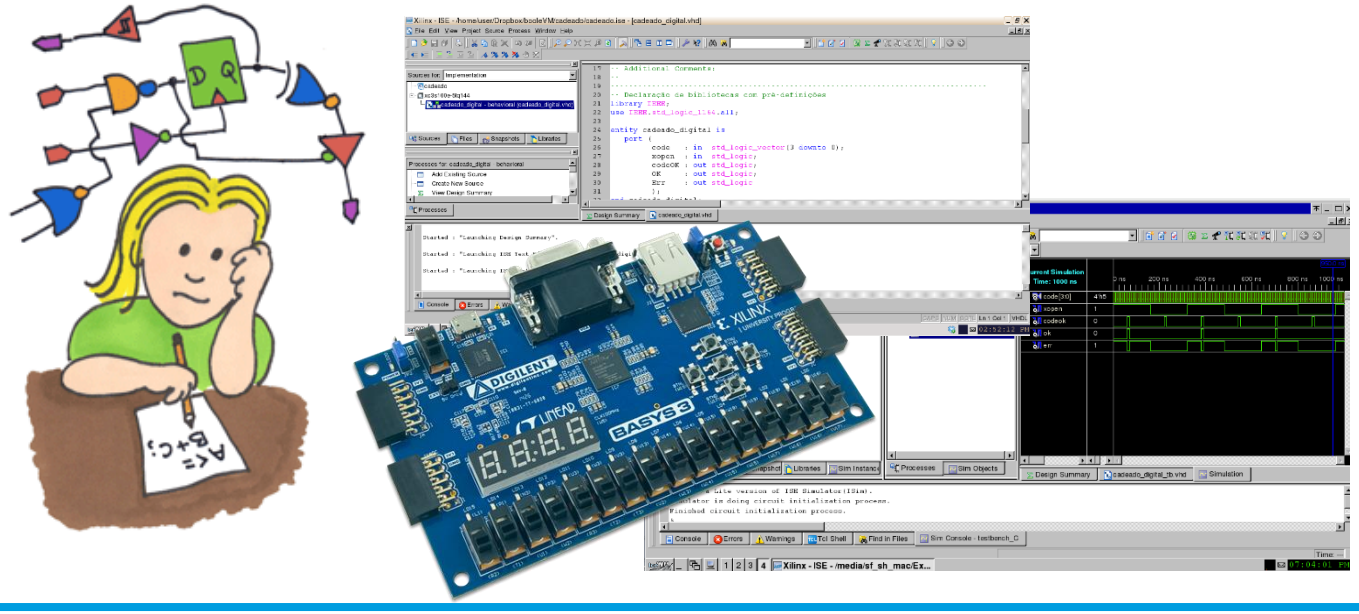
TÉCNICO LISBOA

Sistemas Digitais (SD)

Linguagens de Descrição e Simulação de Circuitos Digitais (apoio ao laboratório)



Aula Anterior

- **Na aula anterior:**
  - Unidade Lógica e Aritmética (ULA)

TÉCNICO LISBOA

Planeamento

SEMANA	TEÓRICA 1	TEÓRICA 2	PROBLEMAS/LABORATÓRIO
19/Set a 23/Set	Introdução	Sistemas de Numeração e Códigos	
26/Set a 20/Set	Álgebra de Boole	Elementos de Tecnologia	P0
3/Out a 7/Out	Funções Lógicas	Minimização de Funções Booleanas (I)	L0
10/Out a 14/Out	Minimização de Funções Booleanas (II)	Def. Circuito Combinatório; Análise Temporal	P1
17/Out a 21/Out	Circuitos Combinatórios (I) – Codif., MUXs, etc.	Circuitos Combinatórios (II) – Som., Comp., etc.	L1
24/Out a 28/Out	Circuitos Combinatórios (III) - ALUs	Linguagens de Descrição e Simulação de Circuitos Digitais	P2
31/Out a 4/Nov	**FERIADO (1/Nov)**	Circuitos Sequenciais: Latches	L2
7/Nov a 11/Nov	Circuitos Sequenciais: Flip-Flops	Caracterização Temporal	P3
14/Nov a 18/Nov	Registos	Contadores **Teste 1**	L3
21/Nov a 25/Nov	Síntese de Circuitos Sequenciais: Definições	Síntese de Circuitos Sequenciais: Minimização do número de estados	P4
28/Nov a 2/Dez	Síntese de Circuitos Sequenciais: Síntese com Contadores	**FERIADO (1/Dez)**	L4
5/Dez a 9/Dez	Memórias	**FERIADO (8/Dez)**	P5
12/Dez a 16/Dez	Máq. Estado Microprogramadas: Circuito de Dados e Circuito de Controlo	Máq. Estado Microprogramadas: Endereçamento Explícito/Implícito	L5
19/Dez a 23/Dez	**Férias Natal**	**Férias Natal**	**Férias Natal**

TÉCNICO LISBOA

Sumário

- **Tema da aula de hoje:**
  - Linguagens de descrição de Hardware
  - Introdução a VHDL:
    - Descrição de estruturas básicas em VHDL
    - Exemplos:
      - Cadeado digital
      - Unidade aritmética

LINGUAGENS DE DESCRIÇÃO DE HARDWARE

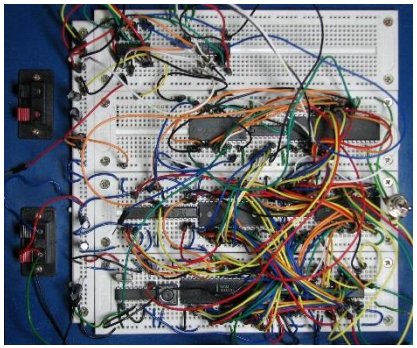
Projecto de um circuito digital

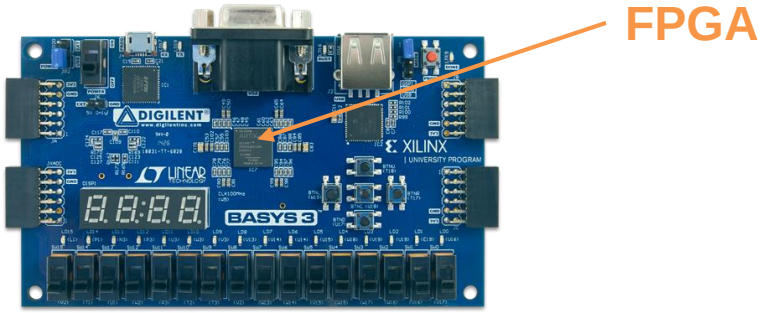
- **Etapas de projecto:**
  1. **Descrição do sistema a projectar**
     - A descrição do sistema é tipicamente feita sob uma forma verbal, não totalmente especificada (tal como aparece nos enunciados de laboratório)
  2. **Especificação do sistema**
     - Divisão do problema em partes (funções lógicas) e especificação de cada uma das partes (funções), geralmente sob a forma de tabelas de verdade
  3. **Derivação das expressões lógicas**
     - Obtenção e minimização das funções lógicas, através da expressão booleana ou mapas de Karnaugh
  4. **Desenho do circuito digital**
     - Desenho do circuito digital usando os elementos básicos de lógica (portas NOT, AND, OR, NAND, NOR, XOR, MUX, DECODER, ...)
  5. **Implementação física**
     - Utilizando circuitos integrados

Projecto de um circuito digital
Implementação física

Implementação física

- Actualmente, é raro fazer-se a implementação de circuitos digitais com componentes discretos (CIs).
  - **PROBLEMA:** Os circuitos são frequentemente muito complexos
    SOLUÇÃO: utilização de **FPGAs** (lógica programável)





- Recorre-se tipicamente a linguagens de descrição de hardware, tais como VHDL ou Verilog
  - Para descrever correctamente o circuito digital em VHDL ou Verilog é necessário conceber primeiro o diagrama lógico

Hardware Description Language (HDL)
VHDL, para que serve?

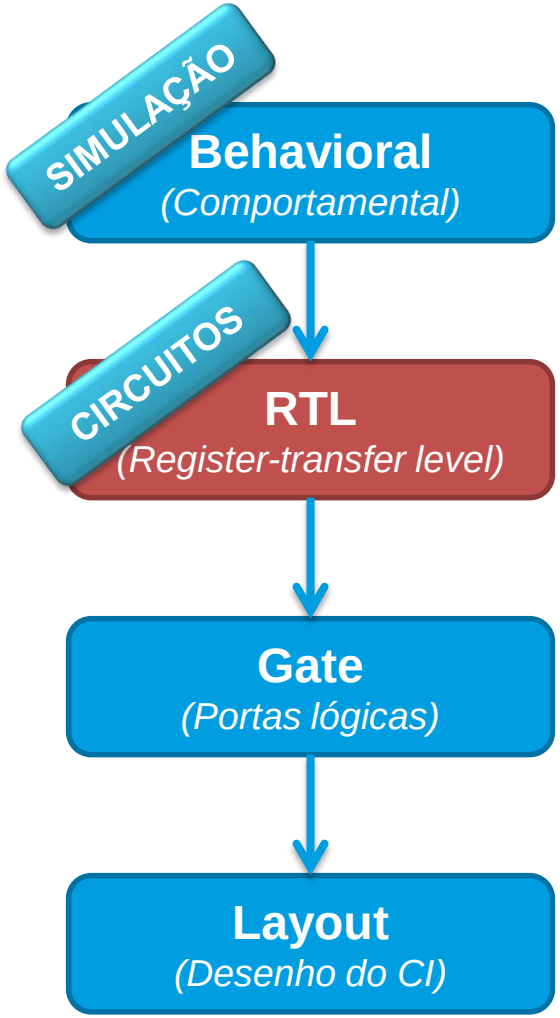
- **As linguagens VHDL e Verilog servem para:**
  - Descrever circuitos digitais
  - Simular circuitos digitais
    - Verificar a funcionalidade, testar e corrigir os erros

- **VHDL (ou Verilog) não é uma linguagem de programação**
  - Os circuitos digitais não se programam... descrevem-se!
  - Escrever código VHDL não é mais do que “descrever” o esquema lógico do circuito digital!

- **NOTAS IMPORTANTES:**
  - Na disciplina de Sistemas Digitais (SD) será sempre **OBRIGATÓRIA** a apresentação do esquema lógico (logigrama)
  - Nem todas as funcionalidade de VHDL serão permitidas!
  - Quaisquer diferenças entre o esquema apresentado e o código VHDL apresentado levarão a penalizações na nota final!
  - O código VHDL deverá ser sempre apresentado em anexo, devidamente comentado.

Níveis de abstração

- **Comportamental (Behavioral)**
  - Descrição funcional do circuito digital
  - Geralmente usada para simular circuitos, mas nem sempre é sintetizável para portas lógicas
- **RTL (Register-Transfer Level)**
  - Descrição do circuito através da instanciação de elementos combinatórios (AND, OR, NOT, multiplexers, descodificadores, somadores, subtractores, ...) e elementos de memória (latches, flip-flops, registos, ...)
- **Gate (portas lógicas AND, OR, ...)**
  - Descrição do circuito através dos componentes principais existentes numa dada biblioteca lógica
- **Layout**
  - Desenho do circuito integrado



TÉCNICO LISBOA

DESCRIÇÃO DE ESTRUTURAS BÁSICAS EM VHDL

Descrição de circuitos em VHDL

- **Um ficheiro VHDL (extensão .vhd) descreve o funcionamento de um circuito digital e pode ser decomposto em duas partes:**
  - **Entidade:**
    - Definição do componente (circuito digital), nomeadamente nome e sinais (fios) de entrada e de saída
  - **Arquitectura:**
    - Descrição da forma como o componente está implementado

Estrutura típica de um ficheiro VHDL

```
-- COMENTÁRIOS

-- Declaração de bibliotecas com pré-definições
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e dos sinais (fios) de entrada/saída
entity <NOME_DO_COMPONENTE> is
   port (
    ...
   );
end <NOME_DO_COMPONENTE>;

-- Descrição da arquitectura (implementação) do componente
architecture <TIPO_DE_ARQUITECTURA> of <NOME_DO_COMPONENTE> is
   -- declaração dos sinais (fios) internos ao componente
Begin
   -- descrição do circuito digital que implementa o componente
   ...
end <TIPO_DE_ARQUITECTURA>;
```

Tipos de sinais

- **Um sinal em VHDL corresponde a um fio num circuito físico**

- **Num circuito digital, o tipo de fio deverá ser:**
  - **bit**, o qual pode ter os valores lógicos 0 e 1

- **No entanto, é comum usar-se outro tipo de sinal, o qual é particularmente útil durante o passo de simulação do circuito:**
  - **std_logic**, o qual pode tomar os valores lógicos:
    - '0' (zero) e '1' (um)
    - 'Z' alta impedância
    - '-' don't care
    - 'U' undefined (o valor do fio não foi definido)
    - 'X' unknown (não é possível determinar o valor do fio)

- **NOTA IMPORTANTE:** na disciplina de SD apenas é permitido a atribuição dos valores 0 e 1 a um sinal! Os valores 'U' e 'X' serão atribuídos automaticamente pela ferramenta (Xilinx Vivado) quando um sinal (fio) não tiver valor atribuído ou não for possível a sua determinação (ex: quando são atribuídos simultaneamente os valores '0' e '1').

