PCS 3115 (PCS2215)

Sistemas Digitais I

Circuitos Combinatórios

Blocos Básicos

Prof. Dr. Marcos A. Simplicio Jr.

versão: 3.0 (Jan/2016)

Blocos básicos

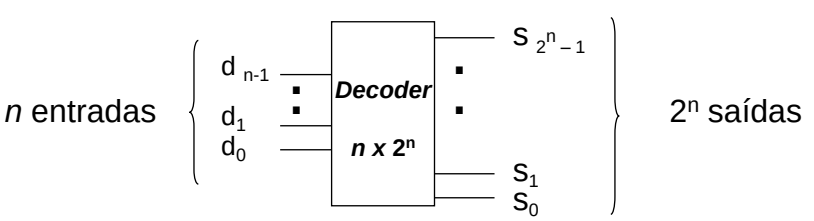
- Codificadores e Decodificadores
  - Drivers de Display
  - Transcoders (BCD – 7 segmentos)
- (De) Multiplexadores e portas tri-state
- Comparadores
- Somadores/Subtratores
- Exercícios

Contexto

- Circuitos SSI: *small scale integration*
  - Até ~20 transistores
  - Circuitos bastante simples, como portas lógicas NOT e AND, OR, XOR de 3 entradas
- Circuitos MSI: *medium scale integration*
  - Até ~200 transistores
  - São os circuitos que veremos neste módulo: codificadores, multiplexadores, somadores, etc.

Decodificador

- Converte código de entrada em código de saída
  - Bloco combinatório lógico que possui n entradas e (até) 2^n saídas
  - Tipo comum: codificador para código 1-de-m
    - Para cada combinação de valores das n entradas, **apenas uma saída é ativada**

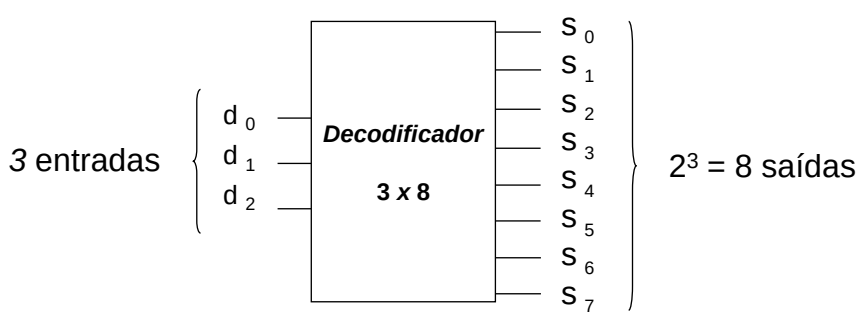


Decodificador 1-de-m

- Funcionamento:
  - Entradas formam uma palavra binária de n bits
  - Palavra de entrada pode assumir os valores de 0 a $2^n - 1$
  - As 2^n saídas são numeradas de 0 a $2^n - 1$
  - Saída ativada é aquela cujo índice corresponde ao valor da palavra binária de entrada
- Exemplo
  - Entradas = 011 (3_{10}); ativa-se a saída S_3
  - Entradas = 101 (5_{10}); ativa-se a saída S_5

Decodificador de 3 bits

- Símbolo funcional (convenção):
  - Entradas à esquerda
  - Saídas à direita
  - Índice menor indica bit menos significativo na palavra binária



Decodificador de 3 bits

- Tabela verdade

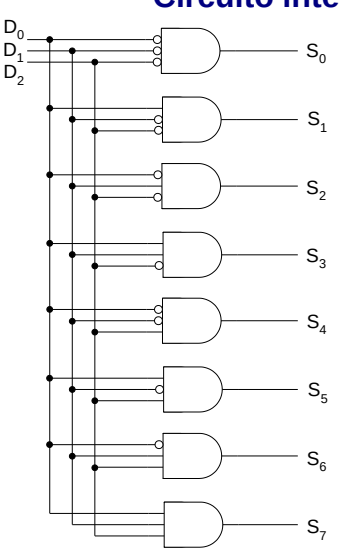
Saída S_2 é ativada...

2 em decimal

ENTRADAS			SAÍDAS							
d_2	d_1	d_0	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0	**1**
0	0	1	0	0	0	0	0	0	**1**	0
0	1	0	0	0	0	0	0	**1**	0	0
0	1	1	0	0	0	0	**1**	0	0	0
1	0	0	0	0	0	**1**	0	0	0	0
1	0	1	0	0	**1**	0	0	0	0	0
1	1	0	0	**1**	0	0	0	0	0	0
1	1	1	**1**	0	0	0	0	0	0	0

Decodificador de 3 bits

Circuito interno: mintermos!



$S_0 = D_2' .D_1' .D_0'$

$S_1 = D_2' .D_1' .D_0$

$S_2 = D_2' .D_1 .D_0'$

$S_3 = D_2' .D_1 .D_0$

$S_4 = D_2 .D_1' .D_0'$

$S_5 = D_2 .D_1' .D_0$

$S_6 = D_2 .D_1 .D_0'$

$S_7 = D_2 .D_1 .D_0$

$S_i = m_i$

Decodificador com Enable

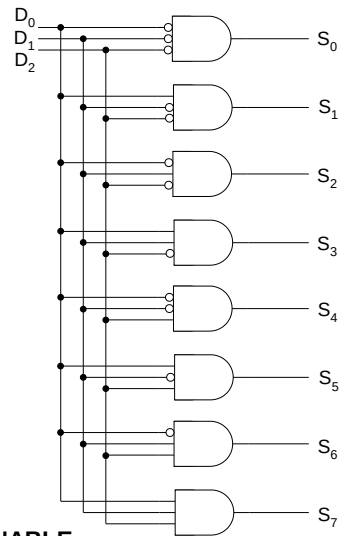
- Entrada adicional: Enable
  - Permite habilitar/desabilitar o bloco todo
- SE Enable ativo
  - Funcionamento normal: apenas uma saída ativa
- SE Enable inativo
  - Nenhuma saída ativa, independente do código nas entradas de endereço
- Atua em todos os mintermos

Decodificador com Enable

Ex: Decodificador 3 por 8, com ***ENABLE*** (EN)

ENTRADAS				SAÍDAS							
EN	d_2	d_1	d_0	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
1	0	0	0	0	0	0	0	0	0	0	**1**
1	0	0	1	0	0	0	0	0	0	**1**	0
1	0	1	0	0	0	0	0	0	**1**	0	0
1	0	1	1	0	0	0	0	**1**	0	0	0
1	1	0	0	0	0	0	**1**	0	0	0	0
1	1	0	1	0	0	**1**	0	0	0	0	0
1	1	1	0	0	**1**	0	0	0	0	0	0
1	1	1	1	**1**	0	0	0	0	0	0	0
0	X	X	X	0	0	0	0	0	0	0	0

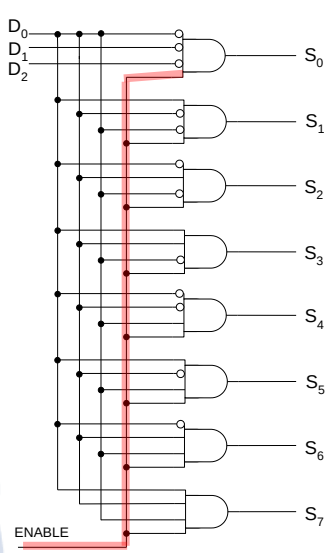
Decodificador com Enable



ENABLE ?

Como adicionar o enable?

Decodificador com Enable

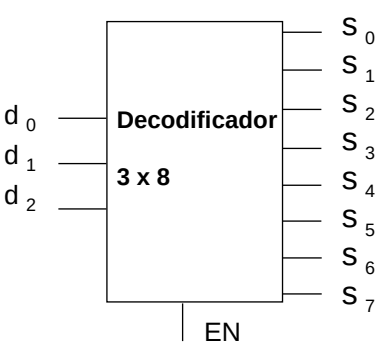


$S_0 = EN .D_2' .D_1' .D_0'$

.

..

$S_7 = EN . D_2 .D_1 .D_0$



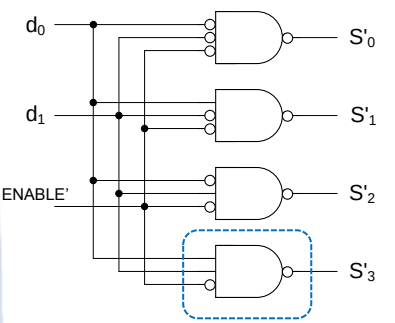
Decodificador: active-low

- Decodificadores podem ter as saídas invertidas (complementadas)
  - Diz-se que as saídas são *active-low* – ativas em ZERO
  - Logo, saídas inativas = 1; a única saída ativa = 0
- Implicações:
  - Na tabela verdade: inversão nas saídas
  - No circuito: uso de **NAND** no lugar de AND
  - Nomenclatura: S_i'
  - Cada saída é um **maxtermo** dos bits de entrada

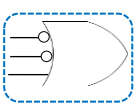
Decodificador: active-low

- Entrada ENABLE também pode ser *active-low*
- SE Enable'=0, circuito habilitado
- SE Enable'=1, saídas todas desabilitadas
- Implicações
  - Na tabela verdade: inversão na entrada
  - No circuito: uso de inversor
  - Nomenclatura: EN'

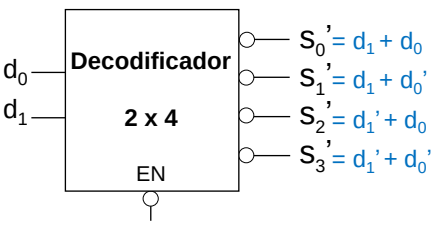
Decodificador: active-low



=



$S_3' = d_1' + d_0'$

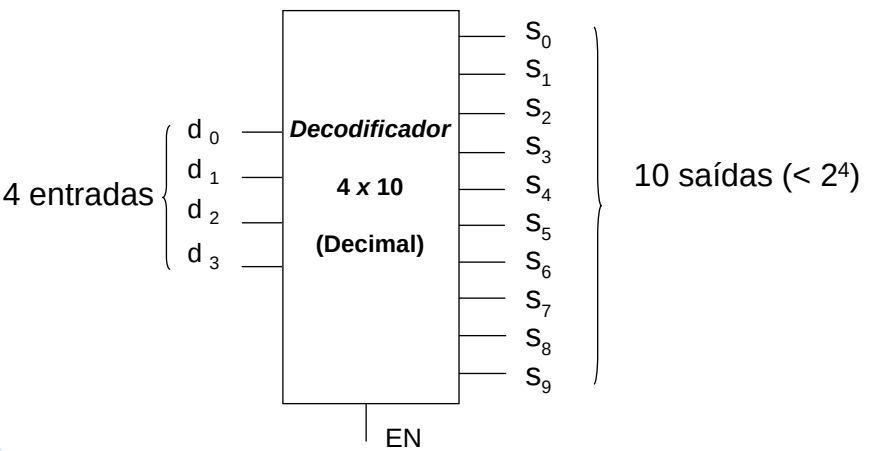


Entradas			Saídas			
EN'	d_1	d_0	S_3'	S_2'	S_1'	S_0'
0	0	0	1	1	1	**0**
0	0	1	1	1	**0**	1
0	1	0	1	**0**	1	1
0	1	1	**0**	1	1	1
1	X	X	1	1	1	1

