

## µProcessador 1 Tutorial de Introdução ao VHDL

Uma HDL é uma *Hardware Definition Language*, ou seja, *não é uma linguagem de programação!* Nós usamos a linguagem VHDL para *especificar circuitos digitais*.

Antes de começarmos, um aviso:

### VHDL é uma linguagem bastante complexa!

Veremos aqui um *subset* da linguagem, uma versão bem *light* mesmo. A internet pode ajudar, mas cuidado pra não se afundar nos muitos detalhes, que serão vistos na disciplina de Lógica Reconfigurável.

### Ferramentas

Precisamos de três ferramentas para o trabalho: um editor de código, um compilador/simulador e um visualizador de formas de onda. O código será compilado e executado com o *ghdl* e os resultados serão vistos com o *gtkwave*. Um documento na página da disciplina contém as instruções de instalação.

Para editar o código, sugere-se o *Notepad++*; em sala de aula, pode-se usar o *Quartus II*. Outras alternativas são sugeridas nas instruções de instalação.

Crie um arquivo novo no editor de textos escolhido. Grave ele com o nome “porta.vhd”.

### Básico: Uma Porta Lógica

Digite o código abaixo no arquivo .vhd, não recorte-e-cole; assim você já vai vendo a cara do bicho (aprenda devagar, é o único jeito). O VHDL é *case-insensitive*, mas cuidado se for usar no Linux...

Vamos por partes. Comece com:

```
library ieee;  
use ieee.std_logic_1164.all;
```

O padrão IEEE para valores lógicos é o 1164, que inclui não só '0' e '1', mas outros sete, incluindo 'X' (desconhecido) e 'U' (não inicializado).

Siga agora com a *definição da interface*, ou seja, as entradas e saídas do bloco:

```
entity porta is  
  port( in_a : in std_logic;  
        in_b : in std_logic;  
        a_e_b: out std_logic  
  );  
end entity;
```

**CUIDADO: veja que o último pino especificado não termina com ponto-e-vírgula!**

A entidade é o bloco que estamos construindo. Observe que o nome da entidade deve “preferencialmente” ser o mesmo nome do arquivo (“porta”, no caso). Ou seja, *use um nome idêntico para o arquivo e para a “entity” e não brigue*.

A keyword “port” indica que vamos especificar as entradas (“in”) e saídas (“out”) do bloco. Os nomes que escolhi para os pinos são *in\_a*, *in\_b* e *a\_e\_b*. Os três são “std\_logic”, ou seja, são só um bit.

Finalmente vamos especificar a *implementação do bloco*, ou a arquitetura dele.

```
architecture a_porta of porta is
begin
    a_e_b <= in_a and in_b;
end architecture;
```

Batize a arquitetura simplesmente colocando um “a\_” na frente (“a\_porta”), pra ficar fácil. Entre o *begin* e o *end* está a circuitaria: uma única linha que define uma porta AND.

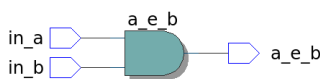
Abra então o terminal de linha de comando (o *shell*) e digite os comandos:

```
ghdl -a porta.vhd
ghdl -e porta
```

Caso haja mensagens, verifique erros de digitação ou cópia<sup>1</sup>. Note que o *-e* tem só o nome da entidade, sem o *.vhd*: é a entidade, não o nome do arquivo.

O primeiro comando (*-a*) executa uma análise do código fonte, procurando validá-lo como VHDL correto. O segundo comando (*-e*) faz a elaboração do circuito, construindo uma representação dele em um formato intermediário.

O circuito produzido é o visto abaixo<sup>2</sup>:



Bom, agora a gente vai simular.

## Arquivo de Simulação

A simulação é *não-interativa*: especificamos algumas entradas e observamos a resposta do circuito nas saídas.

Para especificar as entradas, precisamos criar um arquivo de testes, ou *testbench*. Crie um arquivo “porta\_tb.vhd”<sup>3</sup> no editor de textos e já inicie a digitação do código (aprenda devagar...).

```
library ieee;
use ieee.std_logic_1164.all;

-- a entidade tem o mesmo nome do arquivo
entity porta_tb is
end;
```

Como este arquivo apenas gera sinais, o bloco não possui nem entradas nem saídas. O hífen duplo “--” é o comentário.

Prosseguimos.

```
architecture a_porta_tb of porta_tb is
    component porta
        port( in_a : in std_logic;
              in_b : in std_logic;
              a_e_b: out std_logic -- lembre: sem ';' nessa linha
        );
    end component;
    signal in_a,in_b,a_e_b: std_logic;
---- (continua a seguir...)
```

<sup>1</sup> Se o *ghdl* não der nem sinal de vida, veja o documento de instruções de instalação no site da disciplina. Em especial, verifique se o *path* padrão do sistema está correto.

