



Computação Física



Números e bases numéricas



❑ Conceito de número

- Um número é um conjunto de símbolos (algarismos), com um determinado **peso** cada, destinados a designar uma quantidade ou um código.
- A numeração romana não é ponderada, na medida em que os seus símbolos não têm uma significância posicional.

Números naturais (conjunto \mathbb{N})

Para já, irão considerar-se os números inteiros e positivos. Este conjunto recebe o nome de números naturais, porque é o conjunto que se aprende em primeiro lugar, e o mais simples.



❑ Sistemas de numeração

- Código numérico com significância posicional
- Em cada posição, ter-se-á um algarismo, cujo peso tem a ver com essa posição e cuja quantificação é relativa à base numérica desse número.
- A base determina a variedade de algarismos (símbolos) necessários.
- Os sistemas de numeração têm regras operatórias sobre as quantidades representadas.
- As operações realizam-se sobre os algarismos do mesmo peso, provocando arrasto para o peso seguinte.



□ Significância posicional

$$N = \sum_{i=0}^{L-1} A_i B^i = A_0 B^0 + A_1 B^1 + \dots + A_{L-1} B^{L-1}$$

N – Valor a representar

B – Base de numeração

L – Comprimento (número de algarismos de N)

A_i – Algarismos do número, contendo apenas símbolos possíveis em B

B^i – Peso, ou significância, do algarismo A_i

i – Índice de iteração, indo da direita para a esquerda

Exemplo: $(1987)_{10} = 7 \times 10^0 + 8 \times 10^1 + 9 \times 10^2 + 1 \times 10^3 = 7 + 80 + 900 + 1000 = 1987$



□ Bases de numeração mais utilizadas

- Base 10 (*decimal*) – Base que utilizamos no dia a dia, contendo dez algarismos (símbolos), de 0 a 9.
- Base 2 (*binário*) – Base utilizada em cálculos digitais. Têm-se dois símbolos: 0 e 1. Cada dígito nesta base chama-se *bit* – *binary digit*. Na verdade, o bit é uma medida de informação.
- Base 8 (*octal*) – “Auxiliar” da base 2. Oito símbolos: 0 a 7.
- Base 16 (*hexadecimal*) – “Auxiliar” da base 2. Dezassexis símbolos: 0 a F.

Em qualquer base, de acordo com o comprimento do número utilizado, tem-se uma capacidade máxima de representação que é B^L .



□ Conversão entre bases de numeração

Tendo um número, este é representável de forma diferente consoante a base.

A conversão para decimal faz-se diretamente pela significância posicional.

- Conversão binário → decimal:

$$(0110)_2 \rightarrow N = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 = 2 + 4 = (6)_{10}$$

- Conversão octal → decimal :

$$(614)_8 \rightarrow N = 4 \times 8^0 + 1 \times 8^1 + 6 \times 8^2 = 4 + 8 + 6 \times 64 = (396)_{10}$$

- Conversão hexadecimal → decimal:

$$(6A4)_{16} \rightarrow N = 4 \times 16^0 + 10 \times 16^1 + 6 \times 16^2 = 4 + 160 + 6 \times 256 = (1700)_{10}$$

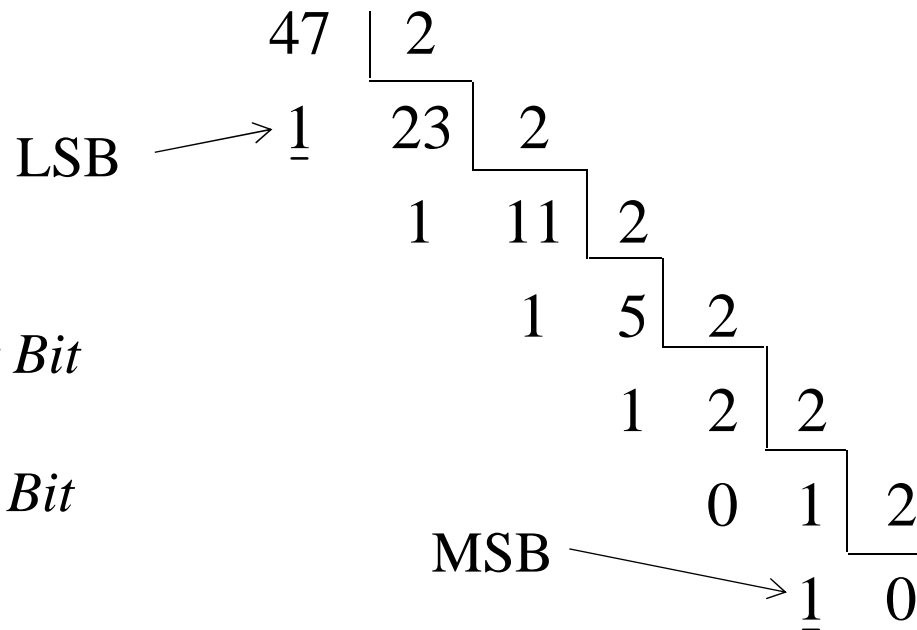


❑ Conversão entre bases de numeração

A conversão de base 10 para a base 2 faz-se através de divisões sucessivas por 2, até obter quociente 0. Vão-se aproveitando os restos das divisões.

- Conversão decimal \rightarrow binário :

$$(47)_{10} \rightarrow (101111)_2$$



MSB – *Most Significant Bit*

LSB – *Least Significant Bit*



□ Conversão entre bases de numeração

A conversão de base 10 para as bases 8 e 16 faz-se através de divisões sucessivas por 8 ou 16, respetivamente, até obter quociente 0. Vão-se aproveitando os restos das divisões.

- Conversão decimal \rightarrow octal :

$$(47)_{10} \rightarrow (57)_8$$

$$\begin{array}{r|l} 47 & 8 \\ \hline 7 & 5 \quad 8 \\ & \underline{5} \quad 0 \end{array}$$

- Conversão decimal \rightarrow hexadecimal :

$$(47)_{10} \rightarrow (2F)_{16}$$

$$\begin{array}{r|l} 47 & 16 \\ \hline 15 & 2 \quad 16 \\ & \underline{2} \quad 0 \end{array}$$

A – 10

B – 11

C – 12

D – 13

E – 14

F – 15



□ Conversão entre bases de numeração

A conversão de base 2 para as bases 8 e 16 faz-se, agrupando o número em conjuntos de 3 ou 4 bits, respetivamente. Dentro de cada um, converte-se como se fosse para decimal.

- Conversão binário → octal :

$$(10011011)_2 \rightarrow (\underbrace{010} \underbrace{011} \underbrace{011})_2 = (233)_8$$

- Conversão binário → hexadecimal :

$$(1001011011)_2 \rightarrow (\underbrace{0010} \underbrace{0101} \underbrace{1011})_2 = (25B)_{16}$$



□ Conversão entre bases de numeração

A conversão de base 8 ou base 16, para base 2, realiza-se fazendo corresponder um algarismo a um grupo de 3 ou 4 bits, respetivamente.

- Conversão octal → binário :

$$(4765)_8 \rightarrow (\underbrace{100}_4 \underbrace{111}_7 \underbrace{110}_6 \underbrace{101}_5)_2$$

- Conversão hexadecimal → binário :

$$(2EB)_{16} \rightarrow (\underbrace{0010}_2 \underbrace{1110}_{E(14)} \underbrace{1011}_{B(11)})_2$$



❑ Arduino e bases de numeração

Na linguagem de programação do Arduino existe a possibilidade de se especificarem números nas bases numéricas já abordadas.

Por *default*, a base numérica de trabalho é a decimal.

Base	Formato	Exemplo	Comentário
10 (decimal)	nenhum	1 2 3	
2 (binário)	Prefixo “B”	B110111	Até 8 bits de comprimento. Só 0 e 1 são válidos.
8 (octal)	Prefixo “0”	0765	Só caracteres 0-7 são válidos.
16 (hexadecimal)	Prefixo “0x”	0xF2E8	Só caracteres 0-9 e A-F (ou a-f) são válidos.



❑ Arduino e bases de numeração

No *serial monitor*, para especificar a escrita de números de acordo com uma determinada base numérica, devem utilizar-se os seguintes modificadores na chamada ao método `Serial.print()` e `Serial.println()`.

Base	Exemplo	Resultado
10 (decimal)	<code>Serial.print(78, DEC)</code>	78
2 (binário)	<code>Serial.print(78, BIN)</code>	1001110
8 (octal)	<code>Serial.print(78, OCT)</code>	116
16 (hexadecimal)	<code>Serial.print(78, HEX)</code>	4E



❑ Operação soma entre dois números binários

Esta operação é idêntica à praticada em base 10, havendo arrasto (*carry*) para o peso seguinte, sempre que se ultrapasse a capacidade do peso corrente.

Exemplo: Somar $A = 100101 (37)_{10}$ com $B = 110111 (55)_{10} \rightarrow S = (92)_{10}$

$$\begin{array}{rcccccccc} & & (C_5) & & (C_4) & & (C_3) & & (C_2) & & (C_1) & & \\ & & 0 & \leftarrow & 0 & \leftarrow & 1 & \leftarrow & 1 & \leftarrow & 1 & \leftarrow & \\ A & = & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 & = & 1 & 0 & 0 & 1 & 0 & 1 \\ B & = & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 & = & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline +B & + & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & \\ S & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & & & & & & \end{array}$$

37
55
92

Olhando a cada índice (peso), de uma forma geral, tem-se $S_i = A_i + B_i + C_i$



❑ Operação subtração entre dois números binários

Algoritmo da subtração (em base 10):

- Tem que se verificar qual é a “distância” do subtrativo para o aditivo

Aditivo	→	49	Peso 0: 5 para 9, são 4 unidades de distância
Subtrativo	→	$\begin{array}{r} - 05 \\ \hline 44 \end{array}$	Peso 1: 0 para 4, são 4 unidades de distância

- E se o algarismo subtrativo for superior ao do aditivo?

$\begin{array}{r} 41 \\ - 28 \\ \hline 13 \end{array}$	<p>Peso 0: Como 8 é superior a 1, a distância tem de ser calculada “pedindo emprestado” uma unidade ao peso superior (1). Pedese $10^1 = 10$ emprestado e este fica com menos uma unidade. Assim, vê-se a distância entre 8 e 11. Dá 3.</p>
--	--

Peso 1: Distância de 2 para 3, porque este peso “emprestou” uma unidade sua ao peso mais baixo e agora já não tem 4. Dá 1.



❑ Operação subtração entre dois números binários

Algoritmo da subtração (em base 10):

- E se o peso imediatamente superior não tiver para emprestar?

$$\begin{array}{r} 201 \\ - 049 \\ \hline 2 \end{array}$$

Peso 0: Como 9 é maior do que 1, tem que “pedir emprestado” às dezenas (peso 1), mas este não tem para lhe emprestar. Tem então que “pedir emprestado” uma unidade às centenas (peso 2). Assim, vê-se a distância entre 9 e 101. Dá 92. Coloca-se o 2 no peso 0 do resultado e o 9 soma-se ao peso 1 de aditivo (não precisa, devolve), e prossegue-se.

$$\begin{array}{r} 191 \\ - 049 \\ \hline 152 \end{array}$$

Peso 1: Distância de 4 para 9. Dá 5.

Peso 2: Distância de 0 para 1. Dá 1.

Final: 152



□ Números relativos (conjunto \mathbb{Z})

- E se o subtrativo for superior ao aditivo? Neste caso, não existe representação sob a forma de número natural. Por este motivo, surgiu o conjunto dos números relativos \mathbb{Z} , englobando positivos e negativos.
- Noção de números simétricos: são dois números, tais que somados um com o outro, o resultado é igual a zero.
- Nos computadores e, em geral, nas máquinas de cálculo, existe uma capacidade máxima de representação numérica porque o *hardware* é finito. Essa representação (n° de dígitos – L), deverá contemplar tanto os números positivos como os negativos.



□ Números relativos (conjunto \mathbb{Z})

- Por exemplo, em decimal ($B = 10$), uma representação com dois dígitos ($L = 2$), permitirá um total de $B^L = 10^2 = 100$ configurações diferentes.

00
⋮
99

E os números negativos? Estes têm que ser também representados dentro destas cem (100) combinações diferentes.

Código de complementos (complemento para dez)

Para obter o simétrico de um número, que se determinar qual a distância desse número para B^L , neste caso, 100.

Por exemplo, o simétrico de +1 (o “-1”), será:

$$\begin{array}{r|rr} & 1 & 0 & 0 \\ - & & 0 & 1 \\ \hline & & 9 & 9 \end{array} \longrightarrow -1$$



❑ Código de complementos (complemento para dez)

O simétrico de +2 (o “-2”), será:

$$\begin{array}{r|rr} 1 & 0 & 0 \\ - & 0 & 2 \\ \hline & 9 & 8 \end{array} \longrightarrow -2$$

O simétrico de +10 (o “-10”), será:

$$\begin{array}{r|rr} 1 & 0 & 0 \\ - & 1 & 0 \\ \hline & 9 & 0 \end{array} \longrightarrow -10$$

Têm que existir, na mesma, 100 combinações diferentes. Por definição, o zero é positivo, logo, tendo metade das configurações positiva e a outra metade negativa, tem-se:

Positivos: 0 a 49

Negativos: 99 a 50

+ 49	49
	⋮
+ 10	10
	⋮
+ 2	02
	01
0	00
- 1	99
- 2	98
	⋮
- 10	90
	⋮
- 50	50



□ Código de complementos (complemento para dez)

Dígito de sinal: 0 a 4 → positivo

5 a 9 → negativo

O simétrico de -10 (código 90) é:

$$\begin{array}{r|l}
 1 & 0 \ 0 \\
 - & 9 \ 0 \\
 \hline
 & 1 \ 0 \longrightarrow +10
 \end{array}$$

Ao somar dois números simétricos, o resultado tem de dar zero.

Exemplo: 9 8 (-2)

+ 0 2 (+2)

1 0 0 → 0

O dígito das centenas está fora da representação.

Truncatura a dois dígitos

+49	49
	⋮
+10	10
	⋮
+2	02
	01
0	00
-1	99
-2	98
	⋮
-10	90
	⋮
-50	50



□ Código de complementos (complemento para dois)

Em binário, ter-se-á uma representação finita a L bits.

$$L = 4 : 2^4 = 16 \text{ configurações}$$

- 8 positivas: 0 a +7: 0000 a 0111
- 8 negativas: -1 a -8: 1111 a 1000

O dígito de maior peso indica o sinal algébrico do número:

0, se for positivo
1, se for negativo

} → bit de sinal

+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000



❑ Código de complementos (complemento para dois)

Obtenção de números simétricos em binário

Algoritmo 1: Complementar todos os bits (complemento para 1) e somar 1.

Algoritmo 2: Percorrer o número da direita para a esquerda até encontrar o primeiro 1, mantê-lo, e inverter todos os bits que se seguem até chegar à extremidade esquerda.

Exemplos: 0010 → 1110

 (+2) (−2)

 0101 → 1011

 (+5) (−5)

Representação a 4 bits

Pesos em complemento para 2:

-8	4	2	1
----	---	---	---

Pesos em binário natural:

8	4	2	1
---	---	---	---



□ Representação com um número finito de bits

Representação com 4 / L bits:

- Binário natural (“só positivos”):

$$\{0, \dots, 2^4 - 1\} = \{0, \dots, 15\}$$

$$\{0, \dots, 2^L - 1\}$$

- Código de complementos (positivos e negativos):

$$\{-2^{(4-1)}, \dots, 0, \dots, +2^{(4-1)} - 1\} = \{-8, \dots, 0, \dots, +7\}$$

$$\{-2^{(L-1)}, \dots, 0, \dots, +2^{(L-1)} - 1\}$$

	N		Z
15	1111	+7	0111
14	1110	+6	0110
13	1101	+5	0101
12	1100	+4	0100
11	1011	+3	0011
10	1010	+2	0010
9	1001	+1	0001
8	1000	0	0000
7	0111	-1	1111
6	0110	-2	1110
5	0101	-3	1101
4	0100	-4	1100
3	0011	-5	1011
2	0010	-6	1010
1	0001	-7	1001
0	0000	-8	1000



□ Representação com um número finito de bits

- Extensão de sinal:

Em representações de maiores dimensões, os números positivos ficam com os bits à esquerda todos a zero e, os negativos, com esses bits todos a 1.

Exemplos:

Número	Representação a 4 bits	Representação a 16 bits (int)
+3	0011	0000000000000011
+1	0001	0000000000000001
-8	1000	11111111111111000
-3	1101	1111111111111101
-1	1111	1111111111111111



□ Subtração entre números binários

- Subtração direta
- Adição ao simétrico do subtrativo (é assim que o *hardware* opera)

Exemplos com subtração direta e adição com simétrico do subtrativo:

$$\begin{array}{r} 7 \\ - 3 \\ \hline + 4 \end{array} \quad \begin{array}{r} 0 \ 1 \ 1 \ 1 \\ - 0 \ 0 \ 1 \ 1 \\ \hline 0 \ 1 \ 0 \ 0 \end{array}$$

$$\begin{array}{r} 2 \\ - 5 \\ \hline - 3 \end{array} \quad \begin{array}{r} 0 \ 0 \ 1 \ 0 \\ - 0 \ 1 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 0 \ 1 \end{array}$$

CBN: $B_w = 1$. Deu 13 e ficou a dever 16.
CBC: O resultado deu -3. Correto.

$$\begin{array}{r} 0 \ 1 \ 1 \ 1 \quad (+7) \\ + 1 \ 1 \ 0 \ 1 \quad (-3) \\ \hline 1 \ 0 \ 1 \ 0 \ 0 \quad (+4) \end{array}$$

$$\begin{array}{r} 0 \ 0 \ 1 \ 0 \quad (+2) \\ + 1 \ 0 \ 1 \ 1 \quad (-5) \\ \hline 0 \ 1 \ 1 \ 0 \ 1 \quad (-3) \end{array}$$

CBN: $C_y = 0 \rightarrow B_w = 1$ (O resultado deu 13 - incoerente)
CBC: O resultado deu -3. Correto.

Carry = 1: é desprezado, pois fica fora da representação.
Borrow = 0: não fica a dever nada. Arrasto, numa subtração, para um peso de ordem superior. Resultado correto.



❑ Indicadores de erro em código binário natural

Como a capacidade de representação do *hardware* é finita, poderão haver resultados decorrentes de somas ou de subtrações que não consigam ficar dentro dessa representação.

Indicadores de erro:

Código binário natural	{	Operação soma: <i>Carry</i> (<i>Cy</i>)
		Operação subtração: <i>Borrow</i> (<i>Bw</i>)

Estes ocorrerão sempre que o resultado não consiga ficar dentro da representação, originando um resultado incoerente com os operandos.



❑ Indicador de erro em código binário de complementos

Em código de complementos, caso o resultado decorrente de um cálculo não se situe dentro da gama de valores permitidos, não existe representação possível. Tal significa que o resultado apresentado nesse número finito de bits será incoerente com a operação e o valor dos operandos.

Indicador de erro (operações soma e subtração): *Overflow (Ov)*

Por definição, o *overflow* ocorre se, tendo como operandos dois números positivos, o resultado for negativo, e vice-versa.

Sempre que se somar (ou subtrair) números de sinais algébricos opostos, o resultado ficará sempre dentro da gama de representação possível.



❑ Indicadores de erro em código binário natural e de complementos

Consoante se esteja a operar no domínio dos números naturais ou no dos números relativos, o número (código binário) resultante de uma operação terá o respetivo significado nesse domínio. Caso o resultado de uma operação aritmética “pretenda” dar um número fora da gama possível, tal será prontamente sinalizado pelo indicador de erro desse domínio.

Habitualmente, o *hardware* realiza as subtrações à custa da adição com o simétrico do subtrativo.

Vejamos alguns exemplos de situações e respetivo resultado e indicação de erro (se for o caso).



□ Exemplos de somas

Soma

$(\mathbb{N})_{10}$

$(\mathbb{Z})_{10}$

$$\begin{array}{rcl} A & = & 1 \ 0 \ 0 \ 0 \\ + \ B & = & + \ 0 \ 1 \ 1 \ 1 \\ \hline & & 0 \ \underline{1 \ 1 \ 1 \ 1} \end{array}$$

$$\begin{array}{r} 8 \\ + \ 7 \\ \hline 15 \end{array}$$

$$\begin{array}{rcl} (-8) & & Cy = 0 \\ + \ (+7) & & Ov = 0 \\ \hline (-1) & & \end{array}$$

$$\begin{array}{rcl} A & = & 1 \ 0 \ 1 \ 0 \\ + \ B & = & + \ 1 \ 0 \ 1 \ 1 \\ \hline & & 1 \ \underline{0 \ 1 \ 0 \ 1} \end{array}$$

$$\begin{array}{r} 10 \\ + \ 11 \\ \hline 5 \end{array}$$

$$\begin{array}{rcl} (-6) & & Cy = 1 \\ + \ (-5) & & Ov = 1 \\ \hline (+5) & & \end{array}$$



□ Exemplos de somas

Soma

$(\mathbb{N})_{10}$

$(\mathbb{Z})_{10}$

$$\begin{array}{rcl} A & = & 0 \ 1 \ 1 \ 0 \\ + \ B & = & + \ 0 \ 1 \ 1 \ 0 \\ \hline & & 0 \ \underline{1 \ 1 \ 0 \ 0} \end{array}$$

$$\begin{array}{r} 6 \\ + \ 6 \\ \hline 12 \end{array}$$

$$\begin{array}{rcl} (+6) & & Cy = 0 \\ + \ (+6) & & Ov = 1 \\ \hline (-4) & & \end{array}$$

$$\begin{array}{rcl} A & = & 1 \ 0 \ 1 \ 0 \\ + \ B & = & + \ 1 \ 1 \ 1 \ 0 \\ \hline & & 1 \ \underline{1 \ 0 \ 0 \ 0} \end{array}$$

$$\begin{array}{r} 10 \\ + \ 14 \\ \hline 8 \end{array}$$

$$\begin{array}{rcl} (-6) & & Cy = 1 \\ + \ (-2) & & Ov = 0 \\ \hline (-8) & & \end{array}$$



□ Exemplos de subtrações

Subtração

$$\begin{array}{r} A = 0110 \\ - B = -0001 \\ \hline 0 \quad \underline{0101} \end{array}$$

⇓

$$\begin{array}{r} A = 0110 \\ \bar{B} = 1110 \\ + 0001 \\ \hline \textcircled{1} \quad \underline{0101} \end{array}$$

→ $Cy = 1 \rightarrow \boxed{B_W = 0}$

$(N)_{10}$

$$\begin{array}{r} 6 \\ - 1 \\ \hline 5 \end{array}$$

$(Z)_{10}$

$$\begin{array}{r} (+6) \\ - (+1) \\ \hline (+5) \end{array} \quad \begin{array}{l} B_W = 0 \\ O_V = 0 \end{array}$$

Quando a subtração é feita por intermédio de uma soma, o Cy daí resultante deve ser transformado em B_W .



□ Exemplos de subtrações

Subtração

$$\begin{array}{r} A = 0110 \\ - B = -0111 \\ \hline 1 \quad \underline{1111} \end{array}$$

⇓

$$\begin{array}{r} A = 0110 \\ \overline{B} = 1000 \\ + 0001 \\ \hline \textcircled{0} \quad \underline{1111} \end{array}$$

→ $Cy = 0 \rightarrow \boxed{Bw = 1}$

$(N)_{10}$

$$\begin{array}{r} 6 \\ - 7 \\ \hline 15 \end{array}$$

$(Z)_{10}$

$$\begin{array}{r} (+6) \\ - (+7) \\ \hline (-1) \end{array} \quad \begin{array}{l} Bw = 1 \\ Ov = 0 \end{array}$$

Na subtração direta, o bit mais à esquerda, que fica fora da representação, é diretamente o indicador (*flag*) de *Borrow*.



□ Exemplos de subtrações

Subtração	$(\mathbb{N})_{10}$	$(\mathbb{Z})_{10}$	
$A = 1\ 0\ 1\ 0$	10	(-6)	$Bw = 0$
$- B = - 0\ 1\ 1\ 1$	$- 7$	$- (+7)$	$ Ov = 1$
$\hline \textcircled{0}\ 0\ 0\ 1\ 1$	$\hline 3$	$\hline (+3)$	
\Downarrow			
$A = 1\ 0\ 1\ 0$			
$\overline{B} = 1\ 0\ 0\ 0$			
$+ 0\ 0\ 0\ 1$			
$\hline \textcircled{1}\ 0\ 0\ 1\ 1$			
\rightarrow	$Cy = 1 \rightarrow$	$\boxed{Bw = 0}$	



□ Exemplos de subtrações

<u>Subtração</u>	$(\mathbb{N})_{10}$	$(\mathbb{Z})_{10}$	
$A = 0111$	7	(+7)	$B_w = 1$
$- B = -1010$	-10	$-(-6)$	$O_v = 1$
$\hline \textcircled{1} \quad 1101$	$\hline 13$	$\hline (-3)$	
\Downarrow			
$A = 0111$			
$\overline{B} = 0101$			
$+ 0001$			
$\hline \textcircled{0} \quad 1101$			
	$\rightarrow Cy = 0$		$\rightarrow B_w = 1$



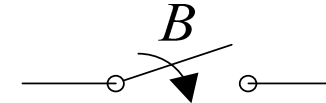
Circuitos combinatórios



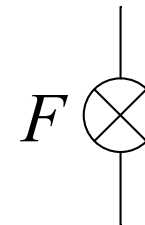
□ Variáveis binárias (booleanas)

Neste contexto, considere-se:

- A e B , variáveis binárias independentes, correspondentes a interruptores com contactos *normally open* (NO). Atribui-se o valor lógico “1” quando são atuados e “0” quando não atuados.



- Considere-se F , variável binária dependente, correspondente a lâmpada acesa (“1”) ou apagada (“0”).





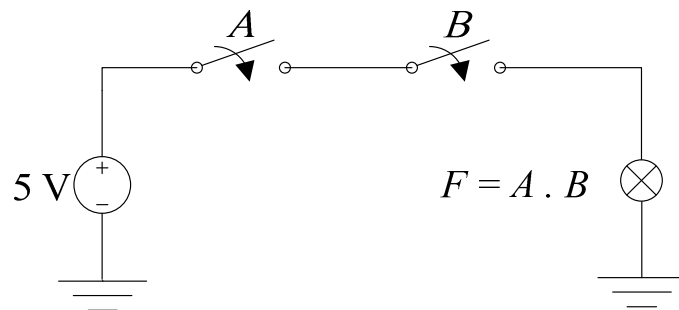
□ Operações lógicas

- Existem várias operações lógicas elementares – AND, OR e NOT.
- A cada operação está atribuído um operador, tal que se podem escrever expressões lógicas à base de operações elementares.
- A partir das operações lógicas elementares, é possível definir outras operações (NAND, NOR, XOR e XNOR), que embora não sendo elementares, podem ser tratadas como tal, com o devido conhecimento das regras a que obedecem.



❑ Operação lógica AND (produto lógico)

Definição: Operação sobre n variáveis, que só toma o valor 1 quando todas essas variáveis tiverem o valor 1.



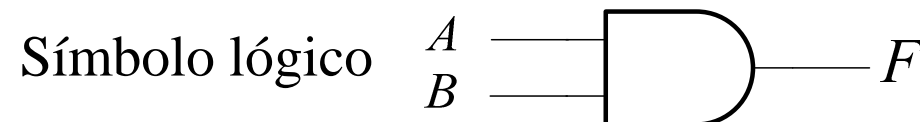
A lâmpada acende se
A e B forem atuados

Expressão
algébrica

$$F = A.B$$

Tabela de
verdade

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



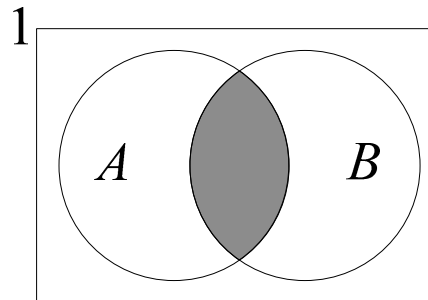


□ Operação lógica AND (produto lógico)

Propriedades:

$$\left. \begin{array}{l} A.0 = 0 \\ A.1 = A \end{array} \right\} \text{Teoremas da interseção}$$
$$A.A = A \quad \text{Teorema da idem potência}$$
$$A.\bar{A} = 0 \quad \text{Teorema da complementação}$$
$$A.B = B.A \quad \text{Teorema da comutação}$$
$$(A.B).C = A.(B.C) = A.B.C \quad \text{Teorema da associação}$$

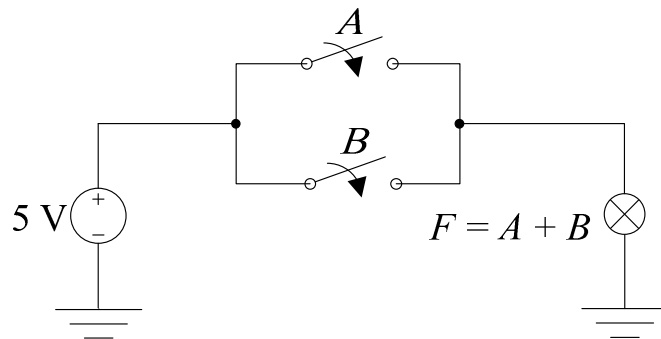
Diagrama de Venn





❑ Operação lógica OR (soma lógica)

Definição: Operação sobre n variáveis, que só toma o valor 0 quando todas essas variáveis tiverem o valor 0.



