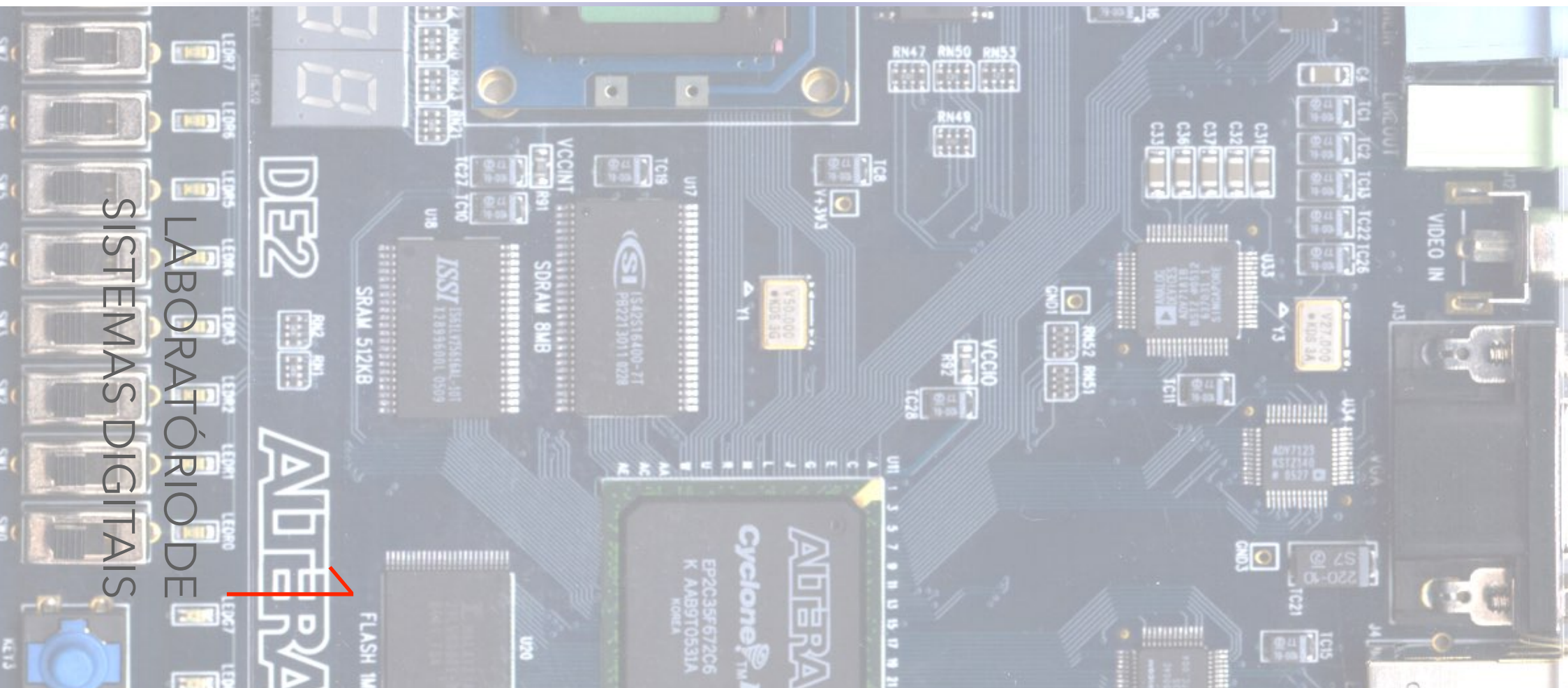
universidade de aveiro
theoria poiesis praxis

"Thus not only the mental and the material, but the theoretical and the practical in the mathematical world, are brought into more intimate and effective connection with each other."

Ada Lovelace



LABORATÓRIO DE SISTEMAS DIGITAIS

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Laboratório de Sistemas Digitais, tal como foi lecionada, no ano letivo de 2014/2015, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

mais informações em ruieduardofalopes.wix.com/apontamentos

Esta disciplina de Laboratório de Sistemas Digitais (a1s2) trabalha os conceitos fundamentais e metodologias estudadas na disciplina do semestre anterior, de Introdução aos Sistemas Digitais. Ao longo destes apontamentos, iremos detalhar processos usando técnicas e procedimentos usados hoje em dia, considerados como *state of the art*. Através da utilização de um kit programável em **VHDL** (linguagem de programação para descrição de hardware) - kit Terasic DE2-115 -, vamos projetar sistemas digitais para uma unidade de processamento FPGA.

VHDL

1. Introdução aos FPGA's

Ao longo desta unidade curricular vamos trabalhar com o equipamento Terasic DE2-115, este, que contém uma unidade FPGA como unidade de processamento. Mas afinal o que é um **FPGA**? Um FPGA, sigla inglesa de *Field Programmable Gate Array*, é um circuito integrado que é usado para suportar equipamentos como routers de alto--desempenho, tal como outros equipamentos de real processamento de elevado desempenho a respeito de imagem, som e restante multimédia ou sinais. O FPGA é assim um circuito que contém uma larga variedade de blocos lógicos integrados, tal como inúmeras entradas e saídas. Na Figura 1 podemos ver o aspeto simbólico de um FPGA.

FPGA

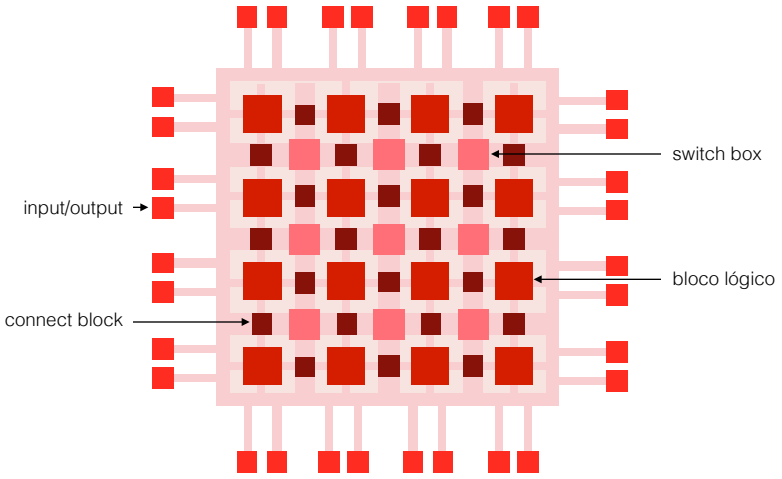


figura 1
FPGA simbólico

Cada um destes blocos lógicos é constituído pela lógica instituída pelo conceito de tabelas de verdade. Através de componentes denominados de **lookup tables** (geralmente designados pelo acrónimo **LUT**), os blocos lógicos que constituem os FPGA's contém também flip-flops e multiplexeres que designam a saída pretendida (quer saída do flip-flop sob forma sequencial, quer saída do próprio LUT). Na Figura 2 está assim representada a lógica interna de cada bloco lógico de um FPGA.

lookup tables
LUT

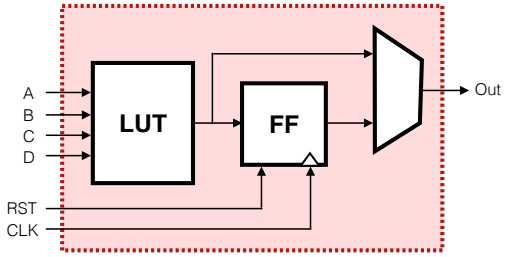


figura 2
bloco lógico de um FPGA

Na Figura 1 temos também, para além dos blocos lógicos, um outro componente importante a estudar ao longo desta disciplina - as **entradas/saídas**. Assim, os blocos de entrada/saída (em inglês *input/output*, muitas vezes abreviado de **I/O**). Na Figura 3 está representada a estrutura lógica interna de cada um destes blocos.

entradas/saídas

I/O

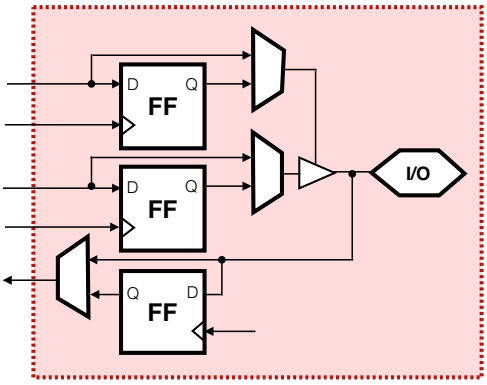


figura 3
bloco de entradas/saídas

Então, mas segundo a Figura 3, como é que o bloco pode processar uma entrada e uma saída? Esta pergunta é a resposta à presença de um buffer 3-state no bloco. Quando se pretende usar o bloco como uma entrada, o buffer irá desimpedir conflitos do componente I/O para o último flip-flop, permitindo assim que os dados fluam no sentido da direita para a esquerda, da Figura 3. Por outro lado, se pretendermos uma saída, então o buffer 3-state irá libertar o valor que reside na sua entrada, para o componente I/O.

Para implementar circuitos num FPGA tem-se que pensar como é que se pode implementar e comunicar os valores de uma tabela de verdade, com quatro entradas (A, B, C e D) para o processador. Temos assim que saber como é que podemos traduzir e desenhar os nossos circuitos sob a forma de LUT's. Vejamos um exemplo de um somador de dois bits com indicação do resultado igual a "zero", numa saída denominada de z (Figura 4).

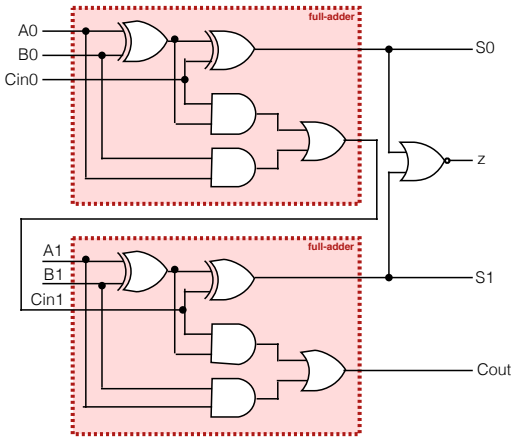


figura 4
somador de dois bits

Para implementarmos com LUT's, o nosso circuito, é importante segmentar o mesmo em várias partes, todas, com um número de entradas não dependentes de estados intermédios do circuito. Assim, dividimos o somador de 2-bits com 3 LUT's, sendo que na LUT2 destaca-se o facto de replicarmos a porta XOR, por esta, no desenho da Figura 4 pertencer ao LUT1. A Figura 5 mostra essa conclusão.

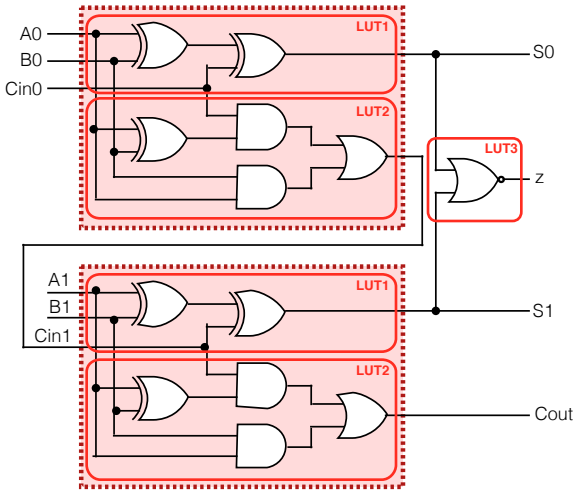


figura 5
implementação com LUT's

Se analisarmos as tabelas das LUT's (Figura 6) podemos verificar que as funções estão bem definidas e que o nosso conceito-base do circuito é cumprido.

LUT1			
A	B	Cin	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

LUT2			
A	B	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

LUT3		
S1	S0	S
0	0	0
0	1	1
1	0	1
1	1	0

figura 6
tabelas de verdade

Criação de um projeto com FPGA

A primeira ideia que é importante frisar aquando da criação de um projeto é que é necessário ter um objetivo (conceito) bem definido do produto que queremos criar. Sendo assim, e partindo desta ideia, é importante ter um esboço de uma função pretendida, perfeita, para estudar e tentar atingir ao longo do projeto. A esta fase podemos dar o nome de **design entry** (em português "entrada do desenho"). Este passo de execução deve ser baseado e suportado com linguagens de descrição de hardware como a linguagem que vamos usar nesta disciplina (VHDL), diagramas de estado e outros diagramas esquemáticos.

design entry

Após a primeira fase, o criador deve passar para uma síntese do projeto. Por uma questão de otimização e rendimento, deve-se prezar por um desenho que não desenvolva muitas redundâncias, o que obriga a que não haja repetições de gerações de sinal, entre outros, tal como vimos em Introdução aos Sistemas Digitais (a1s1). Dado isto, aqui é criada uma **netlist**. Uma netlist é uma lista de componentes de hardware e suas respetivas interconexões, essenciais para a fase seguinte - a de implementação. A esta fase chamamos então de **design synthesis**, que também resulta na criação de estimativas de desempenho do circuito e dos recursos lógicos necessários.

netlist

design synthesis

Tal como já foi referido, o passo seguinte é o de implementação no circuito. Denominada de **design implementation**, esta fase é assim dedicada ao mapeamento da netlist criada anteriormente nas primitivas específicas do nosso FPGA, completando também as interconexões entre elas. Como resultado desta fase é criado um ficheiro de configuração para o FPGA, tal como um relatório detalhado acerca dos recursos usados do FPGA, tempos de atraso, entre outras métricas.

design implementation

Sendo que todas as fases de desenho já foram completadas, só nos falta agora uma única fase - a fase de **device programming**. Nesta fase o ficheiro de configuração gerado anteriormente é transferido para o FPGA, sendo executado e programado. Esta transferência é realizada por via de um cabo de programação adequado (nesta disciplina usaremos um cabo USB). Como o FPGA é baseado, normalmente, em SRAM, tem uma configuração volátil, pelo que se perde aquando do encerramento da máquina. Uma solução a este problema é usar a memória interna FLASH, não-volátil (não será necessário usar no decorrer desta disciplina).

device programming

Cada uma destas fases deve ser intercalada com processos de verificação. Tudo isto conjugado deve seguir, de forma mais sistemática, a Figura 7.