Decodificadores Decimais

- Decodificador BCD para decimal
  - Entrada: 1 dígito BCD (4 bits)
    - Palavras válidas: 0000 a 1001 (0 a 9)
  - Saída: 10 saídas (<2^4)
    - Saída ativa: corresponde à palavra da entrada
    - Saídas inativas (todas):
      - SE Enable = Falso OU
      - SE palavra de entrada > 1001 (inválida)

Decodificadores Decimais



Decodificadores Decimais

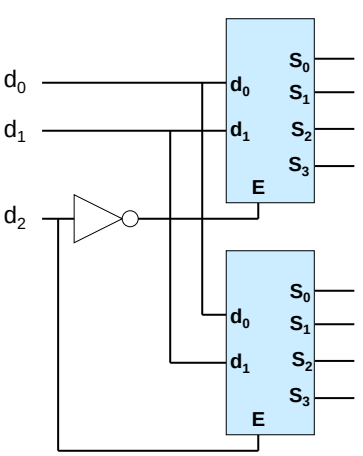
EN	d_3	d_2	d_1	d_0	S_9	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
1	0	0	0	1	0	0	0	0	0	0	0	0	**1**	0
1	0	0	1	0	0	0	0	0	0	0	0	**1**	0	0
1	0	0	1	1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	0	0	**1**	0	0	0	0
1	0	1	0	1	0	0	0	0	**1**	0	0	0	0	0
1	0	1	1	0	0	0	0	**1**	0	0	0	0	0	0
1	0	1	1	1	0	0	**1**	0	0	0	0	0	0	0
1	1	0	0	0	0	**1**	0	0	0	0	0	0	0	0
1	1	0	0	1	**1**	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
	.	.	.	.										
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	X	X	X	X	0	0	0	0	0	0	0	0	0	0

Decodificadores: cascateamento

- Objetivo: expansão de capacidade:
  - Decodificadores 2x4 ➔ 3x8 ➔ 4x16 ➔ 5x32 ➔ n x 2^n
- Estratégia: combinar decodificadores menores, usando entrada de habilitação
- **Exercício**: montar decodificador 3x8 sem entrada de habilitação a partir de dois decodificadores 2x4 com entrada de habilitação

Decodificadores: cascateamento

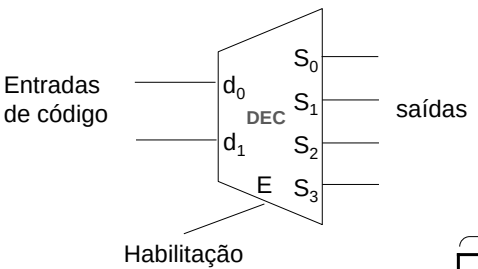
- Decodificador 3x8, sem entrada de habilitação



Exercício: como adicionar uma entrada de habilitação neste circuito?

➔ Adicionar um AND na entrada de habilitação de cada decodificador, fazendo $E \cdot d_2'$ no de cima e $E \cdot d_2$ no de baixo

Decodificadores: Símbolo alternativo

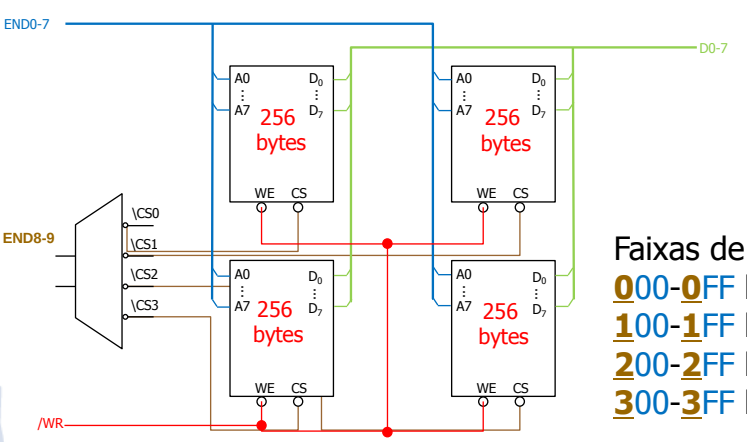


Entradas			Saídas			
E	**d_1**	**d_0**	**S_3**	**S_2**	**S_1**	**S_0**
0	X	X	0	0	0	0
1	0	0	0	0	0	**1**
1	0	1	0	0	**1**	0
1	1	0	0	**1**	0	0
1	1	1	**1**	0	0	0

Tabela verdade

Decodificador: uso (memórias)

Exercício: Dada uma RAM 256 x 1 byte, obter 1ki bytes.



Faixas de endereço
000-0FF h
100-1FF h
200-2FF h
300-3FF h

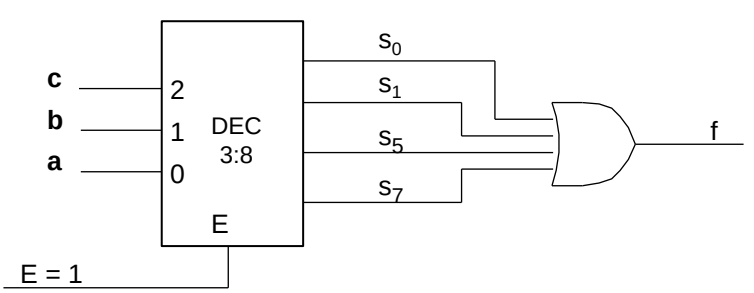
Decodificadores: uso (síntese)

- Síntese de funções de chaveamento
  - Saídas dos decodificadores são os mintermos para as variáveis de entrada
  - 1ª forma canônica: Função = $\Sigma_{mintermos}$
- **Exemplo**:
  - F(d,c,b,a) = Σ(0,1,5,9,12)

Decodificadores: uso (síntese)

- **Exemplo: ativo-alto ➔ soma de mintermos**

  F(c,b,a) = Σ(0,1,5,7)

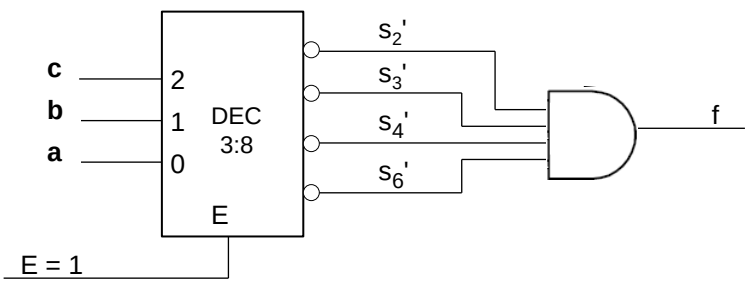


Decodificadores: uso (síntese)

- **Exemplo: ativo-baixo ➔ produto de maxtermos**

  F(c,b,a) = Σ(0,1,5,7)

  = ∏(2,3,4,6)



Decodificadores: uso (síntese)

- FAQ: “Mas isso é usado na prática?!”
- Resposta: não na forma de um decodificador em si, mas ter um circuito com a lista de mintermos “pronto para uso” é **extremamente** interessante
  - Esta é a base para a construção de circuitos de lógica programável (programmable logic device -- PLD)
  - Os conceitos envolvidos são semelhantes aos que aparecem em dispositivos modernos programados usando HDLs, como FPGAs
  - Logo, em projetos com HDL você provavelmente não verá os “mintermos sendo conectados”, mas esse é o tipo de tarefa que será realizada para você pelo sintetizador...

Decodificadores: VHDL

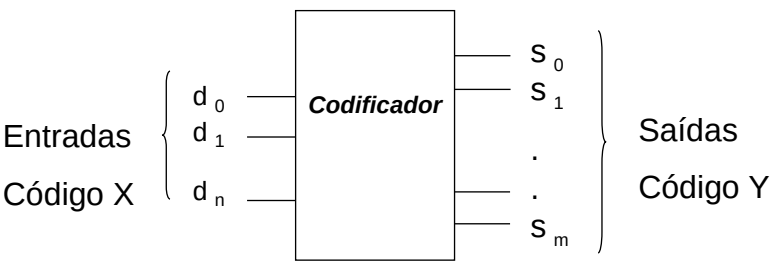
```
library IEEE;
use IEEE.std_logic_1164.all;

entity decoder is
    port (d : in STD_LOGIC_VECTOR (2 downto 0);
          s : out STD_LOGIC_VECTOR (7 downto 0));
end decoder;

architecture decoder_behavior of decoder is
begin
    with d select
        s <=  "00000001" when "000",
              "00000010" when "001",
              "00000100" when "010",
              "00001000" when "011",
              "00010000" when "100",
              "00100000" when "101",
              "01000000" when "110",
              "10000000" when "111",
              "00000000" when others;
end decoder_behavior;
```


Decodificadores: VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity decoder is
    port (d    : in STD_LOGIC_VECTOR (2 downto 0);
          en   : in STD_LOGIC;
          s    : out STD_LOGIC_VECTOR (7 downto 0));
end decoder;