Tipos de sinais

- **Exemplo:**

  Representação dos sinais `acende_LED` e `liga_ALARME` :

```
signal acende_LED : std_logic;
signal liga_ALARME : std_logic;
```

ou

```
signal acende_LED , liga_ALARME : std_logic;
```

Tipos de sinais

- **Por vezes são necessários vários fios para representar um único valor.**

- **Exemplo:**

  Representação do número de um aluno do IST:

  - Considerando que o maior número de aluno é o 100 000
  - São necessários pelo menos log2(100 000) bits = 16,61 bits

  - Portanto o sinal `num_aluno` precisa de pelo menos 17 bits

Tipos de sinais

- **Por vezes são necessários vários fios para representar um único valor.**

- **Exemplo:**

  Representação do número de um aluno do IST:

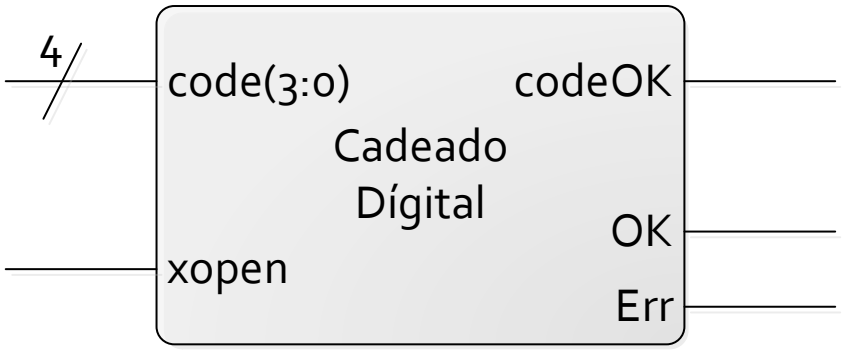
  - Para evitar a declaração (especificação) de 17 sinais (fios) individualmente, utiliza-se o sinal de barramento (*bus*):

```
signal num_aluno : std_logic_vector(16 downto 0);
```

  - Permitindo assim definir os fios (cada um do tipo `std_logic`)

```
num_aluno(16), num_aluno(15), …, num_aluno(1), num_aluno(0)
```

Exemplo: cadeado digital



O cadeado abre com o código $0111_2=7_{16}$

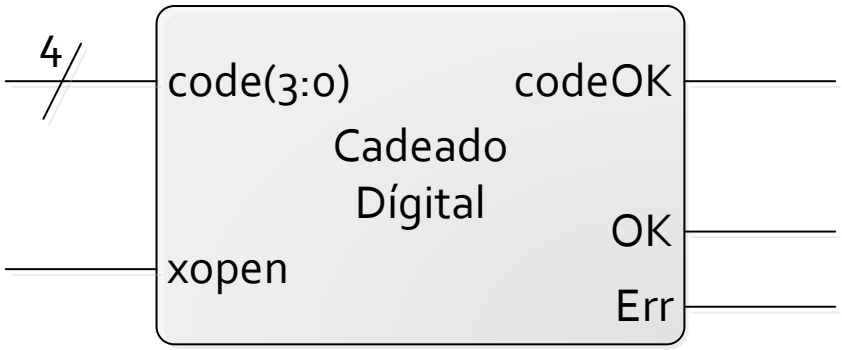
```
-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity <NOME_DO_COMPONENTE> is
   port (
         <NOME_DO_SINAL> : <IN/OUT> <TIPO_DE_SINAL>;
         ...
         <NOME_DO_SINAL> : <IN/OUT> <TIPO_DE_SINAL>
  );
end <NOME_DO_COMPONENTE>;
```

Lista de sinais separados por ";"

A ultima linha não deve ser terminada por ";"

Nota: Já que a palavra open está reservada em VHDL, não é possível representar sinais com este nome. Assim, o sinal de comando para abertura do cadeado foi renomeado para "xopen"

Exemplo: cadeado digital



O cadeado abre com o código $0111_2=7_{16}$

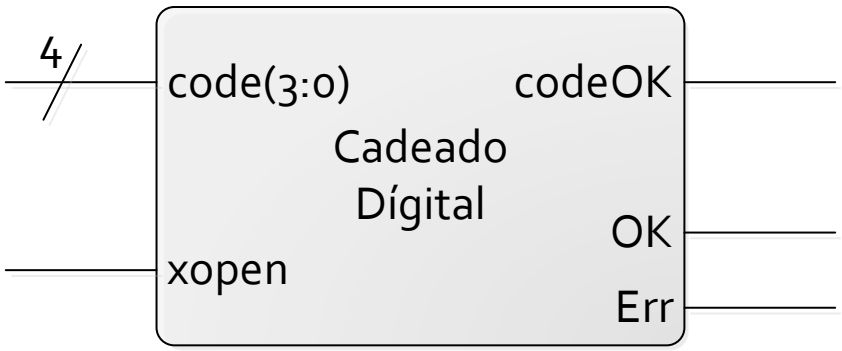
```
entity cadeado_digital is
   port (
         code    : in  std_logic_vector(3 downto 0);
         xopen   : in  std_logic;
         codeOK  : out std_logic;
         OK      : out std_logic;
         Err     : out std_logic
         );
end cadeado_digital;
```

Lista de sinais separados por ";"

A ultima linha não deve ser terminada por ";"

Nota: Já que a palavra open está reservada em VHDL, não é possível representar sinais com este nome. Assim, o sinal de comando para abertura do cadeado foi renomeado para "xopen"

Exemplo: cadeado digital

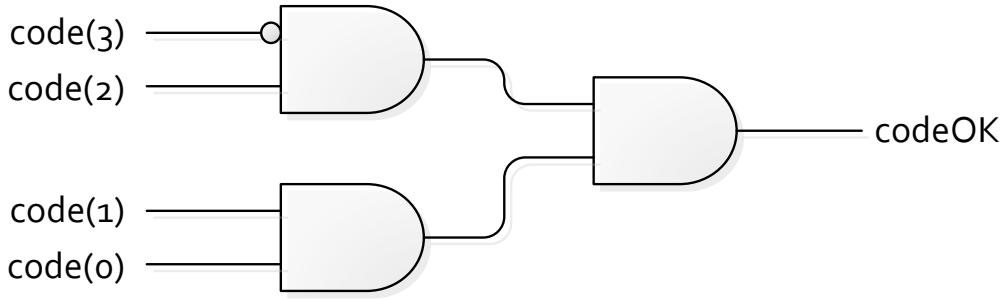


O cadeado abre com o código $0111_2=7_{16}$

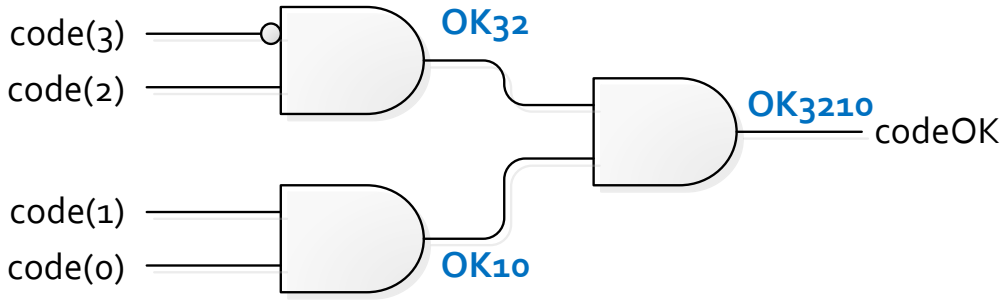
- **Descrição da implementação do circuito "`cadeado_digital`"**
  - Como se descreve a constituição interna do circuito?

Exemplo: cadeado digital

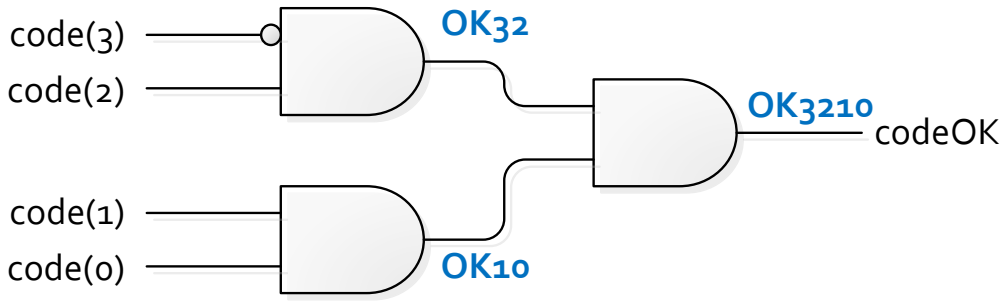
- Após projecto e desenho do logigrama correspondente, obtém-se:



- Em VHDL deverá ser dado um nome a cada um dos fios intermédios:



Exemplo: cadeado digital



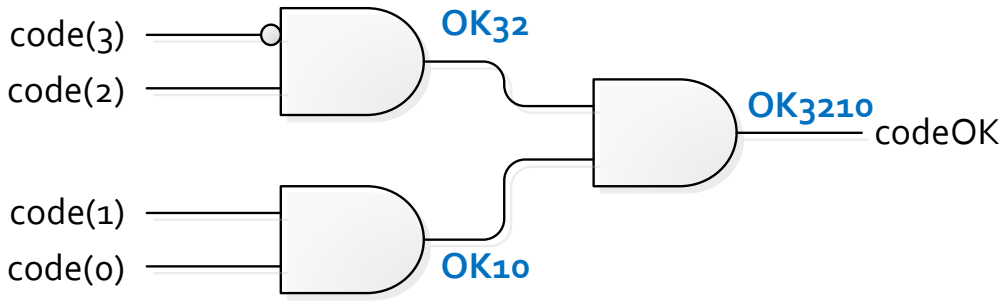
- **Assim, é necessário declarar os sinais OK32, OK10 e OK3210**

- **A declaração de sinais é realizada sob a forma:**

```
signal <NOME_DO_SINAL_1>, <NOME_DO_SINAL_2> : <TIPO_DE_SINAL>;
signal <NOME_DO_SINAL_3> : <TIPO_DE_SINAL>;
signal <NOME_DO_SINAL_4> : <TIPO_DE_SINAL>;
```

- **Naturalmente é possível ter sinais (fios) de diferentes tipos**
  - No entanto, a declaração anterior obriga a que os sinais 1 e 2 tenham o mesmo tipo

Exemplo: cadeado digital



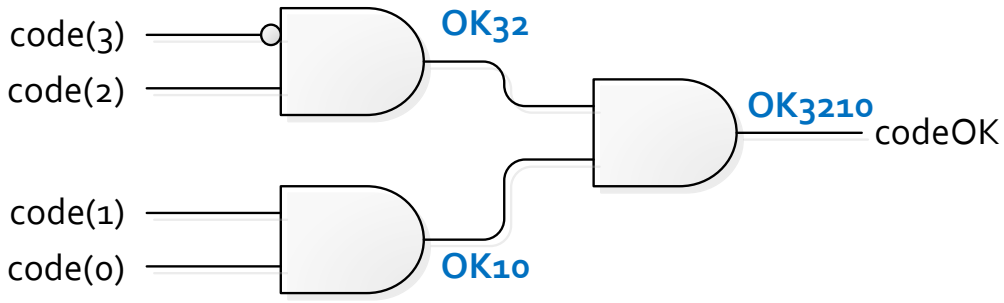
- **Assim, é necessário declarar os sinais OK32, OK10 e OK3210**

```
entity cadeado_digital is
   port (
    ...
   );
end cadeado_digital;

architecture behavior of cadeado_digital is
-- declaração dos sinais (fios) internos ao componente
signal OK32, OK10, OK3210 : std_logic;

Begin
...
end behavior;
```

NOTAS:

- O nome dado à arquitectura não é relevante...
- O Vivado normalmente usa a designação “behavior”

Exemplo: cadeado digital



- Resta-nos agora descrever o circuito projectado
- A atribuição de valores para os sinais é feita da seguinte forma:

```
NOME_SINAL_DESTINO <= OPERAÇÃO_LÓGICA_SOBRE_OPERANDOS;
```

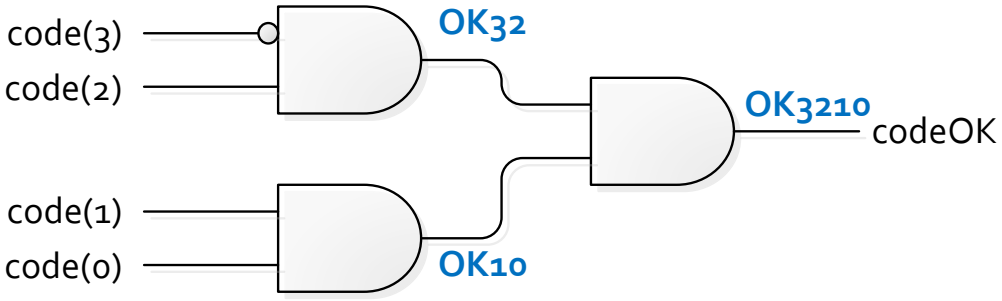
- Existem várias operações típicas:

NOT **AND** **OR** **NAND** **NOR** **XOR** ...

- Deverão ser usados parêntesis sempre que for necessário definir a ordem pela qual as operações são realizadas

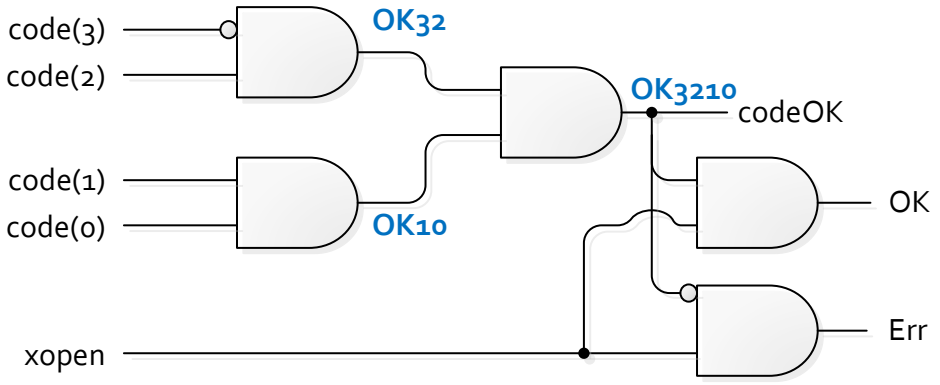
Exemplo: cadeado digital

```
...
architecture behavior of cadeado_digital is
-- declaração dos sinais (fios) internos ao componente
signal OK32, OK10, OK3210 : std_logic;
begin

-- Cálculo do resultado
OK32   <= (not code(3)) and code(2);
OK10   <= code(1) and code(0);
OK3210 <= OK32 and OK10;

-- Atribuição do valor de saída
codeOK <= OK3210;

end behavior;
```



Exemplo: cadeado digital

```
...
architecture behavior of cadeado_digital is
-- declaração dos sinais (fios) internos ao componente
signal OK32, OK10, OK3210 : std_logic;
begin

-- Cálculo do resultado
OK32   <= (not code(3)) and code(2);
OK10   <= code(1) and code(0);
OK3210 <= OK32 and OK10;

-- Atribuição do valor de saída
codeOK <= OK3210;
OK  <= OK3210 and xopen;
Err <= (not OK3210) and xopen;

end behavior;
```

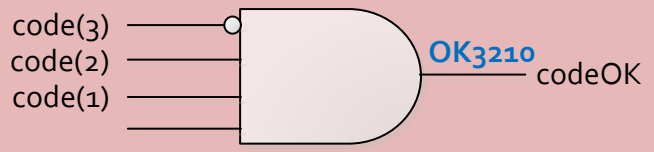


Exemplo: cadeado digital

```
...
architecture behavior of cadea
-- declaração dos sinais (fios
signal OK32, OK10, OK3210 : st
begin

-- Cálculo do resultado
OK3210 <= (not code(3)) and code(2) and code(1) and code(0);
```

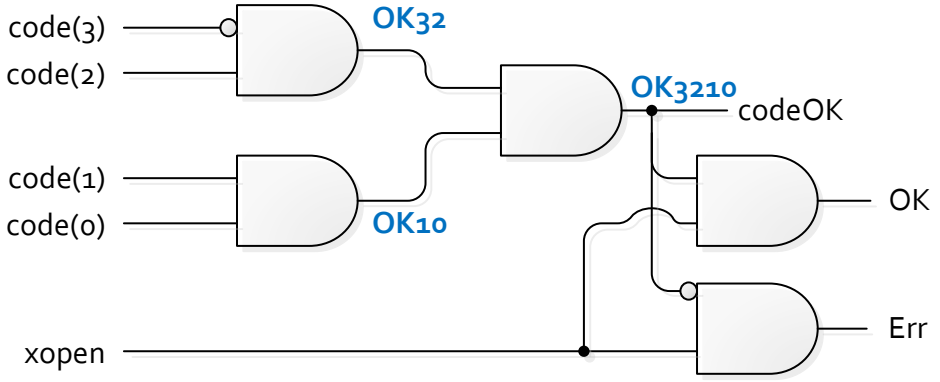
ATENÇÃO!!!