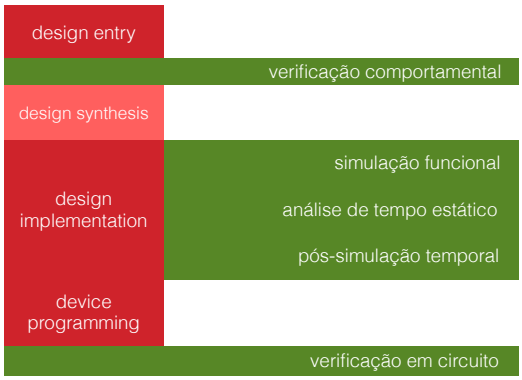


figura 7
projeto em VHDL

2. Linguagem VHDL

A linguagem código que vamos utilizar ao longo desta disciplina é a VHDL, acrónimo inglês de *Very High speed integrated circuits hardware Description Language*. Não sendo propriamente uma linguagem de programação, ela permite descrever o comportamento e a estrutura de hardware digital, utilizando recursos e construções típicas de linguagens de programação. Esta linguagem disponibiliza várias abstrações para as construções típicas do hardware, que por passos, se identificam por caraterizar patamares do processo mostrado na Figura 7. Começando pela criação da interface do módulo a criar em VHDL, existem **entidades** (em inglês *entities*) que são implementadas em **arquiteturas** definidas pelo programador. Estas arquiteturas tanto dependem de **sinais** internos do módulo, como de **portas** que este possa ter como interface com a sua envolvente física.

entidades
arquiteturas
sinais, portas

Um aspeto vital na linguagem VHDL é a **concorrência**, que modela o paralelismo do hardware, isto é, no momento de execução de uma simulação, os componentes (blocos ou módulos) criados serão avaliados temporalmente e por comportamento, de uma forma bastante uniforme e em simultâneo.

concorrência

A linguagem VHDL suporta diversos tipos de dados, sendo todos eles, claramente, orientados para o hardware. Divididos sob pacotes, os tipos de dados lógicos (alto ou baixo nível) estão preservados sob a forma **std_logic** (*standard logic*). Vejamos com mais detalhe os tipos de dados em VHDL na secção seguinte.

std_logic

Tipos de dados

Como referido anteriormente, em VHDL não deixam de existir diversos tipos de dados. Sendo orientados para o hardware, os tipos de dados estão organizados por pacotes, sendo o mais usado (mais útil para esta disciplina e para a manipulação de dados de hardware) o tipo *std_logic*. Este tipo compreende nove sinais, todos com 1 bit de tamanho e que designam o estado baixo com '0', o estado alto com '1' (da mesma forma que também existe o 'L' e o 'H', respetivamente), o estado de alta-impedância 'Z' ou até mesmo o estado don't care, com '-'. Os outros três sinais não são de grande importância para o âmbito desta disciplina, embora sejam o 'X' (conflito entre '0' e '1'), 'W' (conflito entre 'L' e 'H') e o 'U' (não inicializado).

Estes sinais podem ser usados como um só bit ou como um array de bits, aquando de uma conexão que suporta mais do que um bit, usando uma variante do pacote *std_logic*, denominada de **std_logic_vector**. A estas sequências de bits transportadas para uma dada conexão damos o nome de **barramentos**. Para usarmos estes barramentos designamos a palavra reservada do VHDL **downto**, delimitada por dois números, sendo o primeiro o bit mais elevado e o último o bit mais baixo. Por exemplo, se quisermos ligar os LED's 0 até 4, escrevemos `4 downto 0`.

std_logic_vector
barramentos
downto

Existem outros tipos de dados, como referido, que abordaremos mais à frente na disciplina, sendo eles o tipo com ou sem sinal (*signed* e *unsigned*, respetivamente), inteiro (*integer*), enumeração (*enumerated*), booleano (*boolean*), carater (*character*) ou ainda tempo (*time*).

Exemplo de implementação em VHDL

Experimentemos criar um projeto de um multiplexer 2:1 em VHDL. Relembrando Introdução aos Sistemas Digitais (a1s1), sabemos que um multiplexer é um componente com um poder de escolha entre dois sinais de entrada (chamemos-lhe de *input1* e *input0*), provocado por um sinal de seleção (aqui com o nome *sel*), que resulta na saída pretendida por *sel* (chamemos-lhe de *muxOut*).

Em VHDL precisamos sempre de importar bibliotecas, essenciais para a manipulação dos pacotes de tipos de dados que abordámos na secção §§Tipos de dados. Sendo assim, usando as palavras reservadas `library` e `use`, escrevemos o seguinte código (Código 1):

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

código 1
inclusão de bibliotecas

No Código 1, assim, temos uma indicação para a biblioteca IEEE, seguido da informação que necessitamos de utilizar o pacote STD_LOGIC_1164, por inteiro (`all`), onde estão os tipos de dados referidos inicialmente em §§Tipos de dados.

Continuando, agora temos de prosseguir para a criação de uma **entidade**. Como já referimos anteriormente, as entidades são uma forma de designar um objeto, conferindo-lhe a capacidade de possuir entradas e saídas, entre outros. Neste caso, chamando ao nosso multiplexer `Mux2_1`, podemos criar a seguinte entidade (Código 2):

entidade

```
entity Mux2_1 is
  port(sel       : in    std_logic;
       input0    : in    std_logic;
       input1    : in    std_logic;
       muxOut    : out   std_logic);
end Mux2_1;
```

código 2
criação de entidade

Tendo a entidade feita, fazendo corresponder nomes a entradas ou saídas que têm, neste caso, como tipo de dados, `std_logic`, avançamos para a descrição da arquitetura do nosso multiplexer 2:1. Tal como a entidade, a arquitetura também pode ter um nome que nós lhe atribuímos à escolha. Neste caso vamos dar-lhe o nome de `Equations`, por definirmos todas as equações de saída, incluindo sinais de saída internos da arquitetura, que definirão a escolha final para a saída do multiplexer. Esta implementação encontra-se visível no Código 3.

código 3
implementação

```
architecture Equations of Mux2_1 is
  signal s_and0Out, s_and1Out : std_logic;
begin
  s_and0Out <= not sel and input0;
  s_and1Out <=     sel and input1;
  muxOut    <= s_and0Out or s_and1Out;
end Equations;
```

No Código 3, para designar sinais internos da arquitetura usou-se a palavra reservada `signal` para os descrever. Mas há uma forma de dar a volta a esta implementação, como se pode verificar pelo Código 4.

código 4
diferente implementação

```
architecture Behavioral of Mux2_1 is
begin
  process(sel, input0, input1)
  begin
    if (sel = '0') then
      muxOut <= input0;
    else
      muxOut <= input1;
    end if;
  end process;
end Behavioral;
```

A palavra reservada usada no Código 4 `process` permite que haja uma simplificação da designação das portas e sinais internos, criando comportamentos, através do bloco *if...then...else*. Esta construção VHDL inclui uma lista, entre parênteses, denominada de **lista de sensibilidade** que basicamente significa uma lista de sinais/portos sobre os quais dependem processos, isto é, cujos eventos afetam os valores calculados pelo processo.

lista de sensibilidade

O multiplexer que realizámos apenas processa 1-bit de dados. Se quiséssemos aumentar o número de bits, de 1 bit para 8 bits, isto é, para 1 byte, apenas teríamos que re-configurar a entidade, para fornecer um vetor de dados lógicos, ao invés de apenas um bit de dados. Esta é uma das grandes vantagens da linguagem descritiva VHDL. Vejamos então como ficaria o código inteiro, para um multiplexer 2:1 de 8-bits (Código 5).

código 5
modelo completo em VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Mux2_1 is
  port(sel       : in     std_logic;
       input0    : in     std_logic_vector(7 downto 0);
       input1    : in     std_logic_vector(7 downto 0);
       muxOut    : out    std_logic_vector(7 downto 0));
end Mux2_1;

architecture Equations of Mux2_1 is
  signal s_and0Out, s_and1Out : std_logic;
begin
  s_and0Out <= not sel and input0;
  s_and1Out <=     sel and input1;
  muxOut    <= s_and0Out or s_and1Out;
end Equations;

architecture Behavioral of Mux2_1 is
begin
  process(sel, input0, input1)
```

```
  begin
    if (sel = '0') then
      muxOut <= input0;
    else
      muxOut <= input1;
    end if;
  end process;
end Behavioral;
```

Como podemos verificar pelo Código 5, podemos incluir mais que uma implementação no mesmo código, sendo que depois há a necessidade de indicar qual a arquitetura a usar para a execução.

Se pretendêssemos fazer um **comentário** no código, em VHDL, ele pode ser feito de duas formas - através de dois "hífenes", isto é, através dos carateres "--", ou através de um bloco iniciado com "/*" e terminado com "*/", tal como nas linguagens C e Java, por exemplo (Código 6).

comentário

código 6
comentários em VHDL

```
/* Um programa em VHDL
tem a seguinte estrutura:
*/
-- inclusão de bibliotecas:
library IEEE;
use IEEE.STD_LOGIC_1164.all;
-- criação de entidade:
entity NameOfEntity is
  -- blá, blá, blá...
end NameOfEntity;
-- criação de implementação sobre arquitetura:
architecture NameOfArchitecture of NameOfEntity is
begin
  -- blá, blá, blá...
end NameOfArchitecture;
```

Identificadores

A linguagem VHDL não é **case-sensitive**, pelo que não distingue letras maiúsculas de letras minúsculas. No entanto, esta linguagem é sensível a determinados aspetos que devem ser seguidos. Os **identificadores** servem para nomear diversos itens num modelo em VHDL. Podendo ter diversos comprimentos, só podem incluir carateres alfanuméricos ou *underscores*, isto é, letras de 'a' a 'z', 'A' a 'Z', números de '0' a '9' e '_'. Eles devem começar com uma letra e não devem conter o carater '_' tanto no fim, como no início, ou dois seguidos. Tal como noutras linguagens, também não pode usar palavras reservadas do VHDL, como `in` ou `component`.

case-sensitive

identificadores

nota!! Embora não sejam palavras reservadas da linguagem VHDL, as palavras "input" e "output" não devem ser usadas como identificadores, isto porque, no momento de simulação, há uma compilação que é executada sobre outra linguagem de descrição de hardware, o **Verilog**, que contém tais palavras como suas palavras reservadas, podendo criar um erro cuja fonte pode ser difícil de determinar.

nota

Verilog

3. Modelação de Componentes Combinatórios

Em Introdução aos Sistemas Digitais (a1s1) vimos que componentes combinatórios são circuitos cujas saídas apenas dependem do valor das entradas em cada momento. Lá, estudámos portas lógicas elementares, descodificadores, codificadores, multiplexeres, comparadores binários, somadores, entre outros...

Implementação de um descodificador 2:4

Dado que já estudámos também o básico a nível da implementação de circuitos em linguagem VHDL, vejamos como é que podemos implementar um descodificador 2:4, com uma entrada de ativação (a nível alto).

Revendo, um descodificador 2:4, com *enable* a nível alto, tem uma tabela de verdade e um circuito como os representados na Figura 8.

decoder 2:4 com EN						
EN	I1	I0	O3	O2	O1	O0
0	-	-	0	0	0	0
1	0	0	0	0	0	**1**
1	0	1	0	0	**1**	0
1	1	0	0	**1**	0	0
1	1	1	**1**	0	0	0

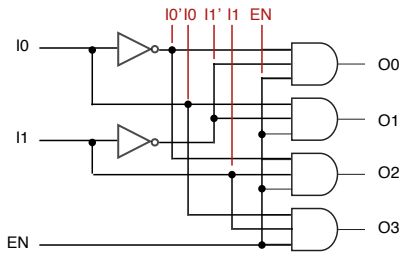


figura 8
descodificador 2:4 e respetiva tabela

Em VHDL, primeiro temos de criar a entidade, à qual, neste caso, vamos dar o nome de `Dec2_4`, o que respeita as regras abordadas em §§Identificadores (Código 7).

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Dec2_4 is
  port(enable    : in     std_logic;
       i         : in     std_logic_vector(1 downto 0);
       o         : out    std_logic_vector(3 downto 0);)
end Dec2_4;
```

código 7
entidade de decoder 2:4

Em termos de arquitetura, tal como já referimos, podemos ter várias abordagens. Uma forma possível, contudo mais trabalhosa, é implementar o nosso circuito de forma **estrutural**, isto é, instanciando e interligando as portas lógicas necessárias, criando todas as portas necessárias e agrupando-as de forma a terminar o projeto, como podemos ver, de forma abreviada, no Código 8.

estrutural

```
architecture Structural of Dec2_4 is
  signal s_nI0, s_nI1 : std_logic;
begin
  s_nI0 <= not i(0);
  s_nI1 <= not i(1);
  and_0 : entity work.AND3(ArchName)
            port(input0   => enable,
                 input1   => s_nI0,
                 input2   => s_nI1,
                 ooutput  => o(0));
-- mais código . . .
  and_3 : entity work.AND3(ArchName)
            port(input0   => enable,
                 input1   => i(0),
                 input2   => i(1),
                 output   => o(3));
end Structural;
```

código 8
arquitetura estrutural

Uma forma de simplificar o Código 8 é criando a concorrência referida atrás. Esta alteração provoca que vários *outputs* sejam executados em simultâneo (Código 9).

```
architecture BehavEquations of Dec2_4 is
begin
  o(0) <= enable and (not i(1)) and (not i(0));
  o(1) <= enable and (not i(1)) and (    i(0));
  o(2) <= enable and (    i(1)) and (not i(0));
  o(3) <= enable and (    i(1)) and (    i(0));
end BehavEquations;
```

código 9
arquitetura com atribuição concorrente (funções lógicas)

Usando atribuições de modo condicional, ainda podemos simplificar o Código 9 no Código 10, com a palavra reservada da linguagem `when`. Este tipo de atribuições só pode ser efetuado diretamente no corpo na implementação, isto é, na arquitetura, mas sempre fora de um corpo de processo.

código 10
arquitetura com atribuição concorrente condicional

```
architecture BehavAssign of Dec2_4 is
begin
  o <= "0000" when (enable = '0')  else
       "0001" when (i = "00")      else
       "0010" when (i = "01")      else
       "0100" when (i = "10")      else
       "1000";
end BehavAssign;
```

Com processos, a arquitetura também é passível de ser definida, sendo que deverá apresentar o seguinte aspeto (Código 11):

código 11
arquitetura com processos

```
architecture BehavProc of Dec2_4 is
begin
  process(enable, i)
  begin
    if (enable = '0') then
      o <= "0000";
    else
      if (i = "00") then
        o <= "0001";
      elsif (i = "01") then
        o <= "0010";
      elsif (i = "10") then
        o <= "0100";
      else
        o <= "1000";
      end if;
    end if;
  end process;
end BehavProc;
```

Se estivéssemos a trabalhar numa linguagem de programação como a Java, muito provavelmente uma solução imediata seria usando um bloco *switch…case*. Em VHDL também existe essa possibilidade, no corpo do processo, de utilizar *case…when*. E do mesmo modo que existe a palavra *default* em Java, em VHDL é `others` que assume o mesmo significado (Código 12).

código 12
arquitetura com bloco case…when

```
architecture BehavSwitch of Dec2_4 is
begin
  process(enable, i)
  begin
    if (enable = '0') then
      o <= "0000";
    else
      case i is
      when "00" =>
        o <= "0001";
      when "01" =>
        o <= "0010";
      when "10" =>
        o <= "0100";
      when others =>
        o <= "1000";
      end case;
    end if;
  end process;
end BehavSwitch;
```

Em suma, no corpo de uma arquitetura pode coexistir a instanciação e interligação de componentes, atribuições concorrentes (que são descritas por funções lógicas), atribuições concorrentes condicionais e processos. Dentro destes processos,

quando relativos a um componente combinatório, eles contêm listas de sensibilidade que devem incluir todos os sinais aos quais o processo se torna sensível (influenciando as saídas).