architecture decoder_behavior of decoder is
begin
    s <= "00000000" when (en = '0')   else   -- com enable
         "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" when (d = "111")  else
         "10000000";
end decoder_behavior;
```

Codificadores

- Codificadores ou Transcodificadores
  - Geram um código específico: transformam um código em outro
  - Ex.: código binário → código 1 entre 2^n



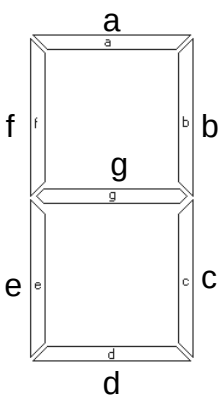
Codificadores

- Codificadores para acionamento de *displays* de 7 segmentos
  - de BCD para 7 segmentos
  - de binário para 7 segmentos
- Entradas: 4 bits
- Saídas: 7 bits, código 7 segmentos

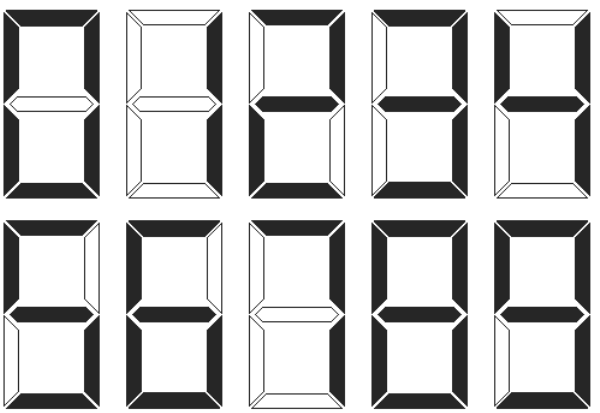
Codificadores

Display de 7 segmentos

Os sete segmentos



O traçado dos algarismos



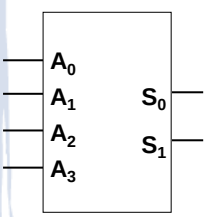
Codificadores: BCD – 7 segmentos

Tabela verdade

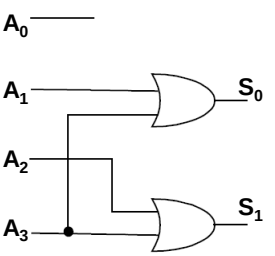
Decimal	EN	d_3	d_2	d_1	d_0	a	b	c	d	e	f	g
0	1	0	0	0	0	1	1	1	1	1	1	
1	1	0	0	0	1		1	1				
2	1	0	0	1	0	1	1		1	1		1
3	1	0	0	1	1	1	1	1	1			1
4	1	0	1	0	0		1	1			1	1
5	1	0	1	0	1	1		1	1		1	1
6	1	0	1	1	0	1		1	1	1	1	1
7	1	0	1	1	1	1	1	1				
8	1	1	0	0	0	1	1	1	1	1	1	1
9	1	1	0	0	1	1	1	1	1		1	1
10	1	1	0	1	0	0	0	0	0	0	0	0
		.	.	.	.							
15	1	1	1	1	1	0	0	0	0	0	0	0
	0	X	X	X	X	0	0	0	0	0	0	0

Codificadores: Binário "2^n : n"

- Codificador binário 4:2
- Problema em potencial:
  - Mais de uma entrada em 1 pode resultar em erro
  - Ex.: 0110 → 11 (mesmo resultado obtido com 1000)



Entradas				Saídas	
A_3	A_2	A_1	A_0	S_1	S_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	1	1	0	1	1



Codificadores: Binário "2^n : n"

- Codificador binário 4:2 **com prioridade**
  - Entradas têm níveis de prioridade: elimina o problema anterior (perceba os "don't care")
  - Resolução: construir o mapa de Karnaugh

Entradas				Saídas	
A_3	A_2	A_1	A_0	S_1	S_0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1
0	1	1	0	1	0

prioridade

$A_3 A_2$ \ $A_1 A_0$	00	01	11	10
00				
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$S_1 = A_3 + A_2$

	00	01	11	10
00			1	1
01				
11	1	1	1	1
10	1	1	1	1

$S_0 = A_3 + A_1 \cdot A'_2$

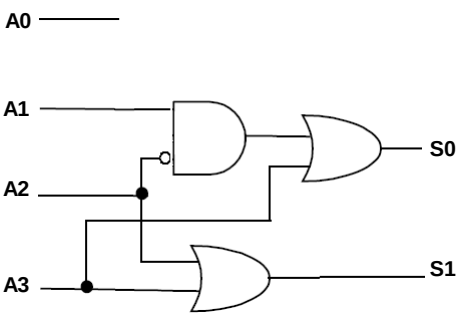
Codificadores: Binário "2^n : n"

- Codificador binário 4:2 **com prioridade**
  - Entradas têm níveis de prioridade: elimina o problema anterior (perceba os "don't care")
  - Usando o mapa de Karnaugh

Entradas | Saídas

A_3	A_2	A_1	A_0	S_1	S_0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1
0	1	1	0	1	0

prioridade



Codificadores: uso

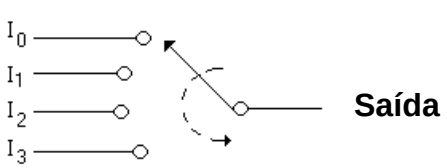
- Reduzem o número de fios e aplicações com diversas entradas
  - Ex.: codificador de teclado (em vez de ter um fio por tecla, valor de cada tecla pode ser antes codificado)
- Regular entradas com diferentes prioridades
  - Ex.: controle de interrupções em computadores
- Conversão entre sistemas numéricos
  - Ex.: o codificador BCD (Binary Coded Decimal – Decimal Codificado em Binário) apresenta 10 entradas, resultando em 4 bits de saída em binário: A5 → 0101, A8 → 1000, etc.

Codificadores: VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity priority_encoder is
    port (A    : in STD_LOGIC_VECTOR (7 downto 0);
          en   : in STD_LOGIC;
          S    : out STD_LOGIC_VECTOR (2 downto 0));
end priority_encoder;

architecture behav of priority_encoder is
begin
    S <= "000" when (en   = '0')  else  -- com enable
         "111" when (A(7) = '1')  else  -- maior prioridade
         "110" when (A(6) = '1')  else
         "101" when (A(5) = '1')  else
         "100" when (A(4) = '1')  else
         "011" when (A(3) = '1')  else
         "010" when (A(2) = '1')  else
         "001" when (A(1) = '1')  else
         "000" when (A(0) = '1')  else -- menor prioridade
         "000";
end behav;
```

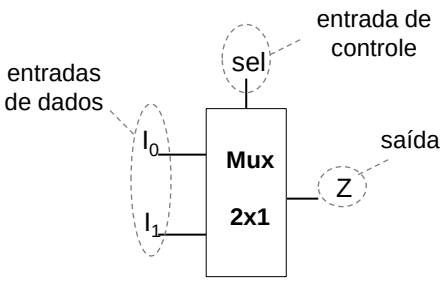

Multiplexador: Conceito

- Multiplexação: seleção da entrada que deve ir para a saída
  - Operação de uma chave seletora



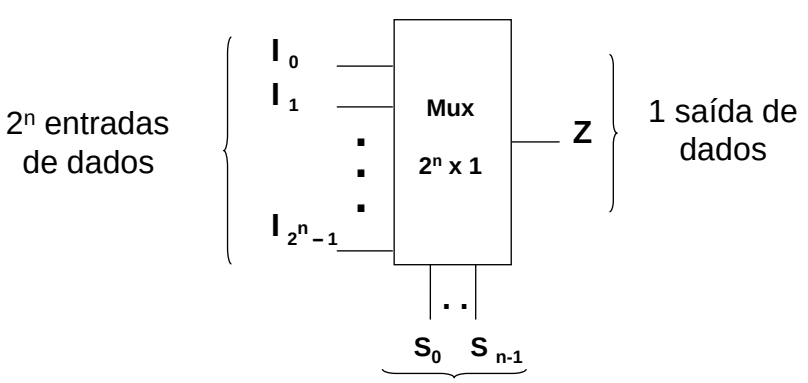
Multiplexador: Conceito

- Multiplexador ou Multiplexer ou Mux
  - Possui 2^n entradas de dados e uma saída: só uma das 2^n entradas é "conectada" à saída
  - Um total de **n entradas de seleção** indicam qual entrada de dados vai para a saída

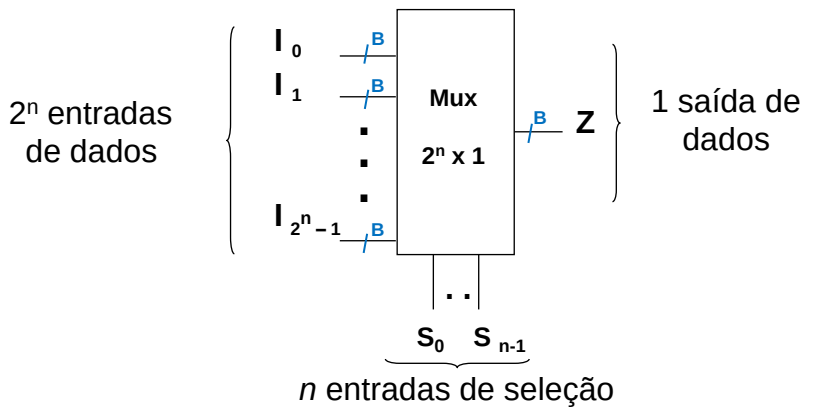


sel	I_0	I_1	Z
0	**0**	0	**0**
0	**0**	1	**0**
0	**1**	0	**1**
0	**1**	1	**1**
1	0	**0**	**0**
1	0	**1**	**1**
1	1	**0**	**0**
1	1	**1**	**1**

Multiplexador: visão geral



Multiplexador: visão geral



Nota: cada entrada/saída pode ser na realidade um barramento com B bits ➔ todos são selecionados ao mesmo tempo. NÃO vamos explorar este conceito nos slides a seguir

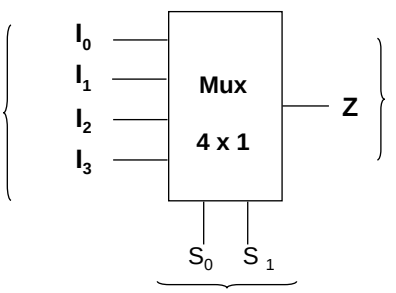
Multiplexador: visão geral

- Qual entrada de dados vai para a saída ?
  - As **entradas de seleção** formam uma palavra binária de *n* bits
    - A palavra pode assumir os valores de 0 a $2^n - 1$
  - As 2^n entradas de dados são numeradas de 0 a $2^n - 1$
    - A saída recebe a entrada de dados cujo índice corresponde ao valor da palavra binária das entradas de seleção
- Exemplo
  - Entradas de seleção = 011 (3_{10}); Z recebe I_3

Multiplexador de 2 bits

- Símbolo funcional (convenção):
  - Entradas de dados à esquerda
  - Saídas à direita
  - Índice menor indica bit menos significativo na palavra binária

$2^2 = 4$ entradas de dados



1 saída de dados

2 entradas de seleção

Multiplexador de 2 bits

Ex: Mux 4 por 1

Entradas de Seleção		Saída
S_1	S_0	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

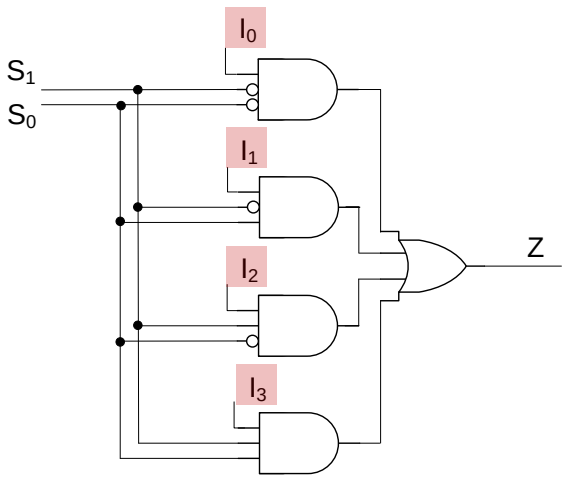
Mapa

S_1 / S_0	0	1
0	I_0	I_2
1	I_1	I_3

$$Z = S_1' \cdot S_0' \cdot I_0 + S_1' \cdot S_0 \cdot I_1 + S_1 \cdot S_0' \cdot I_2 + S_1 \cdot S_0 \cdot I_3$$

mintermos

Multiplexador de 2 bits: circuito interno



$$Z = \underbrace{S_1' \cdot S_0'}_{} \cdot I_0 + \underbrace{S_1' \cdot S_0}_{} \cdot I_1 + \underbrace{S_1 \cdot S_0'}_{} \cdot I_2 + \underbrace{S_1 \cdot S_0}_{} \cdot I_3$$

mintermos

Multiplexador com Enable

- Entrada adicional: Enable
  - Permite habilitar/desabilitar o bloco todo
- SE Enable ativo
  - Funcionamento normal: seleção de entrada
- SE Enable inativo
  - Saída inativa independentemente do código nas entradas de seleção e das entradas de dados
- Atua em todos os produtos (portas E)

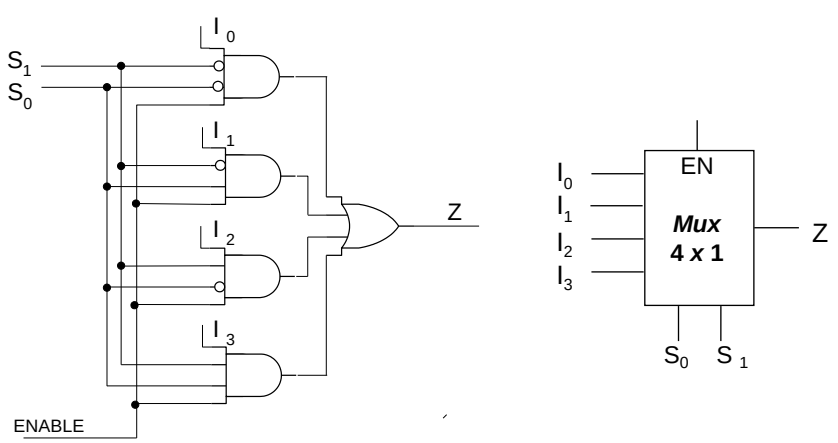
Multiplexador com Enable

- Tabela verdade: Mux 4 por 1 com *Enable* (EN)

