ocorrer num δ ocorrem 'simultaneamente', e *não na ordem em que aparecem no código fonte*. Como num circuito de verdade, todos os sinais que podem mudar num determinado instante, mudam naquele instante, e as consequências destas mudanças se propagam pelo resto do circuito.

Sinais e expressões são avaliados neste delta, mas eventos têm efeito no próximo delta – enquanto houver eventos no delta corrente, estes são avaliados e eventos resultantes são escalonados para o próximo delta. Eventos podem ser escalonados para instantes futuros com **after**.

Considere o modelo de um somador completo, com modelagem da propagação dos sinais através do circuito. Um evento em A em δ_i pode causar um evento em Cout em δ_{i+1}, e em Sum em $\delta_i + 10ns$.

```
Sum  <= A xor B xor Cin after 10 ns;
Cout <= (A and B) or (A and Cin) or (B and Cin);
```

1.6 Da tarefa

Etapa 1 Copie para sua área de trabalho o arquivo com o código VHDL, e extraia seu conteúdo com os seguinte comandos:
(1) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_estrutural.tgz`
(2) expanda-o com `tar xzf l_estrutural.tgz`
(3) o diretório `estrutural` será criado na expansão da *tarball*;
(4) mude para aquele diretório com `cd estrutural`

O arquivo `packageWires.vhd` contém definições para nomes de sinais, uma vez que digitar reg8 é mais econômico do que bit_vector(7 **downto** 0).

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

O arquivo `estrut.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 *mux4*, *mux8*, *demux2*, *demux4*, *demux8*, *decod2*, *decod4*, *decod8*, que foram vistos em sala e estão 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 &`

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 *mux2*.

> *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.*** *Os programadores do GHDL dispenderam um esforço considerável para emitir mensagens (relativamente) úteis em caso de erro. Não desperdice a preciosa ajuda que lhe é oferecida.*

mensagens de erro

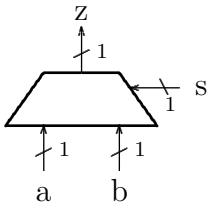
1.6.1 Modelo do multiplexador

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 de um bit.

entity

Programa 1: Entidade mux2.

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

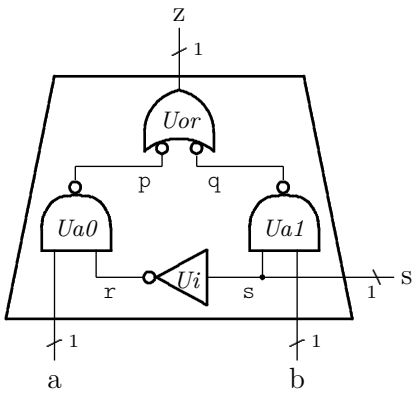


Espaço em branco proposital.

Um modelo estrutural do *mux2* é mostrado no Programa 2. Numa implementação óbvia do *mux2* são necessários três sinais internos, `r` com o complemento de `s`, e `p,q` para interligar as portas *and* e a porta *or*. No Programa 2, 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**.

Programa 2: Arquitetura do mux2.

```
architecture estrutural of mux2 is

  -- declaração dos componentes
  component inv is
    port(A: in bit; S: out bit);
  end component inv;

  component nand2 is
    port(A,B: in bit; S: out bit);
  end component nand2;

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

begin
  -- início da área concorrente

  -- instanciação dos componentes
  Ui:  inv   port map (s, r);
  Ua0: nand2 port map (a, r, p);
  Ua1: nand2 port map (b, s, q);
  Uor: nand2 port map (p, q, z);

  -- fim da área concorrente
end architecture estrutural;
```



Na instanciação, cada componente tem um *label* (opcional) – Uxx: que significa *"design Unit xx"* – 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

port map

```
Ua0: nand2 port map(a,r,p)
```

as portas A, B e S do componente *nand2* são ligados aos sinais `a`, `r`, e `p`, 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.

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

1.6.2 O modelo está pronto, e agora?

Posta a pergunta do título de outra forma: como testar um circuito combinacional?

Iniciemos pela tabela verdade do multiplexador de duas entradas, mostrada ao lado. São três as entradas e portanto as $2^3 = 8$ combinações de entradas devem ser verificadas, para garantir que a saída do circuito é a esperada.

s	a	b	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

O projetista de hardware tem duas obrigações distintas porém inseparáveis: (i) deve projetar um circuito que atenda ao especificado; e (ii) deve prover a garantia de que seu circuito atende à especificação.