Circuitos aritméticos

Um exemplo de circuito combinatório que abordámos em Introdução aos Sistemas Digitais (a1s1) foi toda a classe de circuitos aritméticos. Nele temos, por exemplo, o **somador binário**. Um somador binário de 4-bits é um componente combinatório que recebe duas sequências de 4-bits como entrada e devolve uma só sequência de 4-bits como resultado. Versões mais trabalhadas deste somador incluem uma entrada para transportes de entrada e de saída (*carry in* e *carry out*). Sendo assim, na linguagem VHDL podemos definir a nossa entidade da seguinte forma (Código 13).

somador binário

```
entity Adder4 is
  port(input0    : in     std_logic_vector(3 downto 0);
       input1    : in     std_logic_vector(3 downto 0);
       result    : out    std_logic_vector(3 downto 0);)
end Adder4;
```

código 13
entidade de somador binário

Em termos de arquitetura, nós, mais uma vez, podemos ter muitas implementações, como interligar e instanciar as portas lógicas necessárias (num ponto de vista de uma arquitetura estrutural), escrever as equações lógicas de cada saída (explicitando a tabela de verdade) ou escrever a expressão aritmética da saída. Concentrando-nos nesta última, vejamos como é que podemos criar esta implementação em VHDL (no Código 14).

```
architecture Behavioral of Adder4 is
begin
  result <= input0 + input1; -- OPERAÇÃO ILEGAL EM VHDL
end Behavioral;
```

código 14
arquitetura de somador binário (ilegal)

No Código 14 temos uma operação que é considerada ilegal em VHDL porque o símbolo '+' não está definido como operador das duas quantidades `input0` e `input1`. Para considerar como válida, precisamos de importar o pacote **numeric_std**. Este pacote é que tem diretivas para definir operações entre valores com e sem sinal. Para distinguir estes últimos também é possível usar as palavras reservadas `signed` e `unsigned`. Assim, o nosso código anterior devia ser escrito como o Código 15.

numeric_std

```
architecture Behavioral of Adder4 is
begin
  result <= std_logic_vector(unsigned(input0) + unsigned(input1));
end Behavioral;
```

código 15
arquitetura de somador binário

Mas um somador não pode ter apenas esta constituição, de forma a poder funcionar corretamente. Deverão haver, por exemplo, mecanismos que alertem quando é que o somador entra numa situação de overflow. Sendo assim, é essencial que haja, no mínimo, uma saída de *carry out*. No Código 16[1] podemos ver uma implementação e uma entidade de um somador de 4-bits com saída de *carry out*.

Em Introdução aos Sistemas Digitais (a1s1), e mais à frente, em Arquitetura de Computadores I (a2s1), falamos de um componente essencial em termos de implementação de um processador: **unidade aritmética e lógica** (também abreviada de ALU). Uma unidade aritmética e lógica é um conjunto de componentes combinatórios

unidade aritmética e lógica

1 No Código 16 o operador '&' serve para indicar uma concatenação, pelo que '0' & "1001" é igual a "01001".

capazes de fornecer resultados de operações tanto aritméticas (somas, subtrações, multiplicações, divisões) como lógicas (conjunção, disjunção, diferença, ...). Os blocos ALU têm sempre três entradas, sendo estas os dois operandos e uma entrada de controlo, que serve para escolher a operação a realizar. Em termos análogos há uma semelhança direta com a implementação de um bloco *switch...case* da linguagem de programação Java.

código 16
somador de 4 bits com carry-out

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity Adder4 is
  port (input0    : in  std_logic_vector(3 downto 0);
        input1    : in  std_logic_vector(3 downto 0);
        result    : out std_logic_vector(3 downto 0);
        carryOut  : out std_logic);
end Adder4;

architecture Behavioral of Adder4 is
  signal s_input0, s_input1, s_result : unsigned(4 downto 0);
begin
  s_input0 <= '0' & unsigned(input0);
  s_input1 <= '0' & unsigned(input1);
  s_result <= s_input0 + s_input1;
  result   <= std_logic_vector(s_result(3 downto 0));
  carryOut <= std_logic(s_result(4));
end Behavioral;
```

Criemos então uma entidade respetiva a uma ALU de 4-bits com três entradas e duas saídas. As entradas serão duas para os operandos (`input0` e `input1`) e para a seleção de operação a realizar (`operation`) - não usar `select`, dado que é palavra reservada do VHDL! A respeito de saídas teremos a saída para o resultado (`result`) e uma saída para os bits "extra", necessários para a multiplicação, dado que a multiplicação de duas quantidades de 4-bits gera um resultado de 8-bits. A esta saída dar-lhe-emos o nome de `multHi`. Assim, a nossa entidade será a do Código 17.

código 17
entidade de ALU

```
entity ALU4 is
  port (operation : in  std_logic_vector(2 downto 0);
        input0    : in  std_logic_vector(3 downto 0);
        input1    : in  std_logic_vector(3 downto 0);
        result    : out std_logic_vector(3 downto 0);
        multHi    : out std_logic_vector(3 downto 0));
end ALU4;
```

Para prosseguirmos com a implementação, antes, temos de criar uma tabela na qual fazemos corresponder uma codificação a uma determinada operação. Sendo assim, vejamos a Figura 9[2].

figura 9
tabela de operações da ALU

operações	
código	operação em VHDL
000	+
001	-
010	*
011	/
100	rem
101	and
110	or
111	xor

2 A operação "rem" da Figura 9 é respeitante ao resto da divisão inteira, designada por '%' em linguagem Java e Python.

Em termos de implementação, agora já podemos prosseguir para o código, no qual criamos um sinal interno denominado de `s_multRes`, o qual preservará o resultado total da multiplicação, com 8-bits, do qual iremos ler os bits de 7 a 4 para copiar para `multHi` e os bits 3 a 0 para copiar para `result`, em caso de multiplicação. No Código 18 podemos assim ver uma possível solução para a implementação.

código 18
arquitetura de ALU

```
architecture Behavioral of ALU4 is
  signal s_multRes : std_logic_vector(7 downto 0);
begin
  s_multRes <= std_logic_vector(unsigned(input0) * unsigned(input1));
  process (operation, input0, input1)
  begin
    case operation is
      when “000” =>
        result <= std_logic_vector(unsigned(input0) + unsigned(input1));
      when “001” =>
        result <= std_logic_vector(unsigned(input0) - unsigned(input1));
      when “010” =>
        result <= s_multRes(3 downto 0);
      when “011” =>
        result <= std_logic_vector(unsigned(input0) / unsigned(input1));
      when “100” =>
        result <= std_logic_vector(unsigned(input0) rem unsigned(input1));
      when “101” =>
        result <= input0 and input1;
      when “110” =>
        result <= input0 or input1;
      when others =>
        result <= input0 xor input1;
    end case;
  end process;
  multHi <= s_multRes(7 downto 4) when (operation = “010”) else (others => ‘0’);
end Behavioral;
```

Se pretendermos transformar a nossa composição da ALU de forma a que processe quantidades com sinal, basta substituir `unsigned` por `signed`. Dado que tudo isto se processa em complemento para dois, as adições e as subtrações de quantidades com e sem sinal são iguais em termos de adições e subtrações, mas o mesmo não acontece para as multiplicações e divisões. Ficamos assim com o Código 19.

código 19
arquitetura de ALU permitindo quantidades com sinal

```
architecture Behavioral of ALU4 is
  signal s_multRes : std_logic_vector(7 downto 0);
begin
  s_multRes <= std_logic_vector(signed(input0) * signed(input1));
  process (operation, input0, input1)
  begin
    case operation is
      when “000” =>
        result <= std_logic_vector(signed(input0) + signed(input1));
      when “001” =>
        result <= std_logic_vector(signed(input0) - signed(input1));
      when “010” =>
        result <= s_multRes(3 downto 0);
      when “011” =>
        result <= std_logic_vector(signed(input0) / signed(input1));
      when “100” =>
        result <= std_logic_vector(signed(input0) rem signed(input1));
      when “101” =>
        result <= input0 and input1;
      when “110” =>
        result <= input0 or input1;
      when others =>
        result <= input0 xor input1;
    end case;
  end process;
  multHi <= s_multRes(7 downto 4) when (operation = “010”) else (others => ‘0’);
end Behavioral;
```

Componentes parametrizáveis em VHDL

Se pretendermos alterar o tamanho das sequências de entrada e saída de um determinado componente precisamos de alterar toda a composição do ficheiro VHDL?

Consideremos um somador ao jeito do do Código 16 e imaginemos que agora pretendíamos um somador, não de 4-bits, mas de 16-bits. Será que precisávamos de alterar todas as instâncias `(3 downto …)` do código para `(15 downto …)`? E se num ficheiro `top-level` precisássemos de um somador de 4-bits, dois somadores de 16-bits, um somador de 8-bits e outro de 32-bits. Precisaríamos de 4 ficheiros distintos, para 4 entidades distintas? Não. Estudamos assim uma nova abordagem - componentes **parametrizáveis**. É possível generalizar os componentes criados em VHDL. Vejamos o seguinte código para um somador de n-bits (Código 20).

parametrizáveis

código 20
ALU genérica

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity AdderN is
  generic (N      : positive := 4);
  port (input0    : in  std_logic_vector(N - 1 downto 0);
        input1    : in  std_logic_vector(N - 1 downto 0);
        result    : out std_logic_vector(N - 1 downto 0);
        carryOut  : out std_logic);
end AdderN;

architecture Behavioral of AdderN is
  signal s_input0, s_input1, s_result : unsigned(N downto 0);
begin
  s_input0 <= '0' & unsigned(input0);
  s_input1 <= '0' & unsigned(input1);
  s_result <= s_input0 + s_input1;
  result   <= std_logic_vector(s_result(N - 1 downto 0));
  carryOut <= std_logic(s_result(N));
end Behavioral;
```

No Código 20 podemos assim verificar que não estão estipuladas as instâncias completas dos vectores designados, dado que apenas sabemos que estes se concretizam com um tamanho `N`. A designação de `N` é dada de duas formas: no ficheiro `top_level`, através da atribuição de um valor a `N`, com as palavras reservadas `generic map ()`, ou através da palavra reservada `generic`, no ficheiro atual, designando assim um valor por omissão (por definição), caso nenhum valor seja atribuído. No caso do Código 20, o valor por definição é 4, pelo que se instanciarmos um somador destes, caso não atribuirmos nenhum valor a `N`, este será 4. No Código 21 está um excerto-exemplo de instanciação num ficheiro `top_level`.

código 21
instanciação de componentes parametrizáveis

```
. . .
  adder_1: entity work.AdderN(Behavioral)
             generic map (N      => 4) -- não é necessário - valor por omissão
             port map (input0    => A,
                       input1    => B,
                       result    => E(3 downto 0),
                       carryOut  => E(4));

  adder_2: entity work.AdderN(Behavioral)
             generic map (N      => 4)
             port map (input0    => C,
                       input1    => D,
                       result    => F(3 downto 0),
                       carryOut  => F(4));

  adder_1: entity work.AdderN(Behavioral)
             generic map (N      => 5)
             port map (input0    => E,
                       input1    => F,
                       result    => G(3 downto 0),
                       carryOut  => G(4));
. . .
```

4. Modelação de Circuitos Sequenciais

Um dos tipos mais importantes de circuitos lógicos são os circuitos sequenciais.

Alguns dos circuitos sequenciais que abordámos em Introdução aos Sistemas Digitais (a1s1) foram as latches D, flip-flops do tipo D, registos ou até mesmo contadores. Entre muitos outros, estes são elementares no conceito de sequencialidade, o qual preserva a noção de tempo como o rigor da escala dos seus componentes.

Modelação de latches, flip-flops e registos

Revendo um pouco acerca destes componentes, iniciemos assim o nosso estudo através da implementação de uma **latch D** na linguagem de descrição VHDL. No Código 22 temos uma possível implementação.

latch D

código 22
latch D

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity LatchD is
  port (enable   : in     std_logic;
        dataIn   : in     std_logic;
        dataOut  : out    std_logic);
end LatchD;

architecture Behavioral of LatchD is
begin
  process(enable, dataIn)
  begin
    if (enable = '1') then
      dataOut <= dataIn;
    end if;
  end process;
end Behavioral;
```

Contrariamente ao que tenhamos estudado anteriormente, se repararmos bem, desta vez o Código 22 não cobre todos os casos possíveis para o valor de `enable` e de `dataIn`, pelo que implicitamente, a linguagem VHDL criará uma latch D para englobar tais possibilidades.

Numa latch D, como já devemos saber, a saída copia a entrada quando o valor da entrada de controlo `enable` tiver o valor de '1' lógico. Uma variação deste componente poder-se-á traduzir num outro componente denominado de **flip-flop D**. No Código 23 podemos encontrar uma possível implementação de um destes componentes.

flip-flop D

código 23
flip-flop D

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity FlipFlopD is
  port (clock    : in     std_logic;
        dataIn   : in     std_logic;
        dataOut  : out    std_logic);
end FlipFlopD;

architecture Behavioral of FlipFlopD is
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      dataOut <= dataIn;
    end if;
  end process;
end Behavioral;
```

No Código 23, e comparando com o Código 22, podemos verificar as variadas diferenças entre ambos os circuitos. No caso do flip-flop D passamos a transferir o valor de `dataIn` para `dataOut` apenas quando é verdadeira uma determinada condição traduzida pelo método `rising_edge(clock)`. Este método é, na verdade, uma abstração permitida pela linguagem VHDL para traduzir o comportamento **edge-triggered** dos flip-flops, isto é, traduzir o comportamento de variância de valor quando há uma transição de valores lógicos, neste caso, do valor de `clock` (argumento do método) - no

edge-triggered

sentido ascendente. Uma segunda forma de escrever a condição para a mudança de valor é `if (clock'event and clock = '1') then`, o que procura um atributo de `clock`.