Entradas			Saída
EN	S_1	S_0	Z
1	0	0	I_0
1	0	1	I_1
1	1	0	I_2
1	1	1	I_3
0	X	X	0

$$Z = EN \cdot S_1' \cdot S_0' \cdot I_0 + EN \cdot S_1' \cdot S_0 \cdot I_1 + EN \cdot S_1 \cdot S_0' \cdot I_2 + EN \cdot S_1 \cdot S_0 \cdot I_3$$

Multiplexador com Enable



$$Z = EN \cdot S_1' \cdot S_0' \cdot I_0 + EN \cdot S_1' \cdot S_0 \cdot I_1 + EN \cdot S_1 \cdot S_0' \cdot I_2 + EN \cdot S_1 \cdot S_0 \cdot I_3$$

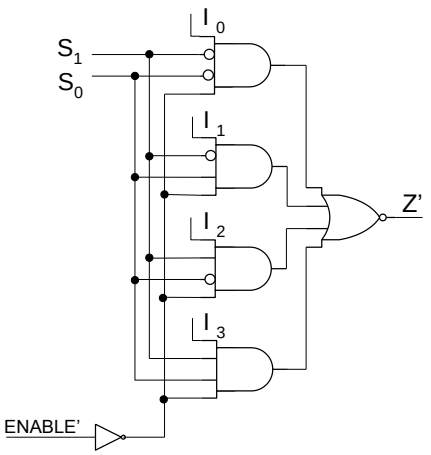
Multiplexador: *active low*

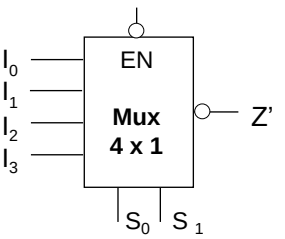
- Multiplexadores podem ter a saída invertida (complementada)
  - Diz-se que as saídas são *active-low* – ativas em ZERO
- Implicações:
  - Na tabela verdade: inversão nas saídas (Z' = I_i')
  - No circuito: uso de **NOR** no lugar de OR
  - Nomenclatura: Z'

Multiplexador: *active-low*

- Entrada ENABLE também pode ser *active-low*
- SE Enable'=0, circuito habilitado
- SE Enable'=1, saídas todas desabilitadas
- Implicações
  - Na tabela verdade: inversão na entrada
  - No circuito: uso de inversor
  - Nomenclatura: EN'

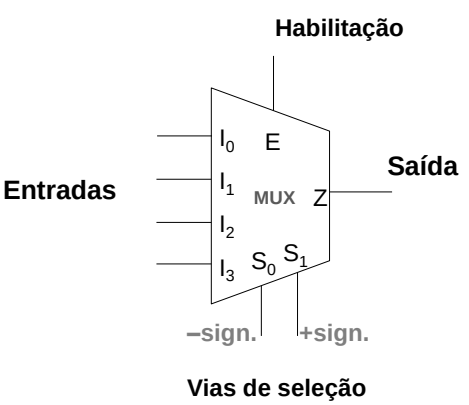
Multiplexador: *active-low*





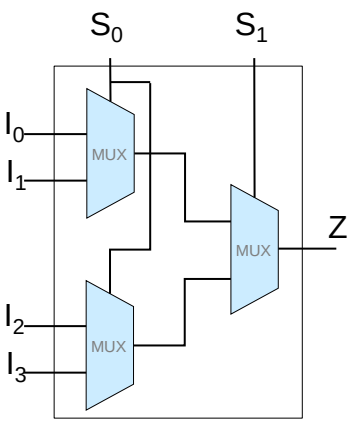
Entradas			Saída
EN'	S_1	S_0	Z'
0	0	0	I_0'
0	0	1	I_1'
0	1	0	I_2'
0	1	1	I_3'
1	X	X	1

Multiplexador: Símbolo alternativo



Multiplexador: Cascateamento

- Multiplexador 4:1 implementação usando Mux 2:1



S_1	S_0	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Multiplexadores: VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4_1 is
    port (sel          : in  STD_LOGIC_VECTOR (1 downto 0);
          I0,I1,I2,I3  : in  STD_LOGIC;
          Y            : out STD_LOGIC);
end mux4_1;

architecture arch_mux of mux4_1 is
begin
    with sel select
       Y <= I0  when "00",
            I1  when "01",
            I2  when "10",
            I3  when "11",
            '0' when others; -- "catch all"
end arch_mux;
```

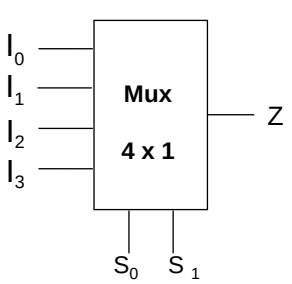
Multiplexadores: VHDL

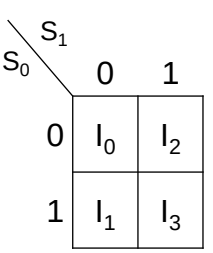
```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in8b is
    port (
        S: in STD_LOGIC_VECTOR (1 downto 0);      -- Select inputs, 0-3 ==> A-D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Data bus input
        Y: out STD_LOGIC_VECTOR (1 to 8)          -- Data bus output
    );
end mux4in8b;

architecture mux4in8b of mux4in8b is
begin
    with S select Y <=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
      (others => 'U') when others; -- this creates an 8-bit vector of 'U'
end mux4in8b;
```

Obs.: No código, entradas/saídas são barramentos de 8 bits

Multiplexadores: Uso

- Síntese de funções
  - Qualquer função de chaveamento de *n* variáveis pode ser implementada com um único MUX $2^n : 1$
  - Ex.: Função de 2 variáveis e Mux 4x1



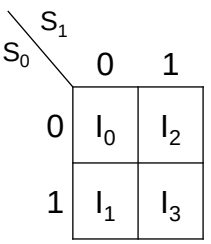


Mapa do Mux 4*x*1

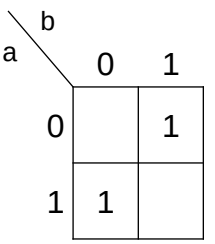
$$Z = S_1'.S_0'.I_0 + S_1'.S_0.I_1 + S_1.S_0'.I_2 + S_1.S_0.I_3$$

Multiplexadores: Síntese

Mapa do Mux 4*x*1



Mapa da Função F(b,a)

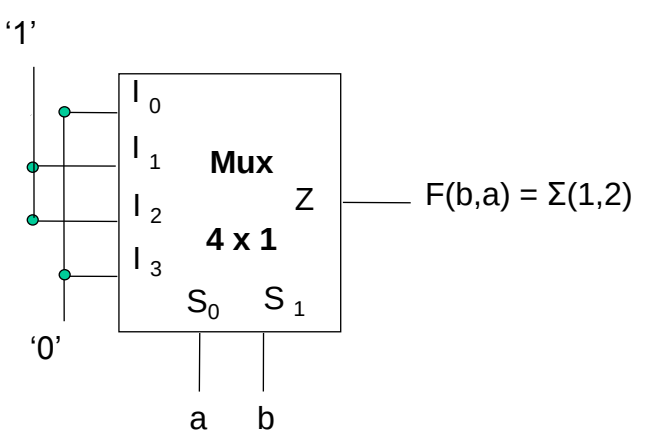


F(b,a) = Σ(1,2)

- Da comparação dos dois mapas obtém-se:

  $I_0 = 0, I_1 = 1, I_2 = 1, I_3 = 0$

Multiplexadores: Síntese



Multiplexadores: Síntese

- Síntese de funções com Mux de **menor ordem**
  - Implementar função de chaveamento de *n* variáveis com um único MUX $2^{n-1} : 1$
  - Exemplo:
    - Função de 4 variáveis
    - Mux 8x1 (3 variáveis de seleção)

Multiplexadores: Síntese

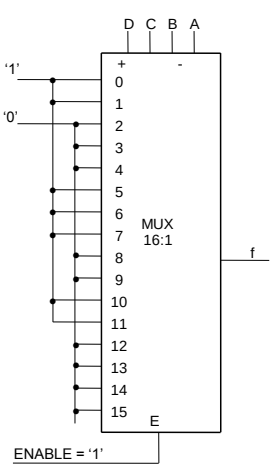
- Solução simples: com Mux de 16x1

BA \ DC	00	01	11	10
00	I_0	I_4	I_{12}	I_8
01	I_1	I_5	I_{13}	I_9
11	I_3	I_7	I_{15}	I_{11}
10	I_2	I_6	I_{14}	I_{10}

MAPA DE MUX 16:1

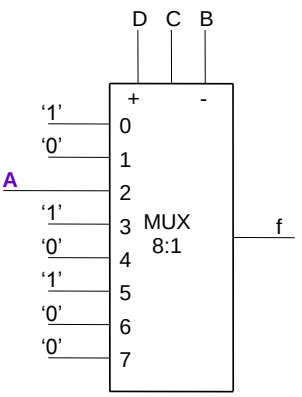
BA \ DC	00	01	11	10
00	1			
01	1	1		
11		1		1
10		1		1

$F = \Sigma(0,1,5,6,7,10,11)$

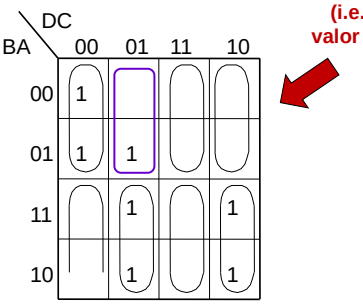


Multiplexadores: Síntese

- Solução 1: Mux de 8x1 e **A na entrada**



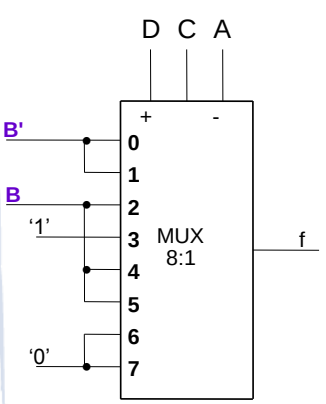
1. Cubos que podem conter A
(i.e., A não tem valor fixo no cubo)



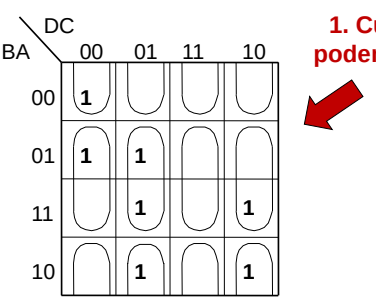
2. Função f: D'C'B' + D'CB'**A** + D'CB + DC'B
0 2 3 5

Multiplexadores: Síntese

- Solução 2: Mux de 8x1 e **B na entrada**



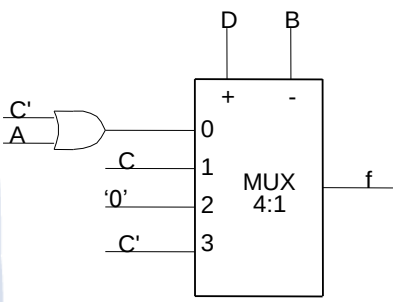
1. Cubos que podem conter B

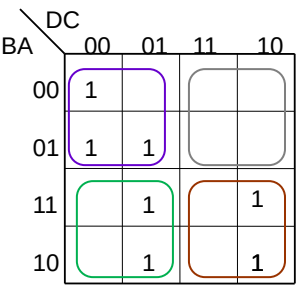


2. Função f = D'C'A'**B'** + D'C'A**B'** + D'CA'**B** + D'CA + DC'A'**B** + DC'A**B**
0 1 2 3 4 5

Multiplexadores: Síntese

- Solução 3: Mux de 4x1 e **A e C na entrada**



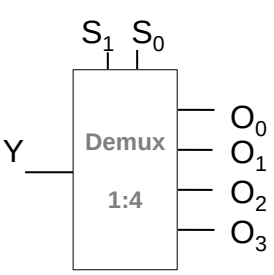


1. Cubos que podem conter A e C

2. Função f = **D'B'**(A+C') + **D'B**C + **DB**C'

Demultiplexadores

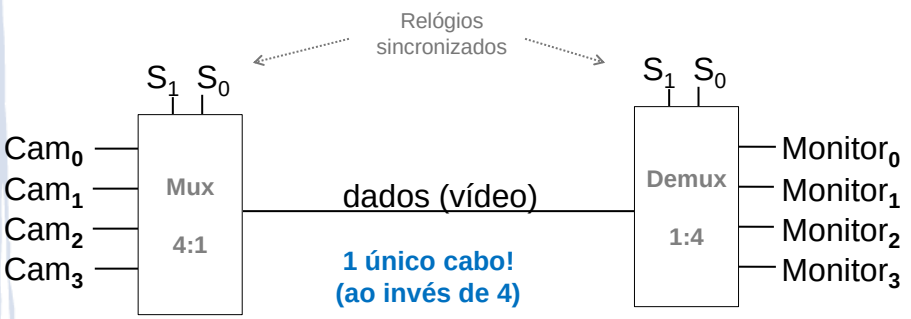
- Função inversa ao multiplexador:
  - Demux 1:2^n direciona a entrada para uma dentre 2^n saídas de dados



S_1	S_0	O_1	O_0	O_2	O_3
0	0	**Y**	0	0	0
0	1	0	**Y**	0	0
1	0	0	0	**Y**	0
1	1	0	0	0	**Y**

(De)multiplexadores: Uso

- Pode ser usado para compartilhamento de canal entre diversos fluxos (multiplexação de vídeo)
  - Ex.: envio serial de vídeo de diferentes câmeras para uma central de vigilância



Demultiplexador: VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity demux4_1 is
    port (S            : in STD_LOGIC_VECTOR (2 downto 0)
          Y            : in STD_LOGIC_VECTOR (7 downto 0);
          O0,O1,O2,O3  : out STD_LOGIC_VECTOR (7 downto 0));
end demux4_1;

architecture arch_demuxZ of demux4_1 is -- com alta impedância
begin
    O0 <= Y when S = "00" else "ZZZZZZZZ"; -- código alternativo:
    O1 <= Y when S = "01" else "ZZZZZZZZ"; -- else (others => 'Z');
    O2 <= Y when S = "10" else "ZZZZZZZZ";
    O3 <= Y when S = "11" else "ZZZZZZZZ";
end arch_demuxZ;

architecture arch_demux0 of demux4_1 is -- desabilitado com 0s
begin
    O0 <= Y when S = "00" else "00000000"; -- código alternativo:
    O1 <= Y when S = "01" else "00000000"; -- else (others => '0');
    O2 <= Y when S = "10" else "00000000";
    O3 <= Y when S = "11" else "00000000";
end arch_demux0;
```