Este código tem a mesma função, mas o mapeamento do circuito para VHDL está errado,



Estes erros serão **penalizados**!!!

```
-- Atribuição do valor de saída
codeOK <= OK3210;
OK  <= OK3210 and xopen;
Err <= (not OK3210) and xopen;

end behavior;
```



Exemplo: cadeado digital

```
-- Declaração de bibliotecas com pré-definições
library IEEE;
use IEEE.std_logic_1164.all;

entity cadeado_digital is
   port (
         code   : in  std_logic_vector(3 downto 0);
         xopen  : in  std_logic;
         codeOK : out std_logic;
         OK     : out std_logic;
         Err    : out std_logic
         );
end cadeado_digital;

architecture behavior of cadeado_digital is
-- declaração dos sinais (fios) internos
-- ao componente
signal OK32, OK10, OK3210 : std_logic;
begin

-- Cálculo do resultado
OK32   <= (not code(3)) and code(2);
OK10   <= code(1) and code(0);
OK3210 <= OK32 and OK10;
```

```
-- Atribuição do valor de saída
codeOK <= OK3210;
OK  <= OK3210 and xopen;
Err <= (not OK3210) and xopen;

end behavior;
```

TÉCNICO LISBOA

CIRCUITOS COMBINATÓRIOS ELEMENTARES

Atribuições simples

```
...
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal A, B, C : std_logic;
signal vec1 : std_logic_vector(2 downto 0);
signal vec2 : std_logic_vector(2 downto 0);
...
begin
...
A <= '0'; -- atribuição do valor lógico zero
B <= '1'; -- atribuição do valor lógico um
C <= A;   -- atribuição do valor lógico dado em A

vec1 <= "011"; -- atribuição do número 3
vec2 <= A & B & C; -- atribuição do resultante da concatenação de
                   -- A, B e C. Assim vec2 tomará o valor 2.
...
end behavior;
```

CONSTRUÇÕES POSSÍVEIS:

Poderão ser usadas quaisquer construções que tenham um mapeamento directo no logigrama original!

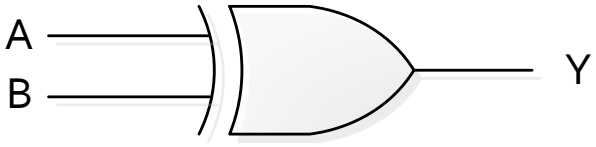
Portas lógicas elementares


```
Y <= A and B and C;
```

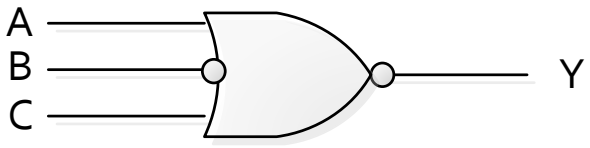
CONSTRUÇÕES POSSÍVEIS:

Poderão ser usadas quaisquer construções que tenham um mapeamento directo no logigrama original!


```
Y <= not (A and B and C);
```

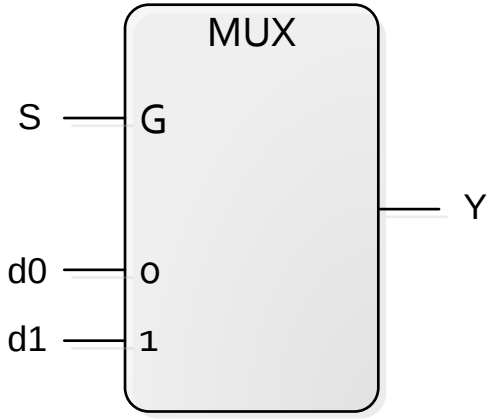



```
Y <= A xor B;
```




```
Y <= not (A or (not B) or C);
```

Multiplexer 2:1 de 1 bit



CONSTRUÇÕES POSSÍVEIS:

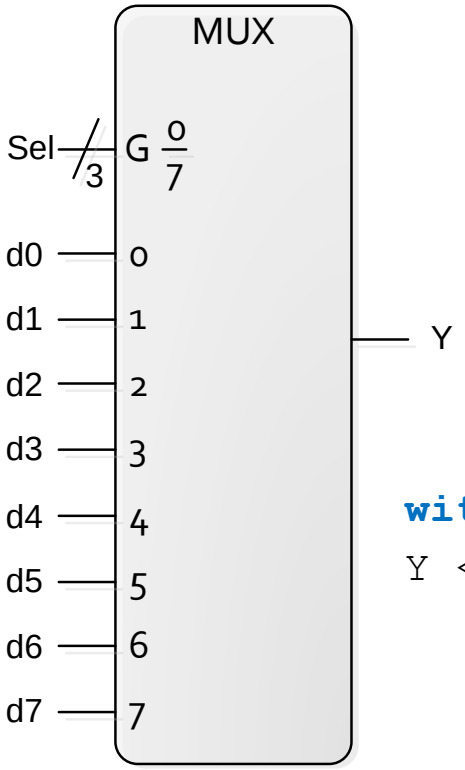
Poderão ser usadas quaisquer construções que tenham um mapeamento directo no logigrama original!

```
Y <= d0 when S='0' else d1;
```

OU

```
with S select
    Y <= d0 when '0',
         d1 when others;
```

Multiplexer 8:1 de 1 bit



CONSTRUÇÕES POSSÍVEIS:

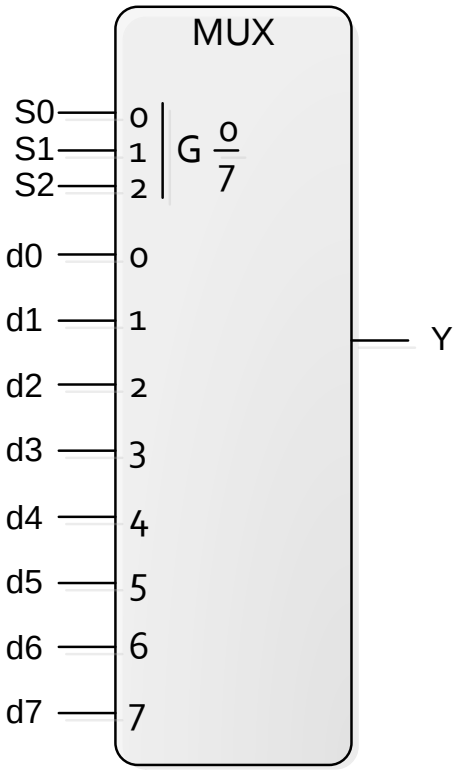
Poderão ser usadas quaisquer construções que tenham um mapeamento directo no logigrama original!

```
with Sel select
Y <= d0 when "000",
     d1 when "001",
     d2 when "010",
     d3 when "011",
     d4 when "100",
     d5 when "101",
     d6 when "110",
     d7 when others;
```

OU

```
Y <= d0 when Sel="000" else
     d1 when Sel="001" else
     d2 when Sel="010" else
     d3 when Sel="011" else
     d4 when Sel="100" else
     d5 when Sel="101" else
     d6 when Sel="110" else
     d7;
```

Multiplexer 8:1 de 1 bit

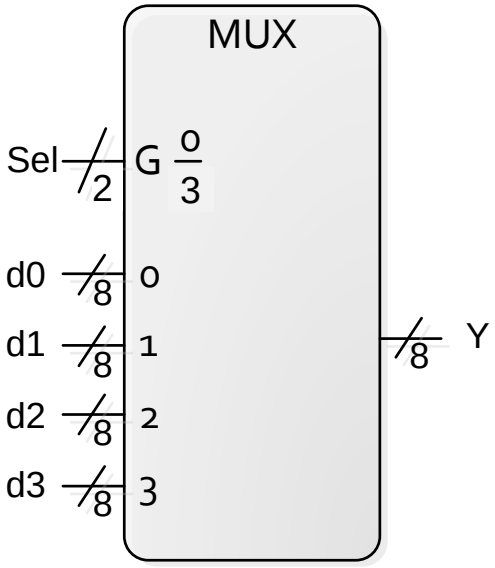


CONSTRUÇÕES POSSÍVEIS:

Poderão ser usadas quaisquer construções que tenham um mapeamento directo no logigrama original!

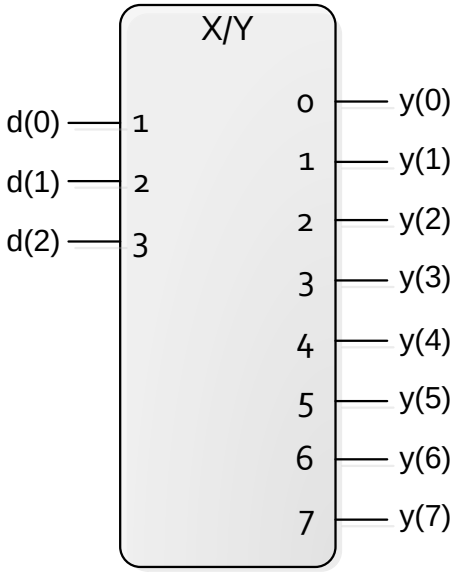
```
...
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal sel : std_logic_vector(2 downto 0);
...
begin
...
-- cálculo (concatenação) do sinal de selecção
Sel <= S2 & S1 & S0;
-- multiplexer 8:1... Ver código do slide anterior
...

end behavior;
```

Multiplexer 4:1 de 8 bits

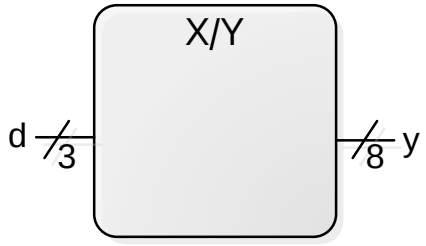



```
...
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal sel : std_logic_vector(1 downto 0);
signal d0,d1,d2,d3,y: std_logic_vector(7 downto 0);
...
begin
...
-- Descrição de 8 multiplexers 4:1, 1 por cada bit.
-- Requer que o número de bits de y, d0, d1, d2 e
-- d3 seja a mesma (neste caso 8 bits)
y <= d0 when sel="00" else
     d1 when sel="01" else
     d2 when sel="10" else
     d3;

end behavior;
```

Descodificador 3:8



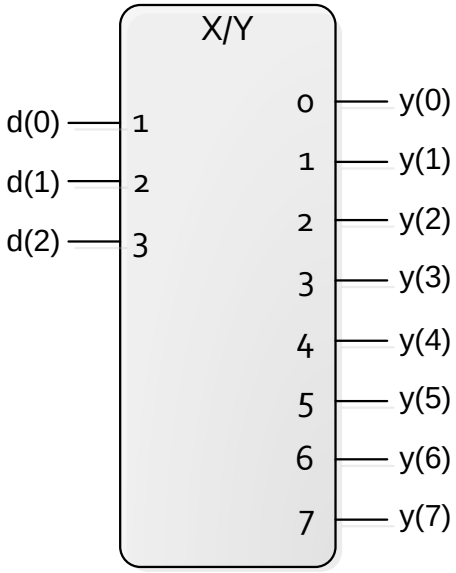
Equivalente a:



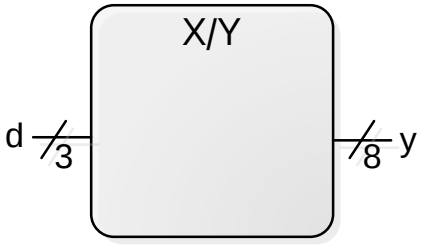

```
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal d : std_logic_vector(2 downto 0);
signal y : std_logic_vector(7 downto 0);
...
begin
...
-- Descrição de 1 descodificador 3:8
y <= "00000001" when d="000" else
     "00000010" when d="001" else
     "00000100" when d="010" else
     "00001000" when d="011" else
     "00010000" when d="100" else
     "00100000" when d="101" else
     "01000000" when d="110" else
     "10000000";

end behavior;
```

Descodificador 3:8

SOLUÇÃO ALTERNATIVA

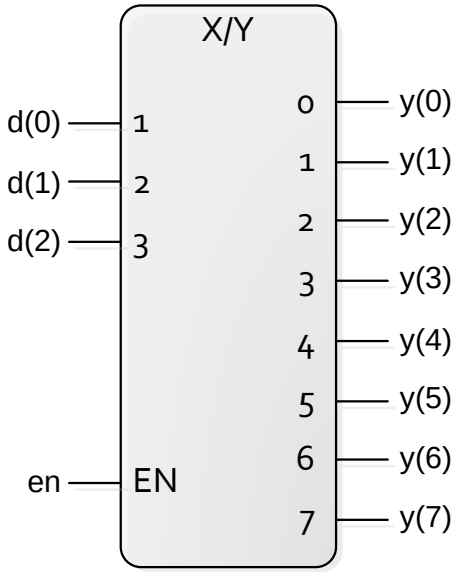


Equivalente a:

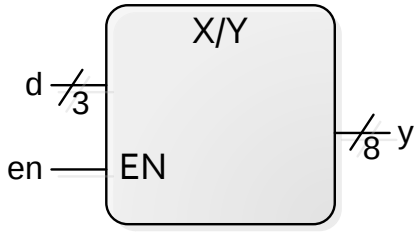



```
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal d : std_logic_vector(2 downto 0);
signal y : std_logic_vector(7 downto 0);
...
begin
...
-- Descrição de 1 descodificador 3:8
With d select
    y <= "00000001" when "000",
         "00000010" when "001",
         "00000100" when "010",
         "00001000" when "011",
         "00010000" when "100",
         "00100000" when "101",
         "01000000" when "110",
         "10000000" when others;

end behavior;
```