Podemos também, como conclusão à nossa análise deste componente, verificar que no flip-flop do tipo D apenas se inclui o sinal de `clock` na lista de sensibilidade dos processos, sendo que este sinal é a única entrada capaz de provocar uma modificação nas saídas.

Vamos agora considerar a situação em que nos interessa adicionar uma entrada ao nosso módulo tal que seja possível restaurar os valores iniciais num determinado instante. Designando essa entrada por `reset`, esta pode ser definida de duas formas distintas no nosso código. No Código 24 podemos ver uma forma de aplicar a entrada de `reset` e uma `enable` que liga ou desliga o módulo.

código 24
flip-flop D com reset assíncrono

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity FlipFlopDReset is
  port (clock    : in    std_logic;
        dataIn   : in    std_logic;
        reset    : in    std_logic;
        enable   : in    std_logic;
        dataOut  : out   std_logic);
end FlipFlopDReset;

architecture Behavioral of FlipFlopDReset is
begin
  process(clock, reset)
  begin
    if (reset = '1') then
      dataOut <= '0';
    elsif (rising_edge(clock)) then
      if (enable = '1') then
        dataOut <= dataIn;
      end if;
    end if;
  end process;
end Behavioral;
```

Se o nosso código coincidir com o Código 24, para a criação de um flip-flop D com reset, este será **assíncrono**, dado que a sua avaliação é feita independentemente do estado do sinal de relógio. Pelo contrário, podemos aplicar uma entrada `reset` **síncrona** como explícito pelo Código 25.

assíncrono
síncrona

código 25
flip-flop D com reset síncrono

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity FlipFlopDResetEnable is
  port (clock    : in    std_logic;
        dataIn   : in    std_logic;
        reset    : in    std_logic;
        enable   : in    std_logic;
        dataOut  : out   std_logic);
end FlipFlopDResetEnable;

architecture Behavioral of FlipFlopDResetEnable is
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        dataOut <= '0';
      elsif (enable = '1') then
        dataOut <= dataIn;
      end if;
    end if;
  end process;
end Behavioral;
```

Estamos agora mais que aptos para prosseguir para a compreensão da construção de um registo. Um **registo** é um componente no qual preservamos uma

registo

sequência de bits de dados. Tomemos, como exemplo inicial, um registo simples, de 8 bits. Este componente é análogo a um flip-flop, com a ligeira diferença de receber 8 bits de dados e, da mesma forma, enviar 8 bits pela saída de dados `dataOut`. No Código 26 podemos ver um exemplo de aplicação.

código 26
registo de 8 bits com reset e enable

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Register8ResetEnable is
  port (clock    : in     std_logic;
        dataIn   : in     std_logic_vector(7 downto 0);
        reset    : in     std_logic;
        enable   : in     std_logic;
        dataOut  : out    std_logic_vector(7 downto 0));
end Register8ResetEnable;

architecture Behavioral of Register8ResetEnable is
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
         dataOut <= (others => '0');
      elsif (enable = '1') then
        dataOut <= dataIn;
      end if;
    end if;
  end process;
end Behavioral;
```

No Código 26 temos assim um registo com entrada `reset` síncrona, dado que a possibilidade de reset ativo depende da transição de ciclo de relógio atual. Também neste código usamos, para fazer o restauro dos valores, a construção `(others => '0')`, o que significa que todos os bits da variável em questão (neste caso é `dataOut`) terão o valor `'0'`.

Tal como já foi referido anteriormente, faz todo o sentido parametrizar, isto é, tornar a implementação o mais global possível, o código do nosso registo. Sendo assim, eis o Código 27.

código 27
registo de n bits com reset e enable

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity RegisterNResetEnable is
  generic (Nbits : positive := 8);
  port (clock    : in     std_logic;
        dataIn   : in     std_logic_vector((Nbits - 1) downto 0);
        reset    : in     std_logic;
        enable   : in     std_logic;
        dataOut  : out    std_logic_vector((Nbits - 1) downto 0));
end RegisterNResetEnable;

architecture Behavioral of RegisterNResetEnable is
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
         dataOut <= (others => '0');
      elsif (enable = '1') then
        dataOut <= dataIn;
      end if;
    end if;
  end process;
end Behavioral;
```

Contadores binários

Tendo já consolidado como implementar circuitos básicos relativos a memórias *bitwise* (flip-flops, latches, ...) podemos agora avançar com circuitos que, baseados em circuitos sequenciais, efetuam operações aritméticas constantes exibindo o seu

resultado por via de uma saída de n bits. A este tipo de circuitos designamos de **contadores**, os quais são caraterizados por efetuarem um incremento constante ao longo do tempo, permitindo ser utilizados como auxílio ao controlo de uma outra operação a ser exercida por uma determinada máquina em construção.

contadores

Criemos assim um contador binário crescente de 8 bits com uma entrada de `enable` e uma entrada de `reset`, com a função de ligar/desligar a contagem e restaurar o valor da contagem para um inicial, designado em código, respetivamente. No Código 28 pode ver-se um exemplo de aplicação possível.

código 28
contador de 8 bits com reset e enable

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity BinaryCounter8ResetEnable is
  port (clock    : in    std_logic;
        reset    : in    std_logic;
        enable   : in    std_logic;
        count    : out   std_logic_vector(7 downto 0));
end BinaryCounter8ResetEnable;

architecture Behavioral of BinaryCounter8ResetEnable is
  signal s_countValue    : unsigned(7 downto 0);
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        s_countValue <= (others => '0');
      elsif (enable = '1') then
        s_countValue <= s_countValue + 1;
      end if;
    end if;
  end process;
  count <= std_logic_vector(s_countValue);
end Behavioral;
```

No Código 28 podemos verificar que foi utilizada a palavra reservada `signal` para criar um sinal, do tipo `unsigned`, de 8 bits. Mas porque é que é necessário fazer este passo? Acontece que um contador não é nada mais do que um somador sequencial de uma quantidade com, neste caso, a quantidade 1. Em VHDL não é possível declarar uma entrada `inNumber` e uma saída `outNumber` e ligar a saída à entrada diretamente, da seguinte forma: `inNumber <= outNumber`. Isto acontece porque não se pode escrever dados nas portas designadas na entidade como entradas. Sendo assim, precisamos de um local onde possamos guardar o valor atual, neste caso de 8 bits, resultante da operação a realizar pelo componente. Cria-se assim um sinal, que se declara como um mero sinal (sem que se especifique `in` ou `out`, dado que não faz sentido), com o qual iremos efetuar todas as nossas operações e, concorrentemente, ligar o sinal, por via de uma conversão para o tipo `std_logic_vector`, de 8 bits, para a saída da entidade `count`.

Pegando no circuito elaborado no Código 28 podemos transitar para um seu critério mais genérico que é a possibilidade de escolha do sentido da contagem, isto é, enquanto que o contador do Código 28 está programado para efetuar a contagem bit-a-bit de forma crescente, podemos agora criar um bit de entrada que especifique se nós pretendemos executar uma contagem crescente (colocando o valor `'1'`) ou uma contagem decrescente (colocando o valor `'0'`). No Código 29 é então possível verificar a construção de um contador binário de 8 bits com possibilidade de escolha relativamente ao sentido de contagem.

Uma última possibilidade é permitir que haja uma entrada que forneça ao contador um valor inicial para o qual ele deve incrementar uma quantidade constante num sentido definido também pela lógica cliente. Criamos assim um contador binário com entrada de carregamento paralelo e sem possibilidade de escolha entre o sentido pretendido para a contagem (é sempre crescente).

código 29
contador de 8 bits com reset e enable e upDown

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity BinaryCounter8ResetEnableUpDown is
  port (clock    : in    std_logic;
        reset    : in    std_logic;
        enable   : in    std_logic;
        upDown_n : in    std_logic;
        count    : out   std_logic_vector(7 downto 0));
end BinaryCounter8ResetEnableUpDown;

architecture Behavioral of BinaryCounter8ResetEnableUpDown is
  signal s_countValue    : unsigned(7 downto 0);
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        s_countValue <= (others => '0');
      elsif (enable = '1') then
        if (upDown_n = '1') then
          s_countValue <= s_countValue + 1;
        else
          s_countValue <= s_countValue - 1;
        end if;
      end if;
    end if;
  end process;
  count <= std_logic_vector(s_countValue);
end Behavioral;
```

No Código 30 podemos a aplicação do contador binário de carregamento paralelo.

código 30
contador de 8 bits com reset e enable e upDown com carregamento paralelo

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity BinaryCounter8ResetEnableUpDownLoad is
  port (clock    : in    std_logic;
        reset    : in    std_logic;
        enable   : in    std_logic;
        loadEn   : in    std_logic;
        upDown_n : in    std_logic;
        count    : out   std_logic_vector(7 downto 0));
end BinaryCounter8ResetEnableUpDownLoad;

architecture Behavioral of BinaryCounter8ResetEnableUpDownLoad is
  signal s_countValue    : unsigned(7 downto 0);
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        s_countValue <= (others => '0');
      elsif (enable = '1') then
        if (loadEn = '1') then
          s_countValue <= unsigned(dataIn);
        else
          s_countValue <= s_countValue + 1;
        end if;
      end if;
    end if;
  end process;
  count <= std_logic_vector(s_countValue);
end Behavioral;
```

Divisores de frequência

O sinal de relógio global das placas de desenvolvimento que usamos nas aulas práticas tem uma frequência de 50 MHz. Dizer que um determinado sinal tem uma frequência de 50 MHz é equivalente a dizer que o sinal muda de valor 50000000 vezes por segundo.

A capacidade de reação de um ser humano é aproximadamente de 1 segundo, pelo que precisarmos de avaliar as transições de um dado sinal, estando este regido ao sinal global de relógio, denominado, no nosso caso, de `CLOCK_50`, é impossível perceber as diferenças entre possíveis estados (se o sinal estiver ligado a um LED será impossível verificar se o mesmo se encontra a piscar ou não, embora esteja, a 50000000 vezes por segundo). Para podermos efetuar tal avaliação podemos atrasar este sinal, considerando uma transição válida de relógio por n transições reais do relógio. De forma a poder tal resultado, necessitamos de criar um componente novo denominado de **divisor de frequência**.

divisor de frequência

Através de contadores binários podemos criar um novo sinal de relógio (válido para o nosso projeto), tal que este divida a nossa frequência natural de 50 MHz em fatores inteiros de potências de base 2 (2^n). Vejamos assim uma possível implementação de divisor de frequência, no Código 31.

código 31
divisor de frequência parametrizável

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity FrequencyDivider is
  generic (K     : positive := 4);
  port (clockIn  : in     std_logic;
        clockOut : out    std_logic);
end FrequencyDivider;

architecture Behavioral of FrequencyDivider is
  signal s_counter       : natural;
begin
  process(clockIn)
  begin
    if (rising_edge(clockIn)) then
      if (s_counter = K - 1) then
        clockOut  <= '0';
        s_counter <=  0 ;
      else
        if (s_counter = K/2 - 1) then
          clockOut  <= '1';
        end if;
        s_counter <= s_counter + 1;
      end if;
    end if;
  end process;
end Behavioral;
```

No contador do Código 31 temos um componente que perfaz uma contagem *free running* com módulo K, isto é, cujos valores rondam entre 0 e (K - 1). A meio de cada contagem a saída, que se traduz pelo nome `clockOut` e quer significar o nosso novo sinal de relógio (válido e mais lento), fica o valor `'1'`, enquanto que no final da contagem volta a `'0'`. Na Figura 10 temos a simulação do divisor de frequência ilustrando precisamente a transição a meia contagem (para $K = 4$).

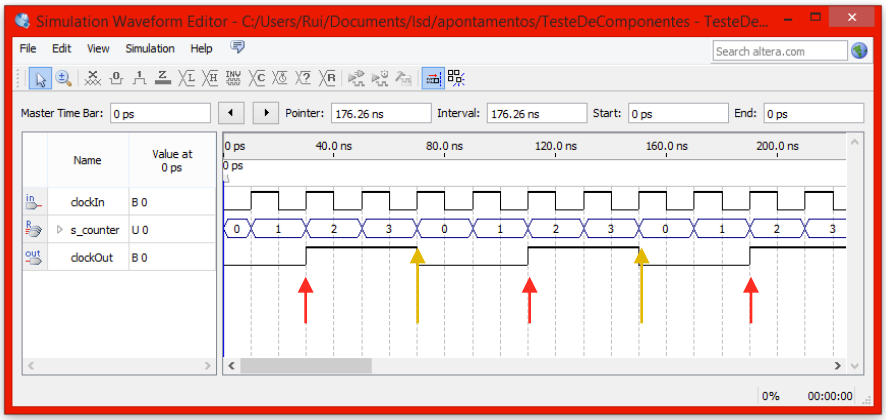


figura 10
simulação de divisor de frequência para $K = 4$

Na Figura 10 estão umas setas que indicam (a vermelho) os momentos, a meia-contagem, em que há uma transição de baixo para alto, enquanto que há setas (a amarelo) que indicam os momentos em que finda a contagem e há uma transição de alto para baixo.

Estamos agora em plenas condições de discutir um novo conceito, o qual pretende definir, em termos percentuais, a quantidade de tempo em que o nosso novo sinal de relógio se encontra em funções na totalidade da contagem. Define-se assim o **duty-cycle** (tempo de serviço).

duty-cycle

Para um $K = 4$ já vimos o efeito da divisão de frequência, mas se agora o nosso K for igual a 5, será que o próprio *duty-cycle* difere da situação anterior? Na Figura 11 encontra-se a simulação do mesmo componente para um $K = 5$.

figura 11
simulação de divisor de frequência para $K = 5$

Em termos de *duty-cycle* podemos reparar então que enquanto que numa contagem completa o relógio da Figura 10 está metade do tempo com o valor `'0'` e a outra metade com o valor `'1'`, no relógio da Figura 11 está menos tempo com o valor `'0'` do que com o valor `'1'`. Isto tem uma consequência em termos percentuais, pelo que podemos dizer que o *duty-cycle* no primeiro exemplo é de 50%, enquanto que no último é maior que 50%.