<sup>2</sup> Usando o *Quartus II* podemos visualizar o circuito produzido pela compilação com *Tools => Netlist Viewers => RTL Viewer* (RTL é *Register Transfer Level*)

<sup>3</sup> Esse padrão de usar “\_tb” como sufixo para o *testbench* de um módulo é bem comum e altamente recomendável. Use-o.

A seção “component” indica que vamos usar um componente pronto de outro arquivo. Precisamos indicar exatamente a mesma interface definida lá.

Adicionalmente, vamos precisar de sinais para fazer as ligações dos pinos do componente e suas atribuições de valores. Um **signal** VHDL é um ponto qualquer no circuito, um nó ou um fio, ao qual damos um nome.

Como padrão, criamos sinais com o *mesmo nome* dos pinos para facilitar, mas o nome pode ser diferente. Note que, embora os pinos possuam sentido (“in” ou “out”), os sinais não possuem.

```
---- (...continuação do anterior)
begin
  -- uut significa Unit Under Test
  uut: porta port map( in_a => in_a,
                      in_b => in_b,
                      a_e_b => a_e_b);
---- (continua a seguir...)
```

“uut” é o nome da instância do componente “porta.” A seguir, é exigido que se faça o mapeamento de *todos os pinos* do bloco para sinais criados neste arquivo. O formato é *pino1 => sinal1, pino2 => sinal2* etc.

Enfim inserimos os dados de simulação.

```
---- (...continuação do anterior)
process
begin
  in_a <= '0';
  in_b <= '0';
  wait for 50 ns;
  in_a <= '0';
  in_b <= '1';
  wait for 50 ns;
  in_a <= '1';
  in_b <= '0';
  wait for 50 ns;
  in_a <= '1';
  in_b <= '1';
  wait for 50 ns;
  wait;
end process;
end architecture;
```

A keyword “process” identifica uma seção sequencial de VHDL, com eventos.

As atribuições parecem simples: '0' e '1' são constantes booleanas atribuídas aos sinais. As aspas simples identificam valores do tipo “std\_logic”.

A construção “wait for \_\_ ns” não pode ser usada em um circuito normal; ela existe apenas para que possamos fazer *testbenches*. O “process” deve terminar com um “wait;” sem tempo.

Assim como para o outro arquivo fonte, precisamos fazer a análise e a elaboração do arquivo com:

```
ghdl -a porta_tb.vhd
ghdl -e porta_tb
```

Okay, agora sim a gente vai simular.

## Como Simular

Vamos fazer o *ghdl* rodar os processos de *testbench* através do seu comando *-r (run)*, que vai gerar formas de onda obtidas a partir dos sinais especificados no arquivo “porta\_tb.vhd” que é o

arquivo *top-level* do nosso projeto, ou seja, inclui todos os outros como dependentes. Digite o comando seguinte no terminal:

```
ghdl -r porta_tb --wave=porta_tb.ghw
```

Isso diz para executar a entidade “porta\_tb” (não é o nome do arquivo, portanto omite o “.vhd”). O comando vai produzir formas de onda de saída da simulação num formato adequado.

Para visualizar estas formas de onda, precisamos invocar o *gtkwave*, que é um visualizador genérico, usado tanto para ondas de circuitos VHDL como para outros propósitos. Execute-o assim:

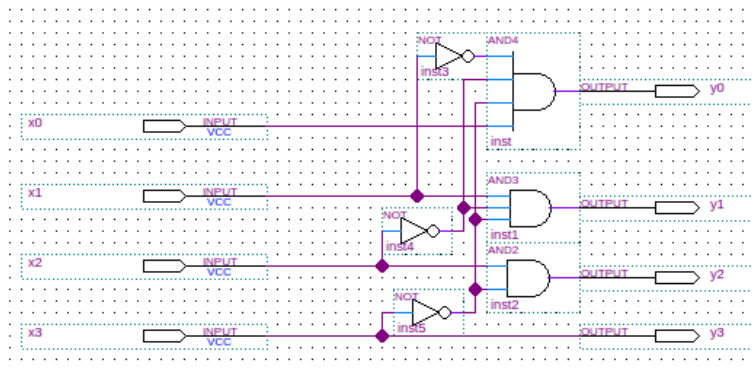
```
gtkwave porta_tb.ghw
```

As ondas não aparecem a princípio pois não foram selecionadas; para vê-las, você deve:

- Expandir a *treeview* que está no painel superior esquerdo (diz SST nele);
- Clicar na entidade *porta\_tb* na árvore;
- Selecionar todos os sinais que aparecem ali (são *a*, *b* e *e*) e clicar *Insert*;
- Ir ao menu *Time => Zoom* e escolher *Zoom Full*;
- As formas de onda devem aparecer no painel preto.