Exercícios

- 6.1. Sintetize a função de chaveamento abaixo utilizando

  $F(d,c,b,a) = \Sigma(0,1,3,8,10)$

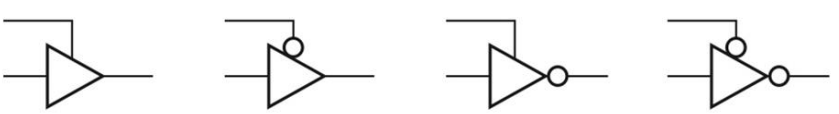
  a. Um Mux de 16x1
  b. Um Mux de 8x1
  c. Um Mux de 4x1

- 6.2. Sintetize a função de chaveamento dada pela tabela verdade ao lado utilizando um decodificador binário 3:8.

c	b	a	F
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	1
1	1	1	0

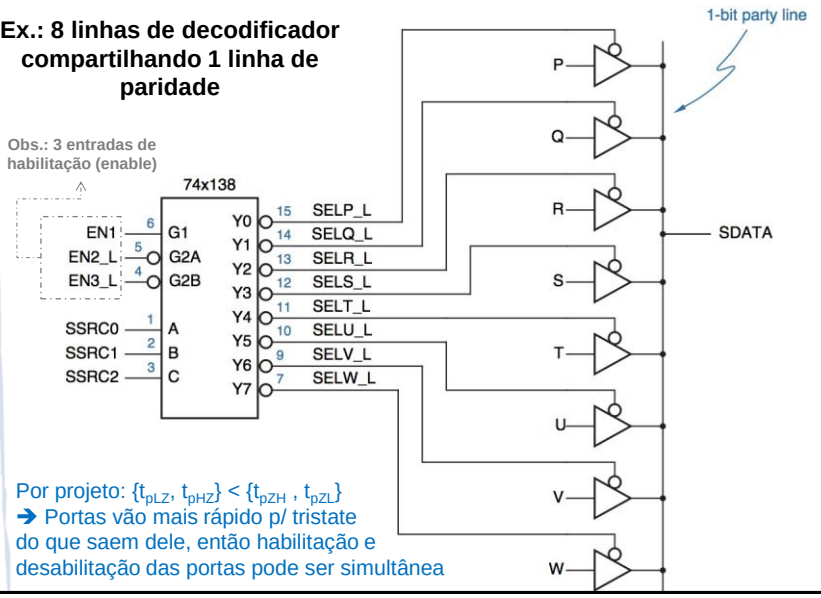
Portas tristate

- Além de saída em 1 e 0, pode-se ter um terceiro estado de alta impedância (HI-Z).
- Usado para conexão em barramentos: apenas circuitos ativos acessam barramento, prevenindo curtos
  a) Buffer (sem inversão): ativo alto
  b) Buffer (sem inversão): ativo-baixo
  c) Inversor tristate: ativo-alto
  d) Inversor tristate: ativo-baixo



Portas tristate

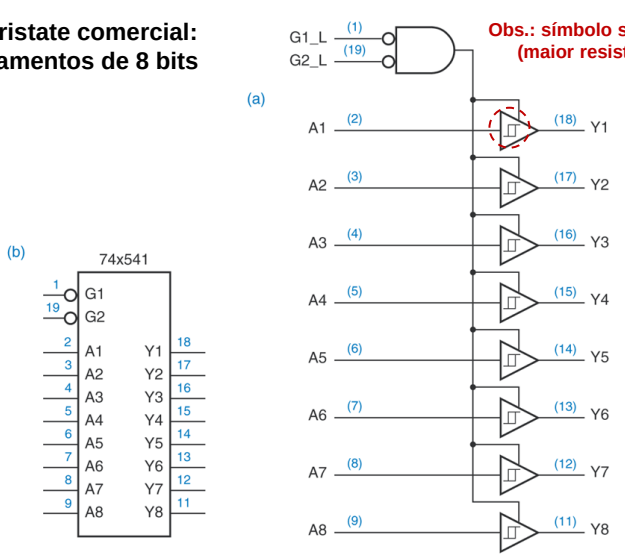
Ex.: 8 linhas de decodificador compartilhando 1 linha de paridade



Por projeto: $\{t_{pLZ}, t_{pHZ}\} < \{t_{pZH}, t_{pZL}\}$
➔ Portas vão mais rápido p/ tristate do que saem dele, então habilitação e desabilitação das portas pode ser simultânea

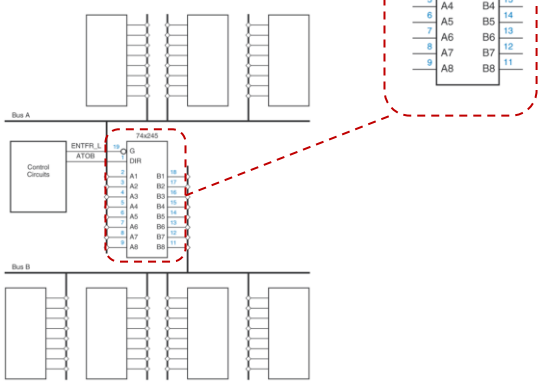
Portas tristate

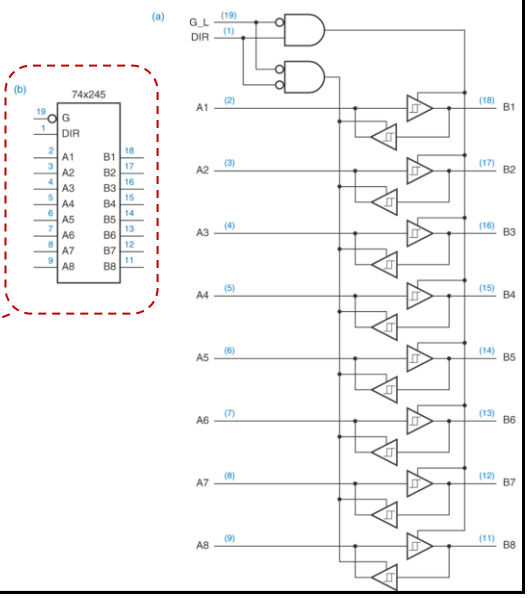
Circuito tristate comercial: para barramentos de 8 bits



Portas tristate

Circuito tristate comercial para conexão de barramentos de 8 bits: direção do fluxo de dados definida por entrada "DIR"



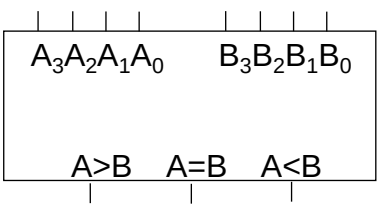


Portas tristate

- Uso comum em computadores modernos: **circuitos de memória** compartilhando o mesmo barramento
  - Apenas um chip de memória pode **escrever** no barramento a qualquer momento
  - Os outros podem **ler** do barramento em paralelo
  - ➔ Não aumenta "capacidade de transmissão", mas permite uso de menos fios para comunicação

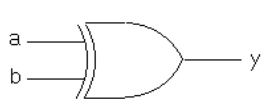
Comparadores

- Comparação entre palavras binárias é uma operação comum em sistemas digitais.
- Comparadores realizam essa função e podem indicar igualdade (= , ≠), e em alguns casos, relação aritmética (> , <).



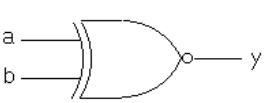
Comparadores

- (Não) OU-exclusivo – X(N)OR
  - São comparadores de 1 bit



Y= a ⊕ b = Dif

Dif = 1, se entradas são diferentes



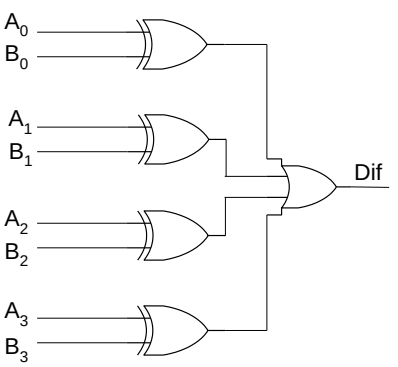
Y= (a ⊕ b)' = Eq

Eq = 1, se entradas são iguais

- **Pergunta**: como fazer um comparador de n bits...?