Com base na tabela verdade – este não é o único método – o projetista gera um *vetor de testes* para exercitar o circuito, e confirmar que seu comportamento é o correto, segundo a especificação.

A especificação, junto com o vetor de testes, são o "contrato de compra e venda" do circuito, que garante ao 'comprador' que o circuito cumpre o que foi prometido pelo 'vendedor'.

1.6.3 *Testbench*

O arquivo `tb_estrut.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 seu código for completado.

A entidade tb_estrut é vazia porque o programa de testes é autocontido e não tem interfaces com nenhum outro componente.

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. Um **record** nada mais é do que uma tupla com o número adequado de componentes. 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

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            -- tupla na vertical
  k  : bit;       -- entrada de bit para demultiplexadores
  a  : reg2;      -- entrada para multiplexadores, 2 bits
  s  : bit;       -- entrada de selecao (para todos circuitos)
  mx : bit;       -- saida esperada do MUX
  dm : reg2;      -- saida esperada do DEMUX, 2 bits
  dc : reg2;      -- saida esperada do DECOD, 2 bits
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  -- mesma tupla na horizontal
  ('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"),
  -- nao alterar estes tres ultimos --
  ('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")
  );
```

O diagrama na Figura 4 mostra as ligações entre os componentes que você deve modelar e o processo que percorre o vetor de testes e gera as sequências de entradas-de-teste e saídas-de-teste. As entradas-de-teste (a, k, s) excitam seus modelos, que produzem saídas de acordo com suas especificações. As saídas-de-teste (mx, dm, dc) são comparadas com as saídas produzidas pelos modelos. O número de bits em a depende da largura do multiplexador, assim como o número de bits (n e m) dos sinais dm e dc.

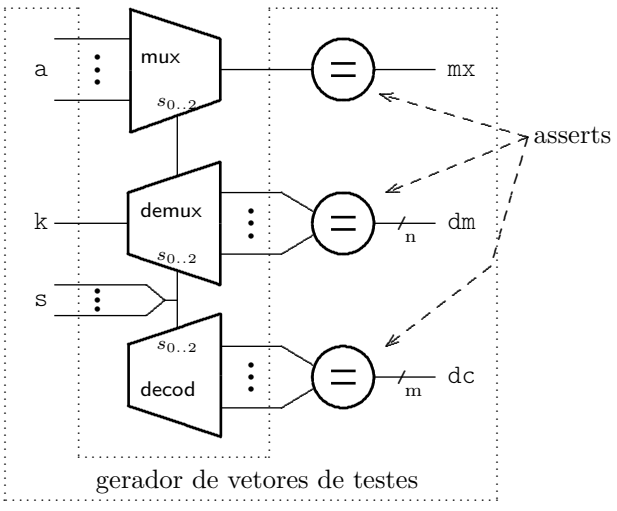


Figura 4: Ligações entre o *testbench* e os modelos.

A sequência de valores de entrada para os testes dos modelos é gerada pelo processo U_testValues, com três laços **for** ... **loop**, um para cada tamanho de circuito. 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. 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** condição **report** string **severity** nível. 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** 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. Note que 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 nos sinais 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*. A seleção de um subcampo de bits é obtida especificando-se quais bits deseja-se selecionar, como discutido na página 7.

& concatenação

seleção de campo

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

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.

1.6.4 Diagramas de tempo do `gtkwave`

O script `run.sh` pode disparar a execução do `gtkwave` para mostrar o diagrama de tempo dos circuitos. Para isso diga
`./run.sh 1 &`. O diagrama de tempo mostra, de cima para baixo:

`s` os bits de seleção, como um vetor de 3 bits – `s[2:0]`, e como três sinais individuais – `s0,s1,s2`;

`MUX` um agrupamento com os sinais para os multiplexadores, com

- **`esperadamux`** a saída esperada para os muxes (verde);
- **`entr_2`** os dois bits de entrada para mux2 (laranja);
- **`saidamux2`** a saída do mux2 (laranja);
- **`entr_4`** os quatro bits de entrada para mux4 (amarelo);
- **`saidamux4`** a saída do mux4 (amarelo);
- **`entr_8`** os oito bits de entrada para mux8 (verde);
- **`saidamux8`** a saída do mux8 (verde);
- **`saidamux8vet`** a saída do mux8vet (violeta);

`DEMUX` um agrupamento com os sinais para os demultiplexadores, com a mesma organização que para os muxes;

- **`inp`** entrada para os demuxes;
- **`esperadademux_2`** a saída esperada para o demux2 (laranja);
- **`sdemux2`** os dois bits de saída do demux2 (laranja);
- **`esperadademux_4`** a saída esperada do demux4 (amarelo);
- **`sdemux4`** os quatro bits de saída do dmux4 (amarelo);
- **`esperadademux_8`** a saída esperada do demux8 (verde);
- **`sdemux8`** os oito bits de saída do demux8 (verde);

`DECODIF` um agrupamento com os sinais para os decdificadores, com a mesma organização que para os demuxes;

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.

1.6.5 Teste dos multiplexadores

A coluna mx de test_vectors é a saída esperada para o *mux-2* e o

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

emitirá mensagem de erro somente se o modelo produzir saída diferente de mx. Se a saída esperada é a produzida, então o **assert** fica silente porque o circuito está correto, segundo o vetor de testes *que você, projetista, escreveu.*

Uma vez que você esteja certo de que o *mux-2* está correto, teste o *mux-4*. Este circuito é composto de três *mux-2*, e você garante que o *mux-2* é um circuito que atende a sua especificação. O *mux-4* tem seis entradas e sua tabela verdade tem $2^6 = 64$ linhas, sendo portanto necessários 64 testes. Certo?

Sim, é certo. Podemos usar de inteligência e descobrir qual é o número *mínimo* de testes para garantir a corretude do *mux-4*. Este número é bem menor do que 64.

Gere o vetor de testes (reduzido) para o *mux-4* e verifique sua corretude. Isso feito, troque 'TST_MUX_4 para false.

O *mux-8* é composto de dois *mux-4* – que você garante que é correto – e de um *mux-2* – que você também garante que é correto. O *mux-8* tem onze entradas e sua tabela verdade tem $2^{11} = 2048$ linhas. São necessários 2048 testes. Certo? Hmm...

Gere o vetor de testes (reduzido) para o *mux-8* e verifique sua corretude.Isso feito, troque 'TST_MUX_8 para false.

Vetor de bits na entrada do *mux8* 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 Programa 5, 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 devem usar seleção de campos de bits.

Programa 5: 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  1 bit
end mux8vet;
```