Descodificador 3:8 com enable



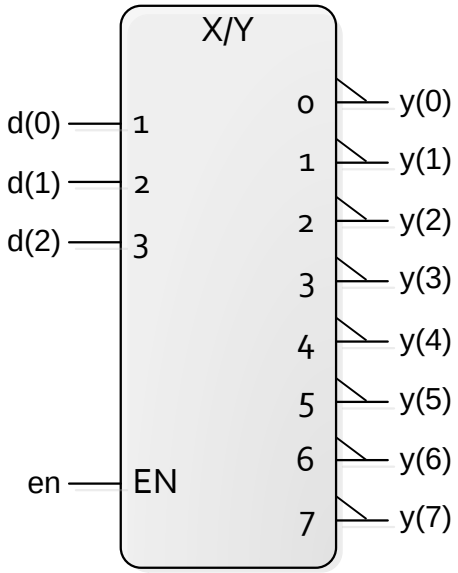
Equivalente a:



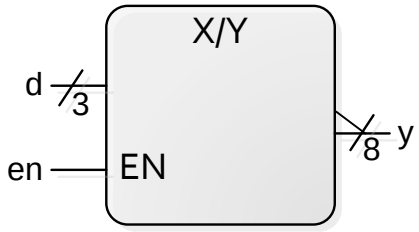

```
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal d : std_logic_vector(2 downto 0);
signal y : std_logic_vector(7 downto 0);
...
begin
...
-- Descrição de 1 descodificador 3:8
y <= "00000000" when en='0' else
     "00000001" when d="000" else
     "00000010" when d="001" else
     "00000100" when d="010" else
     "00001000" when d="011" else
     "00010000" when d="100" else
     "00100000" when d="101" else
     "01000000" when d="110" else
     "10000000";

end behavior;
```

Enable

Descodificador 3:8 com enable (saidas negadas)



Equivalente a:




```
architecture behavior of meu_circuito is
-- declaração do sinal de selecção
signal d : std_logic_vector(2 downto 0);
signal y : std_logic_vector(7 downto 0);
...
begin
...
-- Descrição de 1 descodificador 3:8
y <= "11111111" when en='0' else
     "11111110" when d="000" else
     "11111101" when d="001" else
     "11111011" when d="010" else
     "11110111" when d="011" else
     "11101111" when d="100" else
     "11011111" when d="101" else
     "10111111" when d="110" else
     "01111111";

end behavior;
```

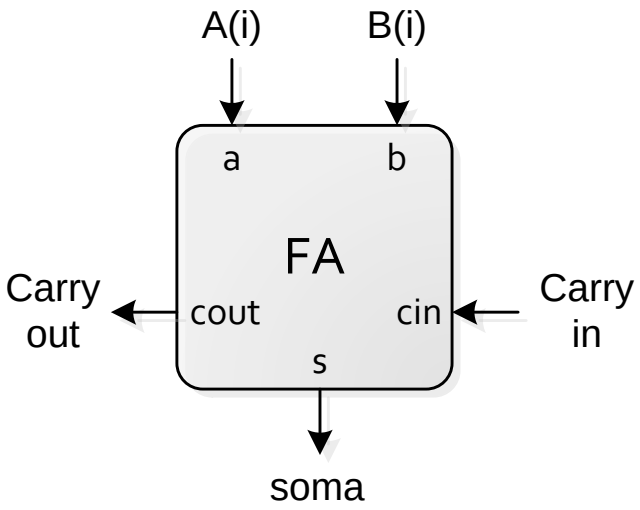
Enable

Saídas Negadas

EXEMPLO:
CIRCUITO SOMADOR/SUBTRACTOR

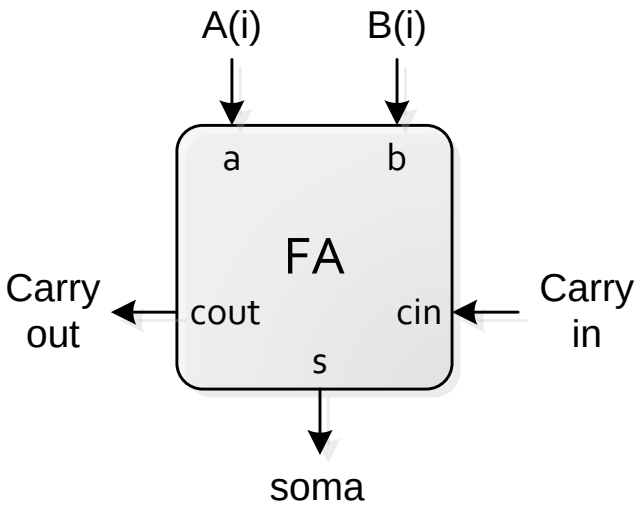
Full-Adder

- **O elemento básico de um circuito somador/subtractor é:**
  - Full-Adder



Full-Adder

1. Identificação do componente:

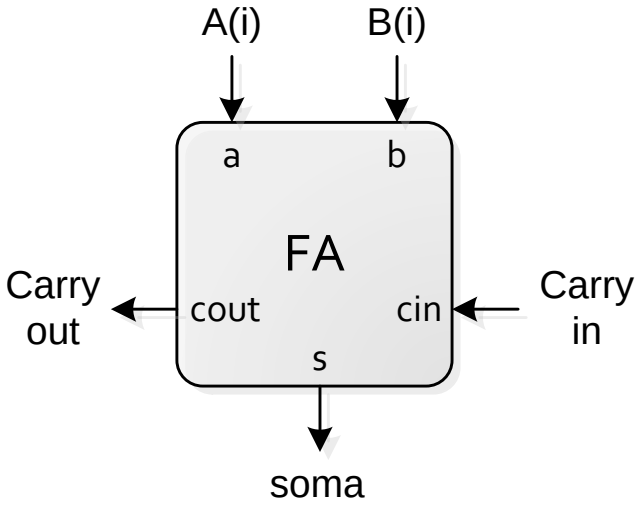


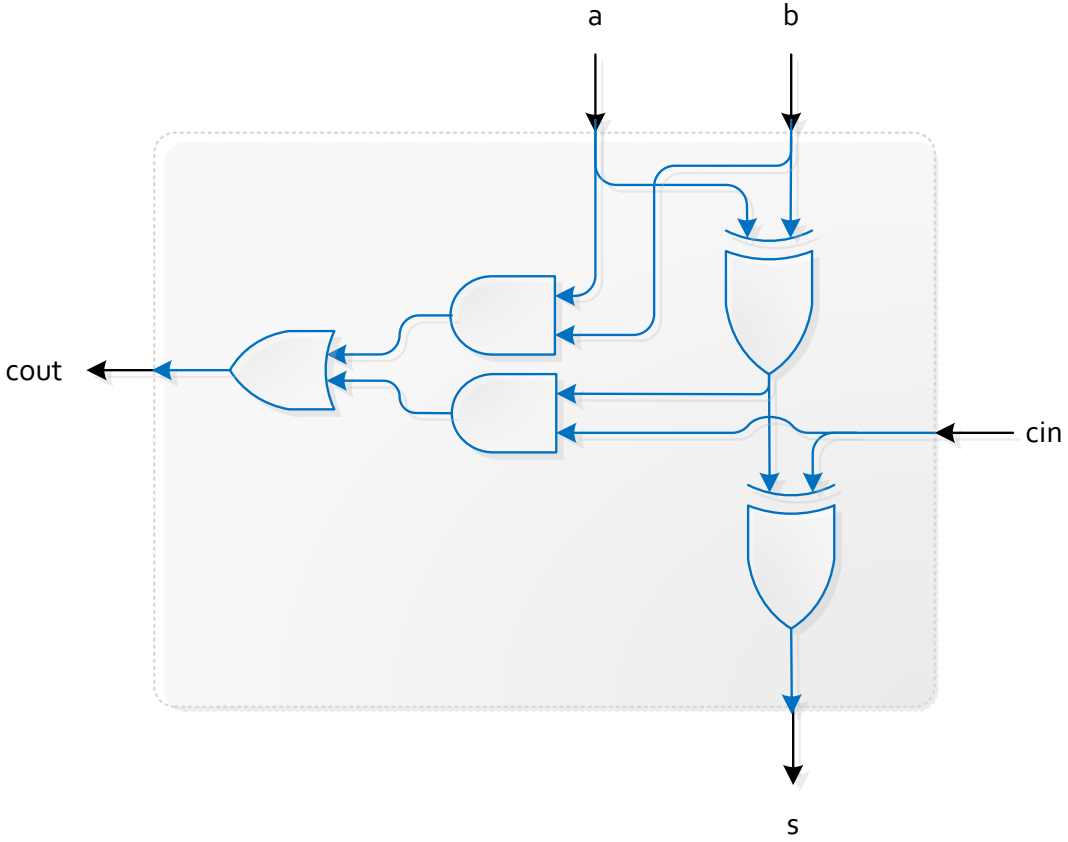

```
-- FICHEIRO full_adder.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity full_adder is
   port (
         a    : in  std_logic;
         b    : in  std_logic;
         cin  : in  std_logic;
         s    : out std_logic;
         cout : out std_logic
);
end full_adder;
```