Comparadores

- Comparador (paralelo) de *n* bits
  - Compara bit a bit, duas palavras de *n* bits,
  - Sumariza o resultado



Comparadores "iterativos"

- Software iterativo:

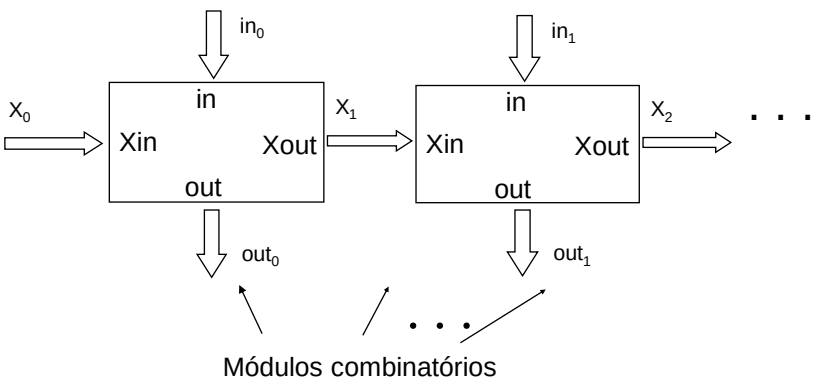
  for (X_0 = valor inicial, i=0 ; i < n ; i++)

  $out_i = f(X_i, in_i)$; $X_{i+1} = g(X_i, in_i)$;

- Circuitos combinatórios "iterativos":
  - *n* módulos idênticos cascateados
  - Cada módulo possui:
    - Entradas e saídas primárias (in e out)
    - Entradas e saídas para **associação em cascata** (X)
  - Adequados para problemas que podem ser resolvidos com algoritmos iterativos.
    - Mas mais lentos que circuitos paralelos equivalentes

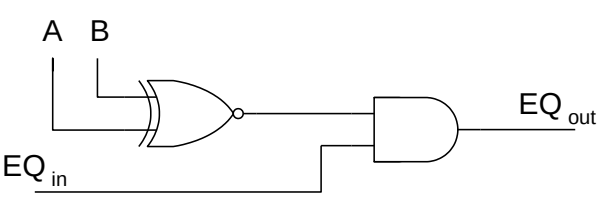
Comparadores “iterativos”

- Cada módulo possui:
  - Entradas e saídas primárias (in e out)
  - Entradas e saídas para **associação em cascata** (X)

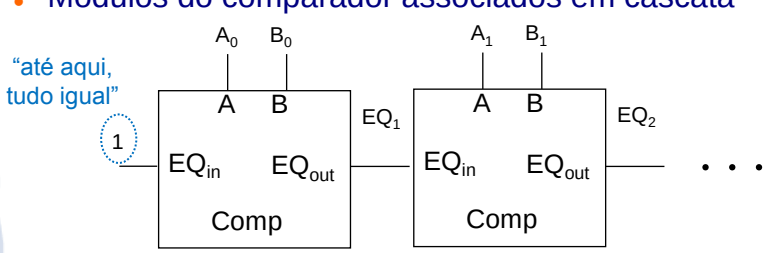


Comparadores “iterativos”

- Módulo combinatório básico do comparador



- Módulos do comparador associados em cascata



Comparador de magnitude

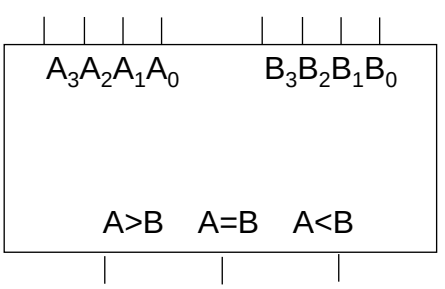
- Indica relação aritmética entre as palavras de dados comparadas
- **Pergunta** (comparação de 1 bit): quando Ai > Bi?
  - Resposta: quando Ai = 1 e Bi = 0
- **Pergunta** (comparação de n bits, sem sinal): como um humano faria essa comparação?
  - **Resposta**: analisar desigualdades dos bits, começando no mais significativo ➔ A > B se
    - A3>B3 (i.e. A3=1 e B3=0)
    - Ou se A3=B3 e A2>B2
    - Ou se A3=B3 e A2=B2 e A1>B1
    - Ou se A3=B3 e A2=B2 e A1=B1 e A0>B0

Comparador de magnitude

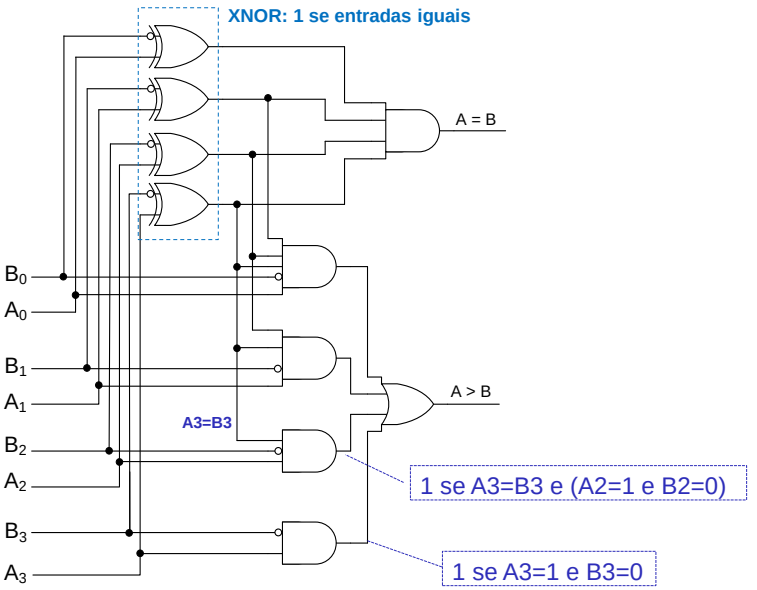
- A=B se
  - A3=B3 e
  - A2=B2 e
  - A1=B1 e
  - A0=B0
- A>B se

Ai > Bi se
Ai=1 e Bi=0

  - **A3>B3**
  - Ou se A3=B3 e **A2>B2**
  - Ou se A3=B3 e A2=B2 e **A1>B1**
  - Ou se A3=B3 e A2=B2 e A1=B1 e **A0>B0**

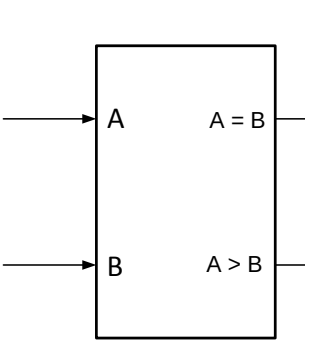


Comparador de magnitude



Comparador de magnitude

- Pergunta: como obter as demais relações?



A dif. B

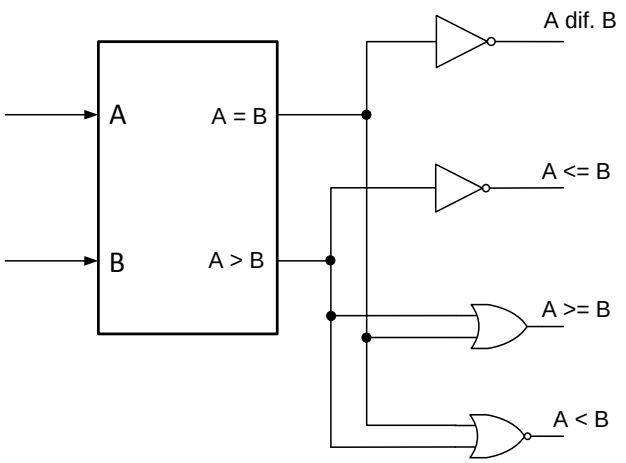
A <= B

A >= B

A < B

Comparador de magnitude

- Pergunta: como obter as demais relações?



Comparador de magnitude: VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity compmag is
 port (inpA,inpB : in std_logic_vector(3 downto 0);
       greater, equal, smaller  : out std_logic);
end compmag;