A lâmpada acende se
 A ou B forem atuados

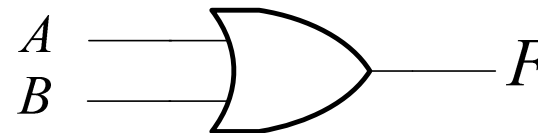
Expressão
algébrica

$$F = A + B$$

Tabela de
verdade

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Símbolo lógico





□ Operação lógica OR (soma lógica)

Propriedades:

$$\left. \begin{array}{l} A + 0 = A \\ A + 1 = 1 \end{array} \right\} \text{Teoremas da união}$$

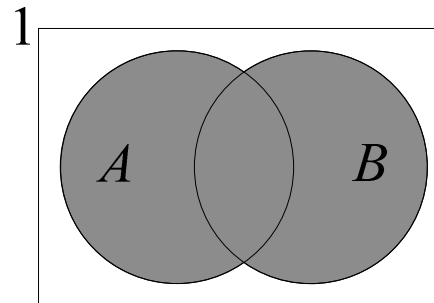
$$A + A = A \quad \text{Teorema da idem potência}$$

$$A + \bar{A} = 1 \quad \text{Teorema da complementação}$$

$$A + B = B + A \quad \text{Teorema da comutação}$$

$$(A + B) + C = A + (B + C) = A + B + C \quad \text{Teorema da associação}$$

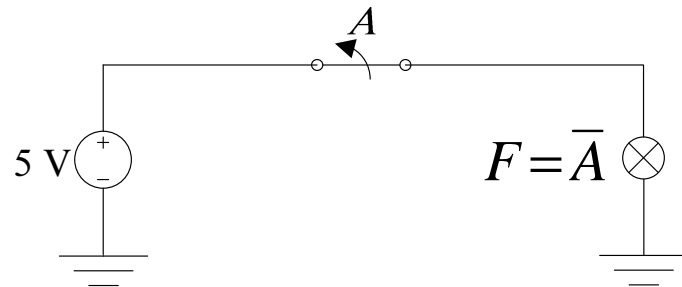
Diagrama de Venn





❑ Operação lógica NOT (negação lógica)

Definição: Operação sobre *uma* variável (ou expressão booleana), de que resulta a inversão do seu valor lógico.



Expressão
algébrica

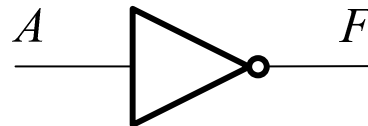
$$F = \bar{A}$$

Tabela de
verdade

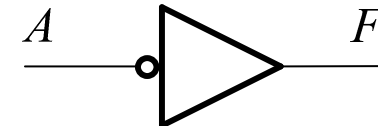
A	F
0	1
1	0

A lâmpada **apaga** se A for atuado

Símbolo lógico



ou



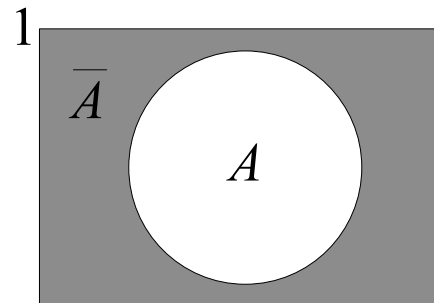


□ Operação lógica NOT (negação lógica)

Propriedades:

$$\overline{\overline{A}} = A \quad \text{Teorema da involução}$$
$$\overline{\overline{\overline{A}}} = \overline{A}$$

Diagrama de Venn





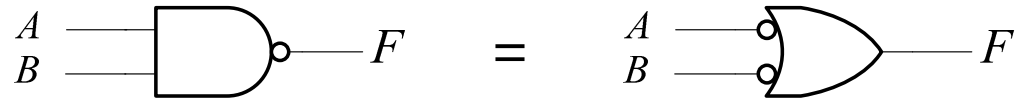
□ Teoremas de De Morgan

Propriedades:

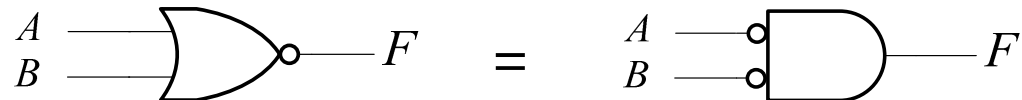
$$\overline{A.B.C} = \overline{A} + \overline{B} + \overline{C}$$
$$\overline{A + B + C} = \overline{A}.\overline{B}.\overline{C}$$

Exemplos de aplicação dos teoremas de De Morgan:

$$\overline{A.B} = \overline{A} + \overline{B}$$



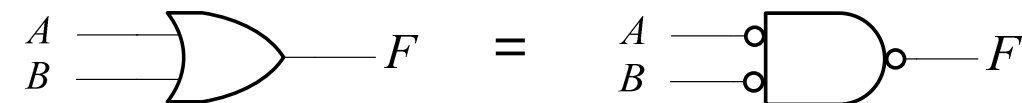
$$\overline{A + B} = \overline{A}.\overline{B}$$



$$A.B = \overline{\overline{A}.\overline{B}} = \overline{\overline{A} + \overline{B}}$$



$$A + B = \overline{\overline{A + B}} = \overline{\overline{A}.\overline{B}}$$





□ Forma AND-OR

Considere-se, por exemplo, a função lógica dada pela sua tabela de verdade.

Extraindo a função pelos 1's, sem simplificação, fica:

<i>n</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>F</i>
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

$$F(C,B,A) = \bar{C}.\bar{B}.A + \bar{C}.B.\bar{A} + \bar{C}.B.A + C.\bar{B}.\bar{A} + C.\bar{B}.A + C.B.A$$

Forma canónica AND-OR (união de interseções – termos produto):

- Em cada termo constam todas as variáveis da função, complementadas ou não complementadas;
- Os termos produto em que intervêm todas as variáveis da função denominam-se “termos mínimos”.

Representação alternativa (apenas válida com a forma canónica):

$$F(C,B,A) = \sum (1,2,3,4,5,7)$$



□ Forma OR-AND

Considere-se, por exemplo, a mesma função lógica dada anteriormente.

Extraindo a função pelos 0's, sem simplificação, fica:

<i>n</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>F</i>
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

$$F(C, B, A) = \overline{\overline{C} \cdot \overline{B} \cdot \overline{A}} + C \cdot B \cdot \overline{A} = (C + B + A) \cdot (\overline{C} + \overline{B} + A)$$

Forma canónica OR-AND (intersecção de uniões – termos soma):

- Em cada termo constam todas as variáveis da função, complementadas ou não complementadas;
- Os termos produto em que intervêm todas as variáveis da função denominam-se “termos máximos”.

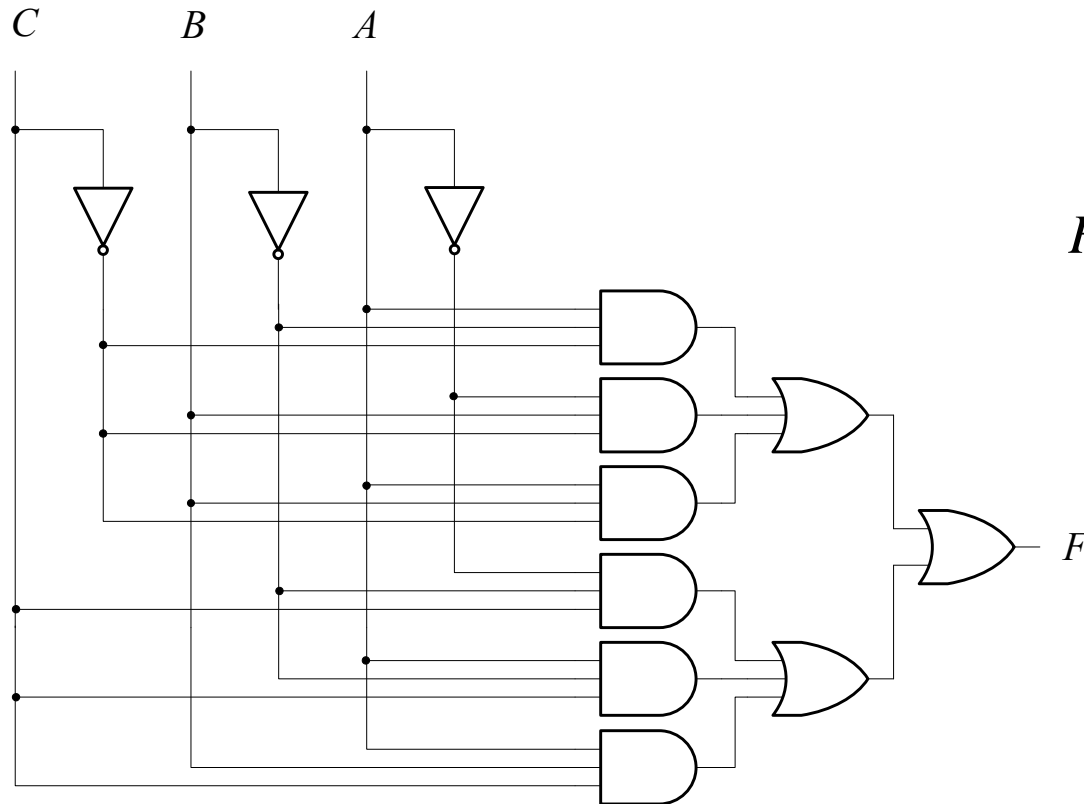
Representação alternativa (apenas válida com a forma canónica):

$$F(C, B, A) = \prod (0, 6)$$



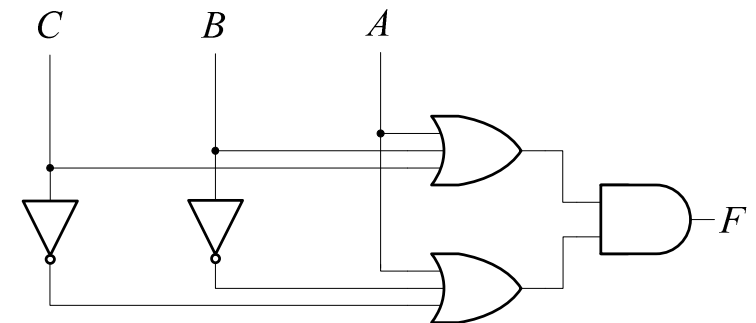
□ Implementação do circuito lógico sob as formas AND-OR e OR-AND

$$F(C, B, A) = \overline{C}.\overline{B}.A + \overline{C}.B.\overline{A} + \overline{C}.B.A + C.\overline{B}.\overline{A} + C.\overline{B}.A + C.B.A$$



SOP – *Sum Of Products*

$$F(C, B, A) = (C + B + A).(\overline{C} + \overline{B} + A)$$



POS – *Product Of Sums*



□ Propriedades das formas AND-OR

Em simplificação algébrica, os critérios de prioridade das várias operações são:

- O símbolo AND tem prioridade sobre o OR e o XOR;
- As operações entre parêntesis têm prioridade face às que estão fora deles.

$$A + B.C = (A + B).(A + C) \quad - \text{Teorema da distribuição}$$

$$A + A.B = A. (1 + B) \quad - \text{Teorema da absorção}$$

$$A.B + A.C = A.(B + C)$$

$$A + \bar{A}.B = A.(1 + B) + \bar{A}.B = A + A.B + \bar{A}B = A + B.(A + \bar{A}) = A + B$$



□ Propriedades das formas OR-AND

Em simplificação algébrica, os critérios de prioridade das várias operações são:

- Idem, como na forma AND-OR.

$$A.(B + C) = A.B + A.C \quad - \text{Teorema da distribuição}$$

$$A.(A + B) = A + A.B = A \quad - \text{Teorema da absorção}$$

$$A.(\bar{A} + B) = A.B$$

$$(A + B).(A + C) = A + A.C + A.B + B.C = A + B.C$$

$$(A + B).(C + D) = A.C + A.D + B.C + B.D$$



□ Mapas de Karnaugh

Dada a morosidade da simplificação algébrica, torna-se necessário utilizar outro método de simplificação de funções lógicas.

n	C	B	A	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Funções não simplificadas, na sua forma canónica:

AND-OR

$$F(C, B, A) = \overline{C}.\overline{B}.A + \overline{C}.B.\overline{A} + \overline{C}.B.A + C.\overline{B}.\overline{A} + C.\overline{B}.A + C.B.A$$

OR-AND

$$F(C, B, A) = (C + B + A).(\overline{C} + \overline{B} + A)$$



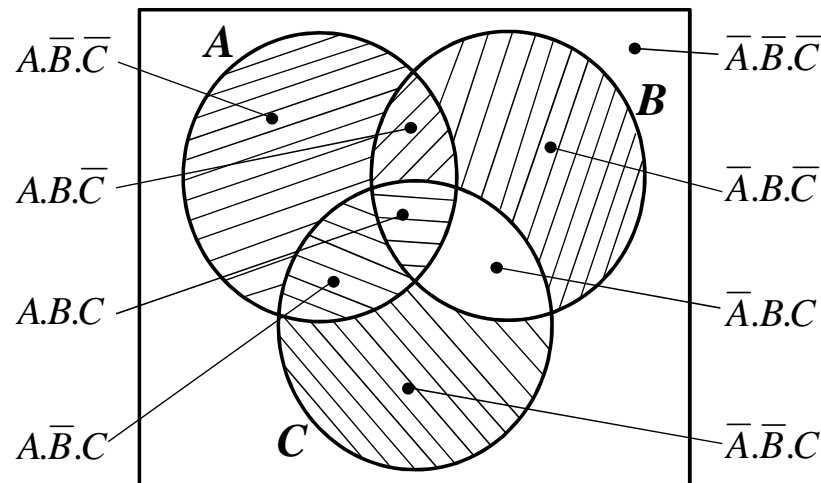
□ Mapas de Karnaugh

$$F(A, B, C) = A.\bar{B}.\bar{C} + \bar{A}.B.\bar{C} + A.B.\bar{C} + \bar{A}.\bar{B}.C + A.\bar{B}.C + A.B.C$$

$$F(A, B, C) = (A + B + C).(A + \bar{B} + \bar{C})$$

<i>n</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>F</i>
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Diagrama de Venn



Linearização do
diagrama de Venn

Mapa da Karnaugh

<i>A</i>			
$\bar{A}.\bar{B}.\bar{C}$ 0	$A.\bar{B}.\bar{C}$ 1	$A.B.\bar{C}$ 3	$\bar{A}.B.\bar{C}$ 2
$\bar{A}.\bar{B}.C$ 4	$A.\bar{B}.C$ 5	$A.B.C$ 7	$\bar{A}.B.C$ 6
<i>B</i>			



❑ Características dos mapas de Karnaugh

- Tanto o diagrama de Venn como o mapa de Karnaugh são formas alternativas de representar a tabela de verdade, logo têm 2^L possibilidades (áreas) distintas.
- De cada uma das áreas do mapa de Karnaugh, para uma qualquer adjacente (ou simétrica), só muda o valor de uma única variável.
- Dentro da quadrícula do mapa coloca-se o valor lógico de saída da função para a configuração de variáveis de entrada respeitante a essa quadrícula.

		A				
		<hr/>				
		$\overline{A}.\overline{B}.\overline{C}$	$A.\overline{B}.\overline{C}$	$A.B.\overline{C}$	$\overline{A}.B.\overline{C}$	
		0	1	3	2	
C		$\overline{A}.\overline{B}.C$	$A.\overline{B}.C$	$A.B.C$	$\overline{A}.B.C$	
		4	5	7	6	
		<hr/>				
		B				



❑ Procedimento de simplificação utilizando mapas de Karnaugh

- A simplificação consiste em agrupar os “1”s, adjacentes ou simétricos, em grupos cuja quantidade seja potência inteira de dois. Por cada agrupamento de 2^x “1”s, reduzem-se x variáveis no respetivo termo produto.
- Deve tentar agrupar-se o máximo número de “1”s.
- “1”s já presentes num agrupamento podem também constar de outro, se daí se obtiver uma maximização do número de “1”s em cada grupo.

		A			
		<hr/>			
		$\bar{A}.\bar{B}.\bar{C}$	$A.\bar{B}.\bar{C}$	$A.B.\bar{C}$	$\bar{A}.B.\bar{C}$
		0	1	3	2
C	$\bar{A}.\bar{B}.C$	$A.\bar{B}.C$	$A.B.C$	$\bar{A}.B.C$	
	4	5	7	6	
		<hr/>			
		B			



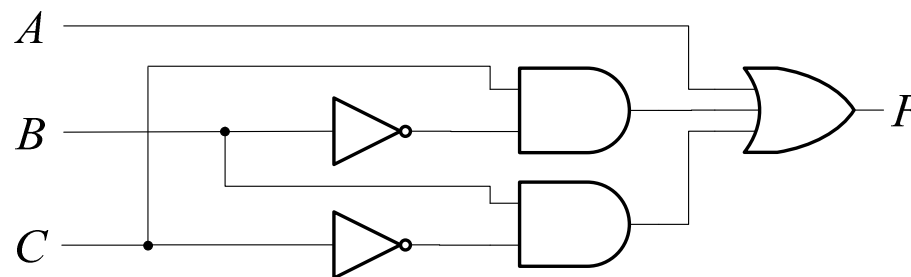
❑ Procedimento de simplificação utilizando mapas de Karnaugh

$$F(A, B, C) = A.\bar{B}.\bar{C} + \bar{A}.B.\bar{C} + A.B.\bar{C} + \bar{A}.\bar{B}.C + A.\bar{B}.C + A.B.C$$

$$F(A, B, C) = (A + B + C).(A + \bar{B} + \bar{C})$$

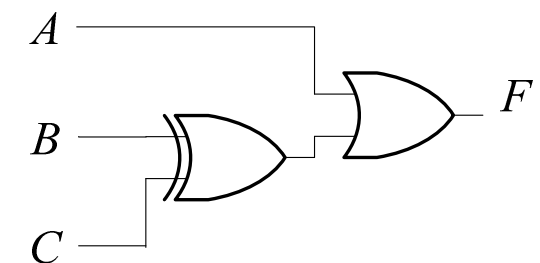
n	C	B	A	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

	A			
C	$\bar{A}.\bar{B}.\bar{C}$ 0	$A.\bar{B}.\bar{C}$ 1	$A.B.\bar{C}$ 3	$\bar{A}.B.\bar{C}$ 2
	$\bar{A}.\bar{B}.C$ 4	$A.\bar{B}.C$ 5	$A.B.C$ 7	$\bar{A}.B.C$ 6
	B			



Simplificação do exemplo dado:

	A			
C	0	1	1	1
	1	1	1	0
	B			





□ Exemplo de simplificação

$$F = \overline{A}.\overline{C} + \overline{A}.D + \overline{A}.B + A.\overline{D} + \overline{B}.D$$

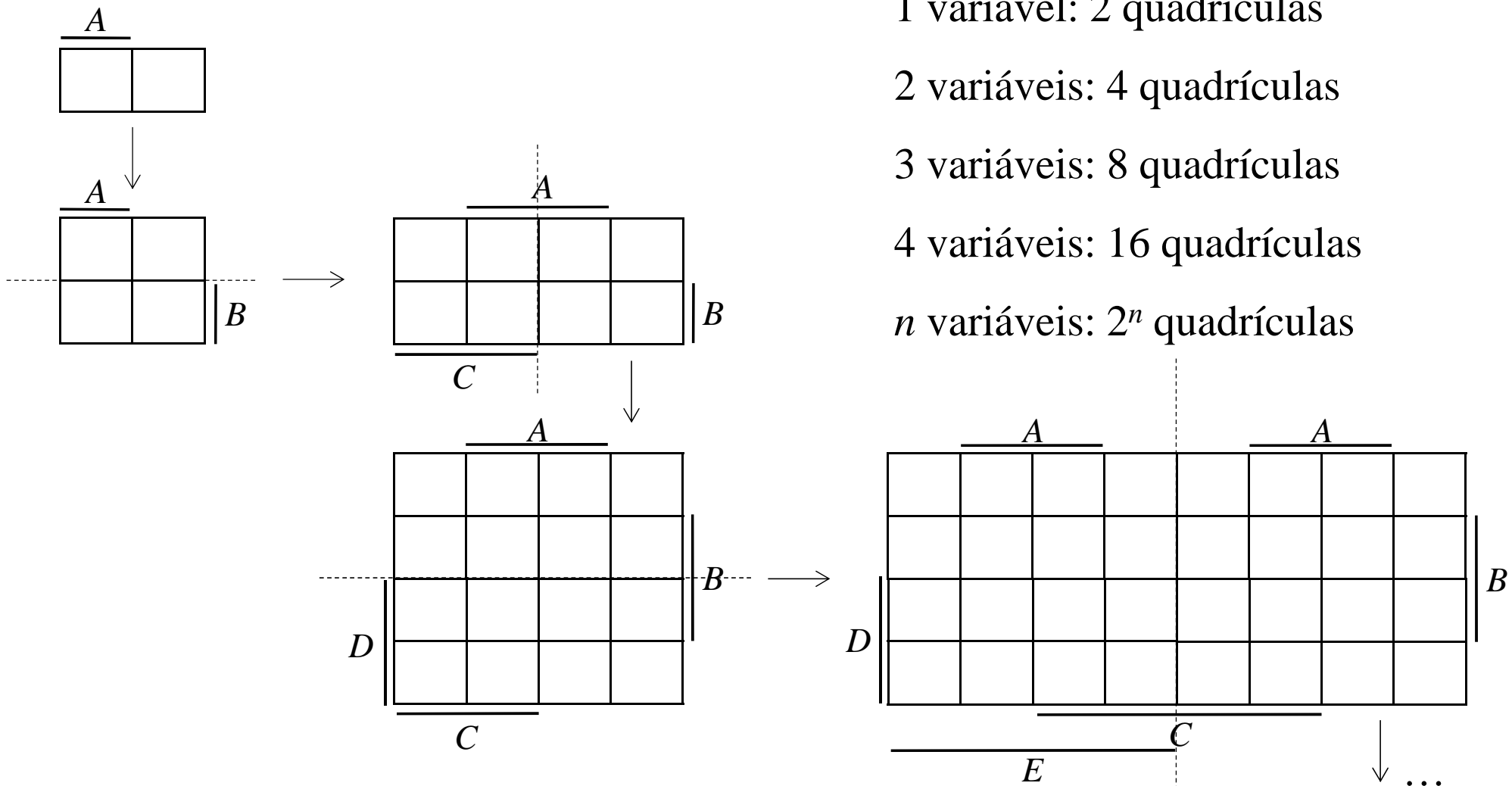
Quatro variáveis: 16 quadrículas

F		A			
C		1	1	1	1
		1	1	1	1
		1	0	0	1
		1	0	0	1
		B			
		D			

A disposição das variáveis no mapa é indiferente. Contudo, tem de ser sempre respeitada a regra da variação de apenas uma variável entre quadrículas adjacentes e entre quadrículas simétricas.



□ Obtenção dos mapas

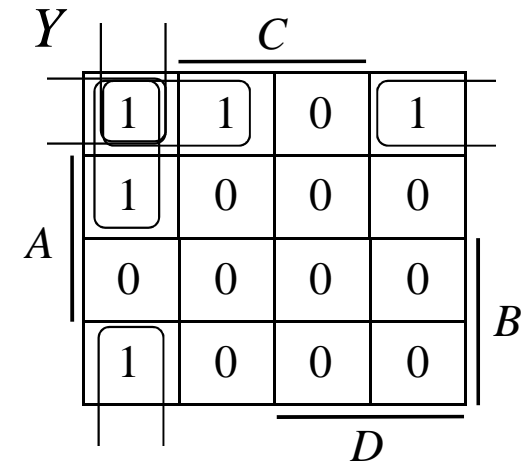
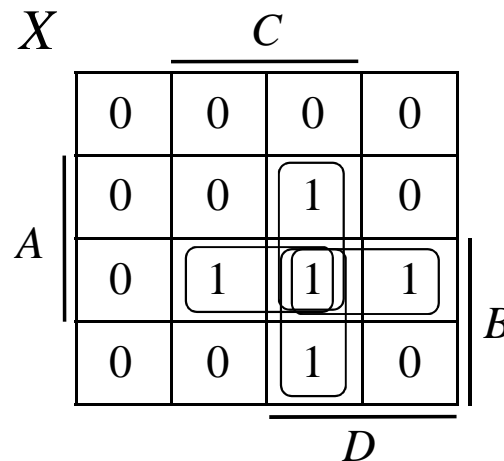




□ Exemplos

A	B	C	D	X	Y
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0

O conselho diretivo de uma escola é formado por 4 membros que têm de votar sobre assuntos de gestão da mesma. Pretende-se um circuito combinatório mínimo que acenda um led verde (X) quando a maioria dos votantes votar afirmativamente e um led vermelho (Y) quando a maioria votar negativamente uma determinada decisão.



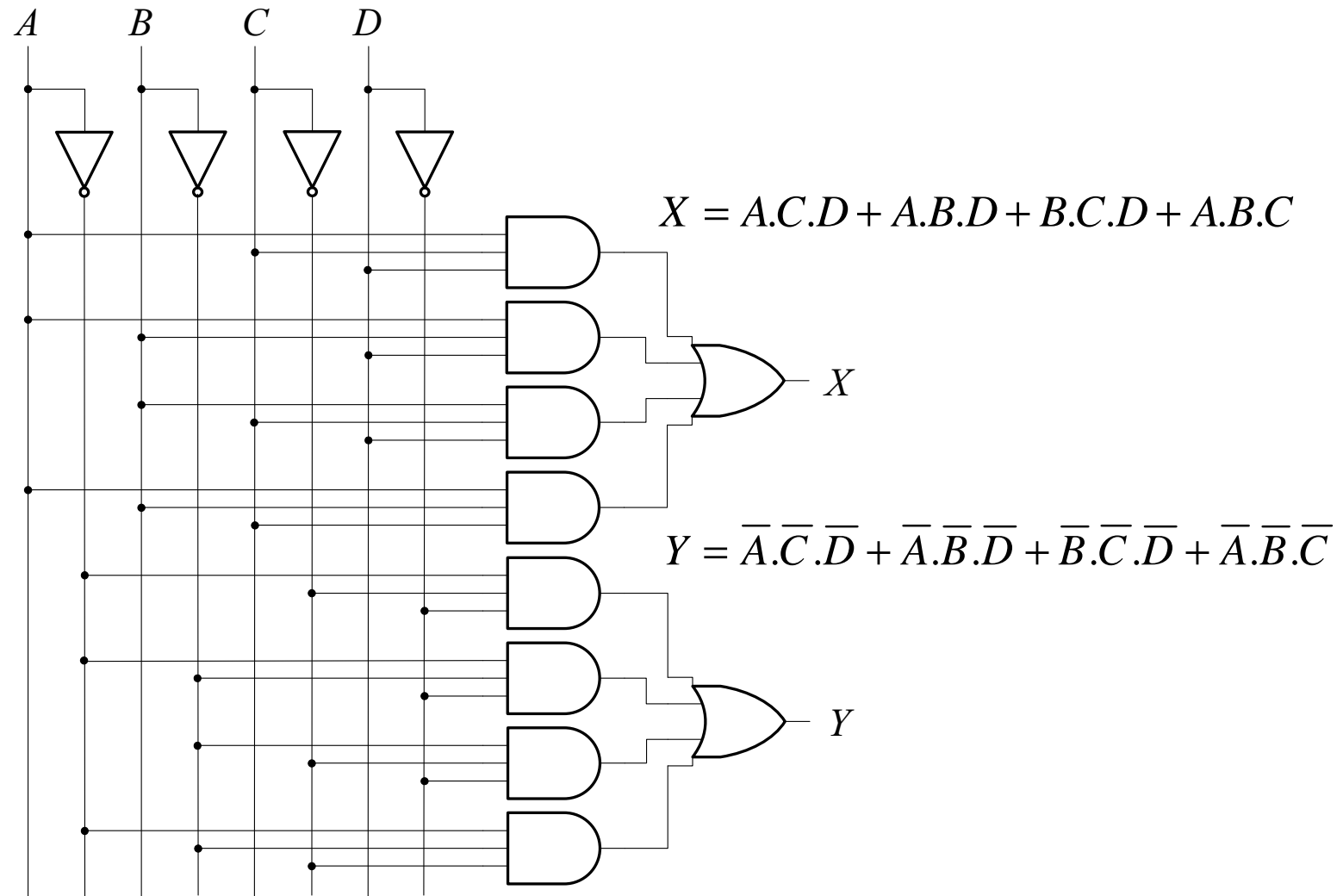
$$X = A.C.D + A.B.D + B.C.D + A.B.C$$

$$Y = \bar{A}.\bar{C}.\bar{D} + \bar{A}.\bar{B}.\bar{D} + \bar{B}.\bar{C}.\bar{D} + \bar{A}.\bar{B}.\bar{C}$$



□ Exemplos

A	B	C	D	X	Y
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0





□ Exercícios

Construir a tabela de verdade e o mapa de Karnaugh de cada uma das seguintes funções lógicas e obter a expressão simplificada para cada função dada:

$$F_1 = A.B + \overline{A}.B + \overline{A}.\overline{B}$$

$$F_2 = \overline{\overline{A} + \overline{B} + \overline{C}}$$

$$F_3 = \overline{B} + A.B.\overline{C}$$

$$F_4 = \overline{A.B} + A.B.\overline{C}$$

$$F_5 = A.\overline{C} + B.\overline{C} + A.C + \overline{B}.C$$

$$F_6 = x.y.z + x.y.\overline{z} + x.\overline{y}.z + x.\overline{y}.\overline{z}$$

$$F_7 = A.\overline{D} + C.B.A.\overline{D} + \overline{C}.\overline{B} + \overline{C}$$

$$F_8 = \overline{x.w.y.z} + \overline{\overline{z} + \overline{x}}$$



❑ Funções incompletamente especificadas

Existem funções em que é indiferente o valor lógico de saída, em função de certas combinações das variáveis de entrada.

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	×
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	×

	A			
	1	1	1	1
B	0	0	×	×
	C			

× - indiferente, “*don't care*”. Pode tomar qualquer valor (1 ou 0), dependendo de como resultar uma maior simplificação.