Full-Adder

2. Desenho do logigrama interno do full-adder:

Nota: o logigrama podia ser alterado de forma a usar só portas NAND e XOR

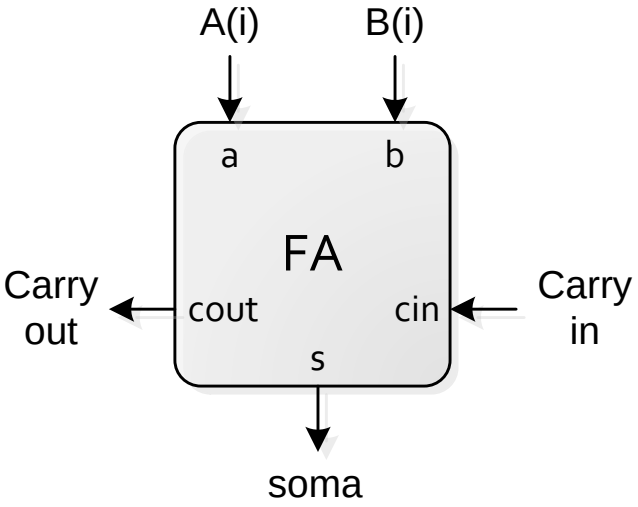


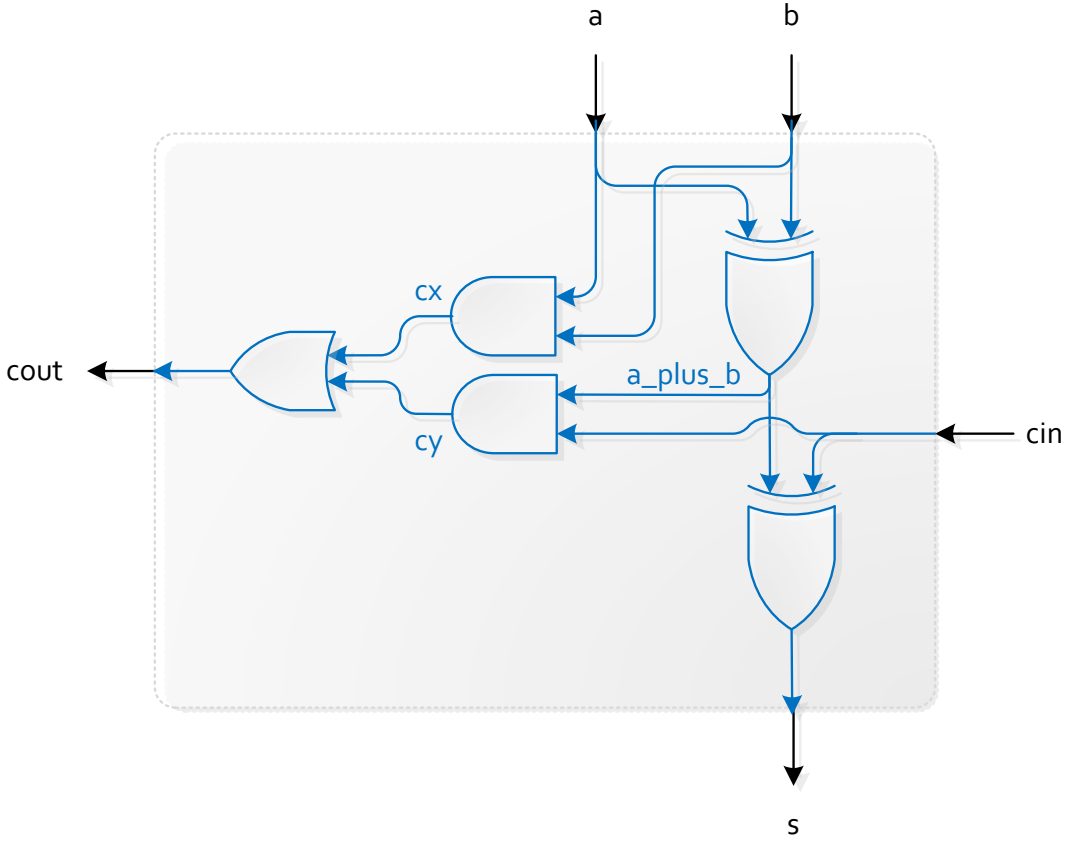


Full-Adder

2. Desenho do logigrama interno do full-adder:

a) é necessário identificar no esquema o nome dos sinais (fios) internos

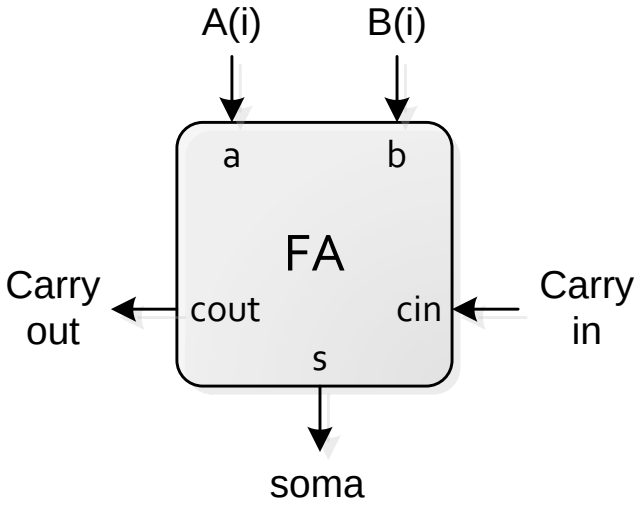


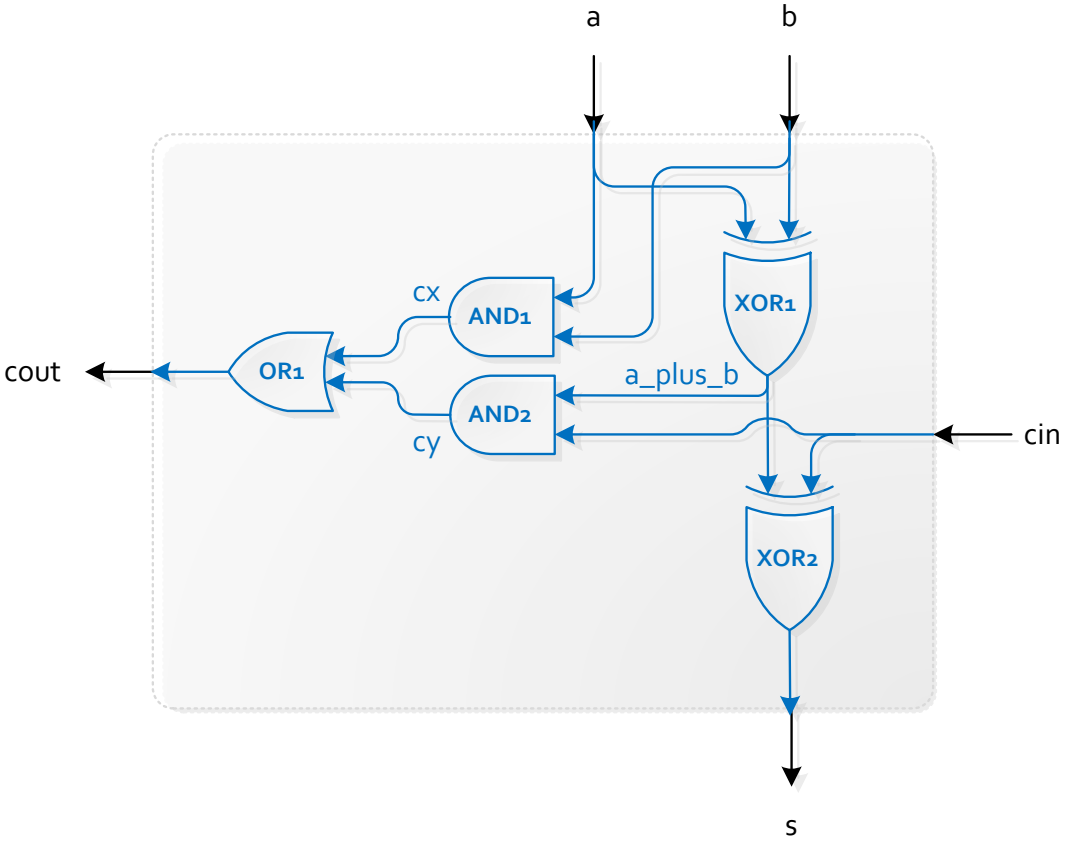


Full-Adder

2. Desenho do logigrama interno do full-adder:

b) vamos ainda identificar cada uma das portas lógicas

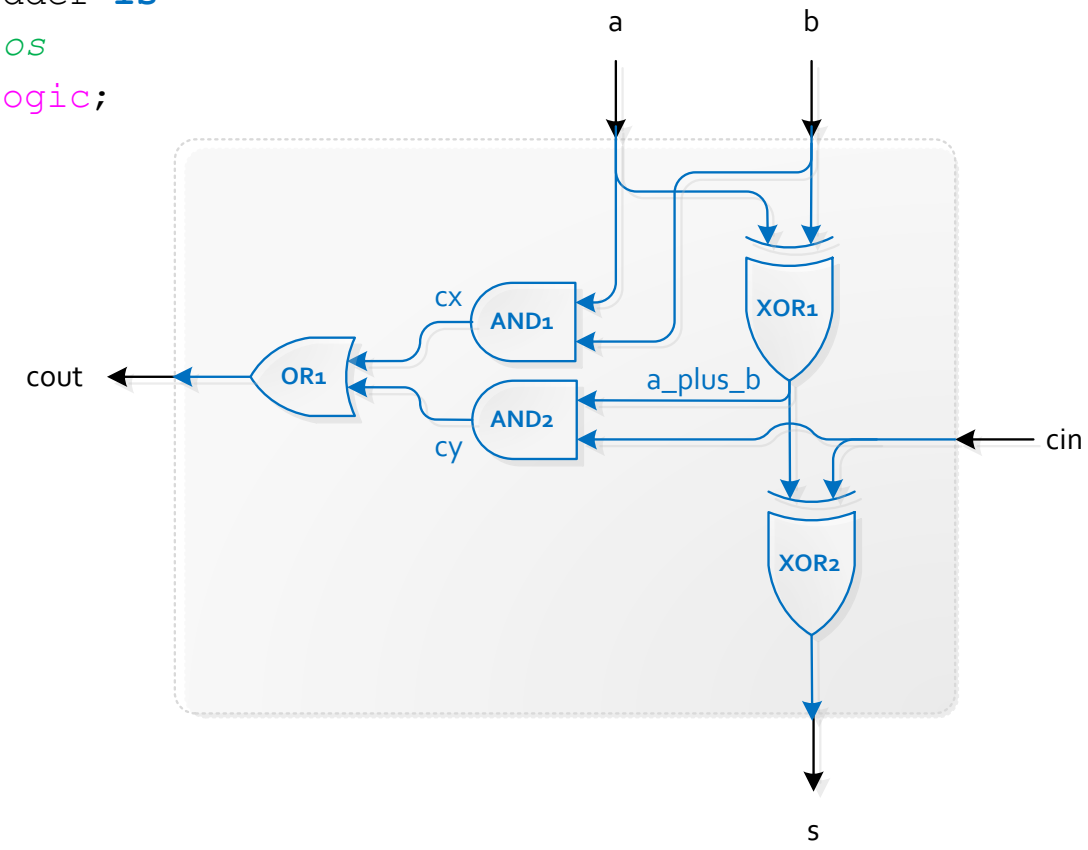




Full-Adder

3. Descrição da arquitectura do circuito `full_adder`

```
architecture behavior of full_adder is
-- declaração dos sinais internos
signal a_plus_b, cx, cy : std_logic;

begin
-- XOR1
a_plus_b <= a xor b;
-- XOR2
s <= cin xor a_plus_b;
-- AND1
cx <= a and b;
-- AND2
cy <= a_plus_b and cin;
-- OR1
cout <= cx or cy;

end behavior;
```



Full-Adder

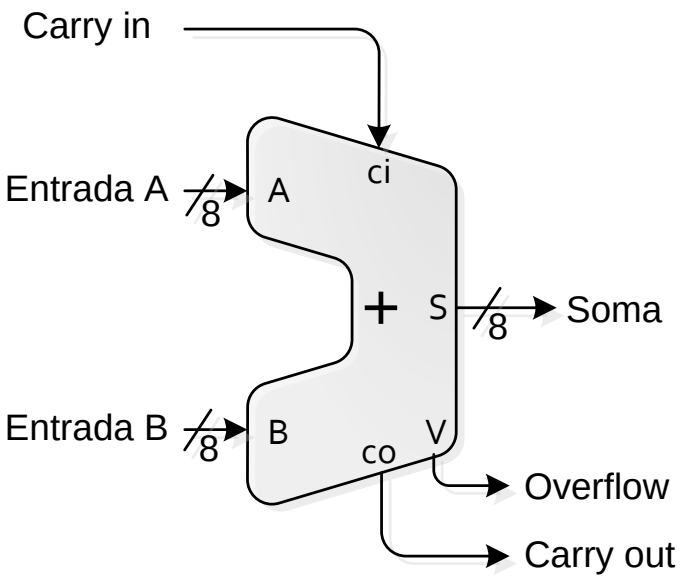
3. Descrição da arquitectura do circuito `full_adder`

```
-- FICHEIRO full_adder.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity full_adder is
   port (
         a    : in  std_logic;
         b    : in  std_logic;
         cin  : in  std_logic;
         s    : out std_logic;
         cout : out std_logic
);
end full_adder;
```

```
architecture behavior of full_adder is
-- declaração dos sinais internos
signal a_plus_b, cx, cy : std_logic;

begin
-- XOR1
a_plus_b <= a xor b;
-- XOR2
s <= cin xor a_plus_b;
-- AND1
cx <= a and b;
-- AND2
cy <= a_plus_b and cin;
-- OR1
cout <= cx or cy;

end behavior;
```

Somador de 8-bits com carry in

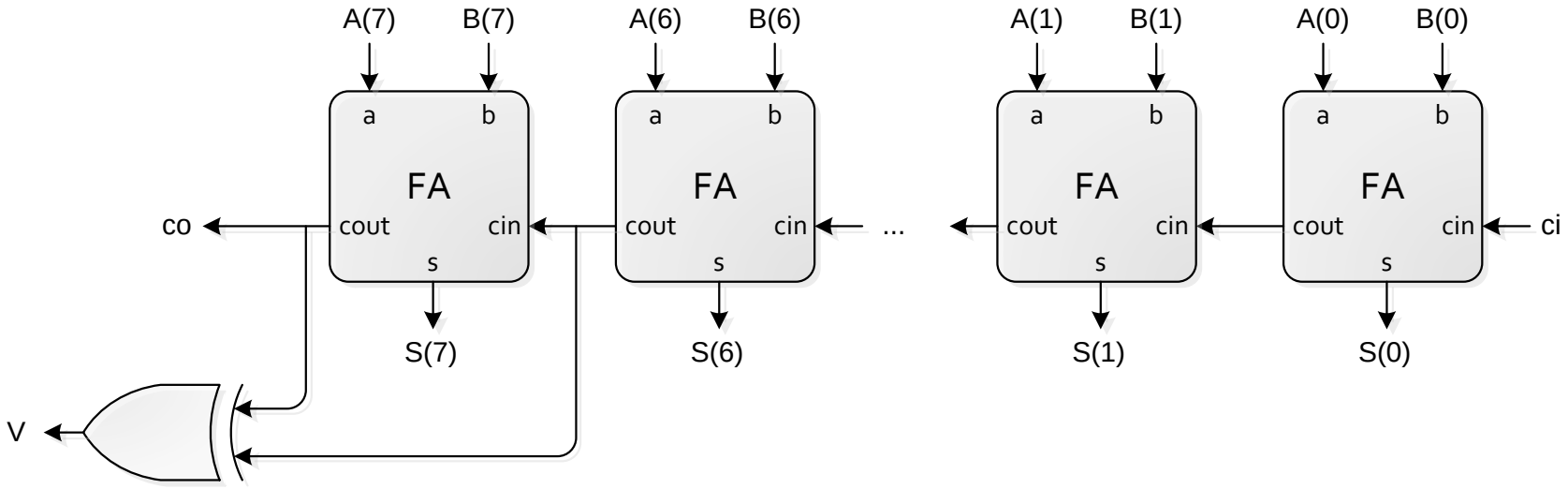
- **Usando o circuito full-adder (full_adder.vhd) é possível construir um somador de 8-bits com carry in**




```
-- FICHEIRO somador8.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity somador8 is
   port (
         A  : in  std_logic_vector(7 downto 0);
         B  : in  std_logic_vector(7 downto 0);
         ci : in  std_logic;
         S  : out std_logic_vector(7 downto 0);
         co : out std_logic;
         V  : out std_logic
         );
end somador8;
...
```