architecture compmag_arch of compmag is
 begin
    -- std_logic e std_logic_vector suportam comparações
       greater <= '1' when (inpA > inpB) else '0';
       equal  <= '1' when (inpA = inpB) else '0';
       smaller <= '1' when (inpA < inpB) else '0';
 end compmag_arch;
```

Somadores binários

- Soma binária: uma das operações aritméticas mais comuns em sistemas digitais
- Somador Binário combina dois operandos aritméticos usando as regras da soma binária
- Regras da soma são as mesmas para números sem sinal e para números em Complemento de 2
- Somador pode realizar uma subtração como a "soma do Minuendo com o complemento do Subtraendo"
  - Ou seja: a – b = a + (– b)

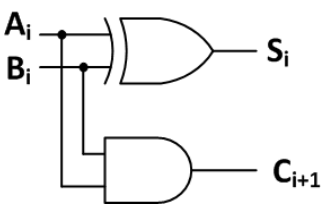
Meio Somador

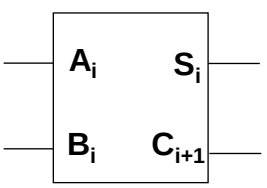
- A soma de dois operandos de 1 bit produz 2 bits
  1. Resultado da soma, e
  2. "Vai-um" (*carry*)
- Operação realizada por "meio somador"
  - Não considera "vem-um"
  - Logo, não funciona para números de 2+ bits
    - Teste: 0010 + 0110 ➔ sem o "vem-um": 0100 (errado...)
      ➔ com o "vem-um": 1000 (correto!!)

A_i	B_i	S_i	C_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Meio Somador

- Qual o circuito que implementa o meio somador?





A_i	B_i	S_i	C_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

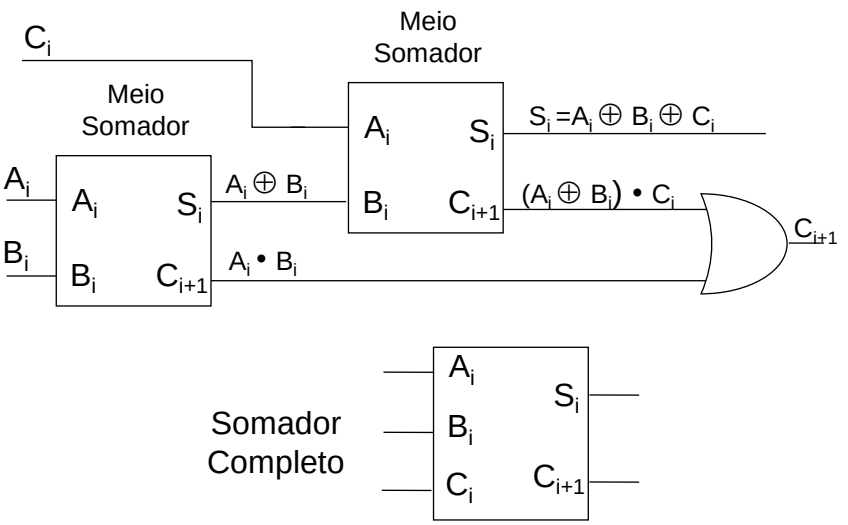
xor and

Somador completo

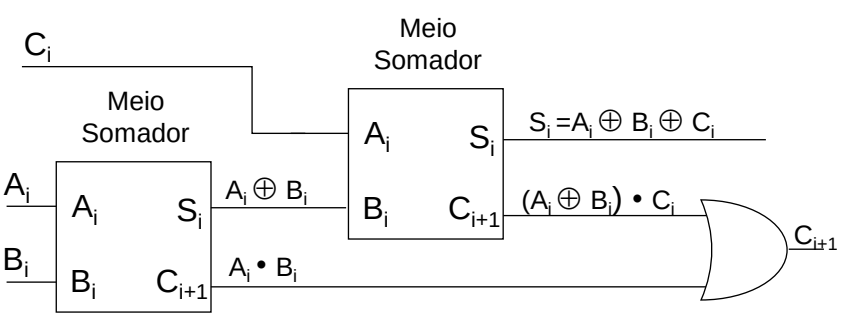
- Usado para soma de operandos com 2+ bits
  - Entrada adicional para tratar o "vem-um" do bloco anterior (entrada "C_i")
- Como implementar um somador completo?
  - Pode-se combinar 2 meio somadores: meia-soma entre A_i e B_i seguida de meia-soma com C_i.
  - "Vai-um" se qualquer das somas levar a "vai-um"

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Somador completo



Somador completo



$S_i = A_i \oplus B_i \oplus C_i = A_i' \bullet B_i' \bullet C_i + A_i' \bullet B_i \bullet C_i' + A_i \bullet B_i' \bullet C_i' + A_i \bullet B_i \bullet C_i$

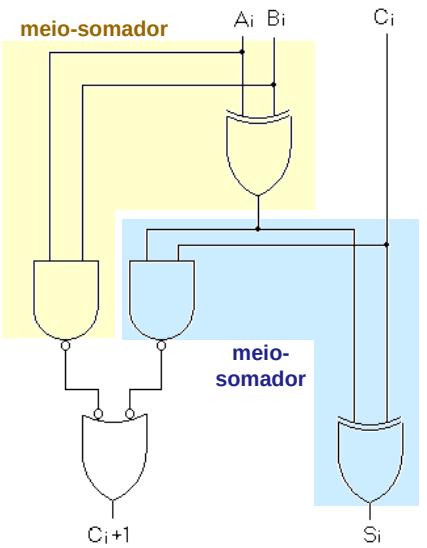
$C_{i+1} = A_i' \bullet B_i \bullet C_i + A_i \bullet B_i' \bullet C_i + A_i \bullet B_i \bullet C_i' + A_i \bullet B_i \bullet C_i$ ← mintermos

$= (A_i \oplus B_i) \bullet C_i + A_i \bullet B_i$ ← meio-somadores

$= (A_i + B_i) \bullet C_i + A_i \bullet B_i$ ← ANDs e ORs

$= A_i \bullet C_i + B_i \bullet C_i + A_i \bullet B_i$ ← AND2

Somador completo



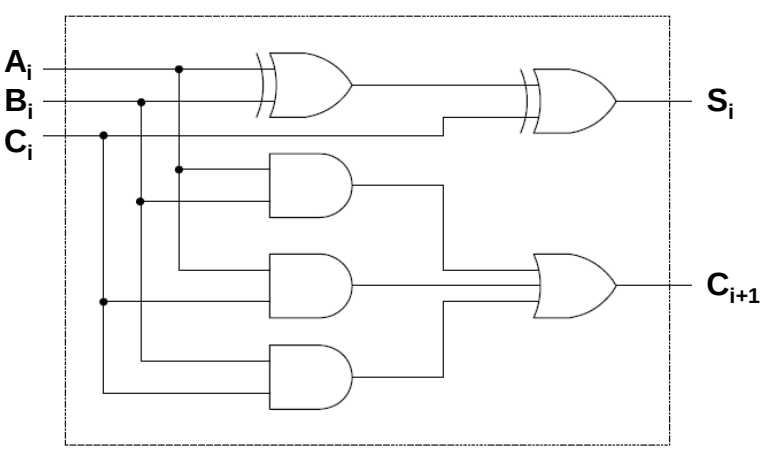
Combinação de meio-somadores:

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = (A_i \oplus B_i) \bullet C_i + A_i \bullet B_i$$

➔ **Latência de C_{i+1}: 3**

Somador completo



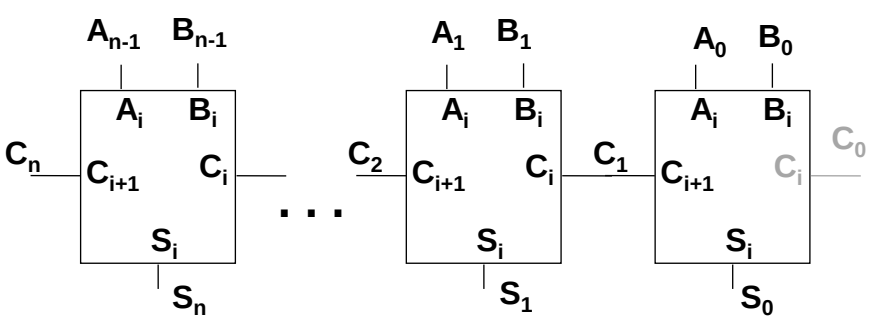
$S_i = A_i \oplus B_i \oplus C_i$; $C_{i+1} = A_i \bullet C_i + B_i \bullet C_i + A_i \bullet B_i$

➔ **Latência de C_{i+1}: 2**

Somador binário de n bits

- **Pergunta**: dado um somador completo de 1 bit, como obter um somador de n bits?
- **Resposta simples**: associação em série (cascateamento) de *n* somadores completos
- **Resultado**: "somadores com propagação de vai-um", ou *ripple adders*:
  - Cada somador completo trata um bit da soma
  - O "Vai-um" de um estágio se conecta ao "Vem-um" do estágio seguinte.
  - O "Vem-um" do estágio menos significativo costuma ficar em 0 (ou usa-se um meio-somador)

Somador binário de n bits



Somador binário de *n* bits com propagação de "vai-um"

➔ **Problema?**

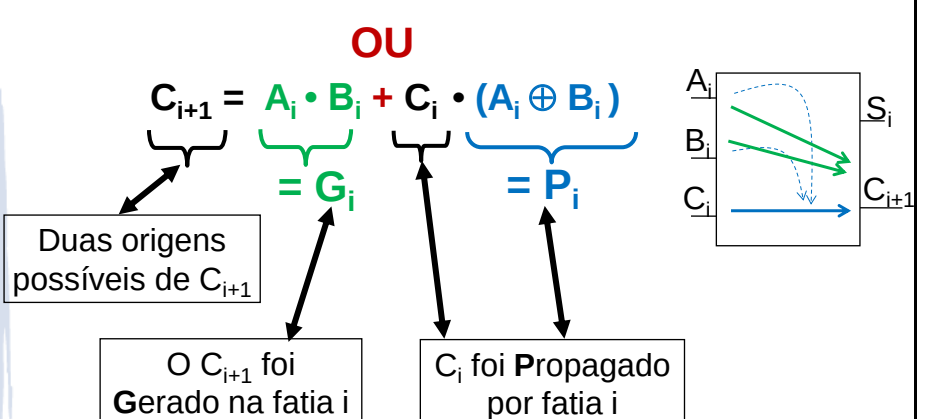
Latência de propagação de "vai-um" do estágio menos até o mais significativo é proporcional a n…

Somador binário de n bits

- **Pergunta**: como solucionar esse problema de latência?
- **Resposta**: montando circuito que calcula saída a partir das entradas diretamente
  - Ex.: Soma de produtos levaria a um atraso de apenas dois níveis (AND e OR)…
- **Resultado**: "somadores com antecipação de vai-um", ou *carry look-ahead*
- Vamos tentar construir esse circuito...

Somador binário de n bits

- Antecipação de carry (*carry look-ahead*)



Somador binário de n bits

- Antecipação de carry (*carry look-ahead*): generalizando

$$C_1 = G_0 + P_0 \cdot C_0$$

$(A_i \cdot B_i)$ $\quad$ $(A_0 \oplus B_0)$

- Prosseguindo com a generalização:

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$$
$$= \underbrace{G_1 + P_1 \cdot G_0}_{\text{geração}} + \underbrace{P_1 \cdot P_0}_{\text{propagação}} \cdot C_0$$

Somador binário de n bits

- Prosseguindo com a generalização:

$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$$
$$= \underbrace{G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot P_0 \cdot G_0 +}_{\text{Geração nas fatias 0, 1 ou 2}}$$
$$\underbrace{P_2 \cdot P_1 \cdot P_0}_{\text{Propagação nas fatias 0, 1 e 2}} \cdot C_0$$

Somador binário de n bits

- Prosseguindo com a generalização:

$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0)$$

$$= \underbrace{G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 +}_{\text{Geração nas fatias 1, 2 ou 3}}$$

$$\underbrace{P_3 \cdot P_2 \cdot P_1 \cdot G_0}_{\text{Geração na fatia 0}} + \underbrace{P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0}_{\text{Propagação nas fatias 0, 1, 2 e 3}}$$

- C_4: útil se bloco for conectado em outro bloco usando carry ripple.

Somador binário de n bits

- Se conexão for feita com outro bloco usando carry look-aheda, podemos escrever:

$$C_{BLOCO4} = G_{BLOCO4} + P_{BLOCO4} \cdot C_0$$