Considerando $\times = 1$:

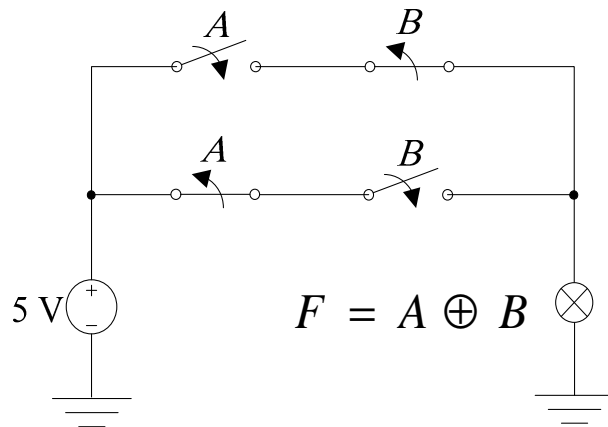
Considerando $\times = 0$:

Neste caso, tornava-se mais vantajoso tomar “×” como igual a 0.



❑ Operação lógica XOR (união exclusiva – soma aritmética módulo 2)

Definição: Operação sobre n variáveis, que só toma o valor 1 quando quando um número ímpar dessas variáveis tomar o valor 1.



A lâmpada acende se ou
A ou B forem atuados

Expressão
algébrica

$$F = A \oplus B \\ \equiv \bar{A}.B + A.\bar{B}$$

Símbolo lógico

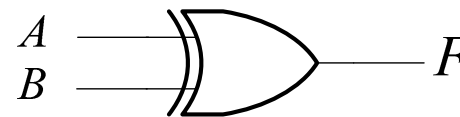


Tabela de
verdade

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0



□ Operação lógica XOR (união exclusiva – soma aritmética módulo 2)

Propriedades:

$$A \oplus 0 = A$$

$$A \oplus 1 = \bar{A}$$

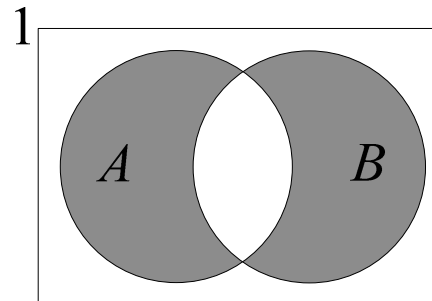
$$A \oplus A = 0$$

$$A \oplus \bar{A} = 1$$

$$A \oplus B = B \oplus A \quad \text{Teorema da comutação}$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C \quad \text{Teorema da associação}$$

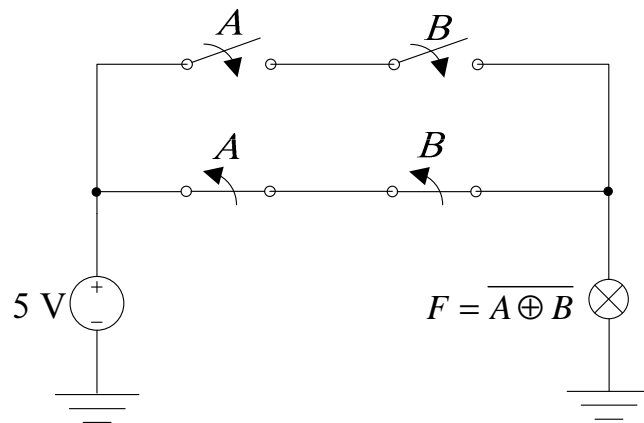
Diagrama de Venn





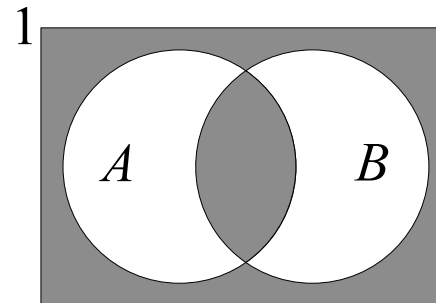
❑ Operação lógica XNOR (equivalência)

Definição: Operação sobre n variáveis, que só toma o valor 1 quando quando um número par dessas variáveis tomar o valor 1.

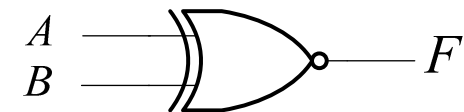


A lâmpada acende se A e B forem atuados ou não atuados **em simultâneo**

$$F = \overline{A \oplus B}$$
$$\equiv \overline{A}.\overline{B} + A.B$$



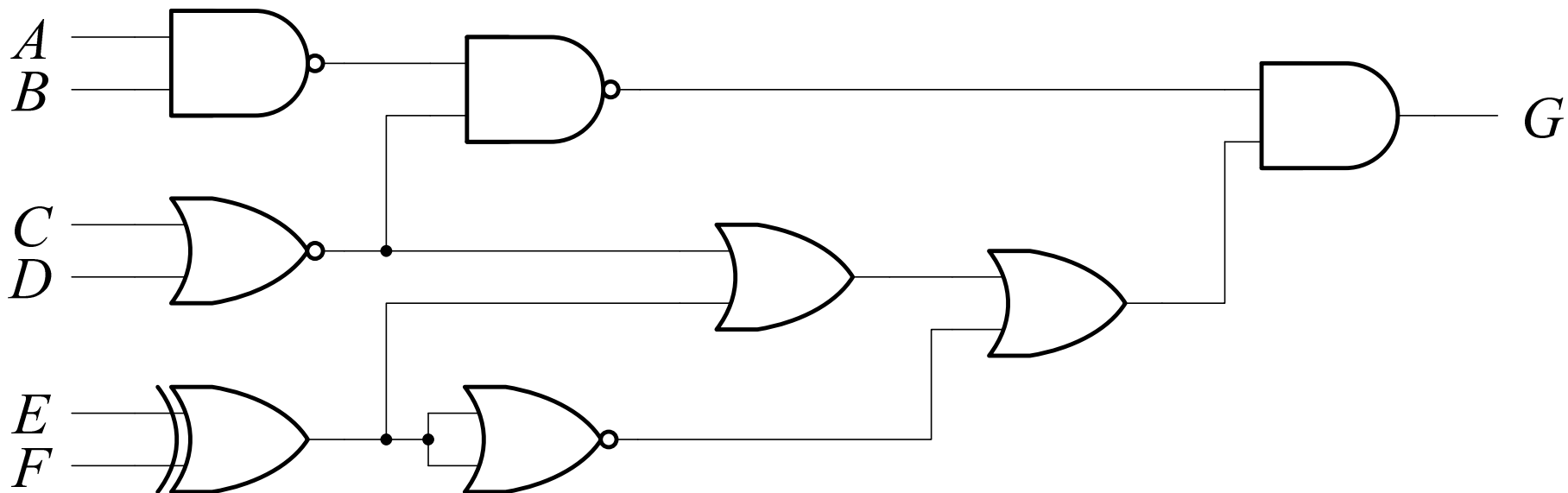
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1





□ Exercício

Obter a expressão simplificada da função G (sugestão: comece do lado das entradas e vá obtendo a expressão lógica à saída de cada porta, simplificando sucessivamente, até chegar à saída).





❑ Variáveis lógicas no Arduino

No Arduino, o tipo de dados `boolean` é o utilizado para conter valores e variáveis lógicas.

- Em essência, as variáveis do tipo `boolean` deveriam ocupar apenas um bit, valendo 0 ou 1. Na realidade, ocupam um byte.
- O valor *true* é qualquer valor diferente de zero. Os símbolos mais utilizados são `true`, `HIGH`, ou simplesmente, 1.
- O valor *false* é o valor zero. Os símbolos mais utilizados são `false`, `LOW`, ou simplesmente, 0.



❑ Operadores lógicos no Arduino

A linguagem utilizada na programação do Arduino dispõe de vários operadores lógicos que podem usar-se para construir expressões lógicas. Existem:

- Operadores *booleanos*: usados quando se pretende compôr expressões das quais resulte um valor `true` ou `false`. Dispõem-se do `&&` (AND), `||` (OR) e `!` (NOT).
- Operadores *bitwise* (bit a bit): usados quando se pretende realizar operações lógicas ao nível do bit. Dispõem-se do `&` (AND), `|` (OR), `~` (NOT) e `^` (XOR).



❑ Operadores lógicos no Arduino

- Pode utilizar-se o Arduino para simular as operações lógicas que os circuitos integrados realizariam em *hardware*.
- As variáveis lógicas externas são lidas para uma variável boolean (lógica) através da função `digitalRead(Dx)`, que lê o valor do pino digital (físico).
- Em termos de variedade e de simplicidade de escrita, podem utilizar-se os operadores *bitwise* para realizar as operações lógicas no Arduino, à exceção da operação NOT, em deve utilizar-se o operador que não é *bitwise*.

(AND) $A \ \& \ B$

(OR) $A \ | \ B$

(XOR) $A \ \wedge \ B$

(NOT) $!A$



❑ Operadores lógicos no Arduino - exemplos

Considere-se variáveis byte $A = B10011001$ e $B = B00111010$

- Com operadores *boolean*,
o resultado fica:

(AND) - $A \ \&\& \ B = 1$ (true)

(OR) - $A \ || \ B = 1$ (true)

(NOT) - $!A = 0$ (false)

- Com operadores *bitwise*,
o resultado fica:

(AND) - $A \ \& \ B = 00011000$ (true)

(OR) - $A \ | \ B = 10111011$ (true)

(XOR) - $A \ ^ \ B = 10100011$ (true)

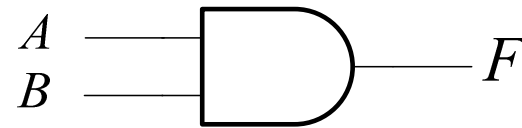
(NOT) - $\sim A = 01100110$ (true)



❑ Operação lógica AND no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = A.B$$



Pode usar-se o operador `&` diretamente, ou definir-se uma função AND de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

```
boolean AND (boolean A, boolean B){  
    return A & B;  
}
```



❑ Operação lógica AND no Arduino - exemplos e variantes

Considerando, por exemplo, $A = B10011001$ e $B = B00111010$, para a função anterior:

```
boolean AND (boolean A, boolean B){  
    return A & B;  
}
```

Evocando-se $F = \text{AND}(A, B)$ resultará em $F = 1$ (true).

Alterando os tipos de dados e redefinindo a função para:

```
byte AND (byte A, byte B){  
    return A & B;  
}
```

Evocando-se $F = \text{AND}(A, B)$ resultará em $F = 00011000$.



□ Operação lógica AND no Arduino - exemplos e variantes

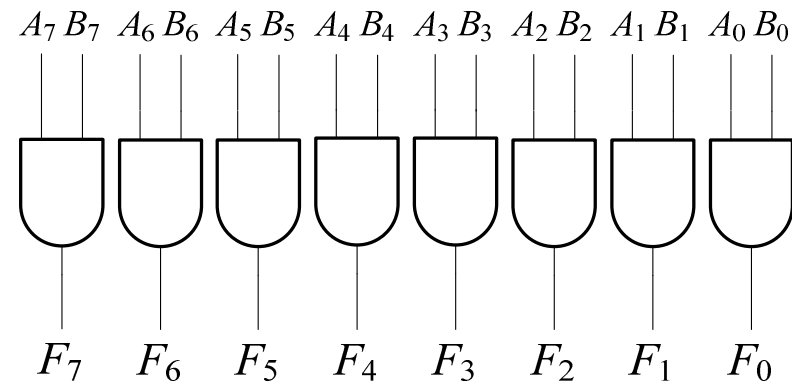
Considerando-se a `byte AND (byte A, byte B) {`
definição anterior, `return A & B;`
`}`

A e B correspondem a $A = A_7A_6A_5A_4A_3A_2A_1A_0$ e $B = B_7B_6B_5B_4B_3B_2B_1B_0$.

Tendo-se $F = \text{AND}(A, B)$, F corresponderá a $F = F_7F_6F_5F_4F_3F_2F_1F_0$.

Se $A = B10011001$ e $B = B00111010$, $F = B00011000$.

A função operará
como se do seguinte
circuito se tratasse:

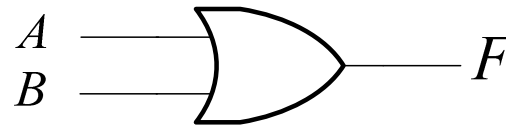




❑ Operação lógica OR no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = A + B$$



Pode usar-se o operador `|` diretamente, ou definir-se uma função OR de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

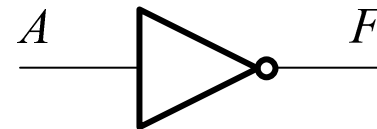
```
boolean OR (boolean A, boolean B){  
    return A | B;  
}
```




□ Operação lógica NOT no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = \overline{A}$$



Pode usar-se o operador `!` diretamente, ou definir-se uma função NOT de uma variável da seguinte forma, em que o valor de retorno e a variável de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

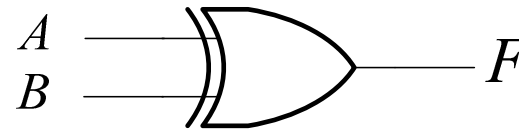
```
boolean NOT (boolean A) {  
    return !A;  
}
```



❑ Operação lógica XOR no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = A \oplus B$$



Pode usar-se o operador \wedge diretamente, ou definir-se uma função XOR de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

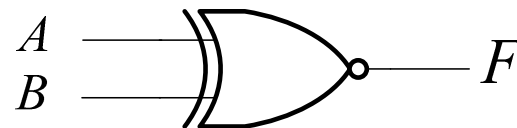
```
boolean XOR (boolean A, boolean B){  
    return A ^ B;  
}
```



□ Operação lógica XNOR no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = \overline{A \oplus B}$$



Podem usar-se os operadores `!` e `^` diretamente, ou definir-se uma função XNOR de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

```
boolean XNOR (boolean A, boolean B){  
    return !(A ^ B);  
}
```



❑ Exemplo de implementação das funções de maioria no Arduino

$$X = F_1(A, B, C, D) = A.C.D + A.B.D + B.C.D + A.B.C$$

$$Y = F_2(A, B, C, D) = \overline{A}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{D} + \overline{B}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}$$

```
// Atribuição de pinos de entrada (A, B, C, D) e de saída (X, Y)
#define pinA 2
#define pinB 3
#define pinC 4
#define pinD 5
#define pinX 6
#define pinY 7

// Variáveis lógicas de entrada e de saída (globais)
boolean A, B, C, D, X, Y;

// Função lógica de maioria de "1"s
boolean F1(boolean A, boolean B, boolean C, boolean D){
    return A & C & D | A & B & D | B & C & D | A & B & C;
}

// Função lógica de maioria de "0"s
boolean F2(boolean A, boolean B, boolean C, boolean D){
    return !A & !C & !D | !A & !B & !D | !B & !C & !D | !A & !B & !C;
}
```



□ Exemplo de implementação das funções de maioria no Arduino (cont.)

$$X = F_1(A, B, C, D) = A.C.D + A.B.D + B.C.D + A.B.C$$

$$Y = F_2(A, B, C, D) = \overline{A}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{D} + \overline{B}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}$$

// Leitura de pinos físicos digitais para as variáveis de entrada

```
void lerEntradas(){  
    A = digitalRead(pinA);  
    B = digitalRead(pinB);  
    C = digitalRead(pinC);  
    D = digitalRead(pinD);  
}
```

// Calcular o valor das funções de maioria

```
void calcularFuncoesCombinatorias(){  
    X = F1(A, B, C, D);  
    Y = F2(A, B, C, D);  
}
```

// Escrita das variáveis de saída em pinos físicos digitais

```
void escreverSaidas(){  
    digitalWrite(pinX, X);  
    digitalWrite(pinY, Y);  
}
```



□ Exemplo de implementação das funções de maioria no Arduino (cont.)

$$X = F_1(A, B, C, D) = A.C.D + A.B.D + B.C.D + A.B.C$$

$$Y = F_2(A, B, C, D) = \overline{A}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{D} + \overline{B}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}$$

```
// Configuração dos pinos de saída e de entrada
void setup(){
  pinMode(pinA, INPUT);
  pinMode(pinB, INPUT);
  pinMode(pinC, INPUT);
  pinMode(pinD, INPUT);
  pinMode(pinX, OUTPUT);
  pinMode(pinY, OUTPUT);
}

// Execução cíclica das funções lógicas
void loop(){
  lerEntradas();
  calcularFuncoesCombinatorias();
  escreverSaidas();
}
```



❑ Exemplo de implementação das funções de maioria no Arduino (cont.)

$$X = F_1(A, B, C, D) = A.C.D + A.B.D + B.C.D + A.B.C$$

$$Y = F_2(A, B, C, D) = \overline{A}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{D} + \overline{B}.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}$$

```
// Função lógica de maioria de "1"s
boolean F1(boolean A, boolean B, boolean C, boolean D){
    return A & C & D | A & B & D | B & C & D | A & B & C;
}

// Função lógica de maioria de "0"s
boolean F2(boolean A, boolean B, boolean C, boolean D){
    return !A & !C & !D | !A & !B & !D | !B & !C & !D | !A & !B & !C;
}

// Calcular o valor das funções de maioria (Versão mais compacta)
void calcularFuncoesCombinatorias(){
    digitalWrite(pinX, F1(digitalRead(pinA), digitalRead(pinB), digitalRead(pinC), digitalRead(pinD)));
    digitalWrite(pinY, F2(digitalRead(pinA), digitalRead(pinB), digitalRead(pinC), digitalRead(pinD)));
}

// Configuração dos pinos de saída (os outros, por default, são entrada)
void setup(){
    pinMode(pinX, OUTPUT);
    pinMode(pinY, OUTPUT);
}

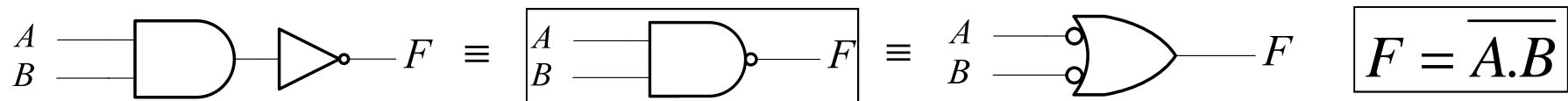
// Execução cíclica das funções lógicas
void loop(){
    calcularFuncoesCombinatorias();
}
```

Esta versão não precisa de declaração de variáveis, dado que se têm em atenção apenas os pinos digitais de entrada.

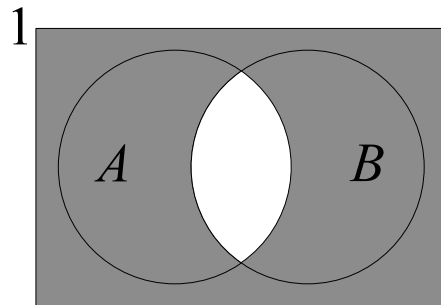


❑ Operação lógica NAND (aglutinação das funções NOT e AND)

Definição: Operação sobre n variáveis, que só toma o valor 0 quando todas essas variáveis tiverem o valor 1.



Propriedades:



$$A \uparrow 1 = \overline{A.1} = \overline{A}$$

$$A \uparrow 0 = \overline{A.0} = 1$$

$$A \uparrow A = \overline{A.A} = \overline{A}$$

$$A \uparrow \overline{A} = \overline{A.\overline{A}} = 1$$

$$A \uparrow B = B \uparrow A$$

$$(A \uparrow B) \uparrow C = \overline{\overline{A.B}.C} = A.B + \overline{C} = \overline{\overline{A} + \overline{B} + \overline{C}}$$

$$A \uparrow B \uparrow C = \overline{A.B.C} = \overline{A} + \overline{B} + \overline{C} \neq (A \uparrow B) \uparrow C$$

• Comutativa

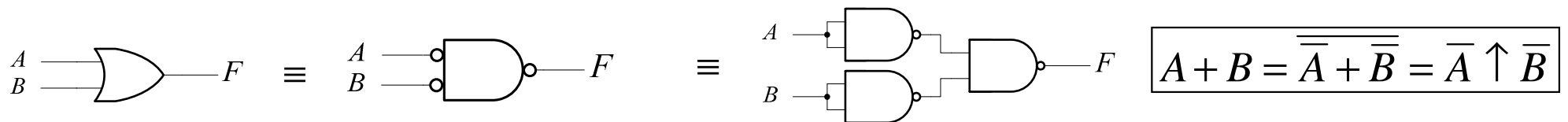
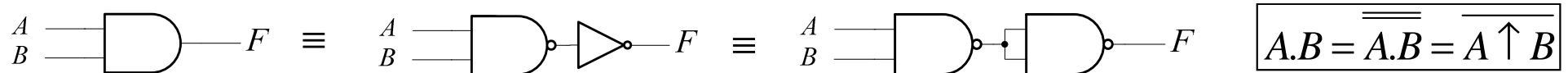
• Não associativa

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



❑ Operação lógica NAND (aglutinação das funções NOT e AND)

A função NAND diz-se funcionalmente completa porque, a partir dela, podem sintetizar-se as funções lógicas básicas NOT, AND e OR.



Assim, qualquer função lógica pode ser realizada usando exclusivamente portas lógicas do tipo NAND.

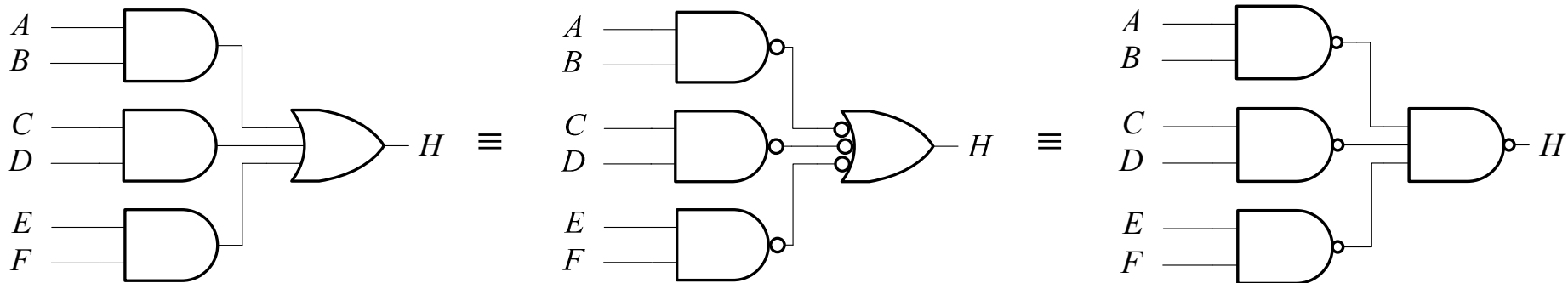


□ Operação lógica NAND (aglutinação das funções NOT e AND)

As expressões AND-OR têm imediata conversão para portas NAND.

Exemplo:

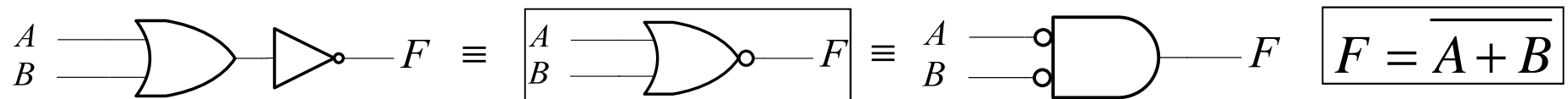
$$H = A.B + C.D + E.F = \overline{\overline{A.B}} + \overline{\overline{C.D}} + \overline{\overline{E.F}} = \overline{\overline{A.B.C.D.E.F}} = (A \uparrow B) \uparrow (C \uparrow D) \uparrow (E \uparrow F)$$



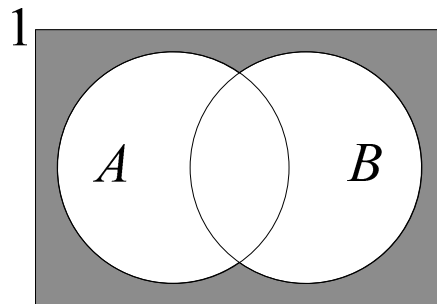


❑ Operação lógica NOR (aglutinação das funções NOT e OR)

Definição: Operação sobre n variáveis, que só toma o valor 1 quando todas essas variáveis tiverem o valor 0.



Propriedades:



$$A \downarrow 0 = \overline{A + 0} = \overline{A}$$

$$A \downarrow 1 = \overline{A + 1} = 0$$

$$A \downarrow A = \overline{A + A} = \overline{A}$$

$$A \downarrow \overline{A} = \overline{A + \overline{A}} = 0$$

$$A \downarrow B = B \downarrow A$$

$$(A \downarrow B) \downarrow C = \overline{\overline{A + B} + C} = \overline{(A + B) \cdot \overline{C}} = \overline{\overline{A} \cdot \overline{B} \cdot \overline{C}}$$

$$A \downarrow B \downarrow C = \overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C} \neq (A \downarrow B) \downarrow C$$

• Comutativa

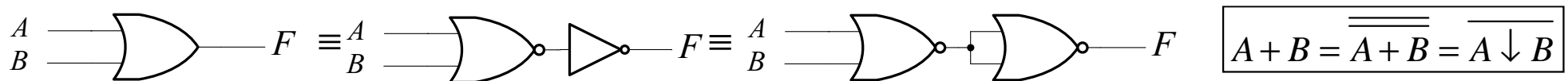
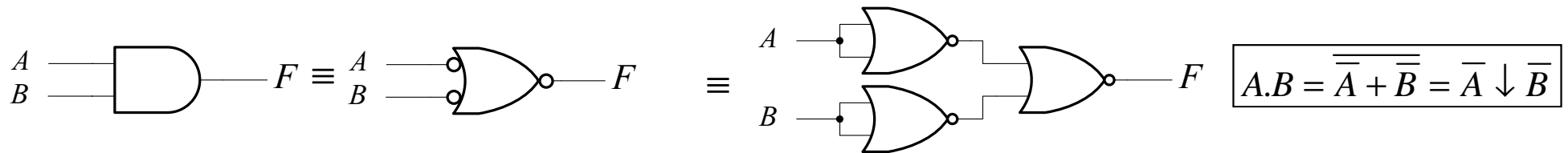
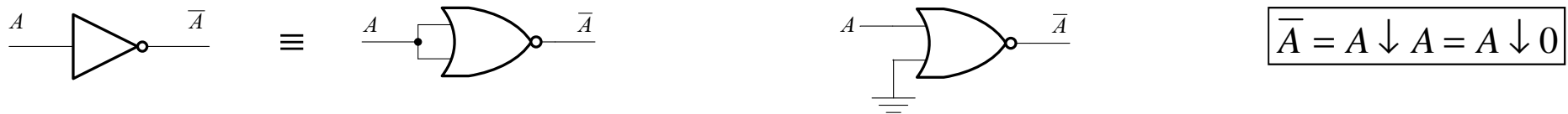
• Não associativa

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



❑ Operação lógica NOR (aglutinação das funções NOT e OR)

A função NOR diz-se funcionalmente completa porque, a partir dela, podem sintetizar-se as funções lógicas básicas NOT, AND e OR.



Assim, qualquer função lógica pode ser realizada usando exclusivamente portas lógicas do tipo NOR.

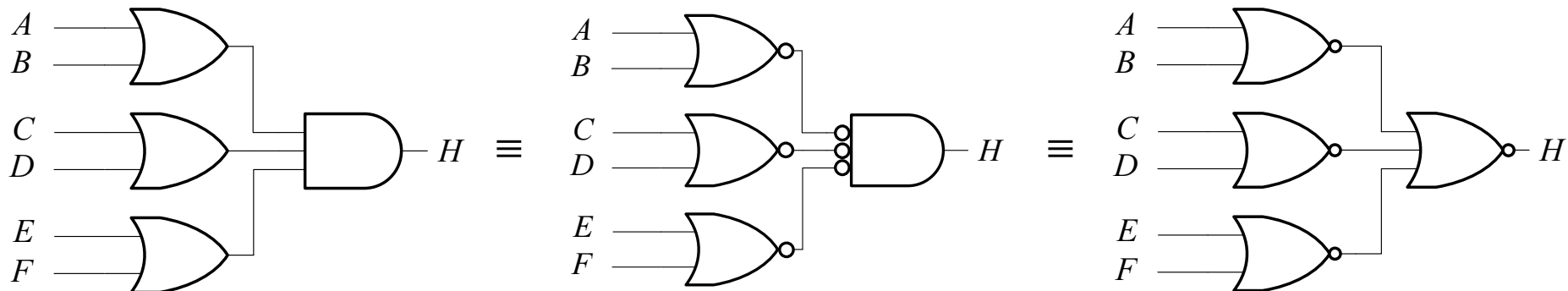


□ Operação lógica NOR (aglutinação das funções NOT e OR)

As expressões OR-AND têm imediata conversão para portas NOR.