Para a entidade mux8, as entradas são a,b,c,d,e,f,g,h, enquanto que para a entidade mux8vet, as entradas são entr(0), entr(1), ..., entr(6), entr(7).

1.6.6 Teste dos demultiplexadores

Usaremos o mesmo procedimento dos *mux-n* para testar os demultiplexadores. Em `tb_estrut.vhd`, mude TST_DEMUX_2 para false.

A tabela verdade das saídas z e w do *demux-2*, é mostrada ao lado, e com base nela, o vetor de testes é mostrado no Programa 3.

s	a	z	w
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Da mesma forma que com os multiplexadores, os *demux-4* e *demux-8* são composições de *demux-2*. Uma vez que o componente 'pequeno' tenha sido verificado, a verificação do componente 'grande' necessita de um número relativamente pequeno de testes para também ser verificada.

Para cada 'tamanho' por testar, lembre de mudar para `false` a constante `TST_DEMUX_n` (`n` $\in \{2, 4, 8\}$) antes do teste.

1.6.7 Teste dos decodificadores

Você já entendeu o processo.

Etapa 2 Acrescente ao arquivo `estrut.vhd` o código VHDL para os modelos dos multiplexadores de 4 e 8 entradas, dos demultiplexadores de 2,4,8 saídas, e dos decodificadores de 2,4,8 saídas. Acrescente e/ou altere os elementos do vetor de testes para verificar a corretude dos seus modelos.

> *Note que é você quem deve ajustar a* saída esperada *para cada um 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* `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.*

a responsabilidade pelos testes é do projetista!

Referências

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

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

EOF

Histórico das Revisões:
`02set2019:` ajustes cosméticos;
`21ago2018:` diagrama do mux2;
`15ago2018:` texto introdutório, exemplos com deltas, mux8vet, gtkwave, (true or) nos asserts;
`03set2016:` vetores de testes separados por tamanho de circuito;
`25ago2016:` incluídas sugestões de Zanata, remoção de FPGA, troca seletor para decodificador;
`12set2015:` ajustes cosméticos, diagrama com simulador;
`29jul2015:` primeira versão.