5. Modelação de Registos de Deslocamento

De forma a podermos prosseguir no nosso estudo em sistemas digitais há que, a partir de agora, tomar partido de todos os componentes que desenhámos anteriormente de forma a conseguirmos compor novos componentes, maiores e mais complexos.

Registos de deslocamento

Um dos novos componentes que podemos desenhar, com maior complexidade, é o **registo de deslocamento**. Um registo de deslocamento não é mais do que flip-flop's ligados em cascata, nos quais podemos efetuar operações de **shift** (deslocamento), para a esquerda ou para a direita (quer lógico ou aritmético). Estes deslocamentos podem ser 3 diferentes tipos: *shift left logical*, *shift right logical* ou *shift right arithmetic*. Começando pelo primeiro, o **shift left logical** (em português deslocamento lógico para a esquerda) é um movimento de um bit para esquerda que é exercido em todos os bits de uma determinada sequência - assim, se tivermos uma sequência 1011 e pretendermos fazer um *shift left logical* de 1 bit obtemos a seguinte sequência: 0110. De forma análoga, o **shift right logical** (em português deslocamento lógico para a direita) é um

registo de deslocamento
shift
shift left logical
shift right logical

movimento de um bit para a direita que é exercido em todos os bits de uma determinada sequência - assim, se tivermos uma sequência 1011 e pretendermos fazer um *shift right logical* de 1 bit obtemos a seguinte sequência 0101. Por último, um **shift right arithmetic** (em português deslocamento aritmético à direita) é o mesmo deslocamento que é feito no caso do *shift right logical* com a diferença que agora a nossa sequência de bits é interpretada como uma quantidade numérica binária com sinal, pelo que o movimento à direita terá de manter o sinal original - assim, se tivermos uma sequência 1011 e pretendermos fazer um *shift right arithmetic* de 1 bit obtemos a seguinte quantidade 1101.

shift right arithmetic

Mais de perto, os deslocamentos são operações que nos permitem simplificar duas operações aritméticas de custo muito elevado na implementação de circuitos: multiplicação e divisão. Quando fazemos um deslocamento para a direita de n bits é equivalente a efetuarmos uma **divisão** por 2^n. Por outro lado, quando fazemos um deslocamento para a esquerda de n bits é equivalente a efetuarmos uma **multiplicação** por 2^n.

divisão

multiplicação

Algumas aplicações típicas são a conversão de dados de paralelo para série (ou vice-versa) em sistemas computacionais de/para as interfaces Ethernet, SATA, PCIe, entre outros, ou até mesmo em algoritmos de deteção e correção de erros em sistemas de comunicação.

Em termos de circuito, na Figura 12 podemos ver um registo de deslocamento (para a direita).

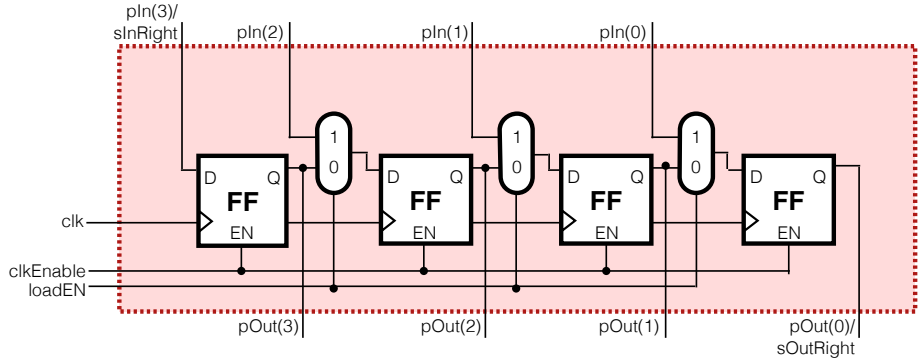


figura 12
registo de deslocamento

O registo de deslocamento da Figura 12 permite mover os bits para a direita e fazer tanto o carregamento de forma paralela como em série. Para fazer o carregamento paralelo a entrada `loadEN` deve encontrar-se com o valor `'1'`, de forma a que esta ative a entrada `1` dos multiplexers, deixando passar as entradas `pIn(3..0)`. Estas, por sua vez, são redirecionadas para cada entrada de cada flip-flop. Por outro lado, se a lógica cliente pretender inserir um valor no registo através da entrada série deve colocar a entrada `loadEN` com o valor contrário e inserir o valor pretendido via entrada `sInRight`. Note-se que para que o circuito possa funcionar a entrada `clkEnable` deverá estar constantemente com o valor `'1'`.

Em termos de código VHDL podemos modelar um componente destes através da instanciação constante de vários componentes flip-flop desenhados anteriormente. No entanto, por motivos educacionais, será mais importante conseguir designar um componente semelhante ao desenhado em circuito na Figura 12, mas com a capacidade de permitir à lógica cliente escolher qual o sentido do deslocamento lógico. Vamos considerar também que, aquando do deslocamento, o novo bit inserido é resultante da entrada série à direita (para deslocamentos para a esquerda) ou da entrada série à esquerda (para deslocamentos para a direita). Assim, no Código 32 podemos encontrar

um registo de deslocamento bidirecional com entradas em série e carregamento em paralelo (por total).

código 32
registo de deslocamento bidirecional

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity BidirectionalRegisterSP is
  port (clock    : in     std_logic;
        loadEn   : in     std_logic;
        sInLeft  : in     std_logic;
        sInRight : in     std_logic;
        dirLeft  : in     std_logic;
        dataIn   : in     std_logic_vector(7 downto 0);
        dataOut  : out    std_logic_vector(7 downto 0));
end BidirectionalRegisterSP;

architecture Behavioral of BidirectionalRegisterSP is
  signal s_shiftReg      : std_logic_vector(7 downto 0);
begin
  process(clock)
  begin
    if (rising_edge(clock)) then
      if (loadEn = '1') then
        s_shiftReg <= dataIn;
      elsif (dirLeft = '1') then
        s_shiftReg <= s_shiftReg(6 downto 0) & sInLeft;
      else
        s_shiftReg <= sInRight & s_shiftReg(7 downto 1);
      end if;
    end if;
  end process;
  dataOut <= s_shiftReg;
end Behavioral;
```

Registo de deslocamento combinatório (barrel shifter)

Os registos de deslocamento que estudamos anteriormente são baseados, especificamente, em circuitos sequenciais. Mas dado que nós não pretendemos guardar valores, mas apenas efetuar deslocamentos, será que é possível criar uma versão de um registo de deslocamento em termos combinatórios? Sim. Para tal serão necessários muitos multiplexers em cascata, numa reprodução a que damos o nome de **barrel shifter**.

barrel shifter

Dada a extensão da representação não vamos verificar qual é o seu esquema, mas em código VHDL podemos realizar o seu desenho, sendo que através do software usado nas aulas práticas (Altera Quartus II) é possível, através da opção *Netlist Viewer*, ver o diagrama de blocos do Código 33.

código 33
barrel shifter

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity BarrelShifter is
  port (dirLeft  : in     std_logic;
        shAmount : in     std_logic_vector(2 downto 0);
        dataIn   : in     std_logic_vector(7 downto 0);
        dataOut  : out    std_logic_vector(7 downto 0));
end BarrelShifter;

architecture Behavioral of BarrelShifter is
  signal s_shAmount               : integer;
begin
  process(dataIn, dirLeft, shAmount)
  begin
    if (dirLeft = '1') then
      dataOut <= std_logic_vector(shift_left(unsigned(dataIn), s_shAmount));
    else
      dataOut <= std_logic_vector(shift_right(unsigned(dataIn), s_shAmount));
    end if;
  end process;
  s_shAmount <= to_integer(unsigned(shAmount));
end Behavioral;
```

Estrutura típica de processos combinatórios e sequenciais

Os processos de circuitos combinatórios e de circuitos sequenciais seguem umas normas que são essenciais para uma boa síntese por parte das linguagens de descrição como o VHDL ou o Verilog.

Em termos de modelação de componentes combinatórios, os processos deverão ter, na sua lista de sensibilidade todas as entradas do processo, enquanto que dentro do próprio processo as saídas devem ser especificadas para todas as combinações de entradas. Caso não sejam cobertas todas as combinações possíveis das entradas do processo, no processo de síntese vão ser inferidas latches, para guardar o último valor válido no processo.

Já no caso dos processos de circuitos sequenciais os processos devem ter, na sua lista de sensibilidade, os sinais de relógio e *set* ou *reset* assíncronos. Por conseguinte, dentro do processo deverá ser feito, primeiramente, um teste para os sinais assíncronos, com as respetivas atribuições, e só depois o teste do flanco ascendente/descendente (dependendo de qual se considera ativo) do sinal de relógio, dentro com os testes para os sinais síncronos e respetivas atribuições.

6. Simulação de Componentes em VHDL

De forma a validar um sistema modelado em VHDL desde as fases iniciais até à sua implementação, tanto de forma económica como muito controlável, há que iniciar processos de simulação dos vários componentes do nosso projeto.

Simulação baseada em testbenches

Uma forma de validarmos os nossos componentes é através das ferramentas de simulação que usam **testbenches** como unidade comum de avaliação. Um testbench é um ficheiro, neste caso (em VHDL), no qual se estabelecem parâmetros temporais e físicos a serem implícitos numa unidade sobre teste (em inglês **unit under test (UUT)**), como serão os casos dos nossos componentes a validar.

testbenches

unit under test (UUT)

A unidade sobre teste, basicamente, será um processo no qual se deve recolher todas as portas de entrada e saída e ligá-los a sinais do nosso ficheiro de testbench. Atuando como ficheiro *top level* no simulador, pode ser escrito de forma textual, independentemente da ferramenta de simulação a usar (é um ficheiro VHDL com uma estrutura ligeiramente diferente, mas universal).

O primeiro aspeto que devemos desenvolver em relação à nossa testbench, é, então, definir a entidade do nosso módulo. Na Figura 13 podemos ver uma representação de um módulo testbench com uma unidade sobre teste genérica.

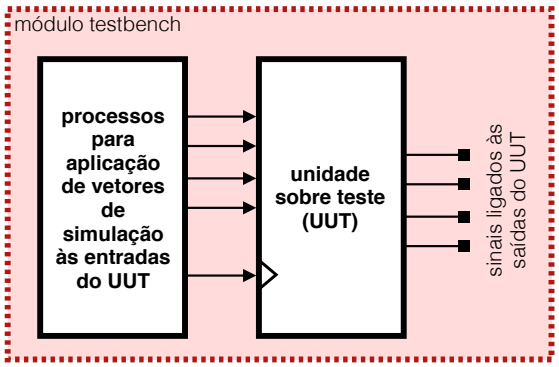


figura 13
módulo testbench

Como podemos verificar pela Figura 13, a entidade do nosso módulo testbench será vazia. No entanto, em termos de arquitetura esta terá de conter tanto sinais quantos forem as entradas e as saídas da nossa unidade sobre teste. Em simultâneo, necessitaremos de ter um ou mais processos que serão correspondentes às várias partes integrantes do nosso módulo testbench, como o representado na Figura 13. Um processo será correspondente à unidade sobre teste e outros mais à lógica restante. O número de processos necessários muda se nós pretendermos testar um componente combinatório ou um componente sequencial. Para o caso de um sequencial será necessário administrar, no mínimo, um sinal de relógio, pelo que deve ser criado um processo exclusivo de um relógio.

Para começar, tentaremos criar um ficheiro VHDL de testbench para um descodificador 2:4. Para tal, tomemos como base o componente `Dec2_4` desenhado anteriormente, no Código 7 e no Código 8. O ficheiro testbench, aqui com o nome de `Dec2_4TB` terá assim uma entidade vazia (não indispensável) com uma arquitetura fundada com dois processos: um para a instanciação da nossa unidade sobre teste (`Dec_4`) e outra para a nossa definição dos testes a serem realizados (que entradas queremos manipular e verificar alterações nas saídas). Para fazer este último passo devemos conhecer uma construção VHDL específica das testbenches que é a `wait for ... ns`, onde "..." pode ser substituído qualquer quantidade e "ns" pode ser "ns" para nanossegundos, "fs" para femtosegundos, "ps" para picossegundos, "us" para microssegundos, "ms" para milisegundos, "sec" para segundos, "min" para minutos ou "hr" para horas[3]. No Código 34 está um exemplo de uma possível testbench para o descodificador 2:4.