onde:

$$G_{BLOCO4} = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0$$

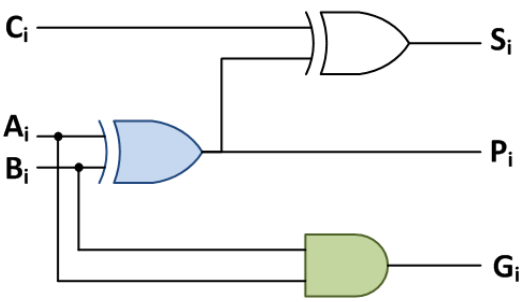
com $G_i = A_i \cdot B_i$

$$P_{BLOCO4} = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$

com $P_i = (A_i \oplus B_i)$

Somador binário de n bits

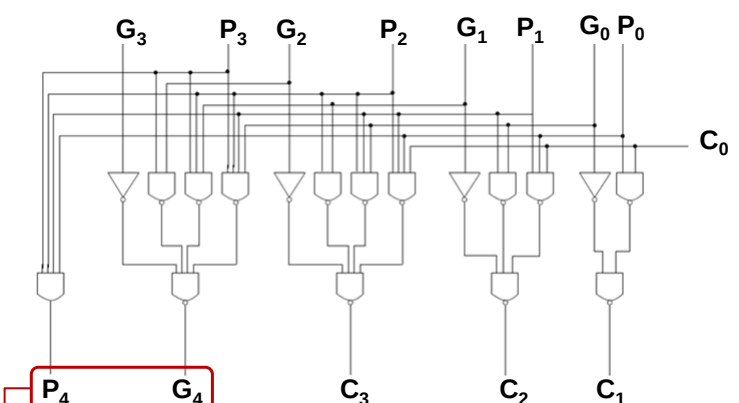
- Carry look-ahead: **Unidade somadora** de 1 bit



$G_i = A_i \bullet B_i$; $P_i = (A_i \oplus B_i)$

Somador binário de n bits

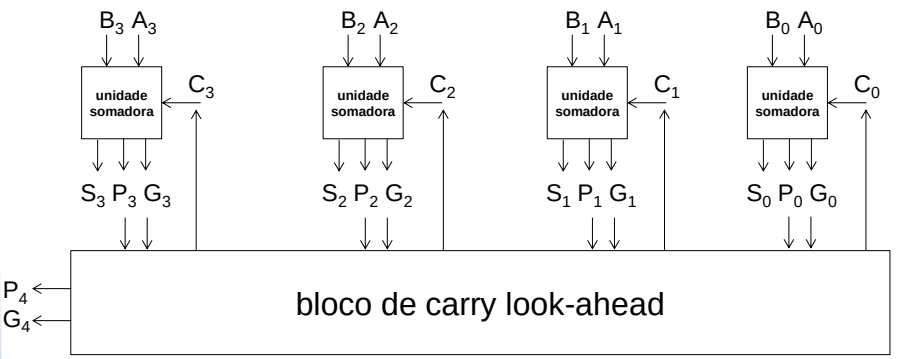
- Carry look-ahead: bloco para **cálculo antecipado** do carry



Podem ser usados para calcular C4 (útil para conectar blocos do tipo carry ripple) ou diretamente em um outro bloco de carry look-ahead

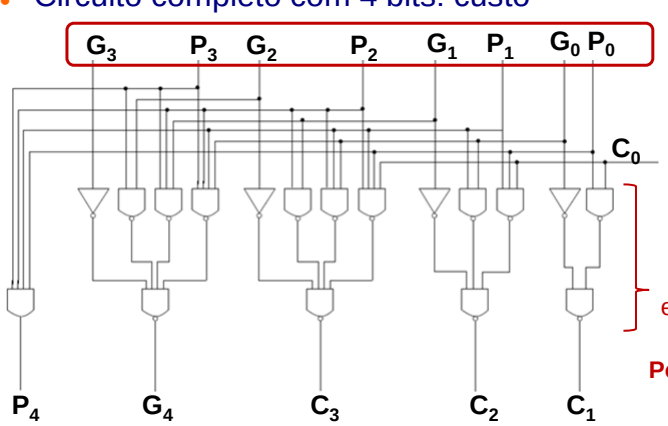
Somador binário de n bits

- Circuito completo com 4 bits

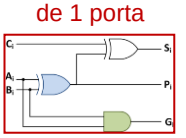


Somador binário de n bits

- Circuito completo com 4 bits: custo



P e G: latência de 1 porta



Latência extra de 2

Por que não fazer em 2 níveis?!

Fazer em 2 níveis seria inviável: 9 entradas ➔ 2^9 mintermos possíveis...
Solução é um compromisso entre latência e tamanho do circuito

Somador BCD

- Relembrando: código BCD
  - 4 bits, representação direta binária de 0 a 9
- Soma BCD: usa a adição binária com 4 bits, mas não é idêntica a ela...
  - Requer correção dos valores inválidos:
    - Aqueles acima de 9
  - Requer correção do “vai-um decimal”:
    - Diferente do vai-um hexadecimal (de 4 bits)

Somador BCD

Comparação entre soma binária (4bits) e BCD

Resultado da soma de 4 bits	Soma Hexa	Vai-um Hexa	Soma BCD	Vai-um BCD	Correção
0 a 9	0 a 9	0	0 a 9	0	-
10 a 15	10-15 (A-F)	0	0 a 5	1	Soma 6 C_{BCD}=1
16 a **19**	0-3	1	6 a 9	1	Soma 6 C_{BCD}=C_{hexa}

Obs.: 9+9+1 (“vem-um”)

Somador BCD

- Correção da soma BCD em relação ao resultado hexadecimal:
  - Se resultado da soma entre A e F, **OU**
  - Se vai-um$_{hexa}$ = 1

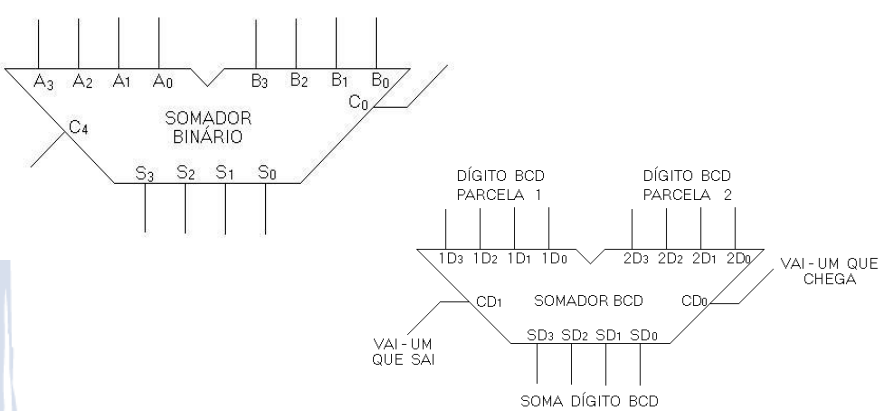
Lógica adicional sobre saída do somador hexa

- Nesse caso, ação a tomar:
  - **Somar 6** (0110) à soma hexa
  - Vai-um$_{BCD}$=1

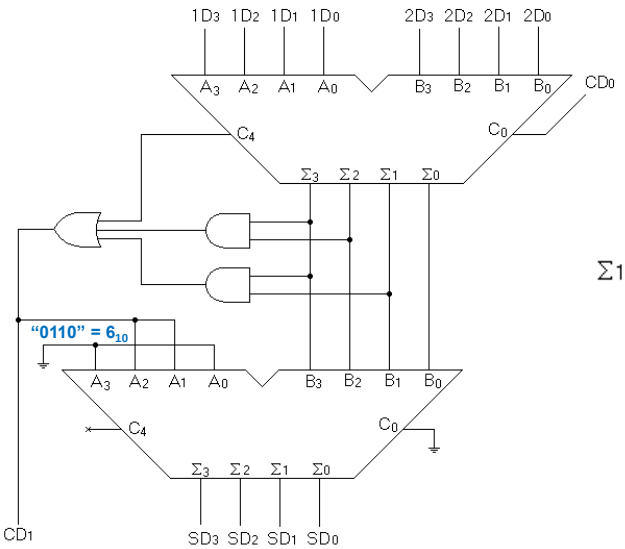
Necessário um segundo somador

Somador BCD

- Desafio: obter um Somador BCD a partir de um Somador Binário de (4 bits)



Somador BCD



10-16

Σ1Σ0 \ Σ3Σ2	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	0	1	1
10	0	0	1	1

Subtrator binário

- Pode ser construído da mesma forma que construímos o somador binário: a partir da tabela verdade…
- ... ou usar o circuito do somador!

entradas			saídas: +		saídas: –	
c_{IN} [b_{IN}]	x	y	Σ (soma)	c_{OUT}	d (diferença)	b_{OUT}
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	1	0	1	1
1	0	1	0	1	0	1
1	1	0	0	1	0	0
1	1	1	1	1	1	1

Subtrator binário

M_i	S_i	B_i	R_i	B_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Tabela Verdade do Subtrator Completo:

$M_i - S_i - B_i$

minuendo – subtraendo – borrow ("empresta um")

$$R_i = M_i' \cdot S_i' \cdot B_i + M_i' \cdot S_i \cdot B_i' + M_i \cdot S_i' \cdot B_i' + M_i \cdot S_i \cdot B_i$$

$$B_{i+1} = M_i' \cdot S_i + B_i \cdot M_i' + B_i \cdot S_i$$

Subtrator binário

$$R_i = M_i'.S_i'.B_i + M_i'.S_i.B_i' + M_i.S_i'.B_i' + M_i.S_i.B_i$$

Comparando com somador

$$R_i = M_i \oplus S_i' \oplus B_i'$$
$$\Updownarrow \quad \Updownarrow \quad \Updownarrow \quad \Updownarrow$$
$$Si = Ai \oplus Bi \oplus Ci$$

negar entrada S_i

saída é B'_{i+1}: usá-la no próximo estágio

$$B_{i+1} = M_i'.S_i + B_i.M_i' + B_i.S_i =$$
$$[B_{i+1}]' = [M_i'.S_i + (M_i' + S_i) \cdot B_i]' =$$
$$B_{i+1}' = M_i \cdot S_i' + (M_i + S_i') \cdot B_i' =$$

Comparando com somador

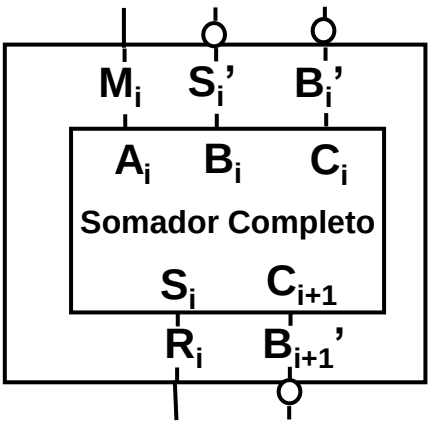
$$B_{i+1}' = M_i \bullet S_i' + M_i \bullet B_i' + S_i' \bullet B_i'$$
$$\Updownarrow \quad \Updownarrow \quad \Updownarrow \quad \Updownarrow \quad \Updownarrow \quad \Updownarrow \quad \Updownarrow$$
$$C_{i+1} = A_i \bullet B_i + A_i \bullet C_i + B_i \bullet C_i$$

Subtrator binário

$R_i = M_i \oplus S_i' \oplus B_i'$

$Si = Ai \oplus Bi \oplus Ci$

Subtrator Completo



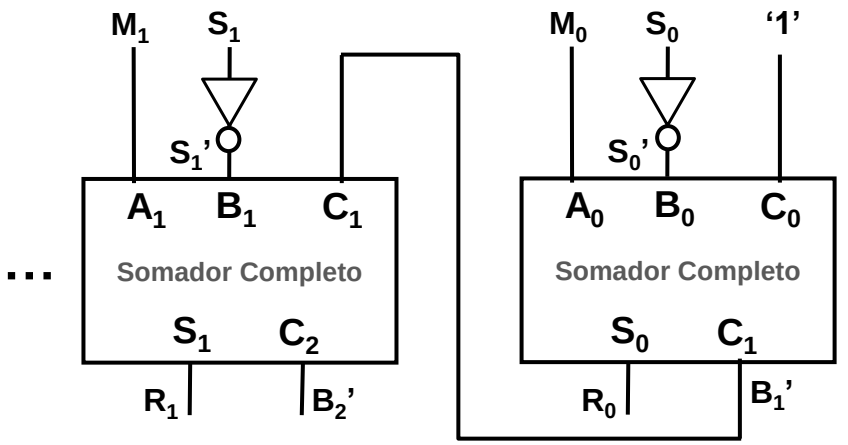
$B_{i+1}' = M_i \cdot S_i' + M_i \cdot B_i' + S_i' \cdot B_i'$

$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$

Subtrator binário

- Essa correspondência tem uma razão simples: m – s = m + (–s)
  - Logo: **operação de subtração** pode ser construída **a partir** de uma **operação de adição** em um **somador completo**
- Procedimento:
  - **Complementar *bit* a *bit*** o subtraendo (**S_i**)
  - Somar resultado a minuendo (**M_i**), lembrando de fazer **$C_0 = 1$** no somador completo para obter o complemento de 2 de (**S_i**)

Subtrator binário



Nota: no final, precisa detectar transbordo (verificar se sinais na entrada diferem dos sinais de saída)

Lição de Casa

- Leitura Obrigatória:
  - Capítulo 6 do Livro Texto, ênfase em 6.4, 6.5, 6.7, 6.8.
- Exercícios obrigatórios:
  - Capítulo 6 do Livro Texto.

Bibliografia Adicional

- Fregni, Edson; Saraiva, Antônio. *Engenharia do Projeto Lógico Digital*. Editora Edgard Blücher Ltda. São Paulo, SP, Brasil, 1.995;
- Ranzini, Edith; Fregni, Edson. Teoria da Comutação: Introdução aos Circuitos Digitais (Partes 1 e 2). Notas de Aula de PCS214, PCS/EPUSP, Agosto de 1.996.
- Tocci, Ronald; Widmer, Neal S. Sistemas Digitais – Princípios e Aplicações. 8ª Edição, Pearson/Prentice-Hall, São Paulo, 2.003.