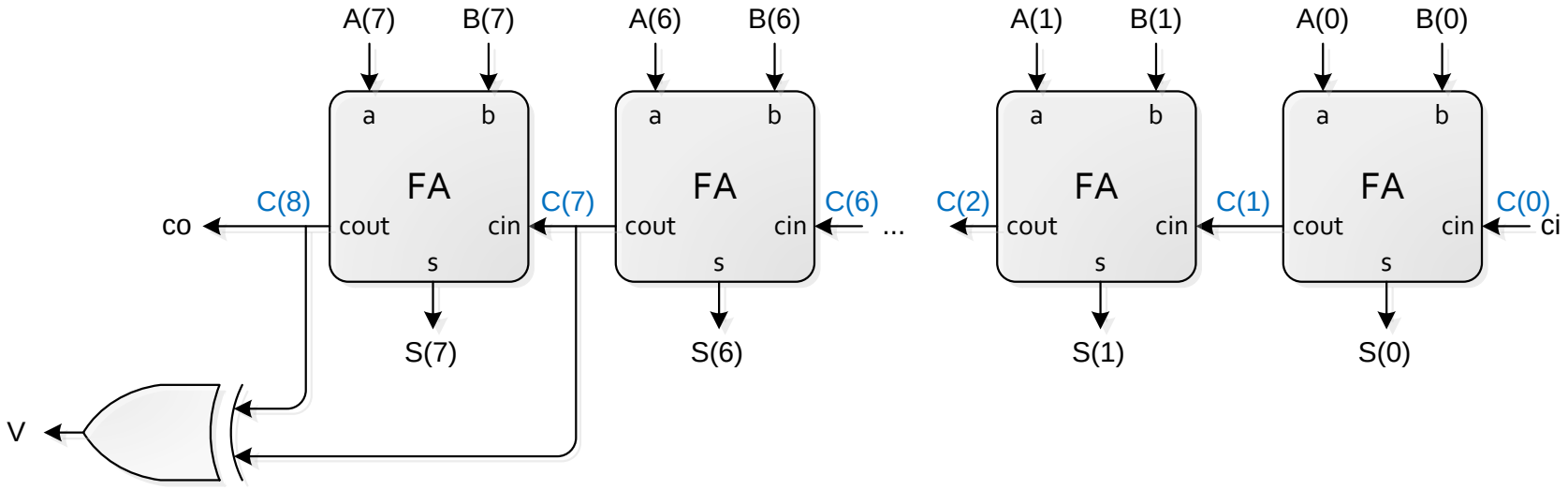
Somador de 8-bits com carry in

- **Logigrama do circuito somador de 8-bits**



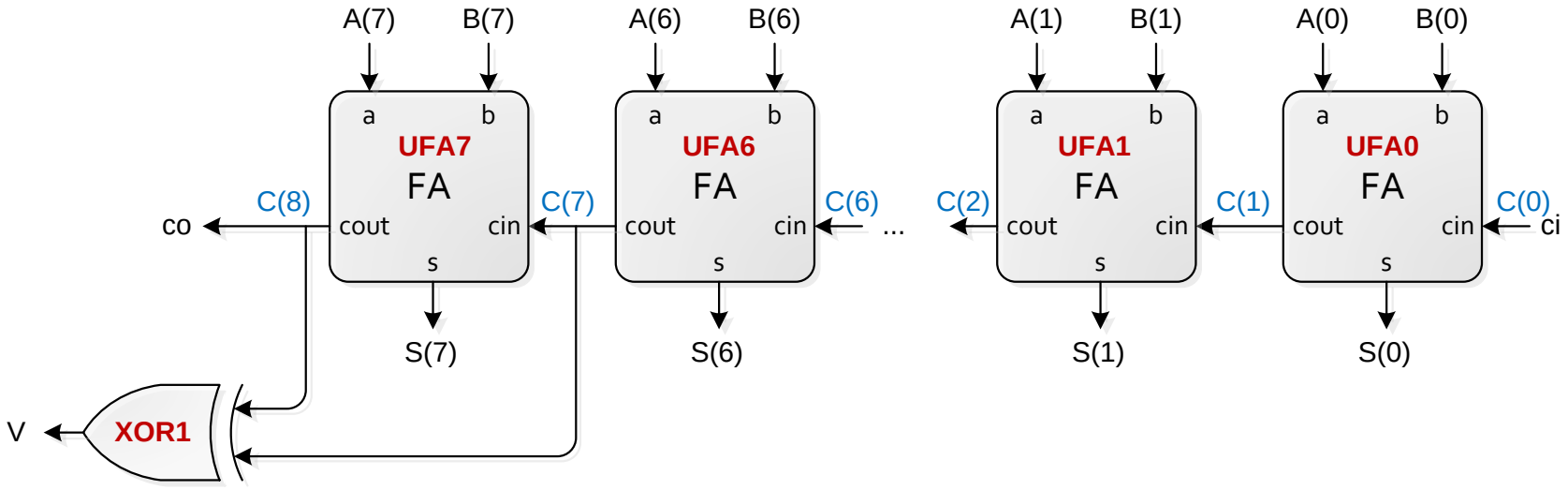
Somador de 8-bits com carry in

- **Logigrama do circuito somador de 8-bits**
  - Sinais internos (azul)



Somador de 8-bits com carry in

- **Logigrama do circuito somador de 8-bits**
  - Sinais internos (azul)
  - Nome das instancias (vermelho)



Somador de 8-bits com carry in

- **Descrição do circuito somador de 8-bits**
  - Declaração do componente full_adder: para utilizar um componente descrito num ficheiro .vhd é necessário declarar a sua existência. A declaração de um componente é a seguinte:

```
-- Declaração da entidade, colocada entre
-- "architecture" e "begin"
component <NOME_DO_COMPONENTE>
   port (
      <SINAL1>  : <IN/OUT> <TIPO_DE_SINAL> ;
      ...
      <SINALN> : <IN/OUT> <TIPO_DE_SINAL>
  );
end component;
```

```
-- Definição da entidade, colocada
-- no topo do ficheiro .vhd
entity NOME_DO_COMPONENTE is
   port (
      <SINAL1> : <IN/OUT> <TIPO_DE_SINAL>;
      ...
      <SINALN> : <IN/OUT> <TIPO_DE_SINAL>
  );
end NOME_DO_COMPONENTE;
```

  - A declaração de um componente é uma cópia quase perfeita da descrição da entidade. As únicas diferenças residem na primeira e última linhas

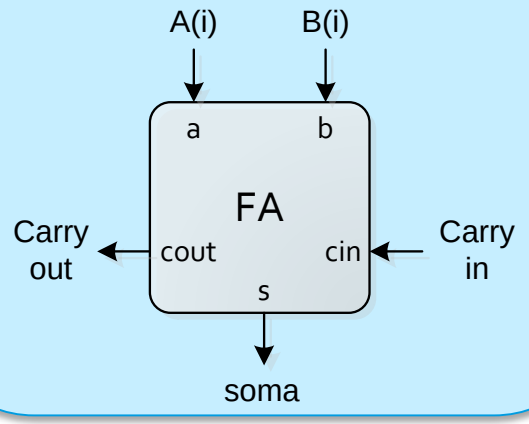
Somador de 8-bits com carry in

Descrição do circuito somador de 8-bits

- Declaração do componente full_adder:

```
architecture behavior of somador8 is
-- declaração do componente full_adder
component full_adder
   port (
         a     : in  std_logic;
         b     : in  std_logic;
         cin   : in  std_logic;
         s     : out std_logic;
         cout  : out std_logic
);
end component;
-- declaração dos sinais internos
signal C : std_logic_vector(8 downto 0);

begin
...
```

Componente **full_adder** descrito no ficheiro **full_adder.vhd**



Nota: a declaração de um componente é uma cópia quase perfeita da descrição da entidade.

Somador de 8-bits com carry in

- **Descrição do circuito somador de 8-bits**
  - Arquitectura do circuito somador:

```
architecture behavior of somador8 is
-- declaração do componente full_adder
...
-- declaração dos sinais internos
signal C : std_logic_vector(8 downto 0);

begin
-- atribuição do valor de C(0)
C(0) <= ci;
-- Utilização de 8 instâncias do componente full_adder
UFA0: full_adder port map( a=>A(0) , b=>B(0) , cin=>C(0) , s=>S(0) , cout=>C(1) );
UFA1: full_adder port map( a=>A(1) , b=>B(1) , cin=>C(1) , s=>S(1) , cout=>C(2) );
UFA2: full_adder port map( a=>A(2) , b=>B(2) , cin=>C(2) , s=>S(2) , cout=>C(3) );
...
UFA7: full_adder port map( a=>A(7) , b=>B(7) , cin=>C(7) , s=>S(7) , cout=>C(8) );

-- sinais adicionais de saida
co <= C(8);
V <= C(7) xor C(8);
end behavior;
```

Somador de 8-bits com carry in

Descrição do circuito somador de 8-bits

- Arquitectura do circuito somador:

```
architecture behavior of somador8 is
-- declaração do componente full_adder
...
-- declaração dos sinais internos
signal C : std_logic_vector(8 downto 0);
begin
-- atribuição do valor de C(0)
C(0) <= ci;
-- Instanciação automática de 8 componentes (0 ... 7)
uGen1 : for i in 0 to 7 generate
   UFA: full_adder port map(
            a=>A(i) , b=>B(i) , cin=>C(i) ,
            s=>S(i) , cout=>C(i+1)
   );
end generate;
-- sinais adicionais de saida
co <= C(8);
V <= C(7) xor C(8);
end behavior;
```

SOLUÇÃO ALTERNATIVA

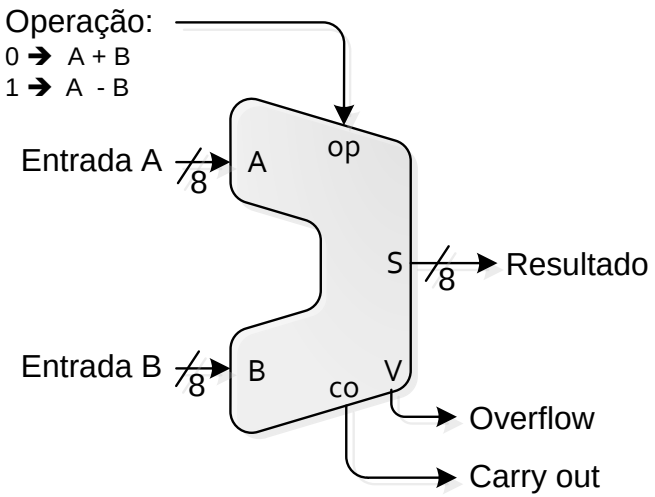
Somador de 8-bits com carry in

```
-- FICHEIRO somador8.vhd
-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;
-- Definição da entidade/sinais de entrada/saída
entity somador8 is
   port (
        A  : in  std_logic_vector(7 downto 0);
        B  : in  std_logic_vector(7 downto 0);
        ci : in  std_logic;
        S  : out std_logic_vector(7 downto 0);
        co : out std_logic;
        V  : out std_logic
        );
end somador8;

architecture behavior of somador8 is
-- declaração do componente full_adder
component full_adder
   port (
        a    : in  std_logic;
        b    : in  std_logic;
        cin  : in  std_logic;
        s    : out std_logic;
        cout : out std_logic
);
end component;
```

```
-- declaração dos sinais internos
--   C(i)   corresponde ao carry-out do full-adder
--          i-1, e serve como carry-in do
--          full-adder i
signal C : std_logic_vector(8 downto 0);

begin

-- atribuição do valor de C(0)
C(0) <= ci;

-- simplificação (opcional) da utilização das
-- várias instancias
uGen1 : for i in 0 to 7 generate
   UFA: full_adder port map(
           a=>A(i) , b=>B(i) , cin=>C(i) ,
           s=>S(i) , cout=>C(i+1)
   );
end generate;
-- sinais adicionais de saida
co <= C(8);
V <= C(7) xor C(8);

end behavior;
```

Somador/Subtractor

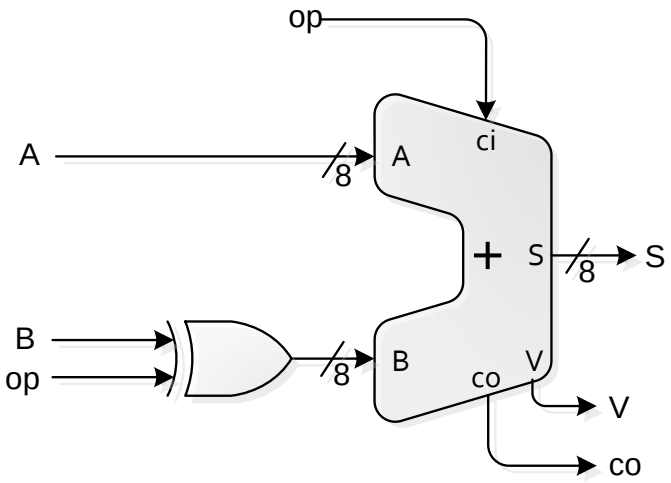
- **Usando o somador de 8 bits (somador8.vhd) é possível construir o circuito somador/subtractor**



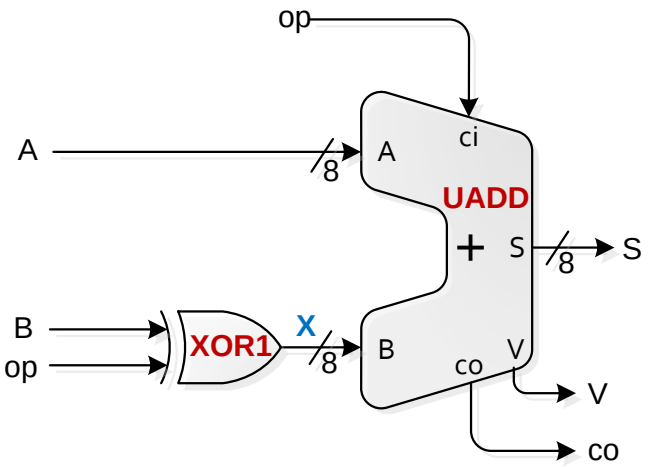

```
-- FICHEIRO arithmetic_unit.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity arithmetic_unit is
   port (
         A  : in  std_logic_vector(7 downto 0);
         B  : in  std_logic_vector(7 downto 0);
         Op : in  std_logic;
         S  : out std_logic_vector(7 downto 0);
         co : out std_logic;
         V  : out std_logic
         );
end arithmetic_unit;
...
```

Somador/Subtractor

- Logigrama do circuito somador/subtractor



Após nomeação dos sinais e circuitos internos

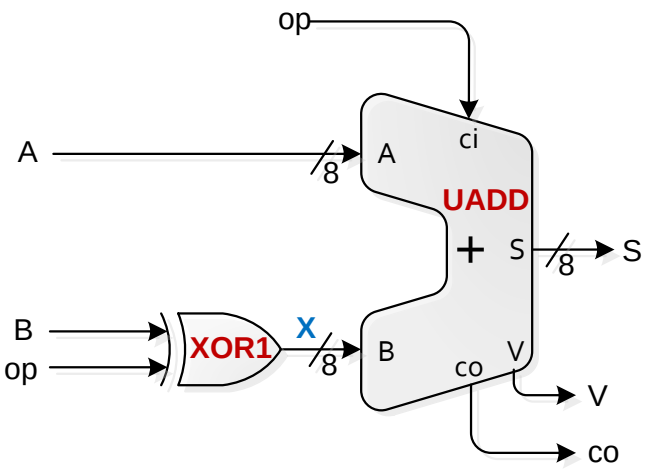


Somador/Subtractor

Logigrama do circuito somador/subtractor

- Declaração do componente **somador8**

```
architecture behavior of arithmetic_unit is
-- declaração do componente somador8
component somador8
   port (
        A  : in  std_logic_vector(7 downto 0);
        B  : in  std_logic_vector(7 downto 0);
        ci : in  std_logic;
        S  : out std_logic_vector(7 downto 0);
        co : out std_logic;
        V  : out std_logic
   );
end component;
-- declaração dos sinais internos
signal X : std_logic_vector(7 downto 0);

begin
...
```



Somador/Subtractor

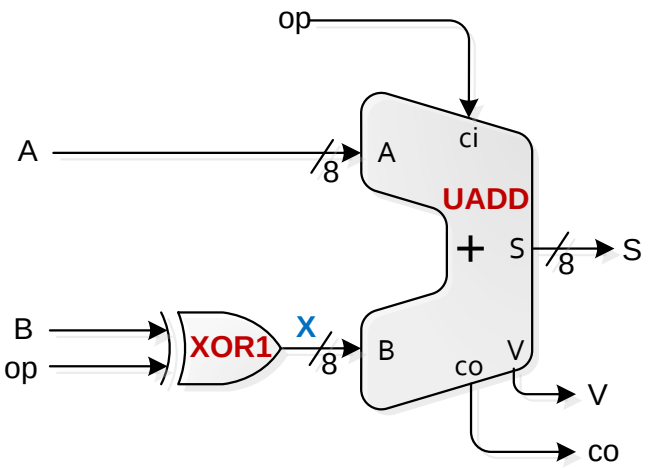
Arquitectura (VHDL) do circuito

```
architecture behavior of arithmetic_unit is
-- declaração do componente somador8
...
-- declaração dos sinais internos
signal X : std_logic_vector(7 downto 0);

begin
-- 8 portas XOR, uma por cada bit, onde cada
-- porta i tem como entradas os bits B(i) e op
X <= B xor (7 downto 0 => op);
-- utilização de uma instancia do componente
-- somador8
UADD: somador8 port map(
          A => A, B => X, ci => op,
          S => S, V => V, co => co
        );
end behavior;
```

Nota: a declaração:

`(n-1 downto 0 => X)`

cria um sinal de n bits, onde todos os bits têm o mesmo valor que o sinal `X`.