Exemplo:

$$H = (A + B).(C + D).(E + F) = \overline{\overline{A + B}.\overline{C + D}.\overline{E + F}} =$$
$$= \overline{\overline{A + B} + \overline{C + D} + \overline{E + F}} = (A \downarrow B) \downarrow (C \downarrow D) \downarrow (E \downarrow F)$$

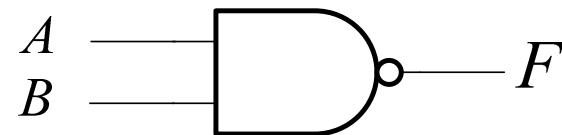




□ Operação lógica NAND no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = \overline{A.B}$$



Podem usar-se os operadores `!` e `&` diretamente, ou definir-se uma função NAND de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

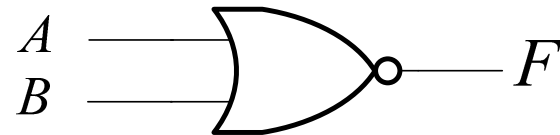
```
boolean NAND (boolean A, boolean B){  
    return !(A & B);  
}
```



❑ Operação lógica NOR no Arduino

Tendo em conta a definição feita anteriormente, ou seja:

$$F = \overline{A + B}$$

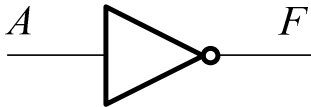
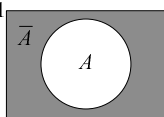
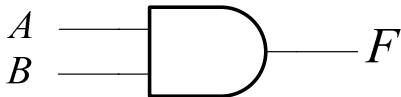
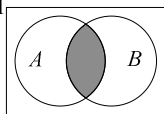
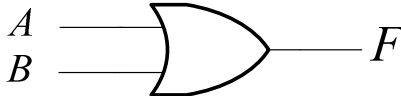
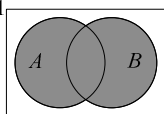
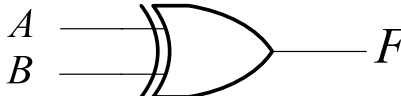
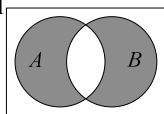
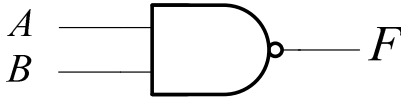
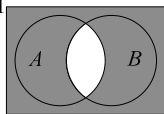
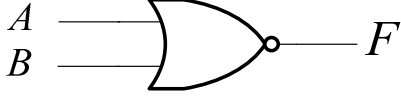
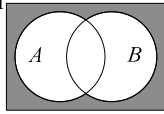
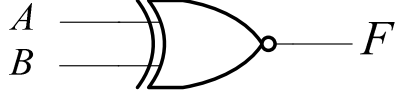
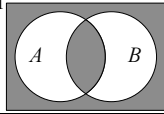


Podem usar-se os operadores `!` e `|` diretamente, ou definir-se uma função NOR de duas variáveis da seguinte forma, em que o valor de retorno e as variáveis de entrada, sendo booleanos, só poderá resultar em *true* ou *false*.

```
boolean NOR (boolean A, boolean B){  
    return !(A | B);  
}
```



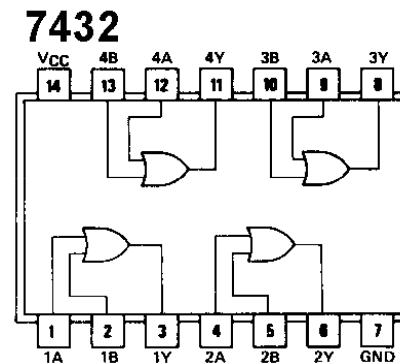
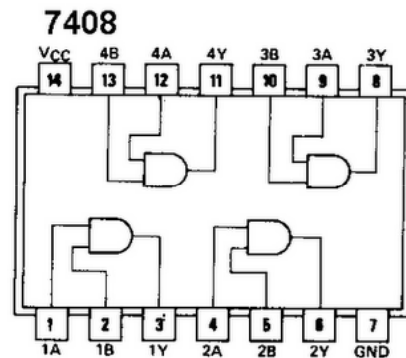
□ Resumo das várias operações (portas) lógicas

Nome da operação	Símbolo lógico	Expressão lógica	Diagrama de Venn
NOT		$F = \bar{A}$	
AND		$F = A.B$	
OR		$F = A + B$	
XOR		$F = A \oplus B$	
NAND		$F = \overline{A.B}$	
NOR		$F = \overline{A + B}$	
XNOR		$F = \overline{A \oplus B}$	

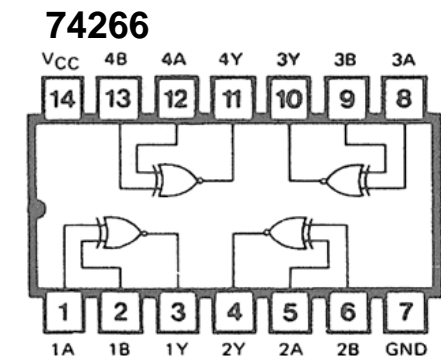
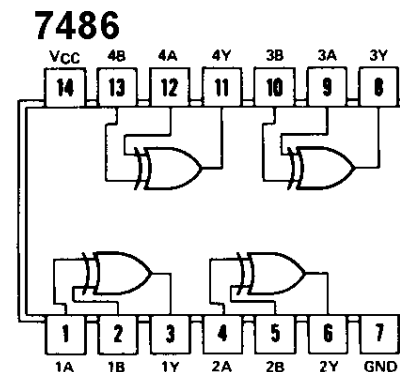
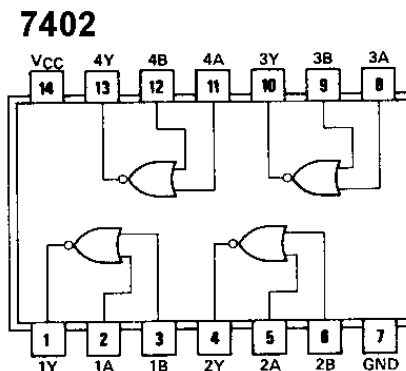
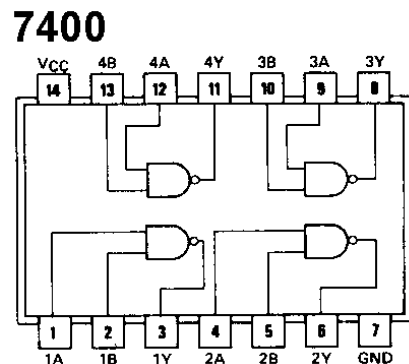
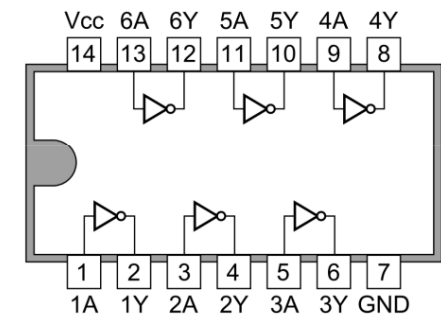


❑ Circuitos integrados

As portas lógicas já estudadas existem comercialmente em circuito integrado, em *chips* que variam na quantidade de portas e no número de entradas por porta.



7404 Hex Inverters

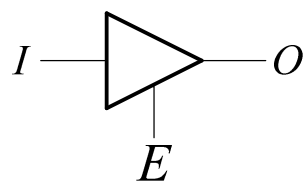




□ Portas *tri-state*

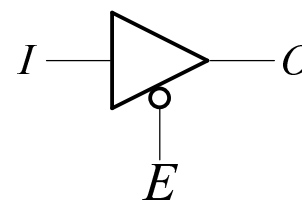
Tal como o nome indica, este tipo de portas possui três estados possíveis: “0”, “1” e “alta impedância” (resistência infinita – circuito aberto), significativo de que a porta se encontra desligada do resto do circuito que tenha a jusante da saída.

Existem algumas variantes possíveis, com o seu respetivo símbolo lógico:



Porta *tri-state* identidade
com controlo *active-high*

E	I	O
0	0	Z
0	1	Z
1	0	0
1	1	1

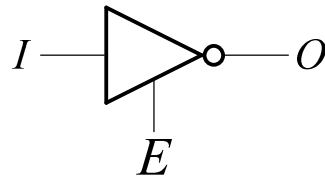


Porta *tri-state* identidade
com controlo *active-low*

E	I	O
0	0	0
0	1	1
1	0	Z
1	1	Z

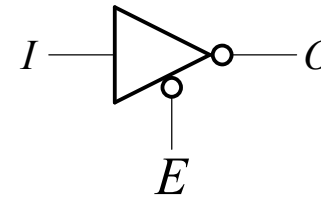


□ Portas *tri-state*



Porta *tri-state* inversora
com controlo *active-high*

<i>E</i>	<i>I</i>	<i>O</i>
0	0	Z
0	1	Z
1	0	1
1	1	0



Porta *tri-state* inversora
com controlo *active-low*

<i>E</i>	<i>I</i>	<i>O</i>
0	0	1
0	1	0
1	0	Z
1	1	Z

O terminal *E* (*enable*) permite que a porta esteja ativa (inserida no circuito), colocando na sua saída o valor que lhe for solicitado, se este sinal estiver ativo (*high* ou *low*). Caso o *enable* não esteja ativo, a saída fica “flutuante”, ou seja, como que desligada fisicamente. O conceito de *enable* é muito comum em sistemas digitais, determinando se um determinado dispositivo ou módulo se encontrará em atividade ou não.



□ Multiplexagem

O conceito de multiplexagem é muito importante e encontram-se exemplos em vários aspetos quotidianos:

- Caixa de velocidades do automóvel
- Único cabo partilhado por várias extensões telefónicas
- Sintonia de postos de rádio
- Lugar sentado na sala de aula

Em cada instante, apenas uma e só uma possibilidade poderá estar presente nos vários cenários acima.

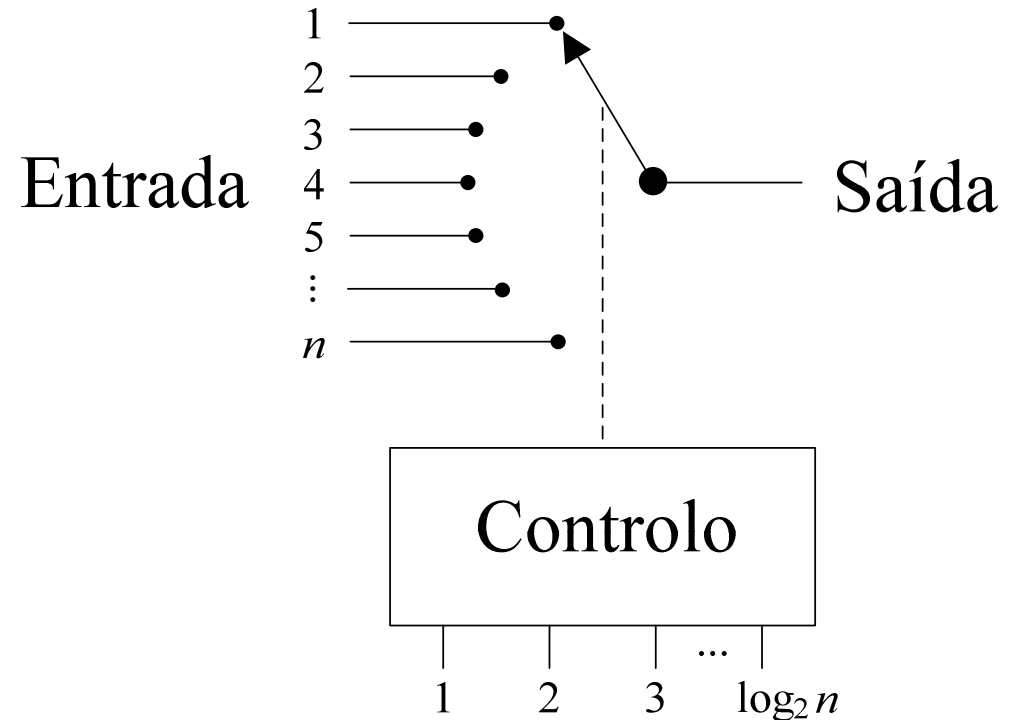


❑ Conceito funcional da multiplexagem

O conceito de multiplexagem consiste em comutar (escolher) apenas uma entre n entradas possíveis e ligá-la à saída.

O bloco de controlo, em função dos seus $\log_2 n$ bits, promove a seleção da entrada a ligar à saída.

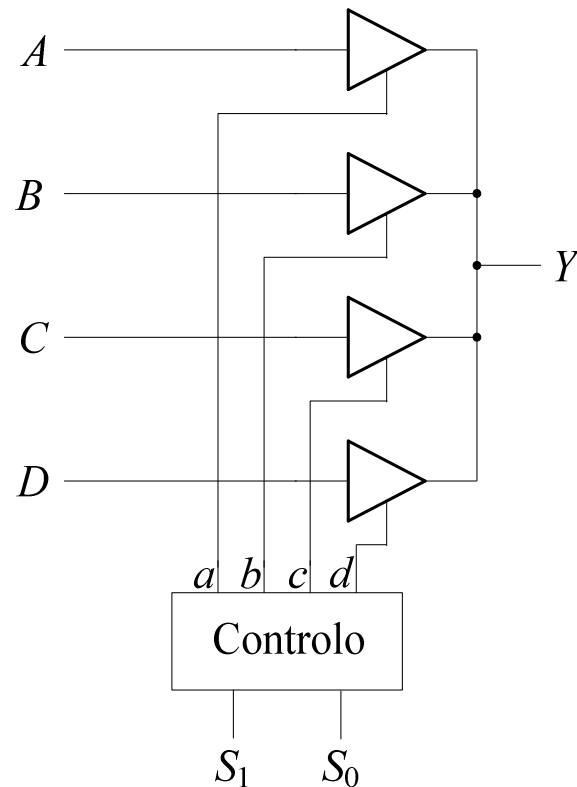
Um sistema com esta funcionalidade denomina-se multiplexador (*multiplexer*).





❑ Exemplo de implementação de um *multiplexer*

Utilizando portas tri-state e um bloco de controlo com o comportamento apropriado, é possível implementar um *multiplexer* como o mostrado de seguida.



Em função da combinação S_1S_0 (bits de seleção), apenas uma das quatro saídas do bloco de controlo ($a - d$) estará ao nível lógico 1, estando as restantes ao nível lógico 0. Tal faz com que apenas uma das entradas ($A - D$) apareça ligada à saída, ficando as restantes em alta impedância.

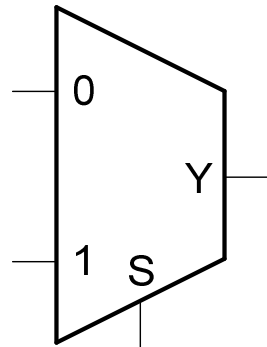
O módulo de controlo descodifica a combinação S_1S_0 e ativa a ligação correspondente entre a entrada e a saída. Este módulo local é um descodificador (*decoder*).



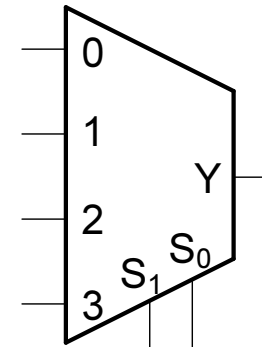
❑ Multiplexer lógico

O *multiplexer* lógico assenta no mesmo princípio já descrito genericamente para os *multiplexers*, ou seja, coloca na sua saída o valor lógico presente na entrada determinada pelos bits de seleção.

Símbolos lógicos:



MUX 2×1

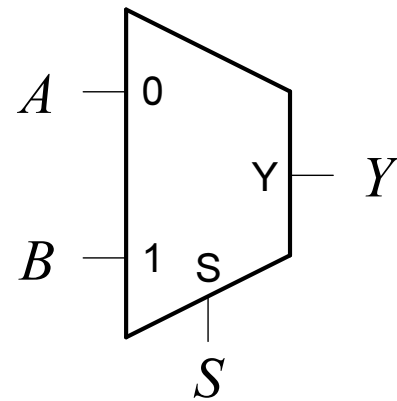


MUX 4×1



□ Síntese do *multiplexer* (MUX) 2×1

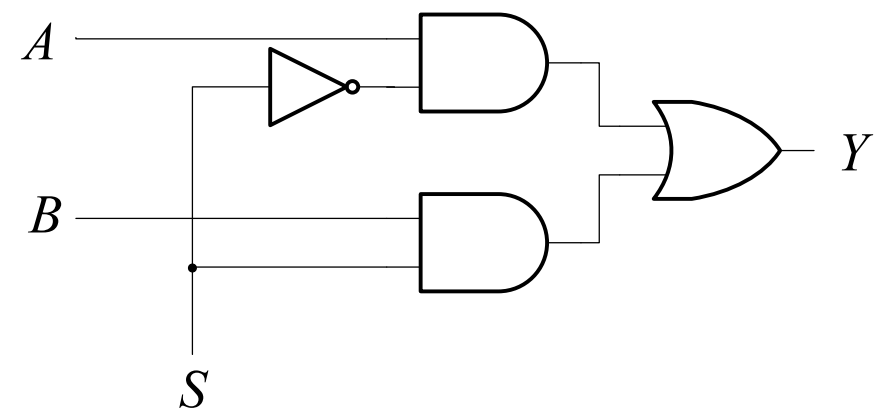
Trata-se de um circuito com três entradas digitais, logo, cuja saída é determinada pela combinação de três variáveis lógicas.



MUX 2×1

S	A	B	Y
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

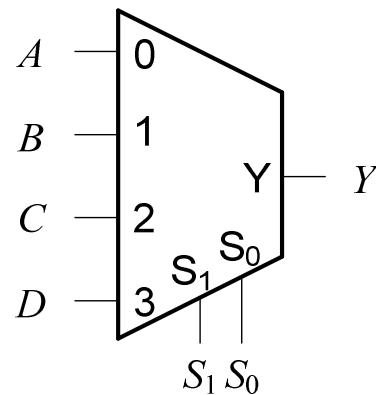
S	A			
	0	1	1	0
0	0	0	1	1





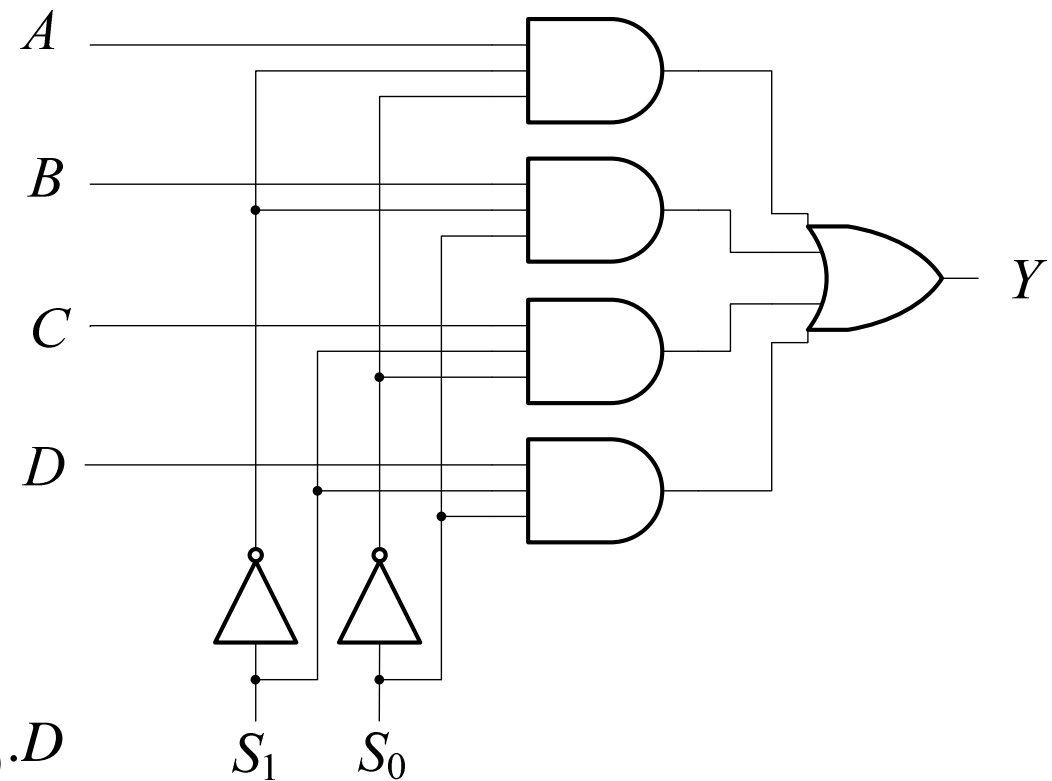
□ Circuito do *multiplexer* (MUX) 4×1

Neste caso, a síntese da função é feita de acordo com uma função de seis variáveis lógicas de entrada.



MUX 4×1

$$Y = \overline{S_1} \cdot \overline{S_0} \cdot A + \overline{S_1} \cdot S_0 \cdot B + S_1 \cdot \overline{S_0} \cdot C + S_1 \cdot S_0 \cdot D$$





❑ Descodificador (*decoder*)

O *multiplexer* utilizando portas *tri-state* fazia recurso a um *decoder* para ativar uma e só uma porta *tri-state*.

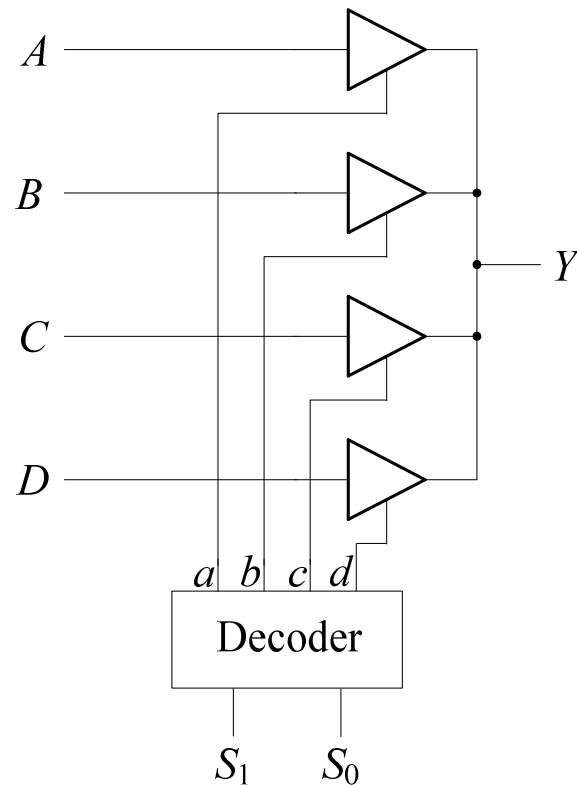
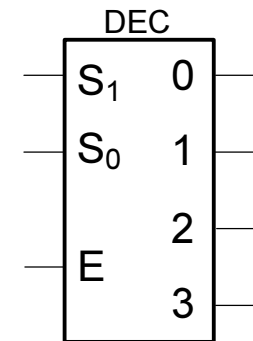


Tabela de verdade e expressões do *decoder*

S_1	S_0	a	b	c	d
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

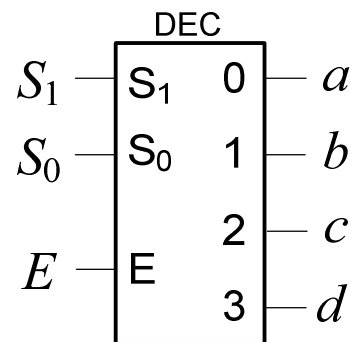
Símbolo lógico





❑ Descodificador (*decoder*)

O *decoder* poderá também ter uma entrada de *enable*, a qual permitirá o funcionamento normal do dispositivo. Se o *enable* for igual a “0”, todas as suas saídas deverão ficar a “0”. Com esta definição, o circuito lógico do *decoder* será:

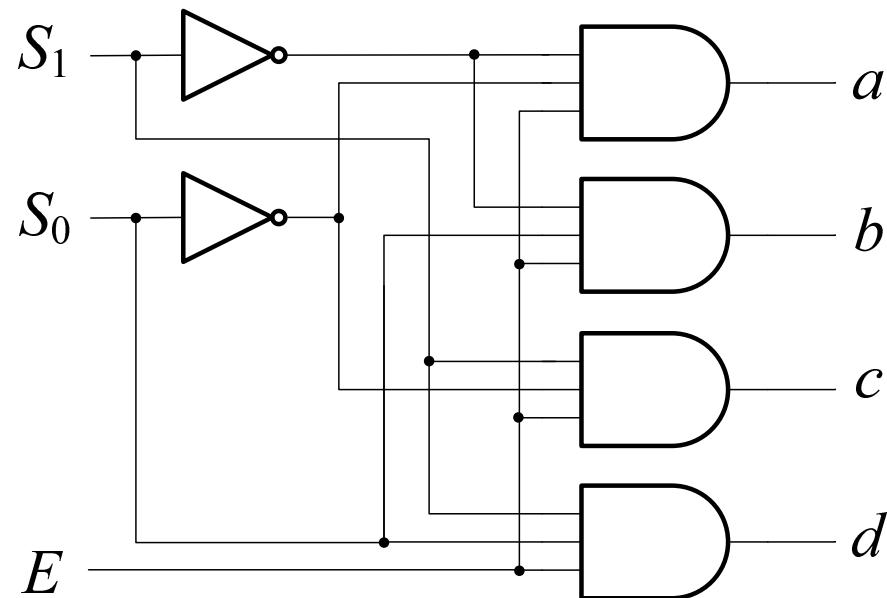


$$a = \overline{S_1} \cdot \overline{S_0} \cdot E$$

$$b = \overline{S_1} \cdot S_0 \cdot E$$

$$c = S_1 \cdot \overline{S_0} \cdot E$$

$$d = S_1 \cdot S_0 \cdot E$$

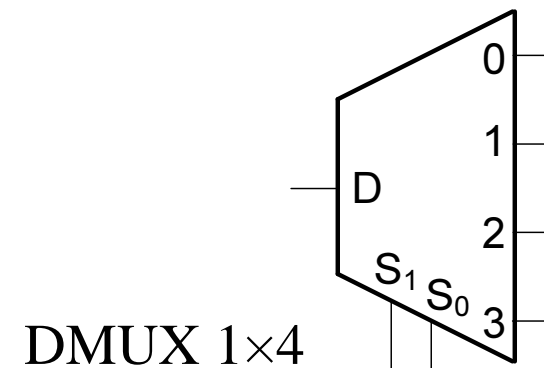
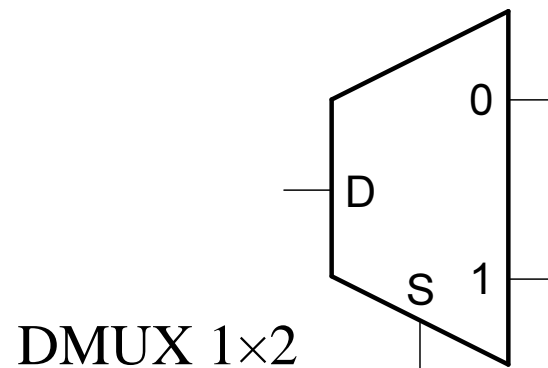




□ Demultiplexer

O *demultiplexer* realiza a função inversa do *multiplexer*, ou seja, tendo uma entrada, coloca o valor lógico que esta apresenta numa das suas 2^n saídas, sendo que essa saída é seleccionada por n entradas de seleção. A sua função é praticamente igual à do decodificador, diferindo no facto deste último colocar sempre “1” na saída pretendida, ao passo que o *demultiplexer* coloca “0” ou “1” dependendo do valor lógico da entrada de informação.

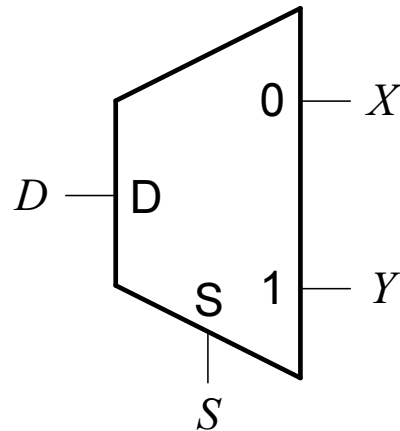
Símbolos
lógicos:





□ Síntese do *demultiplexer* (DMUX) 1×2

Trata-se de um circuito com duas entradas digitais e com duas saídas, que resultam da combinação das duas variáveis lógicas de entrada.



DMUX 1×2

S	D	X	Y
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

$$S = 0: X = D; \quad Y = 0$$

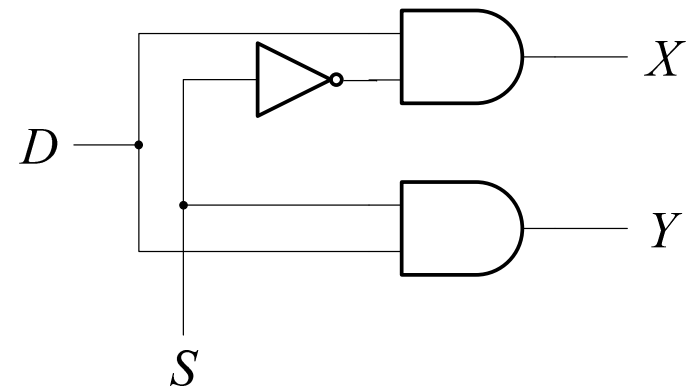
$$S = 1: X = 0; \quad Y = D$$

X	\overline{D}	D
1	0	
0	0	$ S$

$$X = \overline{S} \cdot D$$

Y	\overline{D}	D
0	0	
1	0	$ S$

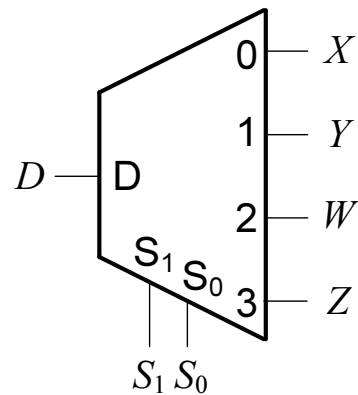
$$Y = S \cdot D$$



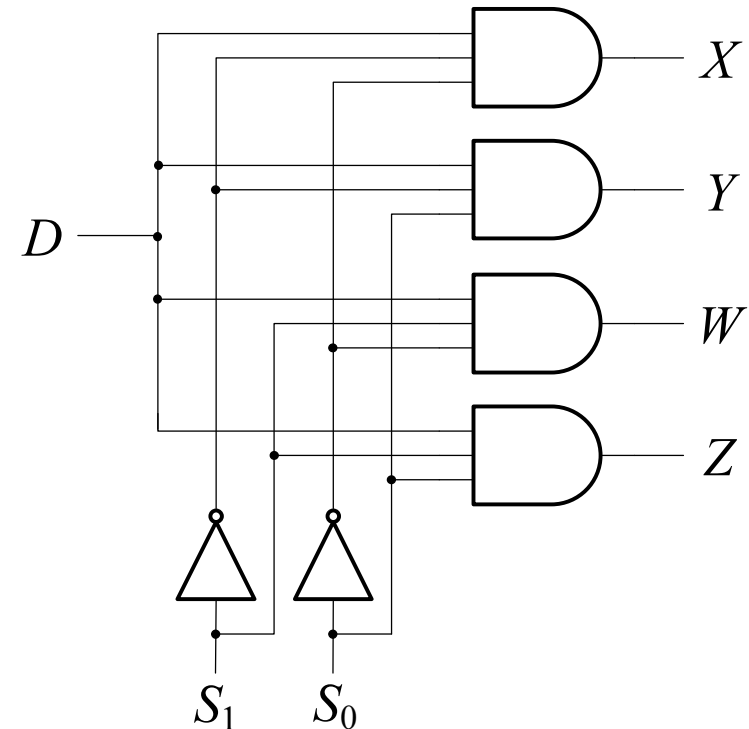


□ Circuito do *demultiplexer* (DMUX) 1×4

Neste caso, a síntese de cada uma das quatro funções de saída é feita de acordo com uma função de três variáveis lógicas de entrada.