## Circuitos Maiores

Para fazermos um circuito como o abaixo, vamos precisar de várias ligações.



Os *x*'s são entradas e os *y*'s são saídas. Aliás, o que faz este circuito? Qual sinal tem maior prioridade, *x3* ou *x0*?

Um código VHDL possível para descrever o circuito é:

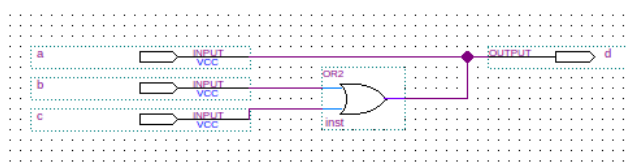
```
y3 <= x3;  
y2 <= x2 and not x3;  
y1 <= x1 and not x2 and not x3;  
y0 <= x0 and not x1 and not x2 and not x3;
```

Perceba que todas as linhas acima são “executadas” *em paralelo*! Isso ocorre porque esta é uma *descrição estática de um circuito*. Isto não é um programa.

Cada comando desta seção cria uma conexão física entre partes do circuito. Por isso, o seguinte trecho é bugado:

```
d <= a;  
d <= b or c;
```

O circuito equivalente é esse:



Ou seja, um curto-circuito indesejável, já que estamos descrevendo duas ligações para um mesmo ponto *d*.

### Drills (Exercícios de Fixação)

É necessário fazer, não é necessário entregar, não vale nota. O método vai ser útil em algum momento.

Um *decoder* é um circuito que seleciona apenas uma das saídas disponíveis. Portanto um *decoder* 2x4 possui dois bits de seleção (entrada) e vai manter apenas em uma das 4 saídas em nível 1 (“ligada”), que é aquela selecionada pelos bits de entrada, deixando as demais em nível 0 (“desligadas”).

►	Projete e simule um <i>decoder</i> 2x4. Faça a tabela verdade, extraia as expressões lógicas e daí sintetize o código VHDL. Teste. <i>Siga obrigatoriamente as recomendações abaixo.</i>
---	--

Então a ordem típica é:

1. Identifique a interface (entradas e saídas)
2. Especifique o componente (entidade)
3. Projete o comportamento *no papel*
4. Só então construa a arquitetura
5. Construa o *testbench*
6. Teste tudo e conserte o que precisar

Mas só para estes exercícios, para fixar bem, **construa o *testbench* antes de construir a arquitetura**. Isso forçará você a lembrar que a implementação é só um detalhe, um ajuste, depois de você compreender bem o comportamento do circuito.

Mais um:

►	Projete e simule um somador completo de 1 bit. Faça a tabela verdade e extraia uma expressão lógica e daí sintetize o código VHDL. Teste o circuito.
---	--

### Resposta Detalhada do Exercício 4 do Laboratório #3

No 8051, é o programador que decide se o número de 8 bits é sinalizado ou não. Dada a conta:  $11001110_2 + 11000111_2$  com resultado num registrador de 8 bits, quais são os valores decimais envolvidos? Houve estouro da capacidade de 8 bits ou não?

Chamemos o valor  $11001110_2$  de *x* e o valor  $11000111_2$  de *y*. Temos quatro possibilidades interpretativas, usando notação da linguagem C:

tipo	<i>x</i>	<i>y</i>
<i>signed de 8 bits</i>	-50	-57
<i>unsigned de 8 bits</i>	206	199

O resultado final da soma será  $10010101_2$ , independentemente de haver sinal ou não, pois a ULA faz as somas sem saber se o número é binário puro ou complemento de 2, já que a operação matemática é a mesma em ambos os casos. Este valor tanto pode ser 149, caso o resultado seja interpretado como *unsigned*, como pode ser -107, se for *signed*.

Perceba que o resultado está correto em todas as possibilidades, exceto quando tanto *x* quanto *y* são *unsigned*:

1.  $-50-57=-107$
2.  $-50+199=149$
3.  $206-57=149$
4.  $206+199=405$ , e  $405=256+149$  (o 256 é o *carry*)

Isso ocorre porque as contas de 8 bits são feitas em *módulo 256*.

Observe que não há *overflow* e há *carry* nesta conta  $11001110_2+11000111_2$ . Ademais, o *overflow* só vale para o caso 1, sinalizado, e o *carry* só deve ser usado no caso 4, não sinalizado. Os casos 2 e 3 são “perigosos” para o programador, misturando representações sinalizada e não sinalizada, mas dão o resultado certo.

Portanto só há estouro da capacidade de 8 bits se estivermos usando números não sinalizados; caso contrário, não há estouro.