Somador/Subtractor (arithmetic_unit.vhd)

```
-- FICHEIRO arithmetic_unit.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade e
-- dos sinais (fios) de entrada/saída
entity arithmetic_unit is
   port (
        A  : in  std_logic_vector(7 downto 0);
        B  : in  std_logic_vector(7 downto 0);
        Op : in  std_logic;
        S  : out std_logic_vector(7 downto 0);
        co : out std_logic;
        V  : out std_logic
        );
end arithmetic_unit;
```

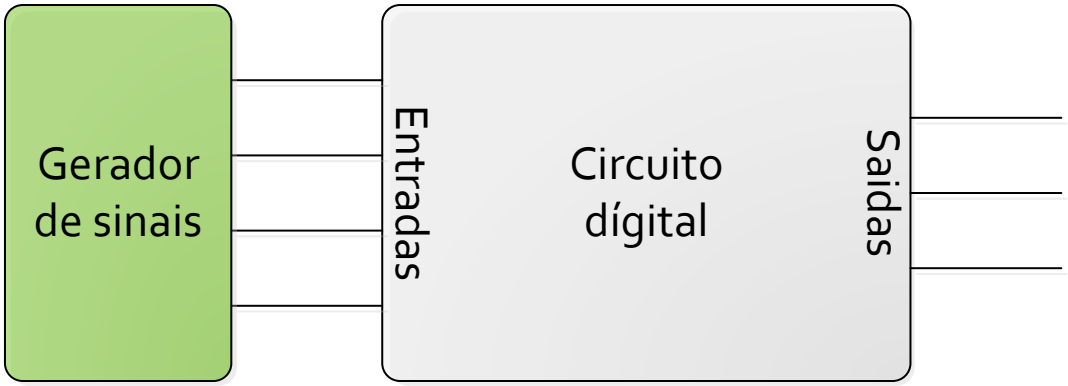
```
architecture behavior of arithmetic_unit is
-- declaração do componente somador8
component somador8
   port (
        A  : in  std_logic_vector(7 downto 0);
        B  : in  std_logic_vector(7 downto 0);
        ci : in  std_logic;
        S  : out std_logic_vector(7 downto 0);
        co : out std_logic;
        V  : out std_logic
   );
end component;
-- declaração dos sinais internos
signal X : std_logic_vector(7 downto 0);

begin
-- 8 portas XOR, uma por cada bit
X <= B xor (7 downto 0 => op);
-- instancia do componente somador8
UADD: somador8 port map(
         A => A, B => X, ci => op,
         S => S, V => V, co => co
       );
end behavior;
```

SIMULAÇÃO DE CIRCUITOS

Criação de um ficheiro VHDL para simular e validar o funcionamento dos circuitos anteriormente descritos

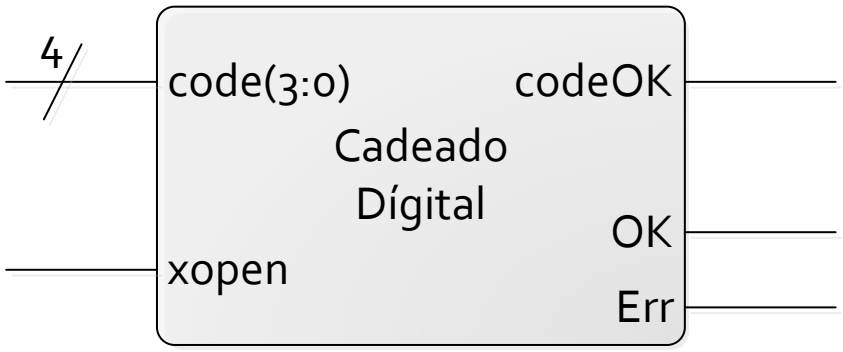
Simulação e teste de circuitos

- **Para validar correctamente o funcionamento de um circuito digital é necessário verificar o valor das saídas do circuito para todas as combinações de entradas**
  - É preciso desenvolver um módulo capaz de gerar os sinais de entrada do circuito digital



  - O gerador de sinais deve gerar todas as combinações de sinais de entrada

Simulação e teste do circuito "Cadeado Digital"
Tabela de verdade



O cadeado abre com o código $0111_2=7_{16}$.

O gerador de sinais a projectar deve ser capaz de gerar todas as combinações de entradas da tabela de verdade de forma a ser possível verificar o valor da saída

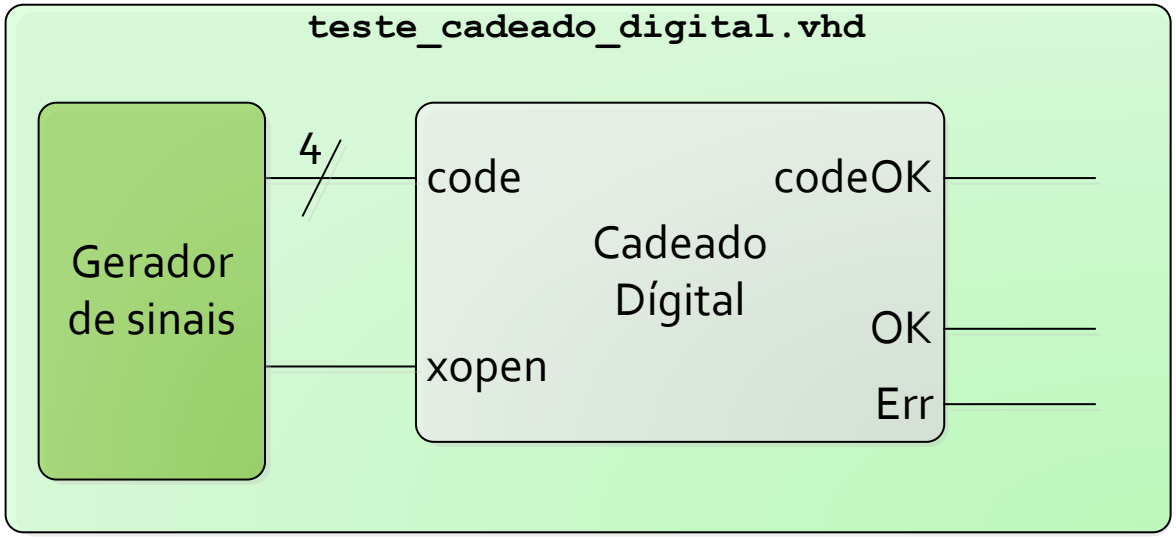
- **Tabela de verdade:**

ENTRADAS — SAIDAS ESPERADAS

code	xopen	codeOK	OK	Err
0000	0	0	0	0
0001	0	0	0	0
…	…	…	…	…
0110	0	0	0	0
0111	0	1	0	0
1000	0	0	0	0
…	…	…	…	…
0110	1	0	0	1
0111	1	1	1	0
1000	1	0	0	1
…	…	…	…	…

Estrutura do ficheiro de simulação

- **Como se pode ver no diagrama abaixo, o ficheiro VHDL para simulação (e teste) de circuitos:**
  - Necessita uma instância do circuito a testar
  - Necessita de gerar os sinais de dados e/ou controlo do circuito
  - Não necessita de entradas ou saídas



Descrição da entidade

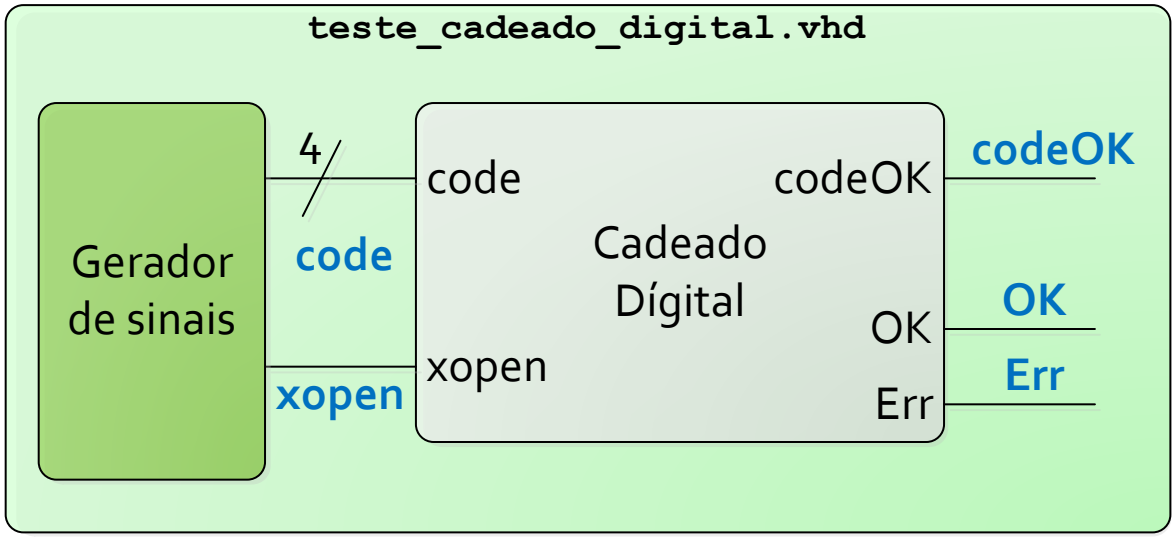
Descrição da entidade

- Sem entradas/saídas

```
-- FICHEIRO cadeado_digital_testbench.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;

-- Definição do nome da entidade, sem qualquer entrada ou saida
entity cadeado_digital_testbench is
end cadeado_digital_testbench;

architecture behavior of cadeado_digital_testbench is
...
```

Descrição da arquitectura

- **Descrição da arquitectura**
  - Declaração de componentes e sinais

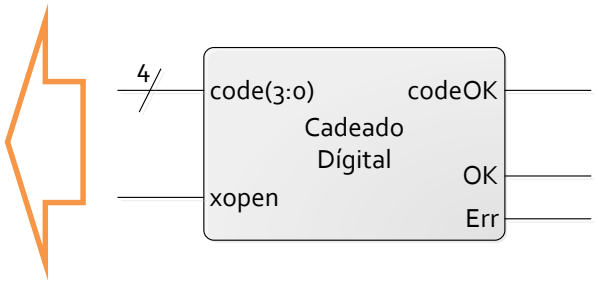


Descrição da arquitectura

Descrição da arquitectura

- Declaração de componentes e sinais

```
...
architecture behavior of cadeado_digital_testbench is
-- Declaração do componente cadeado_digital original
component cadeado_digital
   port (
         code   : in  std_logic_vector(3 downto 0);
         xopen  : in  std_logic;
         codeOK : out std_logic;
         OK     : out std_logic;
         Err    : out std_logic
         );
end component;
-- Declaração dos sinais para o testbench
signal code : std_logic_vector(3 downto 0);
signal xopen, codeOK, OK, Err : std_logic;

begin
...
```



Descrição da arquitectura

- **Implementação**
  - Descrição da unidade para teste

```
...
architecture behavior of cadeado_digital_testbench is
-- Declaração do componente cadeado_digital original
...
-- Declaração dos sinais para o testbench
...
begin

-- Declaração da unidade de teste... O nome dos sinais no circuito é neste
-- casa (não obrigatório) o mesmo que o nome dos sinais no componente
Utest: cadeado_digital port map (
          code => code, xopen => xopen,
          codeOK => codeOK, OK => OK, Err => Err);

-- Descrição do gerador de sinais
...
end behavior;
```

Descrição da arquitectura

- **Implementação**
  - Descrição do gerador de sinais
    - Simulação de 1 linha da tabela de verdade a cada 10 ns

```
...
-- Descrição do gerador de sinais
process
begin
   -- valor dos sinais para a 1ª linha da tabela de verdade
   ...
   wait for 10 ns;
   -- valor dos sinais para a 2ª linha da tabela de verdade
   ...
   wait for 10 ns;
   -- valor dos sinais para a n-ésima linha da tabela de verdade
   ...
   wait; -- end of signal generation
end process;

end behavior;
```

Descrição da arquitectura

- **Implementação**
  - Descrição do gerador de sinais
    - Simulação de 1 linha da tabela de verdade a cada 10 ns

O gerador de sinais acaba na ultima linha da tabela de verdade

```
...
-- Gerador de sinais
process
begin
    -- 1ª linha
    code  <= "0000";
    xopen <= '0';
    wait for 10 ns;
    -- 2ª linha
    code  <= "0001";
    xopen <= '0';
    wait for 10 ns;
    -- 3ª linha
    code  <= "0010";
    xopen <= '0';
    wait for 10 ns;
```

```
    -- 4ª linha
    code  <= "0011";
    xopen <= '0';
    wait for 10 ns;
    ...
    -- 17ª linha
    code  <= "0000";
    xopen <= '1';
    wait for 10 ns;
    -- 18ª linha
    code  <= "0001";
    xopen <= '1';
    wait for 10 ns;
    -- 19ª linha
    code  <= "0010";
    xopen <= '1';
```

```
    wait for 10 ns;
    ...
    -- 31ª linha
    code  <= "1110";
    xopen <= '1';
    wait for 10 ns;
    -- 32ª linha
    code  <= "1111";
    xopen <= '1';
    wait; -- forever
end process;

end behavior;
```

Descrição da arquitectura

- **Implementação**
  - Descrição do gerador de sinais
    - Simulação de 1 linha da tabela de verdade a cada 10 ns