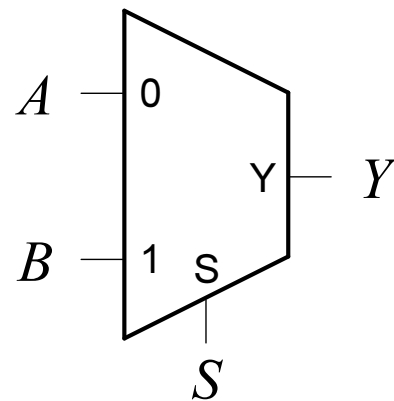
S_1	S_0	D	X	Y	W	Z
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1





❑ Multiplexer no Arduino

A implementação de *multiplexers* no Arduino pode ser feita segundo os mesmos princípios utilizados anteriormente para as outras funções lógicas. Num MUX que opera só com variáveis do tipo `boolean` (bits simples) a função de simulação no Arduino pode ser segundo a definição dada anteriormente. Exemplo para o MUX 2×1 (extensível para o MUX 4×1):



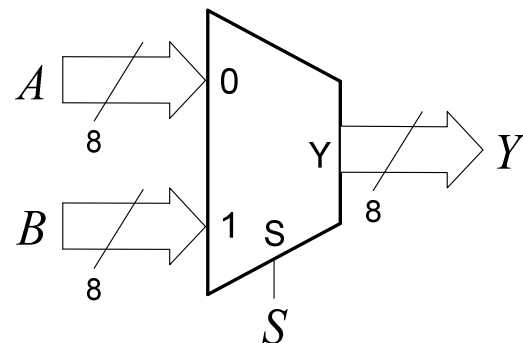
MUX 2×1

```
boolean MUX_2x1 (boolean S, boolean A, boolean B){  
    return !S & A | S & B;  
}
```



❑ Multiplexer no Arduino

Estes módulos são compostos, estão acima do nível da porta lógica. Se se pretender realizar um MUX que tem como entrada variáveis do tipo `byte` e em vez de variáveis `boolean` (bits), o mais importante é que a função de simulação no Arduino opere segundo o princípio funcional pretendido, não tendo a preocupação de operar ao nível da porta lógica. Assim, pode ficar-se com:



MUX 2×1

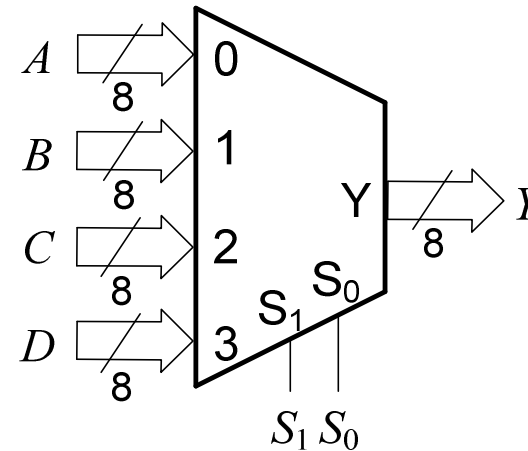
```
byte MUX_2x1 (boolean S, byte A, byte B){
    return S ? B : A;
}
```




❑ Multiplexer no Arduino

Para o caso de um MUX 4×1 que opere com variáveis do tipo `byte`, a função que o simula poderá ser:

```
byte MUX_4x1 (boolean S1, boolean S0, byte A, byte B, byte C, byte D){  
  switch (S1 << 1 | S0){  
    case B00:  
      return A;  
    case B01:  
      return B;  
    case B10:  
      return C;  
    case B11:  
      return D;  
  }  
}
```





□ Operadores “>>”e “<<” no Arduino

Em acréscimo aos operadores *bitwise* já referidos, existem mais dois operadores, “>>”e “<<”, de acordo com a seguinte sintaxe:

variável >> número de bits Operador *shift right*

variável << número de bits Operador *shift left*

Estes operadores realizam o deslocamento dos bits de uma dada variável (ou constante), pelo número de posições especificado, para a direita ou para a esquerda, respetivamente. Com estes operadores e com os outros já conhecidos, é possível realizar-se o que se designa por máscaras.



□ Operadores “>>” e “<<” no Arduino

As máscaras são úteis para se simular o valor lógico dos vários bits num dado barramento digital, tendo como efeito correspondente, a ligação de fios num aparato de *hardware*.

Com o operador “>>”, os n bits mais à direita desaparecem e entram n “1”s ou “0”s do lado esquerdo, dependendo do sinal algébrico do número deslocado.

Com o operador “<<”, os n bits mais à esquerda desaparecem e entram n “0”s do lado direito.

Exemplo: $S_1 = 1$, $S_0 = 0$. Com $(S1 \ll 1 \mid S0)$ obtém-se B10 como pretendido.



□ Operadores “>>”e “<<” no Arduino

Os operadores “>>”e “<<” também podem ser usados para realizar de forma muito rápida e computacionalmente eficiente, divisões e multiplicações por potências de 2, respetivamente. Por cada unidade de deslocamento, ter-se-á um decréscimo para metade, ou um acréscimo para o dobro, respetivamente.

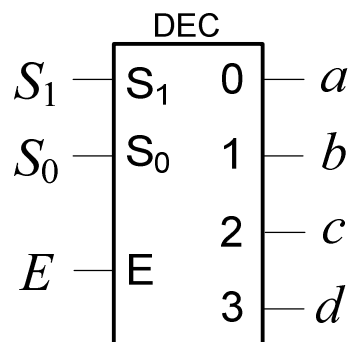
Exemplos:

```
int x = -16;      // B111111111111110000
int y = x >> 3;   // B11111111111111110, decimal: -2
int w = 1000;     // B0000001111101000
int z = w << 2;   // B0000111110100000, decimal: 4000
```



❑ Decoder no Arduino

A implementação de *decoders* no Arduino pode ser feita segundo os mesmos princípios utilizados anteriormente para as outras funções lógicas. Num decoder que opera só com variáveis do tipo `boolean` (bits simples), a função de saída, simulada no Arduino, deve devolver um `byte`, sendo este uma composição dos vários bits de saída na posição certa. Possível implementação:



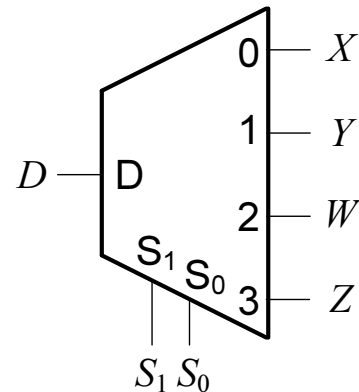
Decoder com dois bits de seleção

```
byte DEC2 (boolean S1, boolean S0, boolean E){  
    return E << (S1 << 1 | S0);  
}
```



❑ Demultiplexers no Arduino

A implementação de DMUXs no Arduino pode ser feita segundo os mesmos princípios utilizados para o *decoder*. A diferença reside na entrada de dados, a qual pode ocasionar, na saída seleccionada, o valor “0” ou “1” consoante o valor desta entrada. Possível implementação (para o DMUX 1×4):



DMUX 1×4

```
byte DMUX_1x4 (boolean S1, boolean S0, boolean D){  
    return D << (S1 << 1 | S0);  
}
```

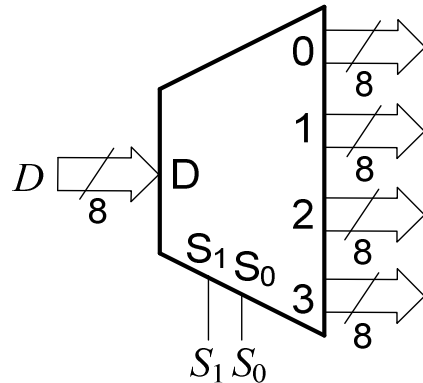
Qual é a conclusão a retirar, face ao *decoder* com *enable*?..



❑ Demultiplexers no Arduino

Para o caso de um DMUX 1×4 que opere com variáveis do tipo `byte`, a função que o simula deverá articular-se com um *array* global que guarde os resultados das saídas do dispositivo. Uma implementação possível poderá ser:

DMUX 1×4



```
#define DIM_DMUX 4

byte DMUX_1x4_out[DIM_DMUX];

void DMUX_1x4 (boolean S1, boolean S0, byte D){
    for (byte i = 0; i < DIM_DMUX; i++){
        DMUX_1x4_out[i] = 0;

        DMUX_1x4_out[S1 << 1 | S0] = D;
    }
}
```



❑ Conceito de *enable*

Para consolidar:

Em situações nas quais se tenha a concatenação de vários módulos, pode ser útil colocar inativo qualquer um deles, consoante o objetivo pretendido. Nestas condições, evidencia-se o sinal de *enable*, o qual quando ativo (pode sê-lo a “1” ou a “0”), permite o funcionamento do respetivo dispositivo em conjunto com os outros aos quais se encontra ligado, ao passo que quando o *enable* está inativo, o mesmo módulo encontrar-se-á inibido de realizar as suas funções. Nestas circunstâncias, as suas saídas encontrar-se-ão, ou em alta impedância, ou a um nível lógico interpretável como que de inoperância.



Circuitos sequenciais



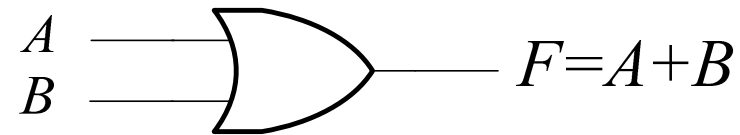
□ Introdução

- Nos circuitos combinatórios, o valor lógico presente nas saídas é unicamente dependente do valor lógico presente, nesse momento, nas entradas.
- Nos circuitos sequenciais, o valor lógico presente nas saídas, num determinado instante, depende não só do valor das entradas nesse instante, mas também do valor lógico que as saídas tomaram anteriormente.
- Existe, assim, a necessidade de que o circuito disponha de “memória” para se “recordar” dos acontecimentos passados.

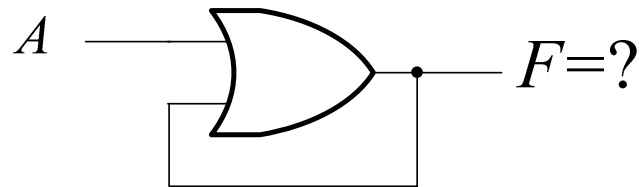


□ Conceito de memória

1. Circuito combinatório – A saída depende exclusivamente do estado das entradas.



2. Circuito com realimentação positiva



A	F^*	F
0	0	0
1	0	1
0	1	1

F^* - Estado anterior

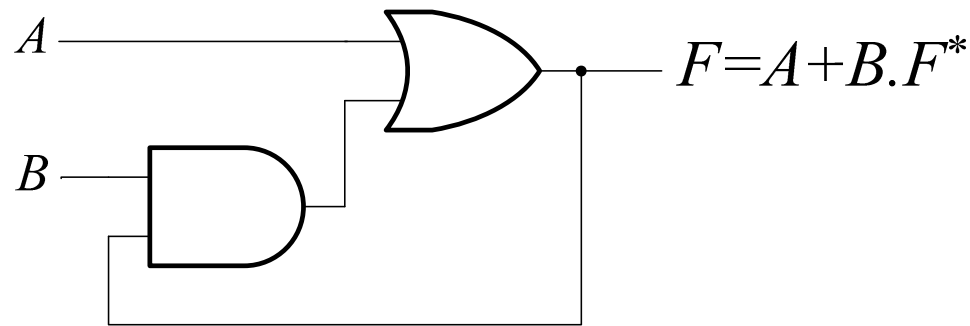
F - Estado atual

A saída fica com o estado anterior, não dependendo apenas do estado das entradas



□ Conceito de memória

3. Circuito com realimentação e entradas



A saída depende de variáveis de entrada e do estado anterior.

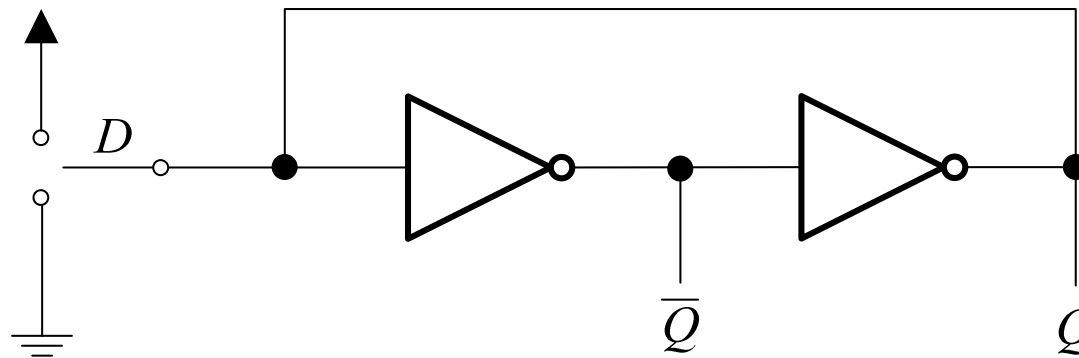
$$A = 0 \rightarrow F = B \cdot F^*$$

A variável B controla a realimentação

Por memória, entende-se a retenção de um estado durante um tempo.



□ Memória digital de 1 bit

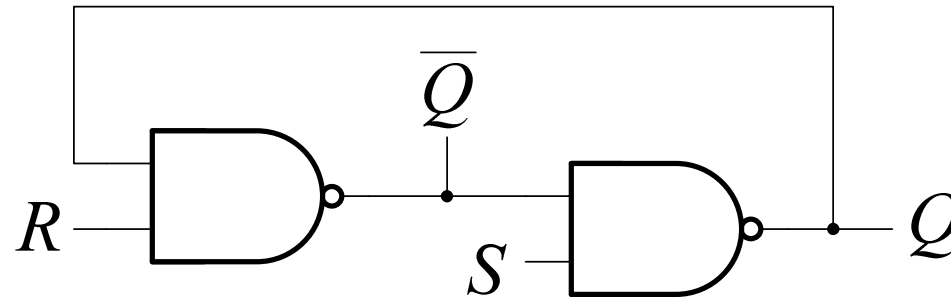


No momento em que se liga a alimentação, é aleatório o que fica em Q , mantendo-se permanentemente este valor (0 ou 1).

D normalmente está “no ar”, e se momentaneamente, em D se aplicar um valor lógico diferente do de Q , será este que ficará memorizado. Note-se que, momentaneamente, ter-se-á um curto-circuito na saída do segundo inversor.



□ Memória digital de 1 bit

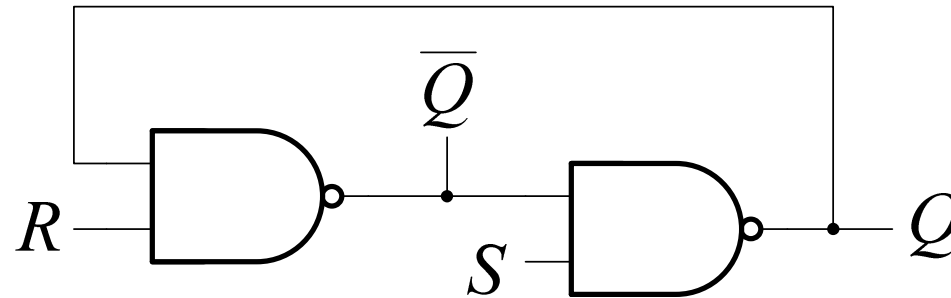


Neste circuito, R e S , em repouso, estão ao nível lógico 1. Desta forma, manter-se-á em Q o valor lógico que já esteja presente, dado que as portas NAND se encontram abertas, possibilitando realimentação.

Quando R é colocado a “0” e S se mantém a “1”, Q ficará com o valor lógico “0”, mantendo-o se já o tiver e transitando de “1” para “0” se fosse “1” que lá estivesse presente.



□ Memória digital de 1 bit

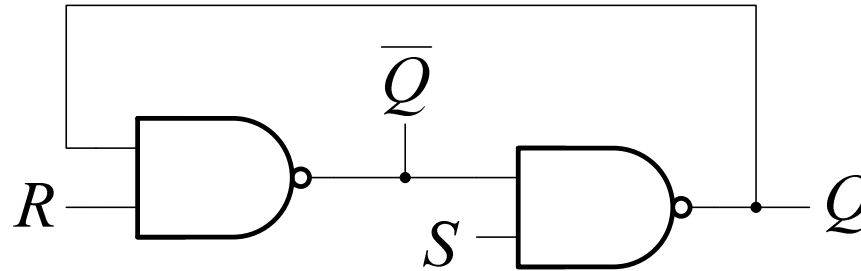


Quando S é colocado a “0” e R se mantém a “1”, Q ficará com o valor lógico “1”, mantendo-o se já o tiver e transitando de “0” para “1” se fosse “0” que lá estivesse presente.

Se R e S forem ambos atuados (colocados a “0”), Q e \overline{Q} ficarão no estado “1”, embora este tipo de combinação seja de evitar. **Só uma das entradas poderá estar atuada!**



□ Flip-flop “S-R latch”



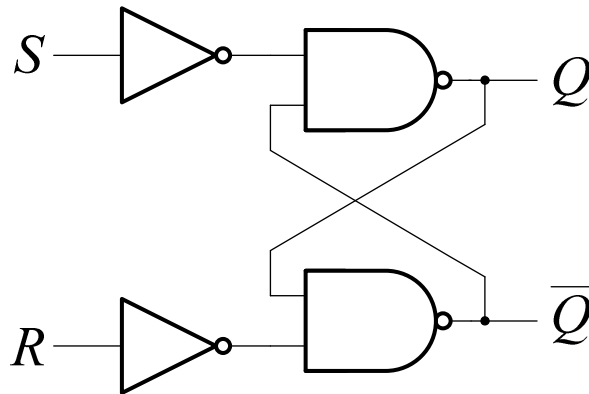
Este circuito chama-se *flip-flop S-R latch* (R : *reset* – colocar a “0”, S : *set* – colocar a “1”). A sua definição formal é:

- Sensível a dois sinais de entrada: S (*set*) para impor o estado 1, R (*reset*) para impor o estado “0”;
- Tem duas saídas: Q , cujo valor define o estado do *flip-flop*, e \overline{Q} (complemento de Q);
- Toma o valor 1, quando $S = 1$ e $R = 0$;
- Toma o valor 0, quando $S = 0$ e $R = 1$;
- Mantém o estado anterior enquanto $S = 0$ e $R = 0$.

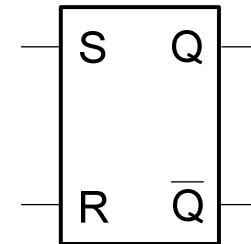


□ Flip-flop “S-R latch”

De acordo com a definição anterior, R e S têm que ser negados no esquema original, resultando no seguinte circuito redesenhado:



Símbolo lógico:



A situação $R = 1$ e $S = 1$ não se deve ser considerada. No caso concreto do presente circuito, Q ficará a 1, significando que a entrada *set* é prioritária sobre a *reset* – *set overrides reset*.



□ Flip-flop “S-R latch”

Tabela de verdade do *flip-flop S-R latch*.

S	R	Q^*	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	×
1	1	1	×

Q^* - Estado anterior

Q - Estado atual

Q	R			
	0	0	×	1
Q^*	1	0	×	1
	S			

Se os *don't care* forem encarados como “1”, Q fica:

Esta é a implementação já conhecida.

Se os *don't care* forem encarados como “0”, Q fica:

$$Q = S.\bar{R} + \bar{R}.Q^* = \overline{\overline{S.\bar{R} + \bar{R}.Q^*}} = \overline{\bar{R}(S + Q^*)} = \overline{\bar{R}} + \overline{S + Q^*} = R + \bar{S} + \bar{Q}^*$$

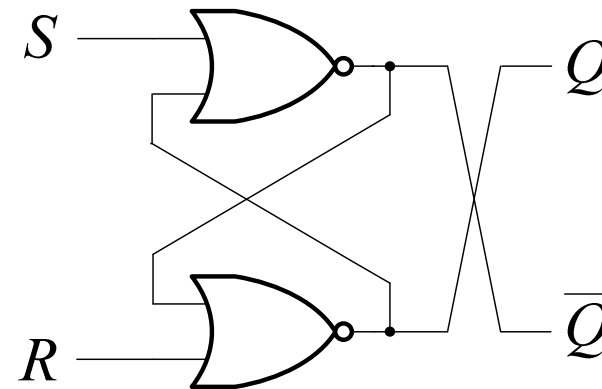


□ Flip-flop “S-R latch”

A função $Q = R + S + \overline{Q}^*$

corresponde ao esquema:

(implementação com portas NOR)



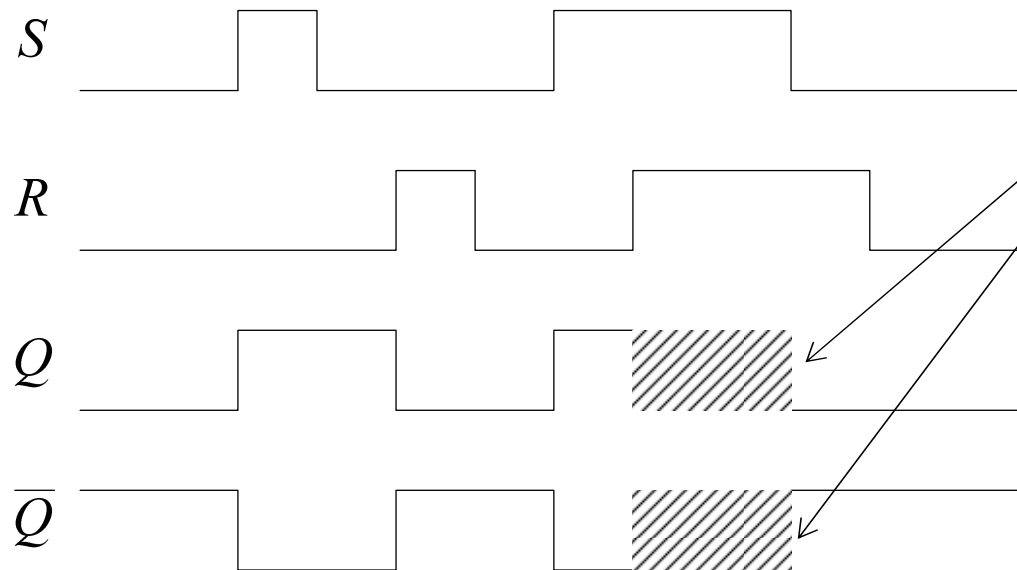
Nesta versão, tudo se passa como na versão realizada com portas NAND, exceto que, no caso em que $S = R = 1$, Q toma o valor 0, ou seja o *reset* é prioritário sobre o *set – reset overrides set*.

Na verdade, a situação $S = R = 1$ não conduz a um valor indeterminado, mas a um valor bem definido, que é função da implementação empregue.



□ Flip-flop “S-R latch”

Comercialmente, poderá desconhecer-se como é que o *flip-flop* é implementado, pelo que a combinação de entradas $S = R = 1$ é de excluir. Outra razão tem a ver com o facto de Q e \bar{Q} tomarem o mesmo valor, o que é incoerente. Diagrama temporal das saídas, função das entradas S e R :



Indeterminado, função da
implementação com portas
NAND ou NOR:

NAND $\rightarrow Q = 1$ e $\bar{Q} = 1$

NOR $\rightarrow Q = 0$ e $\bar{Q} = 0$



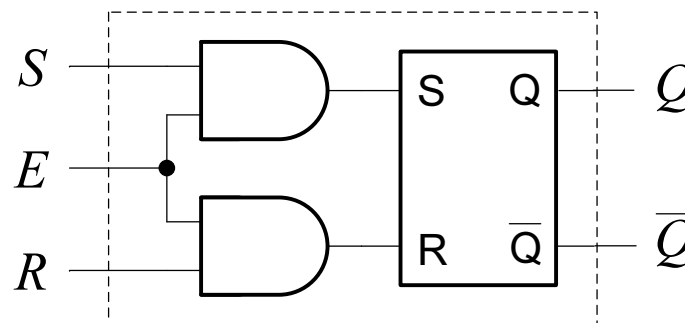
□ Flip-flop “S-R-E latch” (S-R com enable)

Definição:

- Enquanto $E = 0$, é insensível às entradas S e R , mantendo-se no estado anterior;
- Tem o comportamento do flip-flop “S-R latch” quando $E = 1$.

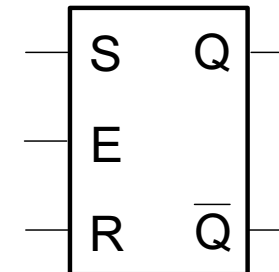
Tabela de verdade do flip-flop S-R-E latch.

E	S	R	Q
0	-	-	Q^*
1	0	0	Q^*
1	0	1	0
1	1	0	1
1	1	1	-



Síntese a partir de um S-R latch.

Símbolo lógico:



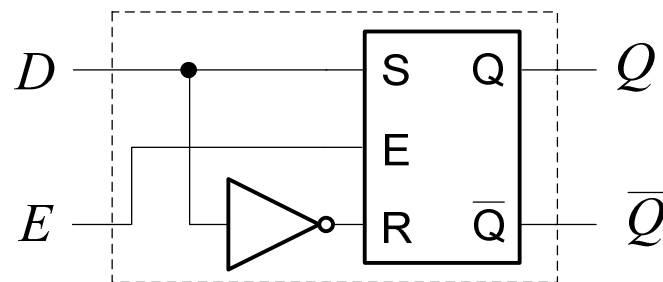


□ Flip-flop “D latch”

Definição:

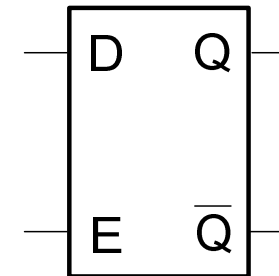
- Enquanto $E = 0$, mantém de memória o último valor assumido pela entrada D antes da transição descendente de E ;
- Mantém transparência da entrada D para a saída Q enquanto $E = 1$.

E	D	Q
0	0	Q^*
0	1	Q^*
1	0	0
1	1	1



Síntese a partir de um *S-R-E latch*.

Símbolo lógico:





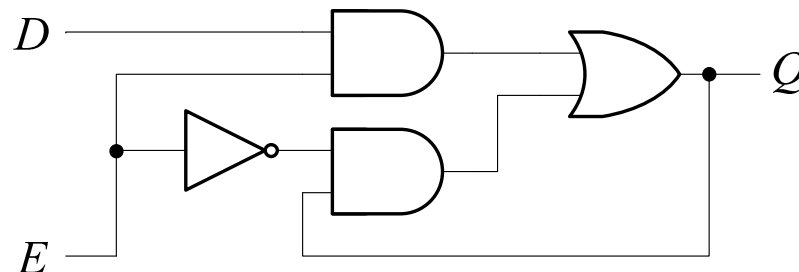
❑ Síntese direta do *flip-flop* “D latch”

A partir da definição, é possível criar a tabela de verdade que rege essa especificação e, a partir daí, sintetizar o circuito lógico mais simples.

Q^*	E	D	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

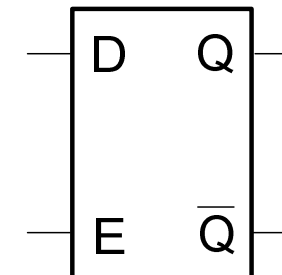
Q	D			
	0	0	1	0
Q^*	1	1	1	0
	E			

$$Q = D.E + Q^*.\bar{E}$$



Síntese a partir da definição

Símbolo lógico:

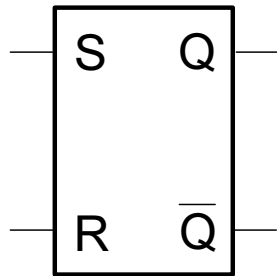




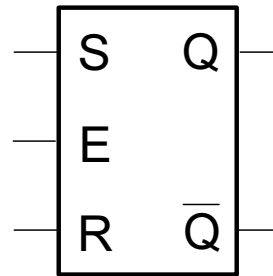
□ Resumo dos *flip-flops* tipo “*latch*”

Os *flip-flops* do tipo *latch* vistos até aqui, respondem de imediato aos estímulos que forem colocados nas suas entradas. Tal tipo de *flip-flop* designa-se por **assíncrono**.

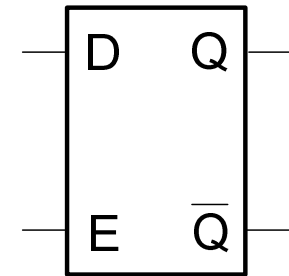
Símbolo lógico dos *flip-flops* assíncronos já abordados:



Latch S-R



Latch S-E-R



Latch D



❑ Conceito de *edge-triggered*

Ao contrário dos *flip-flops* assíncronos, os quais fazem sentir de imediato na sua saída os estímulos das entradas, nalgumas circunstâncias, interessa que as células de memória seja recetivas só em determinados instantes chave.

Por exemplo:

No *latch D*, interessa que a célula de memória seja sensível ao valor lógico presente na entrada *D*, não durante todo o tempo em que a entrada *E* esteja a 1, mas exclusivamente na sua transição de 0 para 1 – transição ascendente de *E*. *Flip-flops* com este comportamento denominam-se “*edge-triggered*” (disparados no flanco, ou na transição), também se denominando por **síncronos**.