código 34
testbench para descodificador 2:4

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Dec2_4TB is
end Dec2_4TB;

architecture Stimulus of Dec2_4TB is
  -- entradas da unidade sobre teste (uut)
  signal s_enable        : std_logic;
  signal s_i             : std_logic_vector(1 downto 0);
  -- saídas da unidade sobre teste (uut)
  signal s_o             : std_logic_vector(3 downto 0);
begin
  uut:           entity work.Dec2_4(Structural)
                 port map (enable => s_enable,
                           i      => s_i,
                           o      => s_o);

  comb_proc:     process
                 begin
                   wait for 100 ns;
                   s_enable  <= '0';
                   wait for 100 ns;
                   s_enable  <= '1';
                   wait for 100 ns;
                   s_i       <= "00";
                   wait for 100 ns;
                   s_i       <= "01";
                   wait for 100 ns;
                   s_i       <= "10";
                   wait for 100 ns;
                   s_i       <= "11";
                   wait for 100 ns;
                 end process;
end Stimulus;
```

Para verificar os resultados da simulação deve ser usada uma ferramenta para o efeito, como a disponibilizada nas aulas práticas (ModelSim Altera) e verificar os

3 Este tipo de dados permitido pela construção VHDL "wait for ..." é time e está definido no pacote STD_LOGIC_1164.

resultados através, por exemplo, de uma interface gráfica garantida por um diagrama temporal.

Se tivermos um componente sequencial e o pretendermos testar, devemos seguir todos os passos expostos anteriormente, sendo que agora necessitamos de, pelo menos, um processo a mais para incluirmos um sinal de relógio, necessário para o teste da unidade sobre teste. Vejamos assim como é que poderíamos fazer um módulo testbench para o componente contador binário com capacidade de escolha entre sentidos de contagem e com reset e enable, desenhado no Código 29 - para tal criamos o módulo `BinaryCounter8ResetEnableUpDownTB`, no Código 35.

código 35
testbench para contador binário do Código 29

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity BinaryCounter8ResetEnableUpDownTB is
end BinaryCounter8ResetEnableUpDownTB;

architecture Stimulus of BinaryCounter8ResetEnableUpDownTB is
  -- entradas da unidade sobre teste (uut)
  signal s_enable         : std_logic;
  signal s_reset          : std_logic;
  signal s_clock          : std_logic;
  signal s_upDown_n       : std_logic;
  -- saídas da unidade sobre teste (uut)
  signal s_count          : std_logic_vector(3 downto 0);
begin
  uut:            entity work.BinaryCounter8ResetEnableUpDown(Behavioral)
                  port map (enable    => s_enable,
                            reset     => s_reset,
                            clock     => s_clock,
                            upDown_n  => s_upDown_n,
                            count     => s_count);

  sync_proc:      process
                  begin
                    wait for 100 ns;
                    s_clock  <= '0';
                    wait for 100 ns;
                    s_clock  <= '1';
                  end process;

  comb_proc:      process
                  begin
                    wait for 100 ns;
                    s_enable   <= '0';
                    s_reset    <= '1';
                    s_upDown_n <= '1';
                    wait for 325 ns;
                    s_reset    <= '0';
                    wait for 25 ns;
                    s_enable   <= '1';
                    wait for 925 ns;
                    s_enable   <= '0';
                    wait for 100 ns;
                    s_upDown_n <= '0';
                    s_enable   <= '1';
                    wait for 100 ns;
                    s_enable   <= '0';
                    wait for 100 ns;
                  end process;
end Stimulus;
```

Então mas afinal quais são os ganhos de uma simulação por testbench VHDL? Em termos de hardware real todos os módulo operam em paralelo, pelo que atualizam as saídas de acordo com o seu estado interno (se assim o for aplicável) e com as entradas de inicialização, sincronização, controlo e dados. Para além disso, em termos de hardware real também temos de considerar atrasos que são naturalmente impostos pela tecnologia e projeto do sistema em causa.

Por outro lado, me termos de simulação temos que considerar que estamos a usar um software que é uma ferramenta de simulação de eventos discretos a executar um ou mais processadores de uso geral. Estamos assim, com as condições perfeitas

para executar testes a todos os níveis, quer comportamentais (iniciais, ideais - sem atrasos), quer funcionais (testes de pós-síntese, sem atrasos) ou até mesmo temporais (pós implementação, considerando os atrasos do circuito).

Depuração do sistema

Uma forma de validar o nosso projeto (especialmente no fim do projeto - validação do ficheiro *top-level*) é através de um processo de **depuração** (*debugging*, em inglês) ao vivo. O que acontece, é que por vezes nós precisamos de ver os sinais a decorrerem ao longo dos nossos processos de simulação, só que nem sempre é possível aproximar o simulado à versão final, executada em hardware. Para tal é necessário executar o nosso projeto numa versão quase final, e colocá-lo à escuta por via de um software. Nas aulas práticas é disponibilizado o software SignalTap II Logic Analyzer que permite iniciar escutas dos acontecimentos, ao vivo, na placa de desenvolvimento.

depuração

7. Modelação de Máquinas de Estados

Na especificação dos passos para a resolução de problemas muitas vezes há parâmetros que, por serem comuns em várias fases do projeto, podem ser simplificados e, evitando redundâncias, generalizados num só evento. Para tal, após tal fase do projeto, é hábito reconhecerem-se construções do tipo **use-cases**, que para determinados casos devem ocorrer determinados procedimentos. Nesta construção, usa-se a ideia geral do conceito de **máquinas de estado finitas** (também vulgarmente designadas por **MEF**'s).

use-cases

máquinas de estado finitas

MEF

Em termos da nossa linguagem de descrição de hardware (VHDL), uma máquina de estados finita é vista como um processo composto por diferentes passos de modelação, otimização e geração de hardware, pelo que cobre, geralmente, uma solução não-única e sub-ótima.

Para a modelação e síntese de uma máquina de estados, quem a projeta, deve saber construir diagramas de estado, tabelas de estados/saídas, tabelas de transição/saídas, tabelas de excitação, diagramas temporais, entre outros... Grande parte deste conhecimento já deve ter sido aprendido na disciplina de Introdução aos Sistemas Digitais (a1s1).

Para elaborar uma síntese de um problema deste estilo devemos entender a especificação inicial, obtendo de seguida uma representação abstrata (por muito que seja crua) da máquina de estados finita - através de um diagrama de estados/saídas, tabelas, ... -, modelando em VHDL e verificando uma simulação funcional da mesma via processos de testbench. Através da linguagem VHDL e das ferramentas de síntese (como a Altera Quartus II) é possível também efetuar uma síntese lógica e implementar em hardware (como na placa de desenvolvimento Terasic DE2-115).

Máquinas de Moore

Consideremos que um determinado cliente nos veio requisitar um serviço para que desenhemos o funcionamento de uma máquina de vendas de bebidas, com os seguintes requisitos: a máquina deve entregar uma lata após o depósito de 0.60€, havendo uma única entrada para moedas, aceitando apenas moedas de 0.20€ e 0.50€ e nunca dando troco.

O primeiro passo a ser feito é perceber o problema, por exemplo, traçando um diagrama de blocos ou um desenho que o exemplifique. Na Figura 14 pode-se ver um **diagrama de blocos** que deve cobrir todo o funcionamento da máquina final.

diagrama de blocos

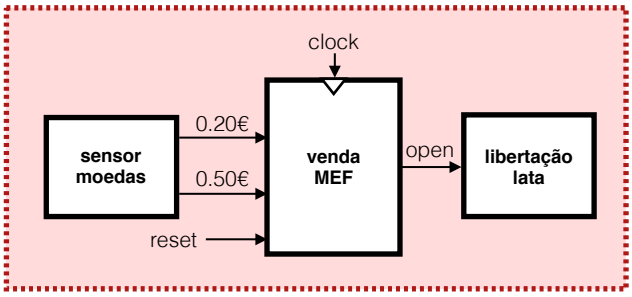


figura 14
diagrama de blocos

De seguida, tendo o diagrama feito (implicitamente indica-se que neste passo já devemos saber minimamente como é que a máquina deve funcionar), deve-se começar por identificar quais são as combinações e sequências entre as moedas (entradas) de 0.20€ e de 0.50€, de forma a formar, no mínimo 0.60€, para a libertação da lata. Para tal, eis as combinações para a libertação da lata:

- 3 moedas de 0.20€ = 0.60€
- 1 moeda de 0.20€ + 1 moeda de 0.50€ = 0.70€ (0.60€ + 0.10€)
- 1 moeda de 0.50€ + 1 moeda de 0.20€ = 0.70€ (0.60€ + 0.10€)
- 2 moedas de 0.50€ = 1.00€ (0.60€ + 0.40€)
- 2 moedas de 0.20€ + 1 moeda de 0.50€ = 0.90€ (0.60€ + 0.30€)

Identificando as nossas entradas como *V* sendo a ativação do sensor para a inserção de uma moeda de 0.20€ e *C* sendo a ativação do sensor para inserção de uma moeda de 0.50€ e a nossa saída como *OPEN*, tal como a Figura 14 indica, podemos agora passar para o desenho do nosso **diagrama de estados**. Assim, na Figura 15, podemos encontrar um digrama de estados que cobre o funcionamento da nossa máquina.

diagrama de estados

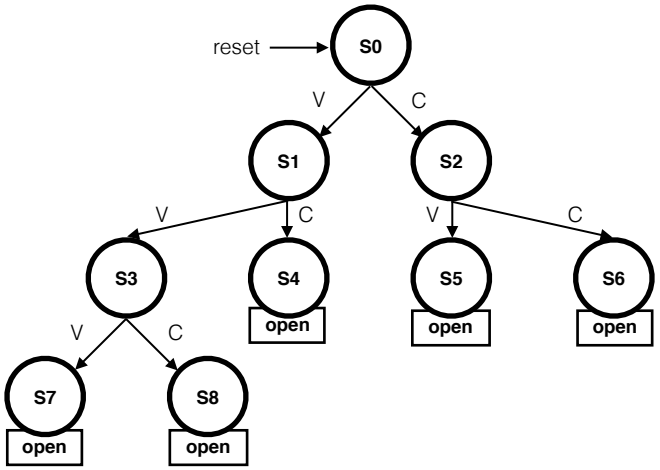


figura 15
diagrama de estados

Tendo o diagrama de estados, numa segunda fase há que tentar simplificar tanto os estados como as suas transições - tentar seguir uma lógica de menor custo, logo menos estados. Segue-se assim a Figura 16, onde temos de novo um diagrama de estados, mais simplificado.

Finalmente podemos gerar uma tabela de estados saídas, para que possamos ver melhor o que se passa em termos de cada um dos estados. Na Figura 17 está representada a tabelas de estados/saídas que decorre diretamente da Figura 16.

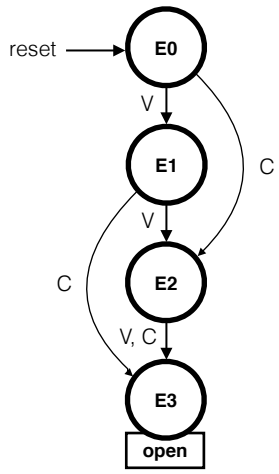


figura 16
diagrama de estados com reutilização de estados

estado atual	entradas		estado seguinte	saída
	V	C		OPEN
E0	0	0	**E0**	0
	0	1	**E2**	0
	1	0	**E1**	0
	1	1	**X**	X
E1	0	0	**E1**	0
	0	1	**E3**	0
	1	0	**E2**	0
	1	1	**X**	X
E2	0	0	**E2**	0
	0	1	**E3**	0
	1	0	**E3**	0
	1	1	**X**	X
E3	X	X	**E3**	1

figura 17
tabela de estados/saídas

A síntese elaborada neste exemplo foi baseada numa **máquina de Moore**, isto é, numa máquina de estados que segue a lógica na qual as saídas dependem apenas e somente do estado atual. Em termos de linguagem VHDL nós vamos poder fazer este raciocínio elaborando uma arquitetura com dois processos. Nestes dois processos, um encarregar-se-á por tratar da parte sequencial (relógio, reset, clear, set, ...) e outro encarregar-se-á pela parte combinatória (lógica operacional, onde há realmente manipulação de dados). Não é obrigatório usarmos apenas dois processos, mas por questões de simplicidade torna-se melhor.

máquina de Moore
⊙ **Edward F. Moore**

Para aplicarmos o nosso conhecimento acerca de máquinas de estados finitos, e para tomar maior atenção à parte respeitante ao código em VHDL, vamos tomar um novo exemplo, da implementação de um detetor de paridade - deteta se um determinado número de '1's ocorridos numa sequência de bits é par ou ímpar. Para a implementação deste bloco precisamos de três entradas (`clock`, `clear`/`reset` (assíncrono) e `dataIn`) e de uma saída `parityOut`. Na Figura 18 podemos encontrar um diagrama de estados deste componente. Note-se também, que na Figura 18 está representado um diagrama de estados por Moore, o que se denota pelo facto de se encontrarem definidas as saídas no interior de cada estado (dado que a saída pertence à definição do

próprio estado e não da transição, como teremos oportunidade de ver mais à frente, nas Máquinas de Mealy).

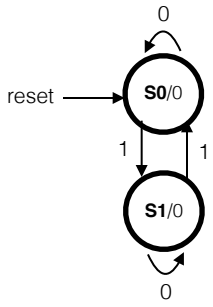


figura 18
diagrama de estados

Passando para o código, no Código 36 eis como modelar o detetor de paridade, baseado numa máquina de estados de Moore, em VHDL.

código 36
detetor de paridade como máquina de Moore

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ParityDetector is
  port (clear    : in    std_logic;
        clock    : in    std_logic;
        dataIn   : in    std_logic;
        parityOut: out   std_logic);
end ParityDetector;

architecture Behavioral of ParityDetector is
  type TState is (S0, S1);
  signal pState, nState   : TState;
begin
  sequ_proc:     process(clock, clear)
                 begin
                   if (clear = '1') then
                     pState <= S0;
                   elsif (rising_edge(clock)) then
                     pState <= nState;
                   end if;
                 end process;

  comb_proc:     process(pState, nState)
                 begin
                   case pState is
                     when S0 =>
                       parityOut <= '0'; -- saída de Moore
                       if (dataIn = '1') then
                         nState <= S1;
                       else
                         nState <= S0;
                       end if;
                     when S1 =>
                       parityOut <= '1'; -- saída de Moore
                       if (dataIn = '1') then
                         nState <= S0;
                       else
                         nState <= S1;
                       end if;
                     when others =>
                       parityOut <= '0'; -- saída de Moore
                       nState <= S0;
                   end case;
                 end process;
end Behavioral;
```

Como podemos verificar via Código 36, a máquina de estados foi elaborada com base numa construção VHDL de definição de um novo tipo (enumerado) que contém dois possíveis ambientes (`S0` ou `S1`), através da linha `type TState is (S0, S1)`. Na linha seguinte, define-se que existem dois sinais que assumem dados do tipo `TState` tal que se chamam `pState` e `nState`.

Máquinas de Mealy

Tal como estudámos anteriormente com as máquinas de Moore, agora com as máquinas de Mealy o estilo de implementação com dois processos na arquitetura também se pode aplicar. Como já vimos, as máquinas de Moore têm como caraterística principal que as suas saídas apenas dependem, exclusivamente, do estado atual da máquina. Por outro lado, as **máquinas de Mealy** têm as suas saídas que dependem tanto do estado atual como das próprias entradas da máquina. Por esta mesma razão, na representação dos diagramas de estados, contrariamente ao que se fazia no caso nas máquinas de Moore (colocar a saída dentro de cada estado), como a saída já não faz parte exclusiva do estado, deixa de ser definida dentro dele, mas antes nas transições de estados, pois agora também depende do valor das entradas, condição para transições.

máquina de Mealy
⊙ George H. Mealy

Para o caso das máquinas de Mealy, tomemos como exemplo um detetor de sequência, que dá valor `'1'` na saída `zOut` cada vez que deteta a sequência 1101, com sobreposição, isto é, permitindo que o componente aproveite o último algarismo da sequência pretendida como o início de uma nova (dá saída `'1'` para `11011101` e para `1101101`). Vejamos assim, na Figura 19 o diagrama de estados e no Código 37, uma possibilidade de implementação do detetor de sequência.

código 37
detetor de sequência 1101 como máquina de Mealy

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Seq1101Detector is
  port (clear    : in    std_logic;
        clock    : in    std_logic;
        xIn      : in    std_logic;
        zOut     : out   std_logic);
end Seq1101Detector;

architecture Seq1101Detector of ParityDetector is
  type TState is (A, B, C, D);
  signal PS, NS  : TState;
begin
  sequ_proc:     process(clock, clear)
                 begin
                   if (clear = '1') then
                     PS <= A;
                   elsif (rising_edge(clock)) then
                     PS <= NS;
                   end if;
                 end process;

  comb_proc:     process(PS, NS)
                 begin
                   zOut <= '0'; -- saída de Mealy
                   case PS is
                     when A =>
                       if (xIn = '1') then NS <= B;
                       else NS <= A;
                       end if;
                     when B =>
                       if (xIn = '1') then NS <= C;
                       else NS <= A;
                       end if;
                     when C =>
                       if (xIn = '1') then NS <= C;
                       else NS <= D;
                       end if;
                     when D =>
                       if (xIn = '1') then
                         NS <= B;      -- sobreposição
                         zOut <= '1'; -- saída de Mealy
                       else NS <= A;
                       end if;
                     when others =>
                       NS <= A;
                   end case;
                 end process;
end Behavioral;
```