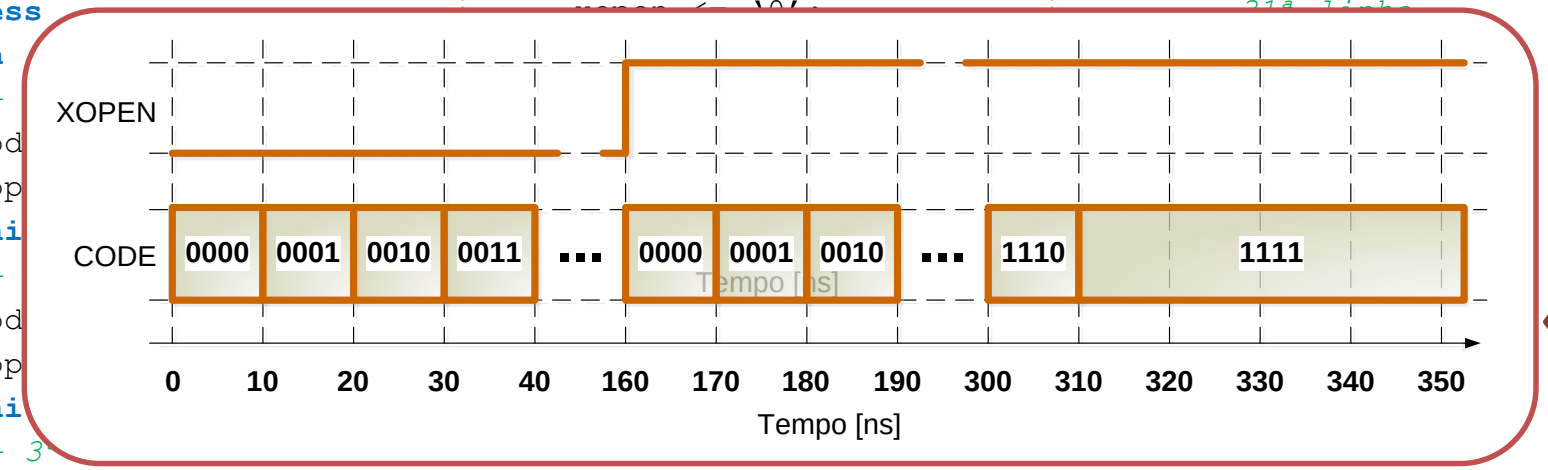
O gerador de sinais acaba na ultima linha da tabela de verdade

```
...
-- Gerador de sinais
process
begin
    --
    cod
    xop
    wai
    --
    cod
    xop
    wai
    -- 3
    code  <= "0010";
    xopen <= '0';
    wait for 10 ns;
```

```
-- 4ª linha
code  <= "0011";
xopen <= '0';
```

```
-- 19ª linha
code  <= "0010";
xopen <= '1';
```

```
wait for 10 ns;
...
-- 21ª linha
```



Descrição da arquitectura

- **Implementação**
  - Descrição do gerador de sinais
    - Simulação de 1 linha da tabela de verdade a cada 10 ns

O gerador de sinais repete após a ultima linha da tabela de verdade

```
...
-- Gerador de sinais
process
begin
    -- 1ª linha
    code  <= "0000";
    xopen <= '0';
    wait for 10 ns;
    -- 2ª linha
    code  <= "0001";
    xopen <= '0';
    wait for 10 ns;
    -- 3ª linha
    code  <= "0010";
    xopen <= '0';
    wait for 10 ns;
```

```
    -- 4ª linha
    code  <= "0011";
    xopen <= '0';
    wait for 10 ns;
    ...
    -- 17ª linha
    code  <= "0000";
    xopen <= '1';
    wait for 10 ns;
    -- 18ª linha
    code  <= "0001";
    xopen <= '1';
    wait for 10 ns;
    -- 19ª linha
    code  <= "0010";
    xopen <= '1';
```

```
    wait for 10 ns;
    ...
    -- 31ª linha
    code  <= "1110";
    xopen <= '1';
    wait for 10 ns;
    -- 32ª linha
    code  <= "1111";
    xopen <= '1';
    wait for 10 ns;
end process;

end behavior;
```

Descrição da arquitectura

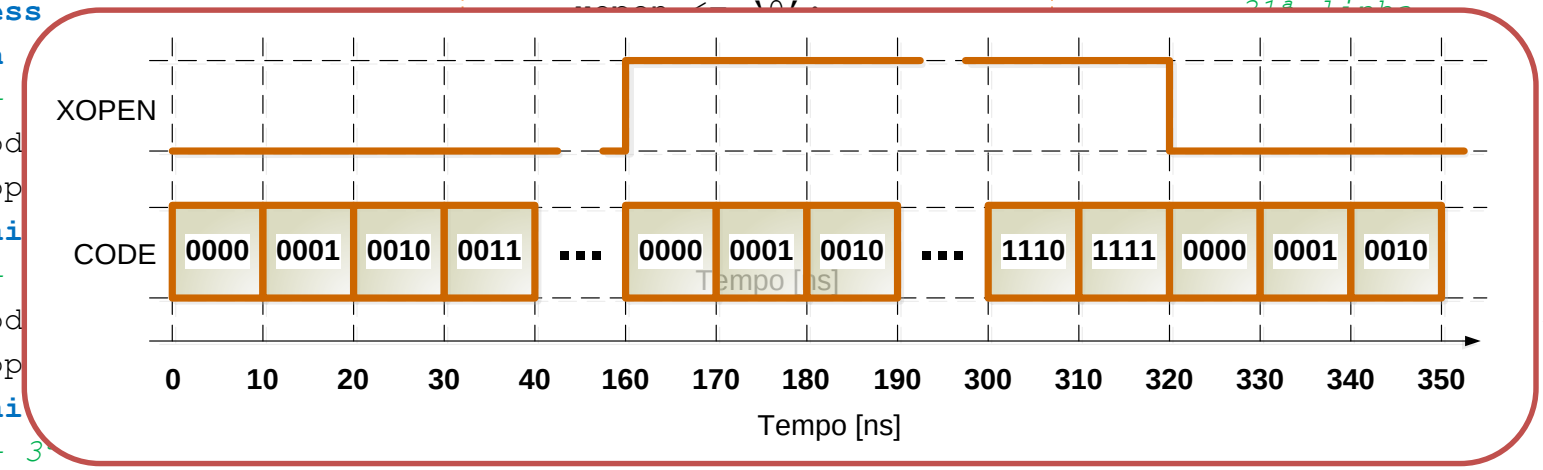
- **Implementação**
  - Descrição do gerador de sinais
    - Simulação de 1 linha da tabela de verdade a cada 10 ns

O gerador de sinais repete após a ultima linha da tabela de verdade

```
...
-- Gerador de sinais
process
begin
    --
    cod
    xop
    wai
    --
    cod
    xop
    wai
    -- 3ª
    code  <= "0010";
    xopen <= '0';
    wait for 10 ns;
```

```
-- 4ª linha
code  <= "0011";
...
-- 19ª linha
code  <= "0010";
xopen <= '1';
```

```
wait for 10 ns;
...
```



Descrição da arquitectura

- **Implementação**

SOLUÇÃO ALTERNATIVA

  - Descrição do gerador de sinais
    - Separação da tabela de verdade em duas partes: sinais `code` e `xopen`

```
...
-- Geração do sinal code
process
begin
   code <= "0000";
   wait for 10 ns;
   code <= "0001";
   wait for 10 ns;
   ...
   code <= "1110";
   wait for 10 ns;
   code <= "1111";
   wait for 10 ns;
end process;
```

```
-- Geração do sinal xopen
process
begin
   xopen <= '0';
   wait for 16*10 ns;
   xopen <= '1';
   wait for 16*10 ns;
end process;

end behavior;
```

Descrição da arquitectura

- **Implementação**
  - Descrição do gerador de sinais
    - Utilização de macros (contador e inversor)

SOLUÇÃO ALTERNATIVA

```
process
begin
    code <= code + 1;
    wait for 10 ns;
end process;

-- Geração do sinal xopen
process
begin
    xopen <= not xopen;
    wait for 16*10 ns;
end process;

end behavior;
```

Requer:
1. **a inicialização dos sinais**
2. **a utilização da biblioteca `ieee.std_logic_unsigned`**

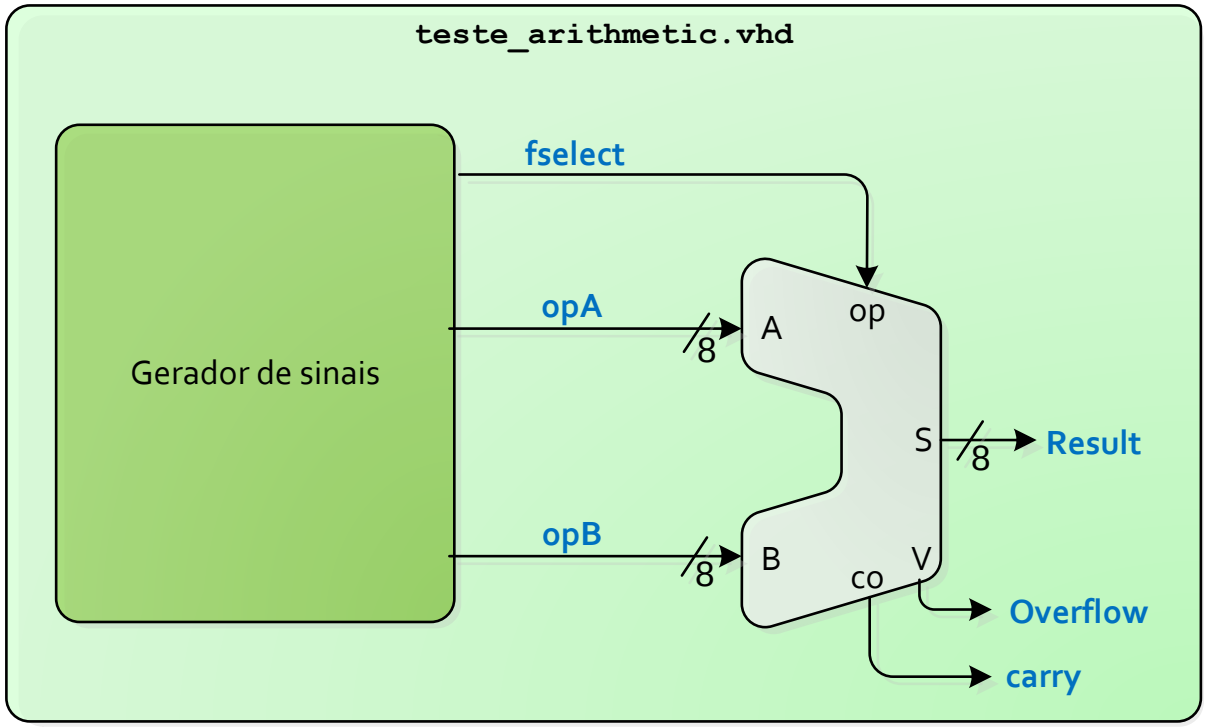
Simulação e teste do circuito "Cadeado Digital" cadeado_digital_testbench_C.vhd

```
-- FICHEIRO cadeado_digital_testbench_C.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Definição da entidade
entity testbench_C is
end testbench_C;

architecture behavior of testbench_C is

-- Declaração do componente
-- cadeado_digital original
component cadeado_digital
 port (
    code    : in  std_logic_vector(3 downto 0);
    xopen   : in  std_logic;
    codeOK  : out std_logic;
    OK      : out std_logic;
    Err     : out std_logic
 );
end component;
```

```
-- Declaração dos sinais para o testbench
signal code : std_logic_vector(3 downto 0) := "0000";
signal xopen, codeOK, OK, Err : std_logic := '0';

begin

-- Declaração da unidade de teste
utest: cadeado_digital port map (
          code => code, xopen => xopen,
          codeOK => codeOK, OK => OK, Err => Err);

-- descrição do gerador para o sinal code
gen_code: process
begin
   code <= code + 1;
   wait for 10 ns;
end process;

-- descrição do gerador para o sinal xopen
gen_open: process
begin
   xopen <= not xopen;
   wait for 16*10 ns;
end process;

end behavior;
```

Descrição da arquitectura

- **Simulação e teste da "Unidade Aritmética"**
  - Componente para teste:



Descrição da entidade

Descrição da entidade

- Sem entradas/saídas

```
-- FICHEIRO tb_arithmetic.vhd

-- Declaração de bibliotecas
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Definição do nome da entidade, sem qualquer entrada ou saida
entity tb_arithmetic is
end tb_arithmetic;

architecture behavior of tb_arithmetic is
...
```

Declaração de componentes e sinais

```
...
architecture behavior of tb_arithmetic is
-- Declaração do componente arithmetic_unit
component arithmetic_unit
   port (
        A  : in  std_logic_vector(7 downto 0);
        B  : in  std_logic_vector(7 downto 0);
        Op : in  std_logic;
        S  : out std_logic_vector(7 downto 0);
        co : out std_logic;
        V  : out std_logic
        );
end component;

-- Declaração dos sinais para o testbench
signal fselect : std_logic := '0';
signal opA, opB : std_logic_vector(7 downto 0) := "00000000";
signal result : std_logic_vector(7 downto 0);
signal overflow, carry: std_logic;

begin
```

Definição do valor inicial do sinal (irá mudar, ao longo da simulação)

Descrição da unidade para teste

Descrição da unidade para teste

```
...
architecture behavior of teste_cadeado_v2 is
-- Declaração do componente arithmetic_unit
...
-- Declaração dos sinais para o testbench
...
begin
-- declaração da instancia para teste
test_unit: arithmetic_unit port map (
          A  => opA, B => opB,
          op => fselect,
          S  => result,
          V  => overflow,
          Co => carry
        );
...
end behavior;
```

Geração dos sinais de dados/controlo

Geração dos sinais de dados/controlo

```
...
begin
-- declaração da instancia para teste
...
-- gerador dos sinais de dados/controlo
process
begin
   fselect <= '0';   -- operação de soma
   opA <= x"08";     -- inicialização em hexadecimal
   opB <= x"FE";     -- inicialização em hexadecimal
   wait for 20 ns;
   fselect <= '1';   -- operação de subtração
   wait for 20 ns;
   fselect <= '0';   -- operação de soma
   opA <= x"A8";     -- inicialização em hexadecimal
   opB <= x"FE";     -- inicialização em hexadecimal
   wait for 20 ns;
   ...
end process;
...
end behavior;
```

Por vezes não é viável testar todos os valores possíveis para as entradas.

Por exemplo: neste caso, o número de combinações possíveis para as entradas (i.e., o número de linhas da tabela de verdade) é de:
$2 \times 2^8 \times 2^8 = 2^{17}$.

Assim deve ser escolhido um conjunto representativo de valores da tabela de verdade de forma a testar o maior número de casos possíveis.

TÉCNICO LISBOA

Próxima Aula

Próxima Aula

- **Tema da Próxima Aula:**
  - Elementos básicos de memória
  - Latches
    - Latch RS
    - Latch RS sincronizado
    - Latch D
  - Flip-Flops
    - Flip-flop master-slave
    - Flip-flop JK
    - Flip-flop edge-triggered

Agradecimentos

Algumas páginas desta apresentação resultam da compilação de várias contribuições produzidas por:

- Guilherme Arroz
- Horácio Neto
- Nuno Horta
- Pedro Tomás