□ Flip-flop “D edge-triggered”

Definição:

- Transfere a entrada D para a saída Q no instante da transição ascendente de *clock* (CLK), mantendo-a memorizada até que ocorra outra transição ascendente de *clock*.

Tabela de verdade
do *flip-flop D*
edge triggered:

CLK	D	Q^*	Q
↑	0	-	0
↑	1	-	1
0	-	Q^*	Q^*
1	-	Q^*	Q^*
↓	-	Q^*	Q^*

Tabela de transição de estados:

Q^*	Q	D
0	0	0
0	1	1
1	0	0
1	1	1



□ Flip-flop “D edge-triggered”

Símbolos lógicos possíveis dos *flip-flops D edge-triggered*:

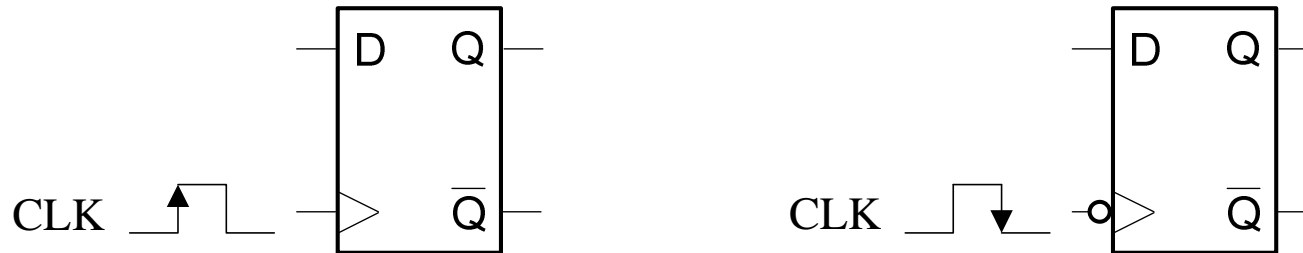
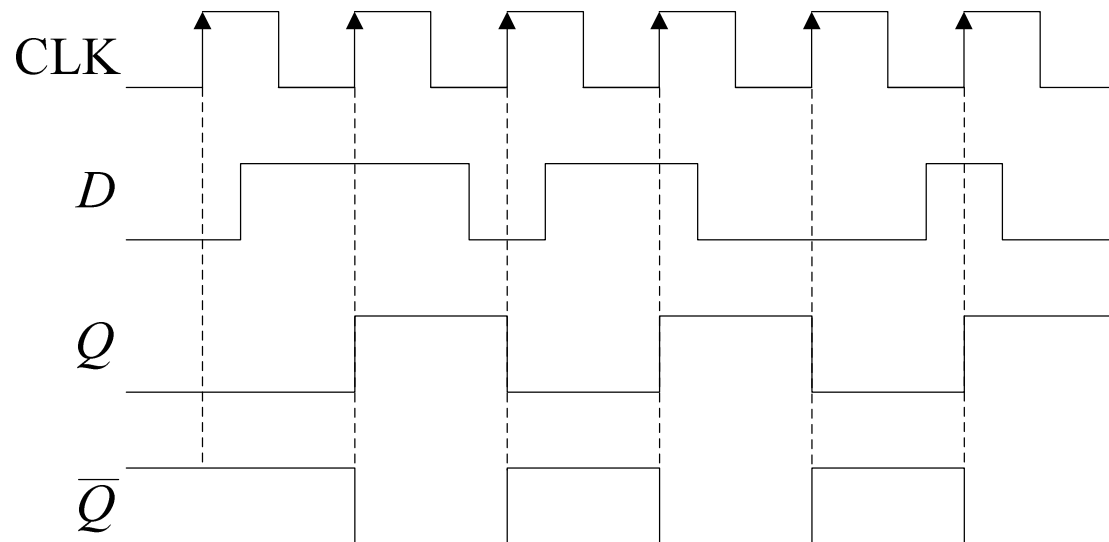


Diagrama temporal
de formas de onda:





□ Flip-flop “J-K edge-triggered”

Definição:

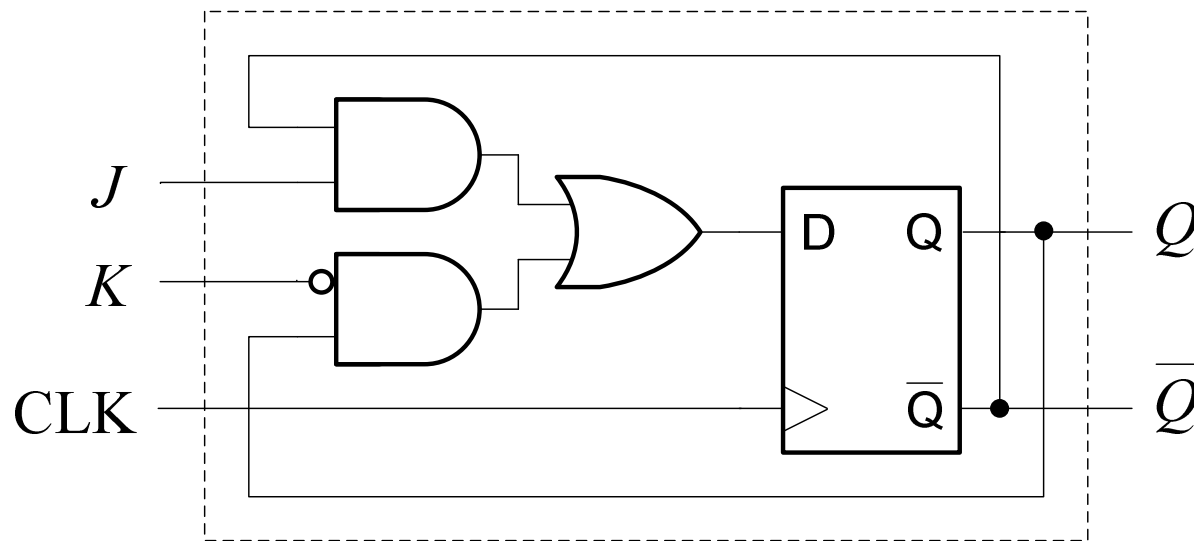
- Na transição ascendente de CLK, toma em consideração os valores presentes nas entradas J e K , de tal modo que , se $Q = 1$, só $K = 1$ poderá levá-lo para 0;
- Se estiver no estado $Q = 0$, só $J = 1$ poderá levá-lo para 1. Quando ambas as entradas J e K estiverem a 1, inverte o estado da saída quando acontece uma transição ascendente de CLK.



□ Flip-flop “J-K edge-triggered”

Síntese do *flip-flop J-K edge-triggered* a partir de um *flip-flop D edge-triggered*, a partir da definição:

Tabela de verdade do *flip-flop J-K edge triggered*:



CLK	J	K	Q^*	Q
↑	0	0	Q^*	Q^*
↑	0	1	-	0
↑	1	0	-	1
↑	1	1	Q^*	$\overline{Q^*}$
0	-	-	Q^*	Q^*
1	-	-	Q^*	Q^*
↓	-	-	Q^*	Q^*



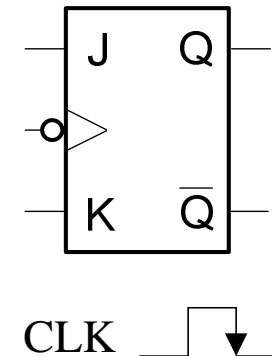
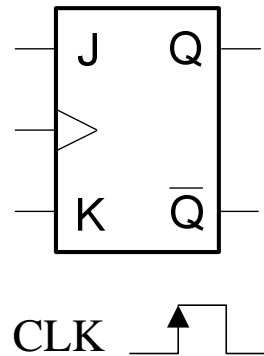
□ Flip-flop “J-K edge-triggered”

A tabela de transição de estados é um auxiliar muito importante para realizar o projeto de circuitos lógicos envolvendo *flip-flops*.

Tabela de transição
de estados:

Q^*	Q	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

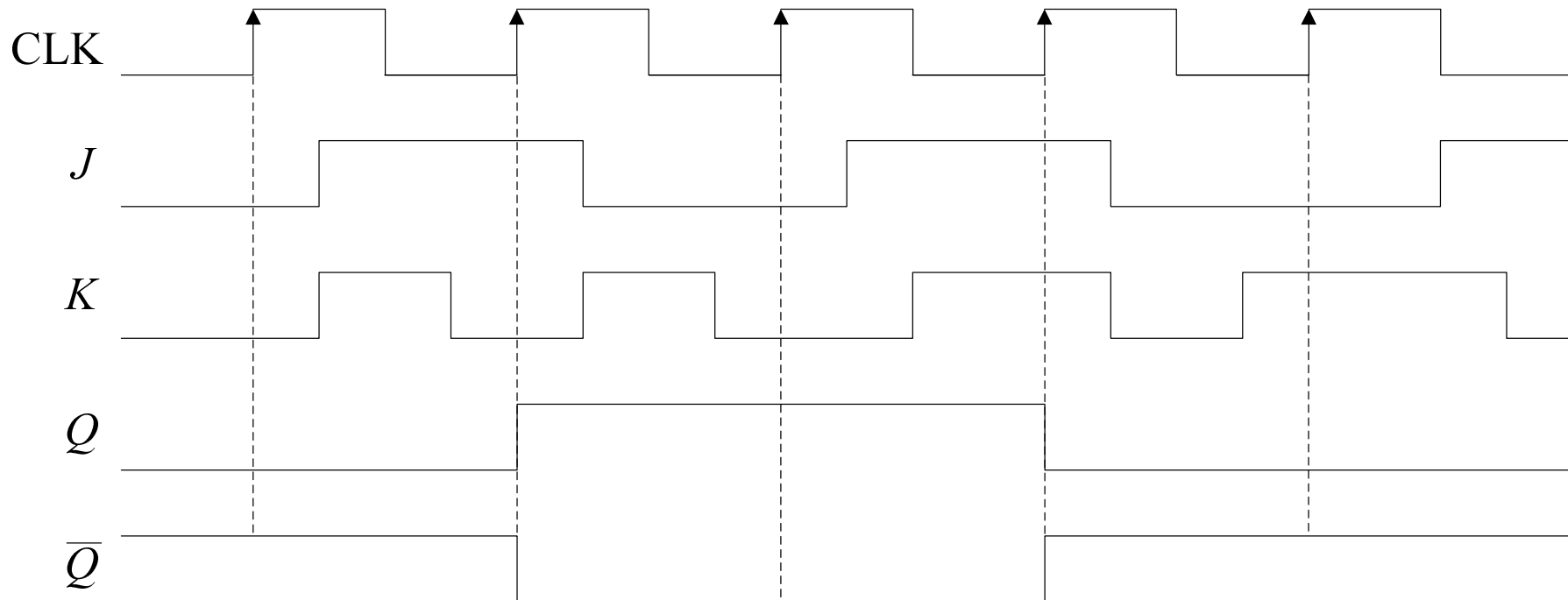
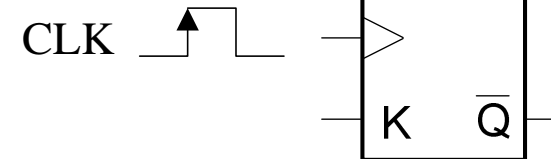
Símbolos lógicos possíveis para
os *flip-flops J-K edge-triggered*:





□ Flip-flop “J-K edge-triggered”

Diagrama temporal de formas de onda:





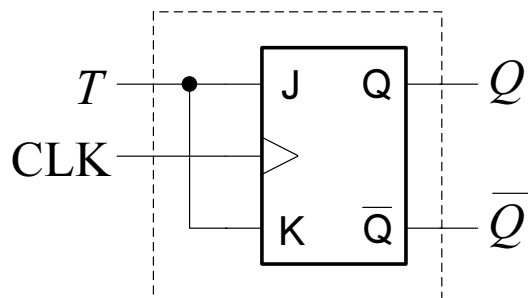
□ Flip-flop “T edge-triggered”

Definição:

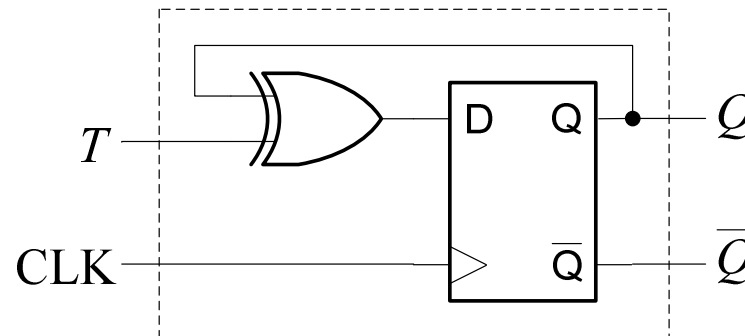
- Sempre que $T = 1$, inverte o estado a cada transição ascendente de CLK;
- Com $T = 0$, permanece no estado anterior, estando insensível ao *clock*.

Dada a definição:

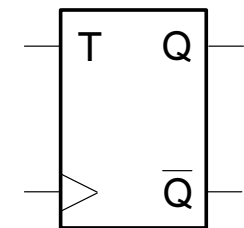
Síntese a partir de um *flip-flop J-K edge-triggered*



Síntese a partir de um *flip-flop D edge-triggered*



Símbolo lógico:





□ Flip-flop “T edge-triggered”

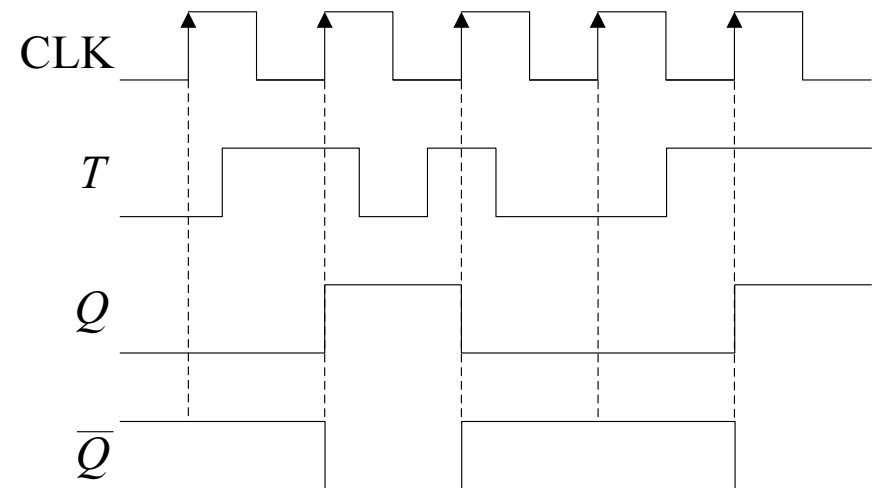
Tabela de verdade do *flip-flop T edge triggered*:

CLK	T	Q^*	Q
↑	0	Q^*	Q^*
↑	1	Q^*	$\overline{Q^*}$
0	-	Q^*	Q^*
1	-	Q^*	Q^*
↓	-	Q^*	Q^*

Tabela de transição
de estados:

Q^*	Q	T
0	0	0
0	1	1
1	0	1
1	1	0

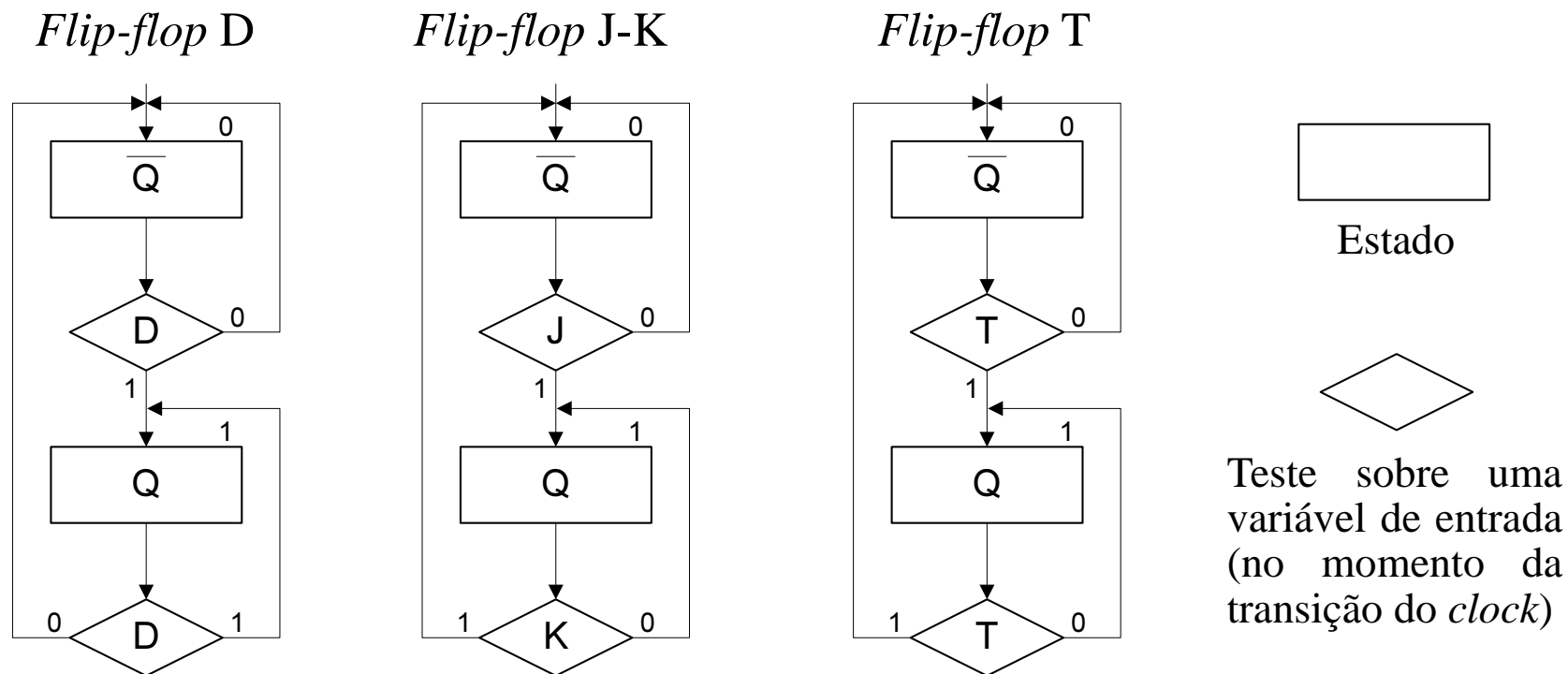
Diagrama temporal de
formas de onda:





□ Diagramas de estado dos *flip-flops*

Trata-se sempre de *flip-flops edge-triggered*. O teste do valor lógico das entradas, que condicionará o próximo estado, ocorre sempre na transição ascendente de *clock*.

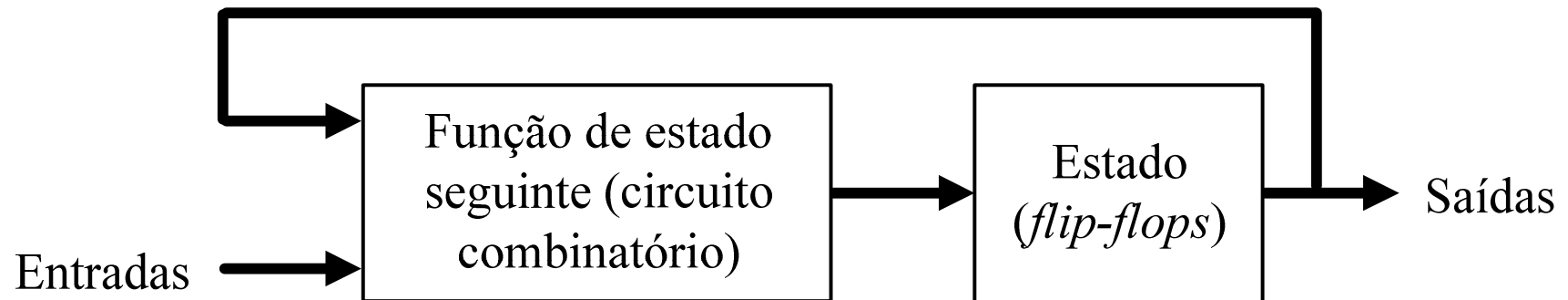




❑ Projeto de circuitos sequenciais (máquinas de estados)

O projeto *hardware* de circuitos sequenciais é baseado em *flip-flops* e designa-se por máquinas de estados. Os modelos existentes são:

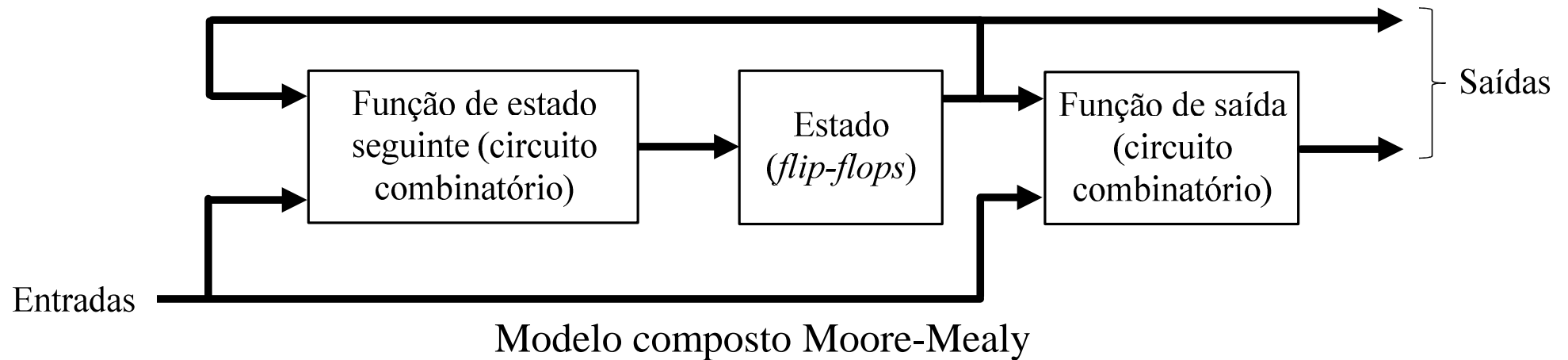
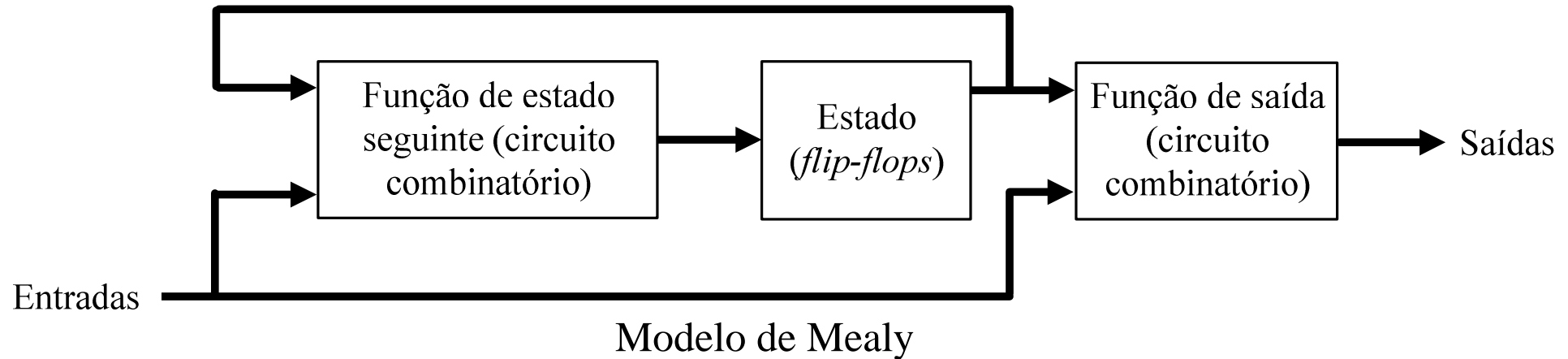
- Modelo de Moore
- Modelo de Mealey
- Modelo composto Moore-Mealy.



Modelo de Moore



❑ Projeto de circuitos sequenciais (máquinas de estados)





❑ Projeto de circuitos sequenciais (máquinas de estados)

Formas de representação de sistemas:

- Diagrama de blocos
- Diagramas de estados (ASM – *Algorithmic State Machine*)

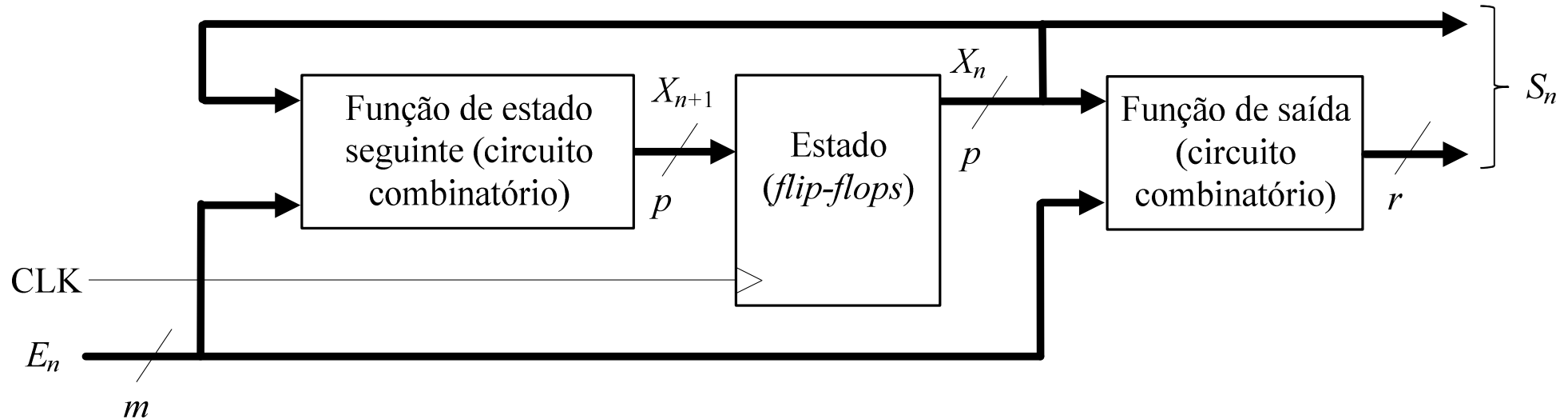
Os sistemas são projetados para cumprir objetivos, realizando um algoritmo.

Algoritmo

- Processo suscetível de ser mecanizado (sistematizado), ou seja, implementado por dispositivos digitais;
- Propriedades: finito, inteligível (não existirem ambiguidades), exequível, caracterizável externamente (relação entre saídas e entradas);
- Formas de representação mais utilizadas: fluxograma, *ASM chart* (para representação de máquinas de estados).



❑ Projeto de circuitos sequenciais (máquinas de estados)

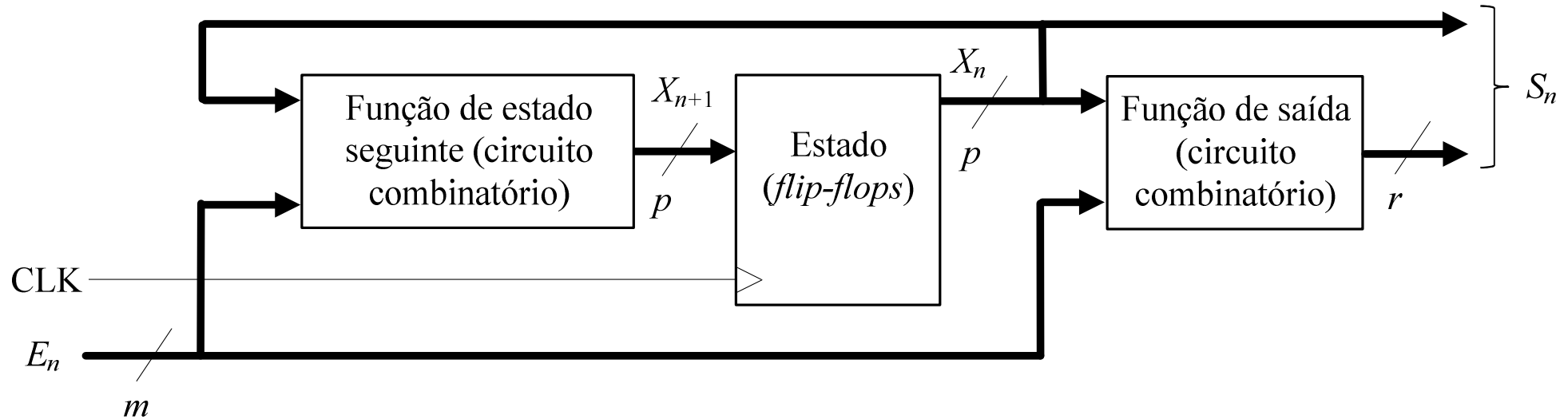


Estes circuitos prestam-se à adoção de métodos de projeto sistemáticos e englobam o conhecimentos de circuitos combinatórios.

A utilização do ASM chart, como passo intermédio de um projeto de circuito sequencial, separa o problema de formulação do algoritmo e da correspondente implementação.