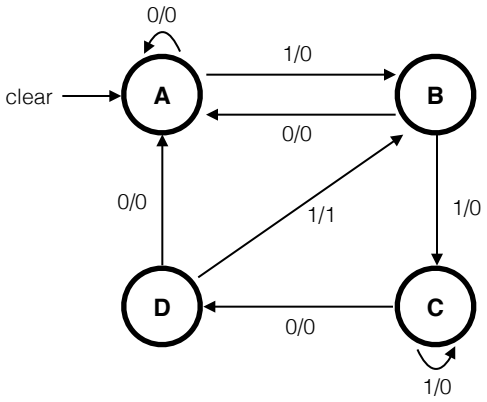


figura 19
diagrama de estados

Na linguagem de descrição que estamos a estudar (VHDL), o recurso a um tipo enumerado para codificar simbolicamente os estados permite uma descrição de alto-nível muito próxima do diagrama de estados, como pudemos ver anteriormente. Conferindo ao sintetizador a tarefa da codificação binária dos estados, o hardware criado, por si, inclui mais flip-flops que o número mínimo possivelmente esperado, dado que a codificação que estes seguem, geralmente, é de **one-hot**, sobre a qual, por exemplo, se tivermos 4 estados, em termos de codificação fará um enumerado como (0001, 0010, 0100, 1000), o que é adequado à implementação em FPGA.

one-hot

Máquinas de estados comunicantes

Tendo várias máquinas de estados é possível fazer com que todas trabalhem de forma síncrona e com variados tipos de protocolos de comunicação entre elas. A estas máquinas damos o nome de **máquinas de estados comunicantes**. O que aqui acontece é que muitas vezes é importante fazer uma partição do nosso sistema global em duas ou mais máquinas paralelas ou sequenciais.

máquinas de estados comunicantes

Um exemplo destas máquinas pode ser um *fader* de áudio por *crossfade* -- componente que permite que na transição de sons diferentes o som antigo diminui linearmente a sua amplitude, enquanto que o som novo aumenta a amplitude do mesmo modo, sendo que ambas amplitudes atingem um ponto em comum (a meia amplitude). Na Figura 20 está representada a máquina de estados comunicante de que é feito este componente.

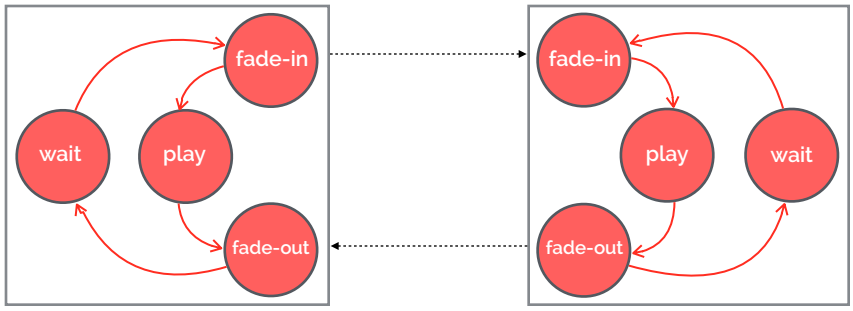


figura 20
máquina de estados comunicantes

Na máquina de estados comunicantes da Figura 20, quando uma máquina está no estado i, a máquina seguinte (semelhante à primeira) está no estado $i + 2$. Assim, se o meu hardware não estiver a reproduzir nenhum som, uma das máquinas encontra-

-se no estado *play* (a reproduzir silêncio) e a outra no estado *wait*, como podemos ver na Figura 21.

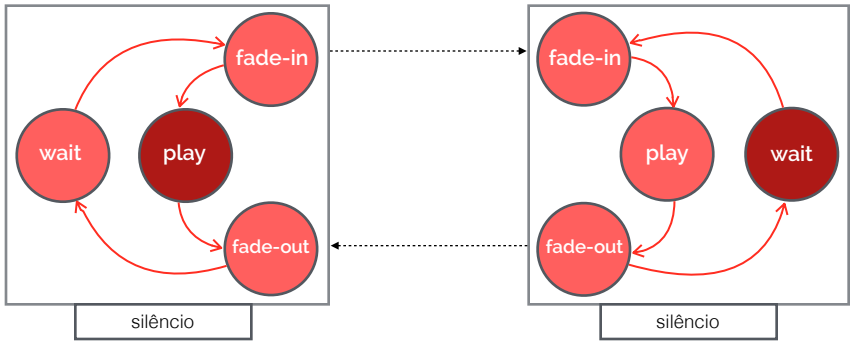


figura 21
máquina de estados comunicantes

Se agora tocarmos uma nota, por exemplo, um Dó índice 3, a máquina que estava no estado *wait* deve receber a nota nova, comunicar à outra máquina que já recebeu nova nota e a outra máquina deve iniciar *fade-out* do silêncio (enquanto a primeira máquina recebe a nota e inicia *fade-in*). Este processo está mostrado na Figura 22.

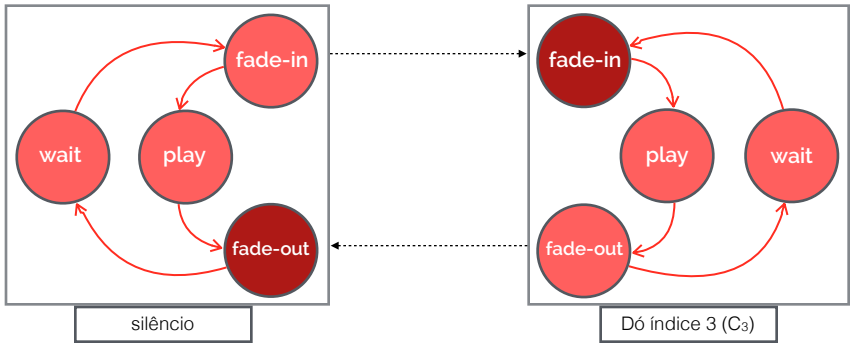


figura 22
máquina de estados comunicantes

Quando o *fade-in* termina, isto é, quando a amplitude atinge o seu valor máximo, a máquina envia um sinal para a sua paralela e ambas transitam de estados, como podemos ver na Figura 23.

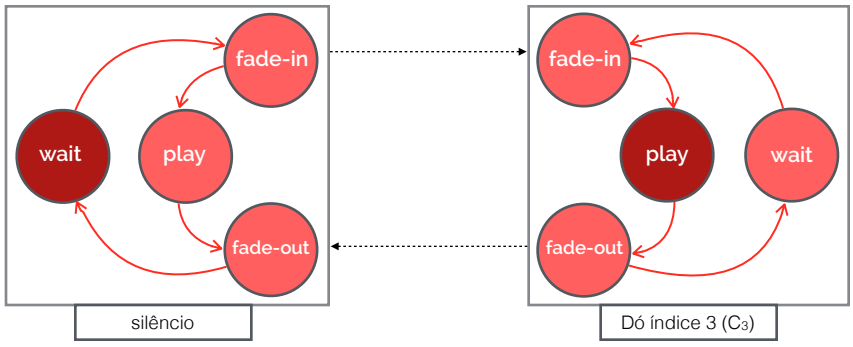


figura 23
máquina de estados comunicantes

Se a qualquer momento decidirmos tocar nova nota, por exemplo, um Sol bemol índice 4, a máquina que está no estado *wait* recebe nova nota, comunicando à outra máquina que a vai receber, transitando, assim, de estado, para *fade-in*, de modo a receber a nova nota, enquanto que a outra máquina transita para o estado *fade-out* para que esta possa desvanecer a nota Dó índice 3, como se vê na Figura 24.

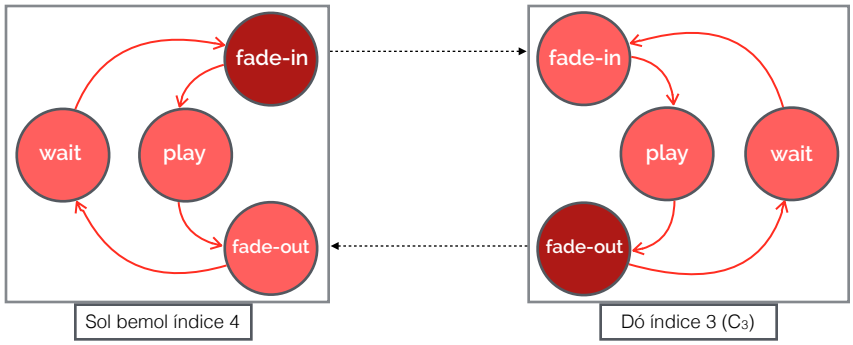


figura 24
máquina de estados comunicantes

Numa fase seguinte, quando a amplitude do Sol bemol índice 4 for máxima, a máquina que a sustém passa para o estado *play* e a outra passa para o estado *wait*, sendo que quando a máquina está no estado *wait*, por muito que ainda tenha a nota nela guardada, não a reproduz. Este processo pode ser visto na Figura 25.

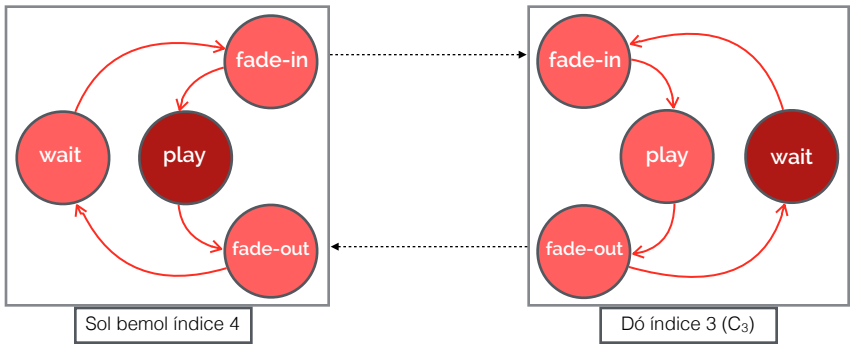


figura 25
máquina de estados comunicantes

Este processo continua sempre da mesma forma, consecutivamente, passando novo som de máquina em máquina.

8. Modelação de Memórias em VHDL

Um elemento importante em termos de criação de projetos com hardware é a gestão de dados e o seu armazenamento. Em questões de armazenamento, já devemos saber que este é feito através de componentes denominados de **memórias**. Uma memória é assim um módulo que tem como principal função a salvaguarda de dados (quer de um só bit ou de uma sequência de b bits).

memórias

Memórias ROM

Um tipo de memória é a **ROM**. Uma memória ROM, tal como o acrónimo o indica, é uma memória apenas de leitura (em inglês, *Read-Only Memory*). De forma análoga a processos de linguagens de programação, uma ROM tem um funcionamento muito próximo a um array de linguagens como a Java ou a Lua. Basicamente, é assim uma matriz (ou uma tabela unidimensional, se assim se preferir) de tamanho fixo, com cada índice acessível a uma complexidade $O(1)$. Assim, para aceder a um determinado dado guardado em memória é necessário indicar o endereço (entrada da ROM) e na saída é- -nos devolvido o valor preservado no endereço especificado da ROM.

ROM

Em termos de hardware, uma ROM tem dimensões $2^n \times b$, onde b é o número de bits por sequência de bits guardada na ROM e 2^n é número de endereços da ROM.

Na Figura 26 podemos ver a estrutura interna de uma ROM (modelo conceptual), onde temos quatro palavras de b bits.