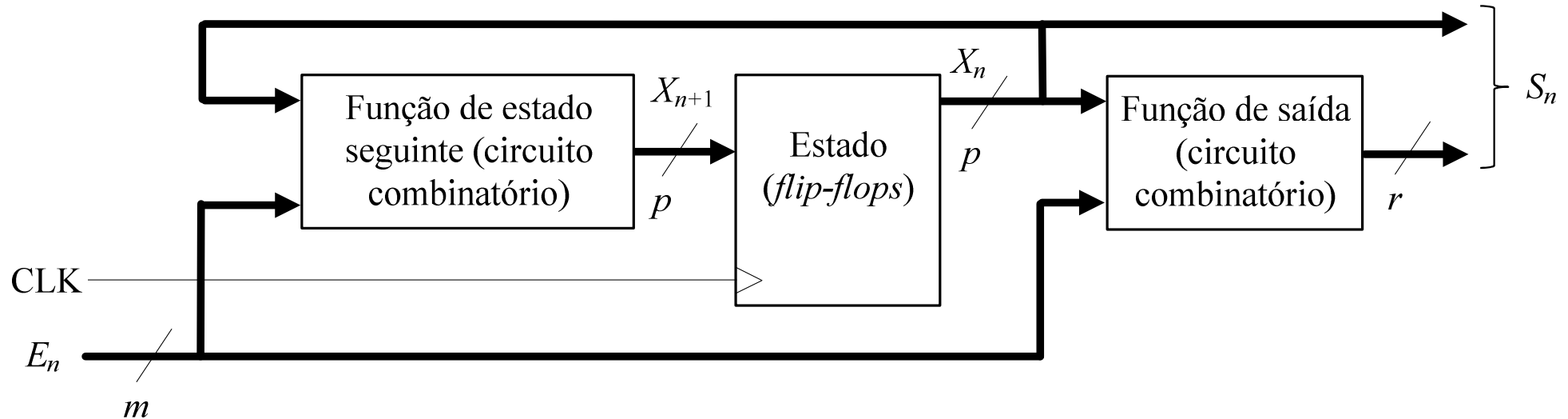
□ Projeto de circuitos sequenciais (máquinas de estados)



- O bloco de “Estado” é um conjunto de *flip-flops edge-triggered*. Se forem do tipo *D* ou *T*, ter-se-ão p entradas. Com o tipo *J-K*, serão $2 \times p$ entradas;
- X_n é o estado presente (saídas Q dos *flip-flops*);
- X_{n+1} é o estado seguinte, para o qual o sistema vai evoluir na próxima transição de *clock*;



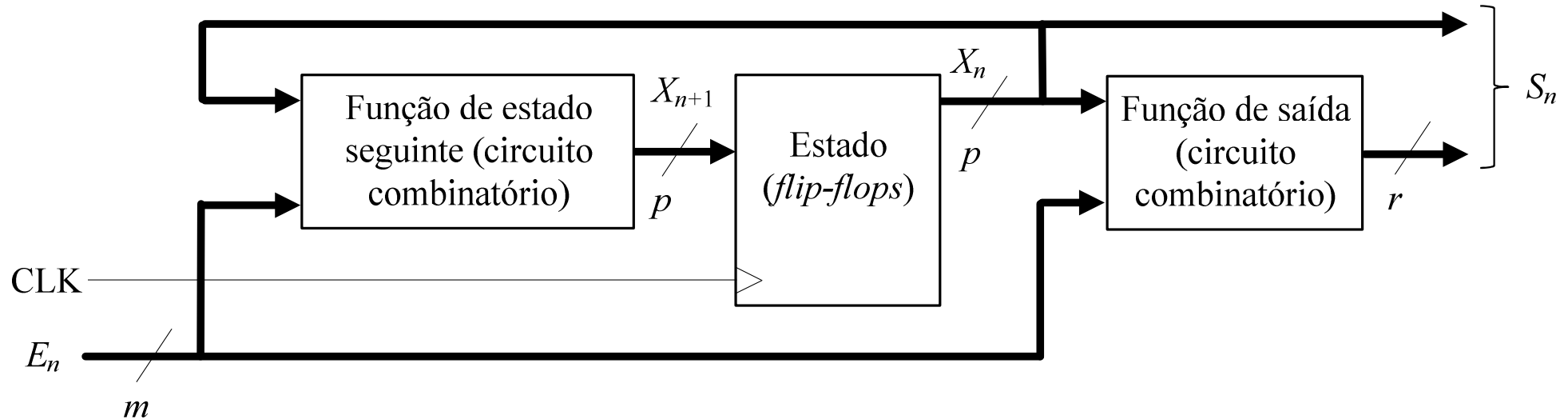
□ Projeto de circuitos sequenciais (máquinas de estados)



- $S_n = f(X_n, E_n)$ é o conjunto de funções booleanas que relaciona as saídas (S_n) com o estado presente (X_n) e as entradas (E_n);
- $X_{n+1} = g(X_n, E_n)$ é o conjunto de funções booleanas que relaciona o estado seguinte (X_{n+1}) com o estado presente (X_n) e as entradas (E_n);
- As p componentes de X_n chamam-se variáveis de estado (saídas dos flip-flops após o n -ésimo impulso de *clock*);



□ Projeto de circuitos sequenciais (máquinas de estados)



- As funções f e g podem ser descritas por ASM ou por uma tabela de “Estado seguinte” e “Saída”, função de “Estado presente” e “Entrada”;

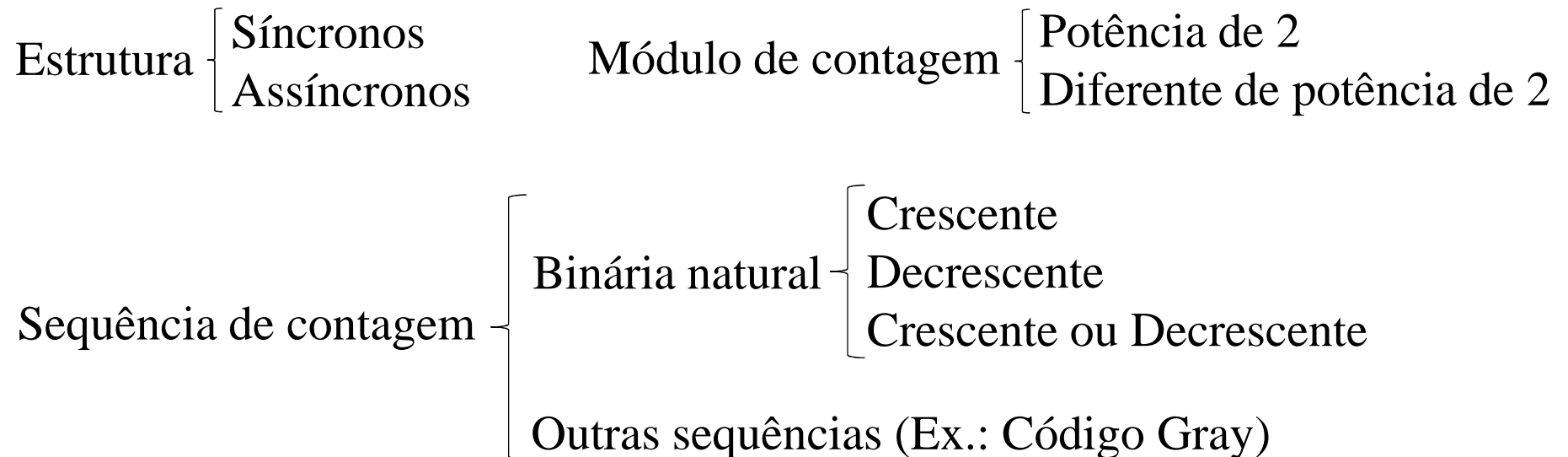
X_n				E_n				X_{n+1}				S_n			
x_p	...	x_1	x_0	e_p	...	e_1	e_0	x_p	...	x_1	x_0	s_p	...	s_1	s_0



□ Contadores

Os contadores são dispositivos destinados a realizar contagens em sequência, possuindo um registo (conjunto de *flip-flops*), sobre o qual a contagem é acumulada.

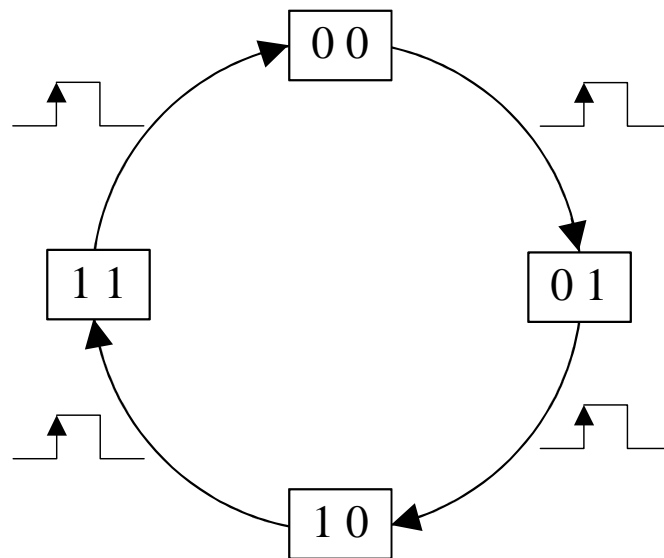
Caracterização dos contadores



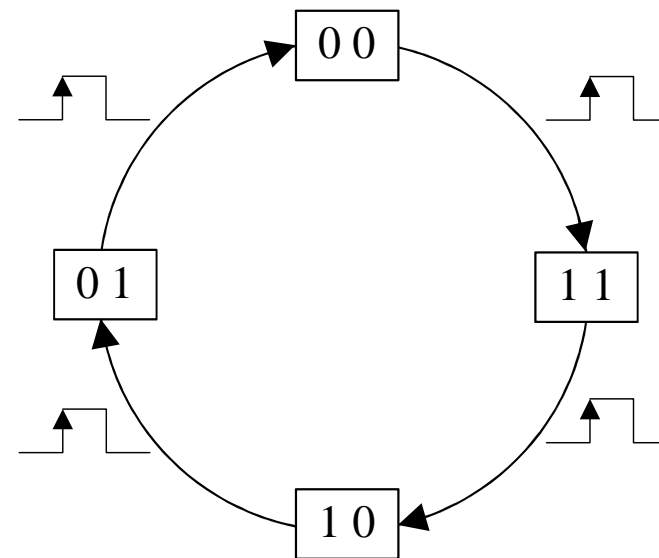


□ Contadores

Evolução da sequência de contagem num contador síncrono, módulo 4, com sequências de contagem crescente ou decrescente, respetivamente.



Crescente



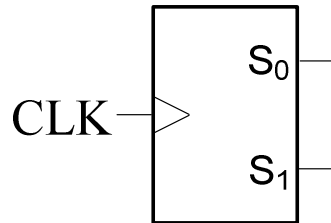
Decrescente



❑ Projeto estruturado de um contador módulo 3, crescente

Este contador terá três estados de contagem, necessitando portanto de dois *flip-flops* para fornecer os bits de saída.

Modelo “caixa preta”
do contador módulo 3



S_1	S_0
0	0
0	1
1	0
<hr style="border-top: 1px dashed black;"/>	
0	0

Tabela de sequência
de estados e saídas

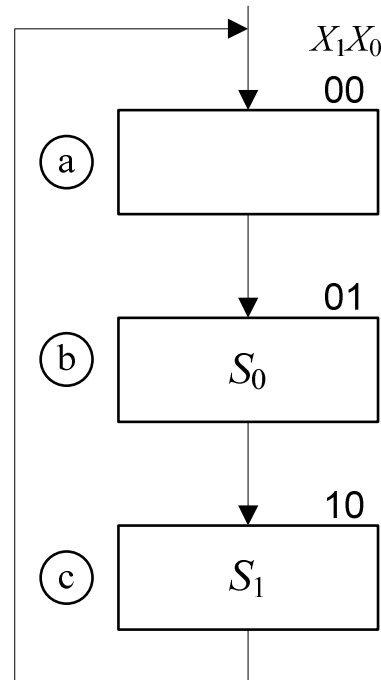
X_n		X_{n+1}		S_n	
x_1	x_0	x_1	x_0	s_1	s_0
0	0	0	1	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	1	-	-	-	-

A saída tomará três estados: 00, 01 e 10. Apesar de 11 ser possível, não é utilizado, sendo considerado *don't care*. Se esta combinação, eventualmente, surgir quando a alimentação é ligada, ao fim de alguns *clocks*, o sistema convergirá.



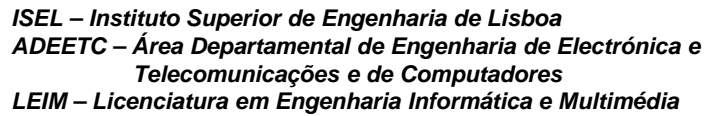
❑ Projeto estruturado de um contador módulo 3, crescente

ASM do contador



- Os estados são representados por retângulos, atribuindo a cada um uma mnemónica que fica do lado esquerdo;
- A cada estado, é atribuído um código binário (*state assignment*), em que cada bit traduz o estado dos *flip-flops* do registo, sendo indicado por cima do retângulo, no lado direito;
- Dentro do retângulo, indica-se o nome das saídas ativas para o respetivo estado.

Neste exemplo, as saídas não precisam de lógica de descodificação, pois coincidem com os bits dos *flip-flops* do registo. Isto acontece porque a contagem é em binário natural.



❑ Projeto estruturado de um contador módulo 3, crescente

X_0		
a	b	
c	-	X_1

Q^*	Q	D	J	K	T
0	0	0	0	-	0
0	1	1	1	-	1
1	0	0	-	1	1
1	1	1	-	0	0

Carlos Carvalho, Abril 2016



❑ Projeto estruturado de um contador módulo 3, crescente

Síntese com *flip-flops* D

D_1	X_0	D_0	X_0
0	1	1	0
0	-	0	-

X_1

$$D_1 = X_0 \quad (- = "1")$$

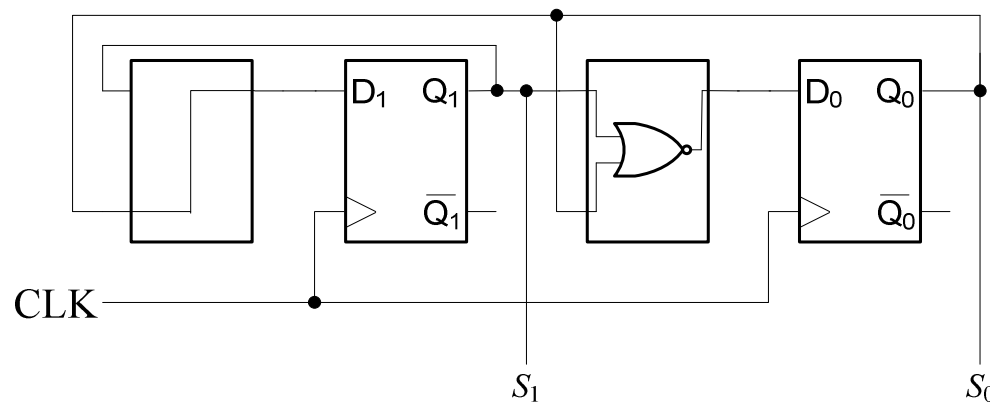
$$D_1 = \overline{X_1} \cdot X_0 \quad (- = "0")$$

$$D_0 = \overline{X_1} \oplus X_0 \quad (- = "1")$$

$$D_0 = \overline{X_1} + X_0 \quad (- = "0")$$

Tabela de transição de estados dos *flip-flops* D, J-K e T

Q^*	Q	D	J	K	T
0	0	0	0	-	0
0	1	1	1	-	1
1	0	0	-	1	1
1	1	1	-	0	0



Por observação do ASM e da tabela de transição de estados dos *flip-flops*, preenchem-se os mapas de Karnaugh de cada entrada dos *flip-flops*, para a transição de cada um dos estados.



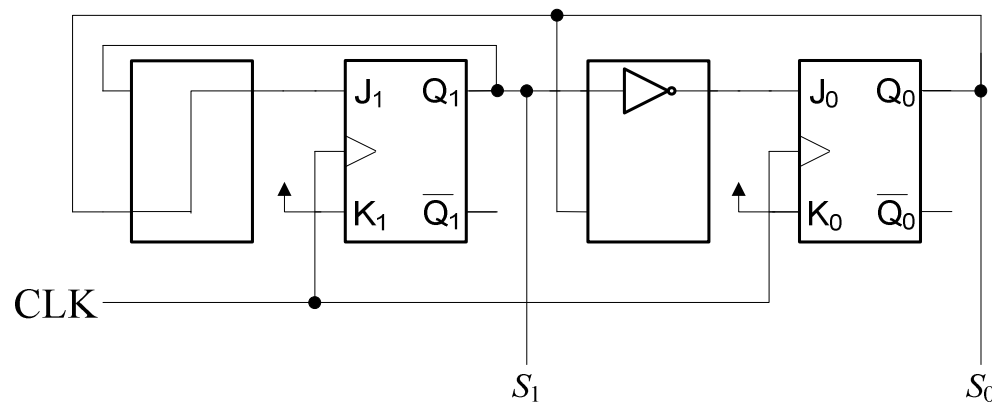
❑ Projeto estruturado de um contador módulo 3, crescente

Síntese com *flip-flops* J-K

J_1	$\overline{X_0}$	K_1	$\overline{X_0}$	J_0	$\overline{X_0}$	K_0	$\overline{X_0}$
0	1	-	-	1	-	-	1
-	-	1	-	0	-	-	-
X_1		X_1		X_1		X_1	
$J_1 = X_0$		$K_1 = 1$		$J_0 = \overline{X_1}$		$K_0 = 1$	

Tabela de transição de estados dos *flip-flops* J-K

Q^*	Q	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0



Na realidade, nesta situação, os blocos combinatórios de FES poderiam não usar portas lógicas, dado que nos *flip-flops* se dispõe do complemento da saída.



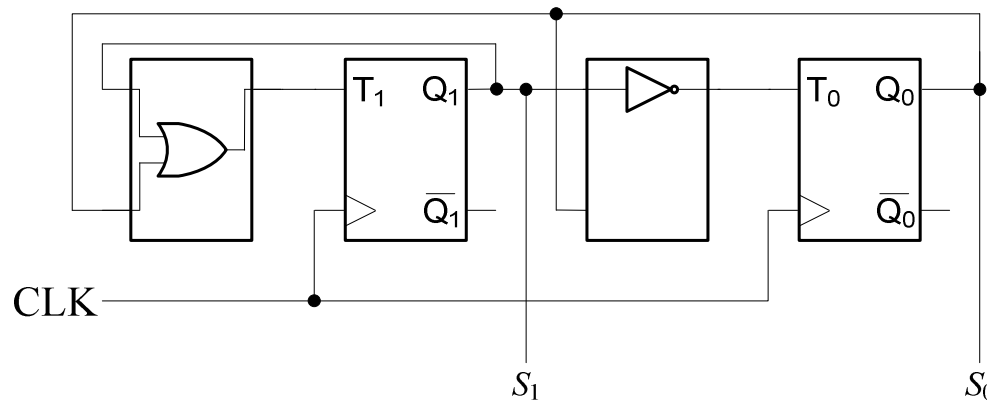
❑ Projeto estruturado de um contador módulo 3, crescente

Síntese com *flip-flops* T

T_1	$\overline{X_0}$		T_0	$\overline{X_0}$	
1	1		0	1	
0	-	X_1	1	-	X_1
$T_1 = X_1 + X_0$			$T_0 = \overline{X_1}$		

Tabela de transição de estados
dos *flip-flops* D, J-K e T

Q^*	Q	D	J	K	T
0	0	0	0	-	0
0	1	1	1	-	1
1	0	0	-	1	1
1	1	1	-	0	0

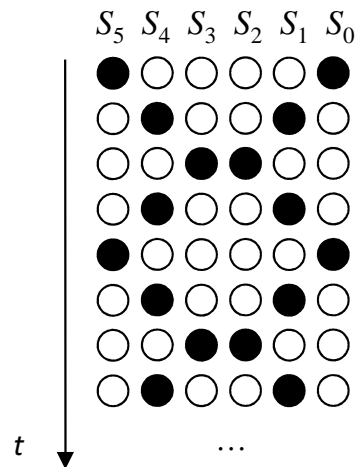


Também nesta situação, o inversor que liga de Q_1 a T_0 poderia ser omitido, utilizando-se diretamente a saída complementar de Q_1 .



□ Projeto de um circuito gerador de padrões sequenciais

É objetivo projetar uma máquina de estados que, ao ritmo dos impulsos de *clock*, acenda seis LEDs de acordo com o seguinte padrão:



Apesar de se terem seis saídas, o número de estados não é necessariamente $2^6 = 64$, dado que só existem três padrões diferentes que se repetem ciclicamente.

Contudo, também não se pode dizer que o número de estados seja igual a três, porque existem, nesse caso, transições de estado ambíguas.

X_n	X_{n+1}	S_5	S_4	S_3	S_2	S_1	S_0
a	b	1	0	0	0	0	1
b	c	0	1	0	0	1	0
c	b	0	0	1	1	0	0
b	a	0	1	0	0	1	0

Considerando apenas três estados, pode notar-se que existe ambiguidade na transição de b para outro estado: num caso, transita para c (2ª linha) e noutro, transita para a (4ª linha).



□ Projeto de um circuito gerador de padrões sequenciais

Como três estados implicam ambiguidade, significa que tem de se aumentar o número de estados. Com quatro estados, já não se têm ambiguidades:

X_n	X_{n+1}	S_5	S_4	S_3	S_2	S_1	S_0
a	b	1	0	0	0	0	1
b	c	0	1	0	0	1	0
c	b	0	0	1	1	0	0
b	a	0	1	0	0	1	0

Apesar de **b** e **d** terem a mesma configuração de saída, na verdade, são estados distintos, uma vez que transitam para um estado sucessor diferente. Como temos quatro estados, estes são codificados com dois bits.

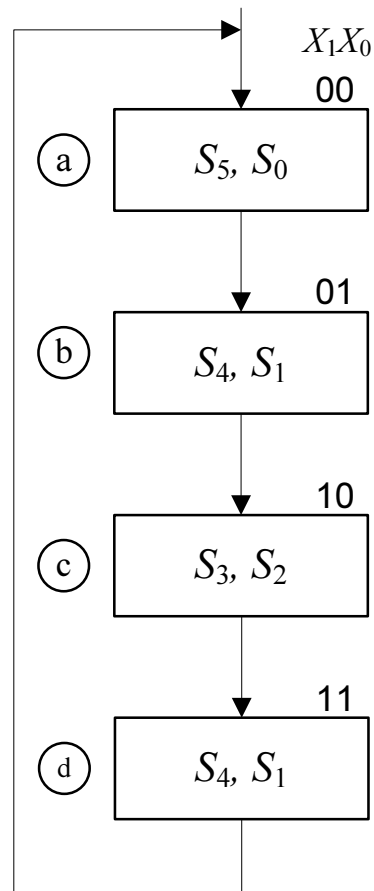
X_n		X_{n+1}		S_n					
X_1	X_0	X_1	X_0	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	1	1	0	0	0	0	1
0	1	1	0	0	1	0	0	1	0
1	0	1	1	0	0	1	1	0	0
1	1	0	0	0	1	0	0	1	0

Note-se que $S_5 = S_0$, $S_4 = S_1$ e $S_3 = S_2$, pelo que ter-se-á que descodificar cada um destes três padrões a partir de X_n .



❑ Projeto de um circuito gerador de padrões sequenciais

ASM chart



Mapa de *state assignment*

X_0	
a	b
c	d

 X_1

Tabela de transição de estados do *flip-flop* D:

Q^*	Q	D
0	0	0
0	1	1
1	0	0
1	1	1

Síntese com *flip-flops* D

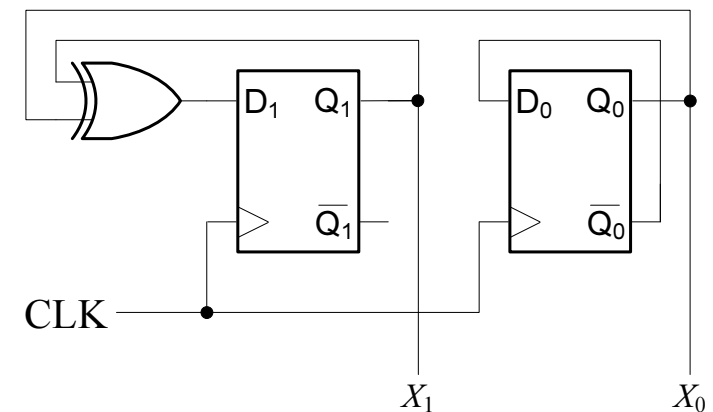
D_1	X_0	
0	1	
1	0	

 X_1

 $D_1 = X_1 \oplus X_0$

D_0	X_0	
1	0	
1	0	

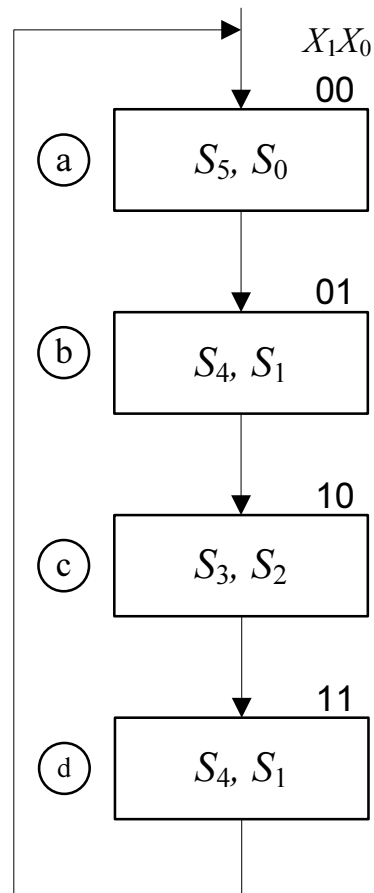
 X_1

 $D_0 = \overline{X_0}$




❑ Projeto de um circuito gerador de padrões sequenciais

ASM chart



Síntese com *flip-flops* J-K

$$J_1 \quad \overline{X_0}$$

0	1
-	-

$$X_1$$

$$J_1 = X_0$$

$$K_1 \quad \overline{X_0}$$

-	-
0	1

$$X_1$$

$$K_1 = X_0$$

$$J_0 \quad \overline{X_0}$$

1	-
1	-

$$X_1$$

$$J_0 = 1$$

$$K_0 \quad \overline{X_0}$$

-	1
-	1

$$X_1$$

$$K_0 = 1$$

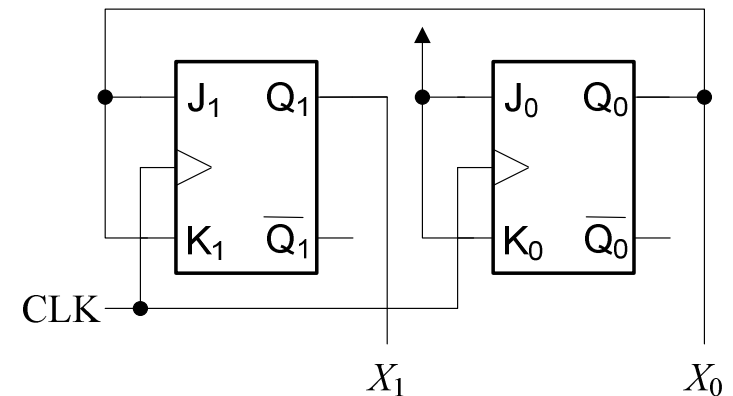
Mapa de *state*
assignment

$$\overline{X_0}$$

a	b
c	d

$$X_1$$

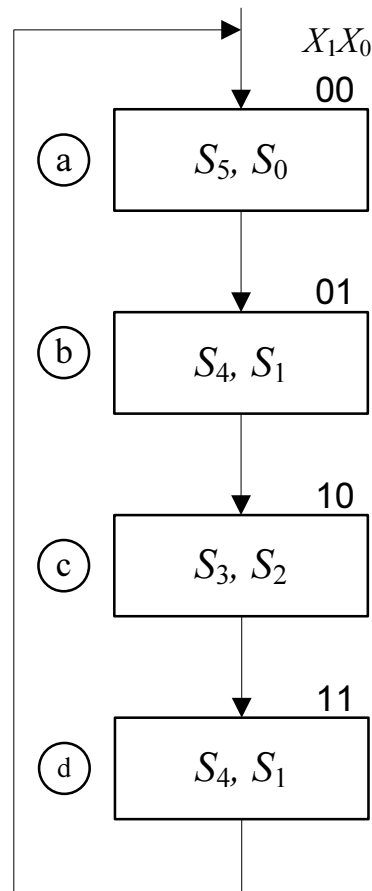
Q^*	Q	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0





❑ Projeto de um circuito gerador de padrões sequenciais

ASM chart



Mapa de *state assignment*

X_0	
a	b
c	d

 X_1

Tabela de transição de estados do *flip-flop* T:

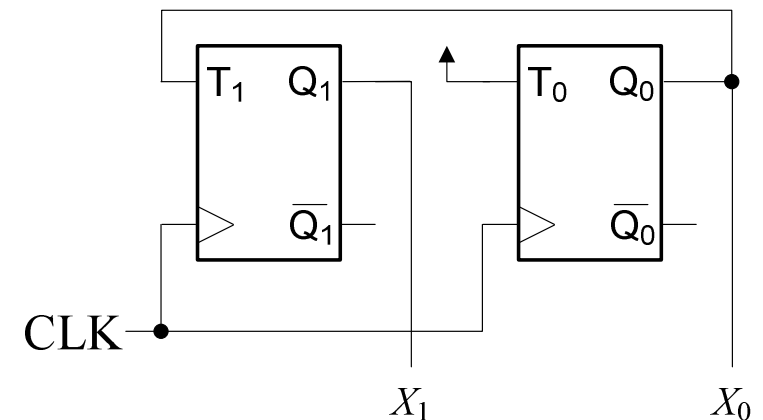
Q^*	Q	T
0	0	0
0	1	1
1	0	1
1	1	0

Síntese com *flip-flops* T

T_1	X_0	
0	1	
0	1	

 $T_1 = X_0$

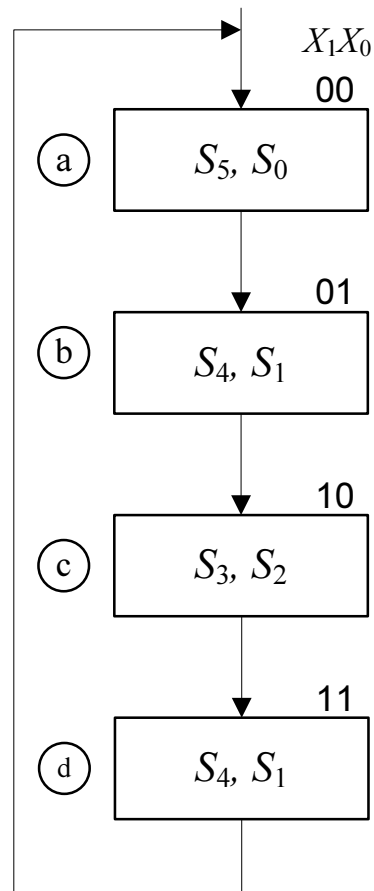
T_0	X_0	
1	1	
1	1	

 $T_0 = 1$




□ Projeto de um circuito gerador de padrões sequenciais

ASM chart

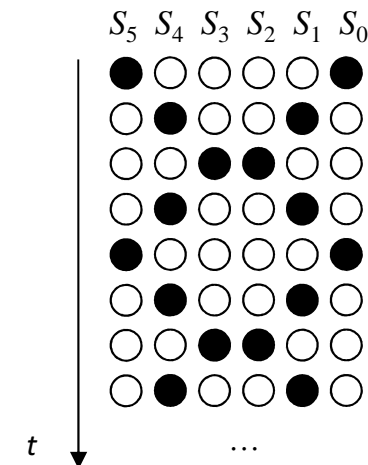
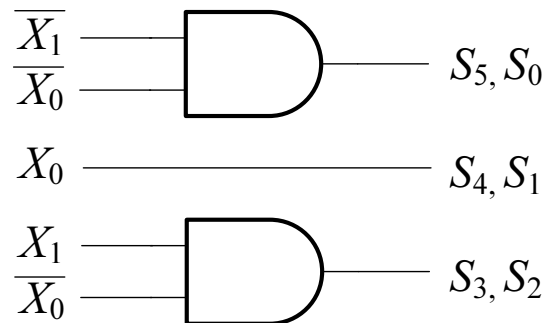


Descodificação das saídas, função das variáveis de estado:

$$S_5 = S_0 = \overline{X_1} \cdot \overline{X_0}$$

$$S_4 = S_1 = \overline{X_1} \cdot X_0 + X_1 \cdot X_0 = X_0 (\overline{X_1} + X_1) = X_0$$

$$S_3 = S_2 = X_1 \cdot \overline{X_0}$$





□ Projeto de um contador de módulo variável

Projetar um contador módulo 3 ou módulo 4, em função de uma entrada M :

- $M = 0$, contagem em módulo 3;
- $M = 1$, contagem em módulo 4.