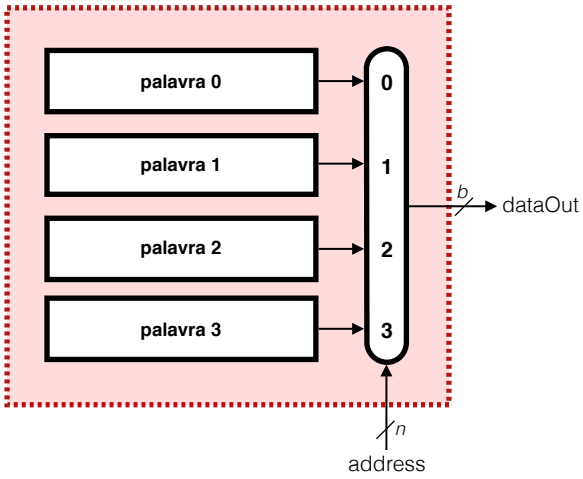


figura 26
memória ROM

Na Figura 26 temos então uma ROM de 4 palavras (*words*) com uma entrada de nome `address` onde se especifica um endereço (`"00"`, `"01"`, `"10"` ou `"11"`) e uma saída de nome `dataOut` onde sai a palavra escolhida por via do endereço inserido e do multiplexer, com b bits (geralmente potências de base 2). Neste dispositivo a leitura é feita de modo totalmente assíncrono, dado que se trata de um processo meramente combinatório. Caso tivessemos uma memória de 1M palavras, aí, seriam necessários 20 bits de endereços, dado que lb $2^{20} = 20$.[4]

Em termos de código VHDL, para definir uma memória ROM necessitamos de usar um novo conceito da linguagem que é o de **array**. Assim, usamos uma construção algo semelhante à da criação de uma máquina de estados, mas para criar um novo tipo de dados, com a forma de um array de um tipo específico de dados. Por exemplo, se escrevermos `type ArrayDeInteiros is array (0 to 15) of integer` dizemos que queremos criar um array de inteiros, com 16 entradas (tamanho 16), com o nome `ArrayDeInteiros`. De uma forma genérica, no Código 38, podemos ver como definir um tipo do array em VHDL, onde XX é (tamanho do array - 1).

array

```
type nome_do_tipo is array (0 to XX) of tipo_de_elementos;
```

código 38
array em VHDL

Para criarmos uma ROM em VHDL, apenas precisamos de gerar o Código 39.

código 39
memória ROM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity ROM_8x4 is
  port (address  : in     std_logic_vector(2 downto 0);
        dataOut  : out    std_logic_vector(3 downto 0));
end ROM_8x4;

architecture Behavioral of ROM_8x4 is
  subtype TDataWord is std_logic_vector(3 downto 0);
  type TROM is array (0 to 7) of TDataWord;
  constant c_memory: TROM := ("0000","0001","0010","0011",
                              "0100","0101","1010","1111");
begin
  dataOut <= c_memory(to_integer(unsigned(address)));
end Behavioral;
```

4 "lb" significa logaritmo binário, isto é, logaritmo de base 2.

No Código 39, na definição da ROM em código VHDL, podemos reparar que foi criada uma constante, e não um sinal. Porquê? Uma ROM é uma memória exclusiva para leitura, pelo que não se pode escrever lá. Sendo assim, os valores lá preservados, uma vez inseridos (aqui por vias da programação), não podem ser alterados, pois permanecem sempre iguais - daí o uso da palavra reservada `constant`, ao invés da `signal`.

Memórias RAM

RAM

Um segundo tipo de memórias são as memórias **RAM**. Uma memória RAM, tal como o acrónimo o indica, é uma memória que pode ser acedida de modo aleatório, mas que cujo conteúdo pode ser alterado (em inglês diz-se *Random-Access Memory*). Numa memória deste tipo, num dado instante de tempo pode ser lida ou escrita qualquer posição de memória. Estes componentes são úteis para o armazenamento de informação arbitrária que pode ser alterada durante o funcionamento do sistema. Sendo volátil, esta pode ser estática ou dinâmica, como teremos oportunidade de verificar e estudar na disciplina de Arquitetura de Computadores II (a2s2). Estas memórias podem disponibilizar uma porta, fornecendo apenas uma operação de leitura ou escrita num dado instante, ou mais portas, fornecendo várias operações de leitura e de escrita num dado instante.

Em termos de estrutura interna esta memória tem uma complexidade maior que a memória ROM, como podemos averiguar na Figura 27.

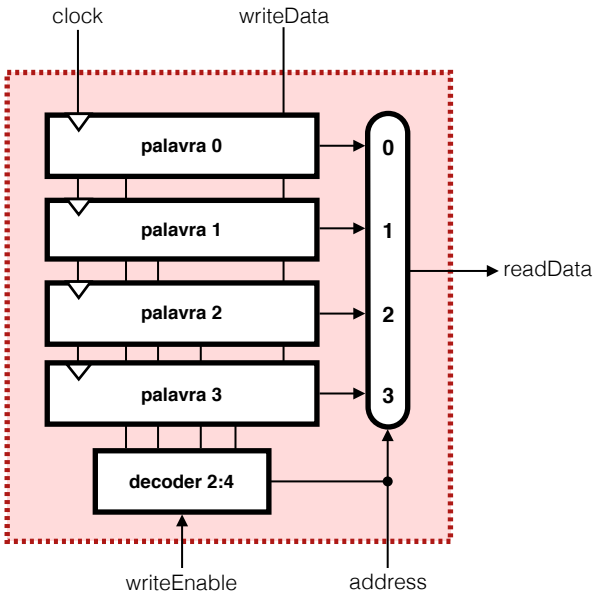


figura 27
memória RAM

Na RAM representada (modelo conceptual) na Figura 27 temos quatro entradas (`writeEnable` para permitir a escrita, `clock` como sinal de relógio, `writeData` como entrada de n bits para escrita e `address` como endereço de escolha) e uma saída de nome `readData`, pela qual é debitada a palavra acedida pelo endereço inserido. Este tipo de memória carateriza-se por ter momentos de escrita síncrona e leitura assíncrona.

Em termos de código VHDL, uma memória RAM de 32 palavras (32×8) com uma porta, escrita síncrona e leitura assíncrona é feita de acordo com o exibido no Código 40.

código 40
memória RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity RAM_32x8_ESLAS is
  port (wrClock  : in     std_logic;
        wrEnable : in     std_logic;
        wrData   : in     std_logic_vector(7 downto 0);
        address  : in     std_logic_vector(4 downto 0);
        readData : out    std_logic_vector(7 downto 0));
end RAM_32x8_ESLAS;

architecture Behavioral of RAM_32x8_ESLAS is
  constant NUM_WORDS : integer := 32;
  subtype TDataWord is std_logic_vector(7 downto 0);
  type TMemory is array (0 to NUM_WORDS-1) of TDataWord;
  signal s_memory: TMemory;
begin
  process(wrClock)
  begin
    if (rising_edge(wrClock)) then
      if (wrEnable = '1') then
        s_memory(to_integer(unsigned(address))) <= writeData;
      end if;
    end if;
  end process;
  readData <= s_memory(to_integer(unsigned(address)));
end Behavioral;
```

Tal como foi referido, podemos criar uma RAM com mais portas. Sendo assim, tentemos desenhar uma RAM com duas portas - uma de escrita (síncrona) e uma de leitura (assíncrona) - (Código 41).

código 41
memória RAM com 2 portas

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity RAM_32x8_2PESLAS is
  port (wrClock  : in     std_logic;
        wrEnable : in     std_logic;
        wrAddress: in     std_logic_vector(4 downto 0);
        wrData   : in     std_logic_vector(7 downto 0);
        rdAddress: in     std_logic_vector(4 downto 0);
        readData : out    std_logic_vector(7 downto 0));
end RAM_32x8_2PESLAS;

architecture Behavioral of RAM_32x8_2PESLAS is
  constant NUM_WORDS : integer := 32;
  subtype TDataWord is std_logic_vector(7 downto 0);
  type TMemory is array (0 to NUM_WORDS-1) of TDataWord;
  signal s_memory: TMemory;
begin
  process(wrClock)
  begin
    if (rising_edge(wrClock)) then
      if (wrEnable = '1') then
        s_memory(to_integer(unsigned(wrAddress))) <= writeData;
      end if;
    end if;
  end process;
  readData <= s_memory(to_integer(unsigned(rdAddress)));
end Behavioral;
```

Do mesmo modo, podemos também criar uma RAM de duas portas, mas com uma porta de escrita síncrona e uma porta de leitura também síncrona. Para fazermos tal memória, em VHDL, necessitamos de criar um paralelismo entre dois processos: um dos processos será responsável pela leitura e um outro pela escrita. No Código 42 podemos assim verificar como implementar esta memória.

As memórias são componentes que facilmente são parametrizáveis. Sendo todo o código análogo ao restante código fornecido ao longo da disciplina, o único pormenor que difere é no cálculo do número máximo do índice das memórias, que pode ser criado através de `constant NUM_WORDS : integer := (2 ** tamanho_do_bus_de_endereço);`.

código 42
memória RAM com escrita síncrona e leitura síncrona

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity RAM_32x8_ESLS is
  port (wrClock  : in     std_logic;
        wrEnable : in     std_logic;
        wrAddress: in     std_logic_vector(4 downto 0);
        wrData   : in     std_logic_vector(7 downto 0);
        rdClock  : in     std_logic;
        rdAddress: in     std_logic_vector(4 downto 0);
        readData : out    std_logic_vector(7 downto 0));
end RAM_32x8_ESLS;

architecture Behavioral of RAM_32x8_ESLS is
  constant NUM_WORDS : integer := 32;
  subtype TDataWord is std_logic_vector(7 downto 0);
  type TMemory is array (0 to NUM_WORDS-1) of TDataWord;
  signal s_memory: TMemory;
begin
  writ_proc:     process(wrClock)
                 begin
                   if (rising_edge(wrClock)) then
                     if (wrEnable = '1') then
                       s_memory(to_integer(unsigned(wrAddress))) <= writeData;
                     end if;
                   end if;
                 end process;

  read_proc:     process(rdClock)
                 begin
                   if (rising_edge(rdClock)) then
                     readData <= s_memory(to_integer(unsigned(wrAddress)));
                   end if;
                 end process;
end Behavioral;
```

9. Bibliotecas e Pacotes de VHDL

Ao longo desta disciplina, fomos utilizando variados recursos da linguagem VHDL. Estes recursos estão organizados por bibliotecas e por pacotes. Uma **biblioteca** (em inglês *library*) é um conjunto de um ou mais **pacotes** (em inglês *packages*). As bibliotecas contêm definições úteis para os vários projetos que possam ser criados em VHDL, entre eles, tipos, constantes, componentes, funções, entre outros. Dentro de cada biblioteca há, como dito, um conjunto de um ou mais pacotes, estes, que contêm definições úteis para os projetos e/ou módulos.

biblioteca
pacotes

As bibliotecas que são standard para o VHDL (comuns) são as IEEE e a STD. Existe uma outra biblioteca que é automaticamente definida que tem o nome de WORK, que consiste na biblioteca que nós próprios criamos aquando de um projeto, no qual estão especificados todos os detalhes acerca dos ficheiros que vamos criando.

Alguns dos pacotes mais utilizados do VHDL são, para síntese, a `IEEE.STD_LOGIC_1164` e a `IEEE.NUMERIC_STD`. Em termos de simulação, embora não tenhamos usado, é sempre possível usar `IEEE.STD_LOGIC_TEXTIO`, a `IEEE.MATH_REAL`, a `IEEE.MATH_COMPLEX` ou a `STD.TEXTIO`. Outros pacotes menos usados são o `IEEE.NUMERIC_BIT`, o `IEEE.STD_LOGIC_SIGNED`, o `IEEE.STD_UNSIGNED` e o `IEEE.STD_LOGIC_ARITH`.

Criar um pacote

Criar um pacote em VHDL é um processo relativamente simples. Para tal, consideremos o Código 43, onde criamos o ficheiro `LSDApontamentos.vhd`, no qual se define o pacote `LSDApontamentos`.

código 43
pacote LSDApontamentos

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package LSDApontamentos is
  subtype TNibble is std_logic_vector(3 downto 0);
  subtype TWord   is std_logic_vector(31 downto 0);
  constant WORD_MAX       : TWord := (others => '1');
  function min(a, b : integer) return integer;
  function boolean2std_logic(v : boolean) return std_logic;
end LSDApontamentos;

package body LSDApontamentos is
  function min(a, b : integer) return integer is
  begin
    if (a < b) then
      return a;
    else
      return b;
    end if;
  end min;

  function boolean2std_logic(v : boolean) return std_logic is
  begin
    if (v = false) then
      return '0';
    else
      return '1';
    end if;
  end boolean2std_logic;
end LSDApontamentos;
```

Pacote STD_LOGIC_1164

O pacote `STD_LOGIC_1164` da biblioteca `IEEE` tem um tipo (`STD_LOGIC(_VECTOR)`) que por si contém subtipos lógicos standard que podem ser divididos em dois grandes grupos: **resolved** (valores que são específicos e legíveis em termos de hardware) e **unsolved** (valores que não são atribuídos explicitamente, mas geridos pelo simulador). Como tipos resolved temos o `'0'` que traduz o nível lógico baixo forte, o `'1'` que traduz o nível lógico alto forte, `'L'` que traduz o nível lógico baixo fraco, `'H'` que traduz o nível lógico alto fraco, `'Z'` que traduz o estado de alta-impedância (tri-state) e o `'-'` que traduz o *don't care*. Já nos tipos unsolved temos o `'X'` que traduz um conflito forte, o `'W'` que traduz um conflito fraco e o `'U'` que traduz um sinal não inicializado durante a simulação.

resolved
unsolved

Os **conflitos** que aparecem ao longo da descrição anterior relacionam-se com a injeção de dois sinais de entrada num mesmo ponto. Dependendo da lógica de cada uma das entradas e da sua força, é possível obter conflitos ou resolver os conflitos. Por norma, sempre que num mesmo ponto se encontrem dois níveis com lógicas diferentes, o que tiver mais força resolve o conflito. Quando há situações em que ambos os sinais têm a mesma força ocorrem conflitos fortes (se forem ambos sinais fortes) ou conflitos fracos (se forem ambos sinais fracos). Aos conflitos fracos damos o nome de **conflitos passivos**, e estes geram uma corrente de +VDD/2, enquanto que aos conflitos fortes damos o nome de **conflitos ativos**, que resultam sempre em **curto-circuito**.

conflitos
conflitos passivos
conflitos ativos,
curto-circuito

Uma forma de resolver situações em que se pretende unir duas entradas num ponto é através de um buffer tri-state, conforme foi abordado em Introdução aos Sistemas Digitais (a1s1).

Outros tipos do pacote `STD_LOGIC_1164` são os enumerados (como usámos nas máquinas de estados), os booleanos (resultados de condições e expressões booleanas), os carateres e strings (para manipulação de carateres e arrays de carateres), os reais (para simulação com quantidade reais), `time` (para simulação e construção de testbenches) e os `bit` e o `bit_vector` (pouco usados dada a existência do `std_logic(_vector)`.

Apontamentos de Laboratórios de Sistemas Digitais
2ª edição - junho de 2015

lsd

Autor: Rui Lopes
Fontes bibliográficas: Digital Design: An Embedded Systems Approach Using VHDL, ASHENDEN, P.J.; Free Range VHDL, MEALY, B. et TAPPERO, F.; The Designer's Guide to VHDL, ASHENDEN, P.J.; Computer Science, An Overview, BROOKSHEAR, Glenn.
Outros recursos: Recursos fornecidos na página web da disciplina.
Agradecimentos: professor Arnaldo Oliveira, Guilherme Campos, Iouliia Skliarova e Bernardo Cunha.

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.

apontamentos