Existe uma entrada S que para a contagem, mantendo o valor, ou deixa avançar a contagem:

- $S = 0$, avanço normal;
- $S = 1$, para a contagem no valor máximo de contagem.

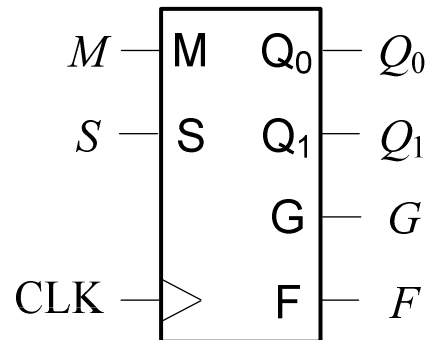
Tem duas saídas:

- F , o número é ímpar;
- G , o número é ≥ 2 , quando em contagem em módulo 4.



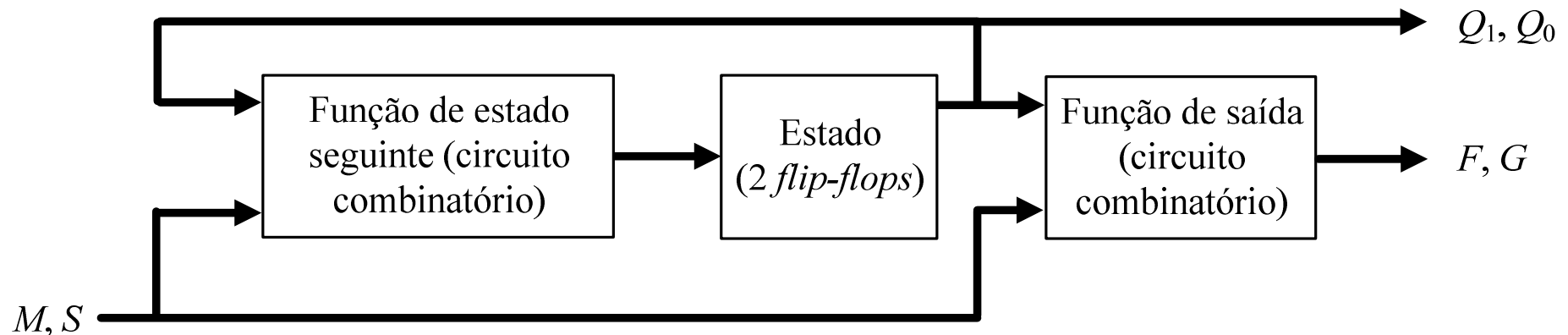
❑ Projeto de um contador de módulo variável

Modelo “caixa preta”



Aspetos especiais a ter em atenção:

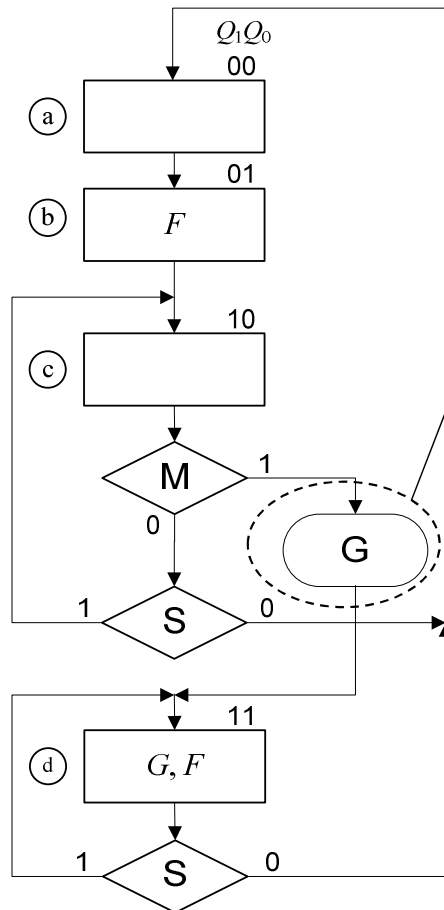
- As entradas M e S não são síncronas com o *clock*;
- A saída G também não é síncrona com o *clock* e é dependente não só do estado, mas também da entrada M .





❑ Projeto de um contador de módulo variável

ASM do sistema



Mapa de *state assignment*:

Q_0	
a	b
c	d

 Q_1

Saída função de estado e entrada. Significa que no estado 10, se $M = 1$, G fica ativada. No estado d, esta permanece ativa, mas se tal não acontecesse, G poderia ativar-se apenas momentaneamente (*glitch*). Nas presentes condições, G pode estar ativa até desde o início do estado c.

Síntese com *flip-flops* D

Q_0		Q_1	
0	1		
$S+M$	S		

- Se M , fica 1
- Se $\overline{M} \cdot S$, fica 1
- Se $\overline{M} \cdot \overline{S}$, fica 0

$\overline{M} \cdot S + M = S + M$

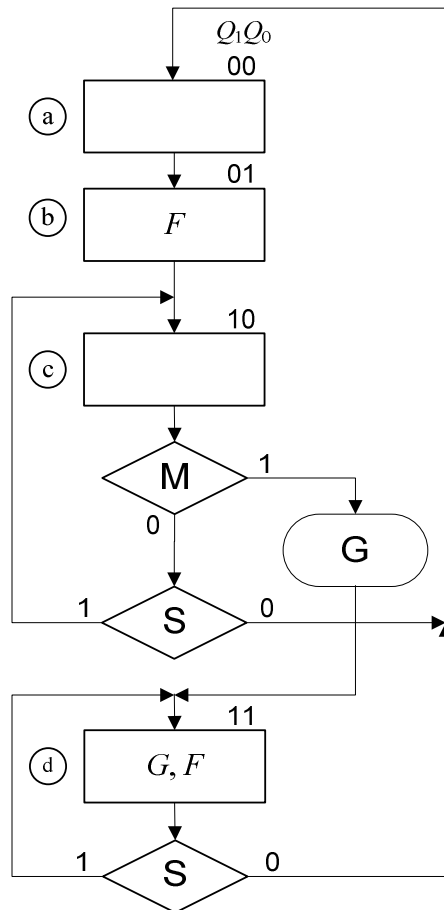
- Se S , fica 1
- Se \overline{S} , fica 0

$$D_1 = \overline{Q_1} \cdot Q_0 + Q_1 \cdot Q_0 \cdot S + Q_1 \cdot \overline{Q_0} \cdot (S + M) = \overline{Q_1} \cdot Q_0 + Q_1 \cdot S + Q_1 \cdot \overline{Q_0} \cdot M$$



❑ Projeto de um contador de módulo variável

ASM do sistema



Mapa de *state assignment*:

Q_0	
a	b
c	d

 Q_1

Extração de expressões simplificadas em mapas de Karnaugh com variáveis inseridas:

- Extrair os “1”s presentes no mapa, considerando como “0” as variáveis, e aproveitando os eventuais *don't care* para simplificação;
- Extrair as variáveis inseridas, considerando também os “1”s como *don't care*.

D_1	Q_0	
	0	1
$S+M$	0	1
S	1	0

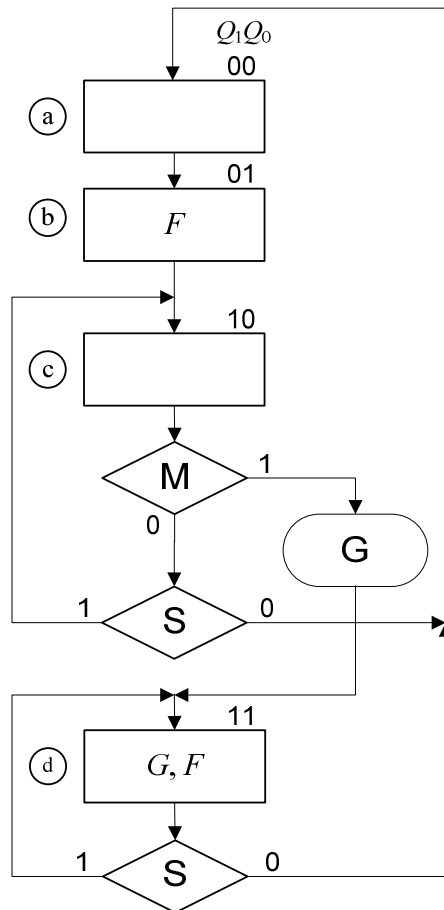
 Q_1

$$D_1 = \overline{Q_1} \cdot Q_0 + Q_1 \cdot Q_0 \cdot S + Q_1 \cdot \overline{Q_0} \cdot (S + M) = \\ = \overline{Q_1} \cdot Q_0 + Q_1 \cdot S + Q_1 \cdot \overline{Q_0} \cdot M$$



❑ Projeto de um contador de módulo variável

ASM do sistema



Mapa de *state assignment*:

Q_0	
a	b
c	d

 Q_1

D_0	Q_0	
1	0	• Se M , fica 1
M	S	• Se $\overline{M} \cdot S$, fica 0

 Q_1

- Se S , fica 1
- Se \overline{S} , fica 0

$$D_0 = \overline{Q_1} \cdot \overline{Q_0} + Q_1 \cdot \overline{Q_0} \cdot M + Q_1 \cdot Q_0 \cdot S = \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_0} \cdot M + Q_1 \cdot Q_0 \cdot S$$

Funções das variáveis de saída:

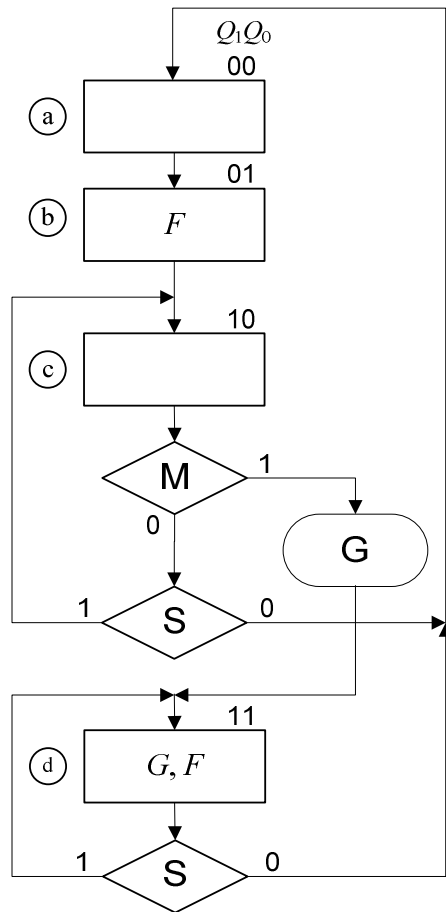
$$F = \overline{Q_1} \cdot Q_0 + Q_1 \cdot Q_0 = Q_0$$

$$G = Q_1 \cdot \overline{Q_0} \cdot M + Q_1 \cdot Q_0 = Q_1 \cdot (\overline{Q_0} \cdot M + Q_0) = Q_1 \cdot (M + Q_0)$$



❑ Projeto de um contador de módulo variável

ASM do sistema



Síntese com *flip-flops* D

Funções de estado
seguinte:

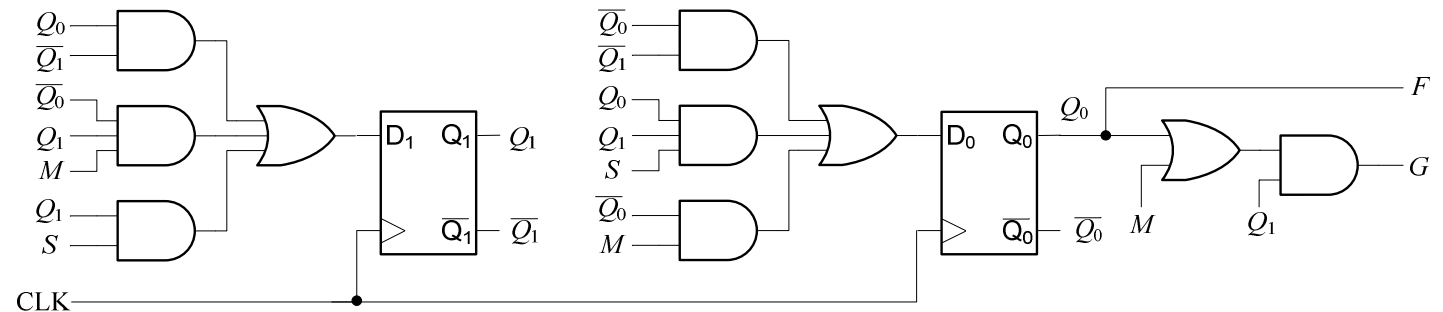
$$D_1 = \overline{Q_1} \cdot Q_0 + Q_1 \cdot S + Q_1 \cdot \overline{Q_0} \cdot M$$

$$D_0 = \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_0} \cdot M + Q_1 \cdot Q_0 \cdot S$$

Funções de saída:

$$F = Q_0$$

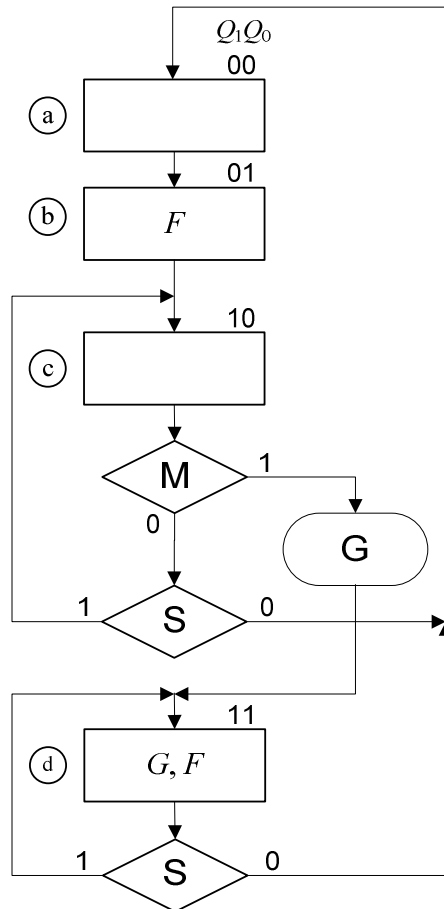
$$G = Q_1 \cdot (M + Q_0)$$





❑ Projeto de um contador de módulo variável

ASM do sistema



Mapa de *state assignment*:

Q_0	
a	b
c	d

 Q_1

Síntese com *flip-flops* JK

J_1	Q_0	
	0	1
	-	-

 Q_1

$$J_1 = Q_0$$

K_1	Q_0	
	-	-
	$\overline{M} \cdot \overline{S}$	\overline{S}

 Q_1

- Se S , fica 1
- Se $\overline{M} \cdot S$, fica 1
- Se $\overline{M} \cdot \overline{S}$, fica 0

- Se S , fica 1
- Se \overline{S} , fica 0

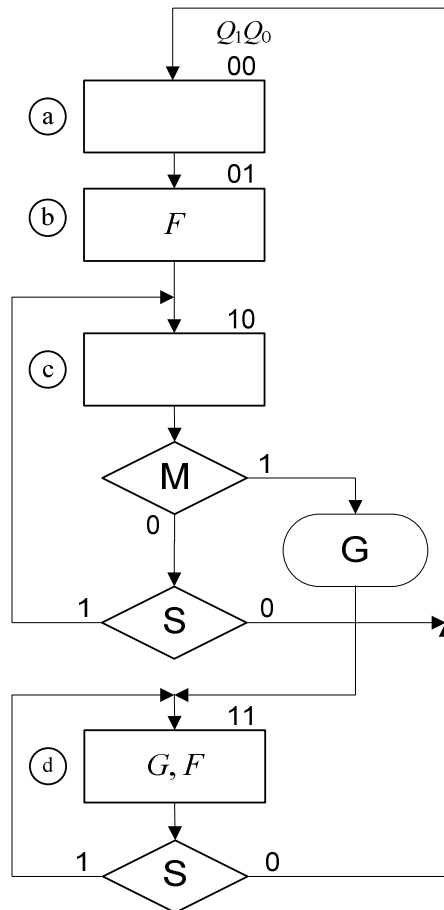
Neste estado, as transições possíveis são $1 \rightarrow 1$ ou $1 \rightarrow 0$. Pela tabela de transição, vê-se que K fica com a negação de Q , por isso, vai pelo “0”.

$$K_1 = Q_0 \cdot \overline{S} + \overline{Q_0} \cdot \overline{M} \cdot \overline{S} = \overline{S} \cdot (Q_0 + \overline{M})$$



❑ Projeto de um contador de módulo variável

ASM do sistema



Mapa de *state assignment*:

$\overline{Q_0}$	
a	b
c	d

 Q_1

Síntese com *flip-flops* JK

J_0	$\overline{Q_0}$		
1	-	Q_1	
M	-		

- Se M , fica 1
- Se $\overline{M} \cdot S$, fica 0
- Se $\overline{M} \cdot \overline{S}$, fica 0

$$J_0 = \overline{Q_1} + M$$

K_0	$\overline{Q_0}$		
-	1	Q_1	
-	\overline{S}		

- Se S , fica 1
- Se \overline{S} , fica 0

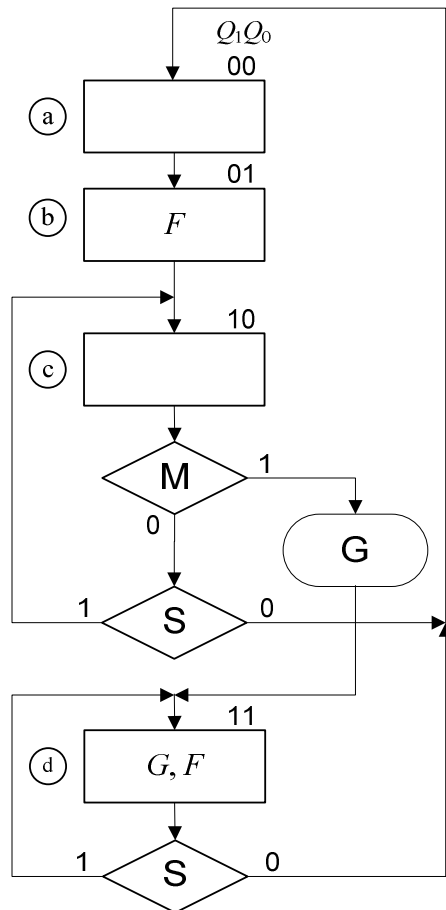
$$K_0 = \overline{Q_1} + \overline{S}$$

Q^*	Q	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0



❑ Projeto de um contador de módulo variável

ASM do sistema



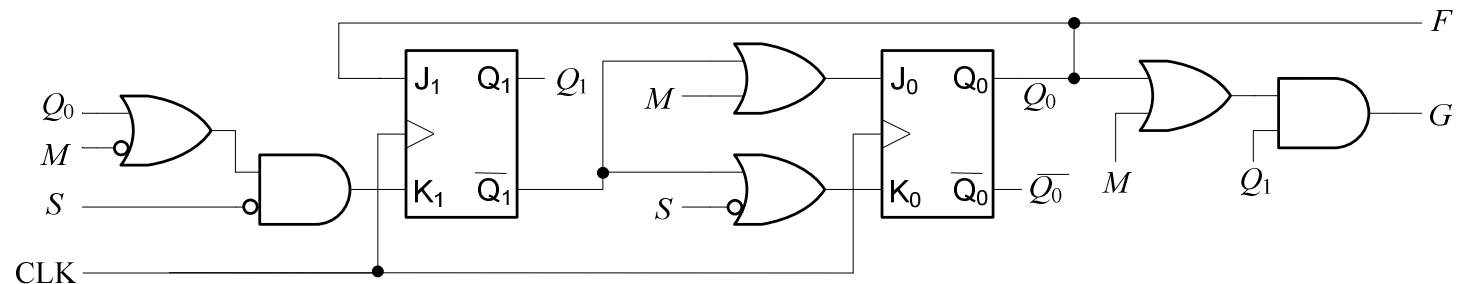
Síntese com *flip-flops* J-K

Funções de estado
seguinte:

$$\begin{aligned} J_1 &= Q_0 \\ K_1 &= \bar{S} \cdot (Q_0 + \bar{M}) \\ J_0 &= \bar{Q}_1 + M \\ K_0 &= \bar{Q}_1 + \bar{S} \end{aligned}$$

Funções de saída:

$$\begin{aligned} F &= Q_0 \\ G &= Q_1 \cdot (M + Q_0) \end{aligned}$$





□ Simulação da operação de *flip-flops* no Arduino

- É possível simular a operação de *flip-flops* no Arduino.
- Apesar de serem estruturas de *hardware* existentes em circuito integrado, é possível estabelecer em *software* o mecanismo através do qual se pode simular a operação de cada um dos tipos de *flip-flop* já abordados.
- Dar-se-á particular ênfase aos *flips-flops edge-triggered*, dado estes se adequarem melhor ao tipo de circuitos que se pretende desenvolver.
- Como a atualização do estado dos *flip-flops* depende da existência de um *clock*, este terá que ser gerado externamente.



□ “Interrupções” no Arduino

- A transição ativa de *clock* origina a resposta de dado tipo de *flip-flop*.
- Como esta transição do sinal de *clock* é um evento externo, tem que se lidar com ele recorrendo a interrupções (*interrupts*).
- Os *interrupts* são úteis quando se pretende ter uma dada resposta do programa em Arduino à ocorrência de algum estímulo externo proveniente de algum dispositivo ou do próprio utilizador.
- Se não se utilizassem *interrupts*, o programa teria continuamente que verificar a ocorrência desse estímulo, realizando *polling*.



□ “Interrupções” no Arduino

- O interrupt consiste em desviar a execução do programa, de modo a ser executada uma função específica. Quando esta termina, volta-se à execução original, no ponto onde esta tinha ficado antes de acontecer a interrupção
- Função que permite associar um *interrupt* ao Arduino, no `setup()`:

`attachInterrupt(interrupt, function, mode);`

Parâmetros: `interrupt` – nº do *interrupt* [0 (pino 2) ou 1 (pino 3)]

`function` – Nome da função a ser executada no *interrupt*

`mode` – Modo de sensibilidade ao evento físico de *interrupt*



❑ Parâmetros da função `attachInterrupt ()` no Arduino

- `interrupt`

São possíveis dois *interrupts* físicos no Arduino, 0 ou 1, atribuídos aos pinos digitais 2 ou 3, respetivamente.

Pode usar-se a macro `digitalPinToInterrupt(pin)` por forma a mapear o pino digital para o número do *interrupt* correspondente.

O *interrupt* 0 é mais prioritário do que o *interrupt* 1.



❑ Parâmetros da função `attachInterrupt ()` no Arduino

- `function`

Função a ser executada quando o *interrupt* físico ocorre. Também se designa por ISR (*Interrupt Service Routine*).

Esta função não pode ter argumentos e não deverá devolver valor algum.

Deve ser uma função o mais eficiente possível, do ponto de vista computacional, de modo a demorar o menor tempo possível a ser executada.

As variáveis globais que forem manipuladas devem ter o qualificador `volatile`, por forma a garantir que sejam guardadas em memória principal.



□ Parâmetros da função `attachInterrupt ()` no Arduino

- `mode`

Define como o *interrupt* deve ser “disparado”. São usadas quatro constantes para definir os correspondentes modos de disparo do *interrupt*:

LOW - Quando o pino está a “0”;

CHANGE - Quando o pino muda de valor (de “1” para “0” e vice-versa);

RISING - Quando o pino muda de “0” para “1”;

FALLING - Quando o pino muda de “1” para “0”.



❑ Parâmetros da função `attachInterrupt()` no Arduino

Após se estabelecer a *Interrupt Service Routine* e, em que circunstâncias esta deve ser executada, deve evocar-se a função `attachInterrupt()` na função `setup()`, para que o Arduino seja sensível a uma dada interrupção.

Exemplo:

```
void setup() {  
  
    ...  
  
    attachInterrupt(0, ISR, RISING);  
  
}
```

Para desligar uma dada interrupção, deve usar-se a função `detachInterrupt(interrupt)`.



❑ Ligar ou desligar a totalidade das interrupções no Arduino

De maneira a, de uma forma geral, se poder ligar ou desligar os *interrupts* no Arduino, deve usar-se, respetivamente:

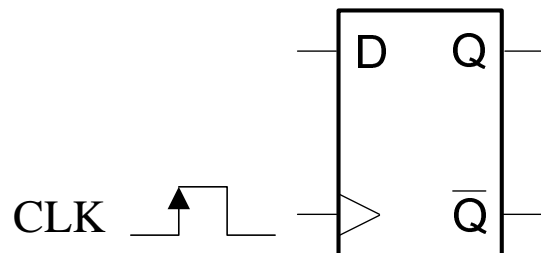
```
interrupts(); // Ligar as interrupções no Arduino  
noInterrupts(); // Desligar as interrupções no Arduino
```

`noInterrupts()` utiliza-se em certas zonas nas quais o código a executar é crítico em termos de tempo, não devendo ser interrompido.



❑ Simulação da operação de *flip-flops* no Arduino

- Tirando partido das interrupções, a operação de memorização de informação por um *flip-flop* será realizada no momento da transição dos impulsos de *clock*, tipicamente, na transição ascendente. Essa operação é realizada por uma função de interrupção (ISR).
- A ISR de simulação de um *flip-flop* D é feita pela sua definição:



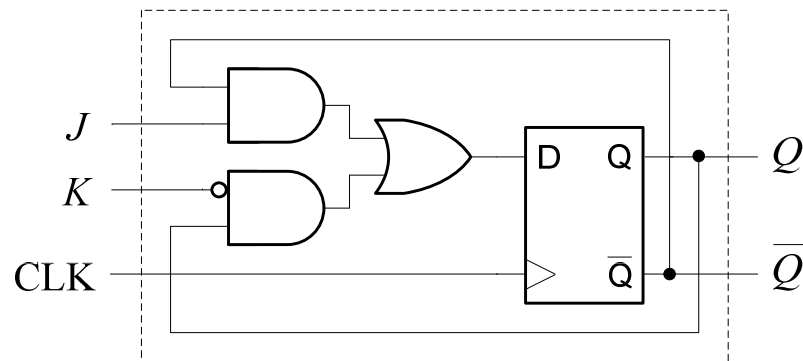
```
boolean D; // Entrada D
volatile boolean Q; // Saída Q

void flip_flop_D() {
    Q = D;
}
```



❑ Simulação da operação de *flip-flops* no Arduino

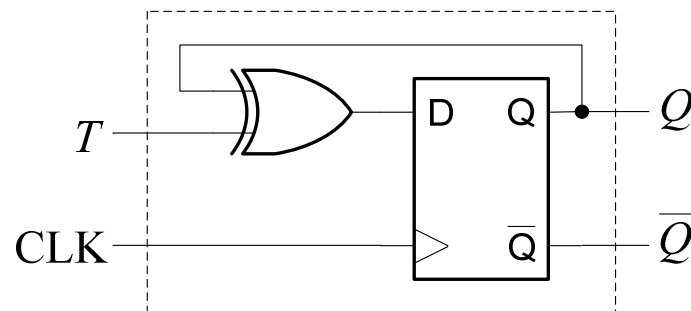
- A ISR de simulação de um *flip-flop* J-K é feita pela sua definição:



```
boolean J, K; // Entradas J e K
volatile boolean Q; // Saída Q

void flip_flop_JK(){
    Q = Q & !K | !Q & J;
}
```

- A ISR de simulação de um *flip-flop* T também é feita pela sua definição:



```
boolean T; // Entrada T
volatile boolean Q; // Saída Q

void flip_flop_T(){
    Q = Q ^ T;
}
```