

Apostila de Circuitos Digitais

Nathalie Godoi, Julia Acras, Isabella Stuart, Rafaela Leitoles
PET Computação UFPR

Tutor: Luiz Carlos Pessoa Albini

20/02/2026

Sumário

1	Números binários	4
1.1	Bytes	4
1.2	Bases numéricas	4
1.3	Sistema posicional	4
1.4	Conversão de bases	5
1.4.1	Conversão de uma base qualquer para base 10	5
1.4.2	Conversão da base 10 para uma base qualquer	6
1.4.3	Conversão da base 10 para base 2, para números racionais	6
1.4.4	Conversão de base n para base potência de n	8
1.4.5	Conversão de base potência de n para base n	8
1.5	Exercícios	9
1.6	Operações básicas com números binários	10
1.6.1	Adição	10
1.6.2	Subtração	10
1.6.3	Multiplicação	10
1.6.4	Divisão	11
1.7	Exercícios	11
1.8	Representação de números negativos em binário	12
1.8.1	Complemento de 1	12
1.8.2	Complemento de 2	12
1.9	Exercícios	13
1.10	Representação de números reais	14
1.10.1	Notação científica $\times$ Ponto flutuante	14
1.10.2	IEEE 754	14
1.10.3	Expoente com viés (<i>bias</i>)	15
1.10.4	Precisão simples e dupla	15
1.10.5	Casos especiais	15
1.10.6	Exemplo de conversão para IEEE 754	16
1.11	Exercícios	16
2	Terminologia e Símbolos	17

3	Tabela Verdade	17
3.1	Introdução	17
3.2	Construção da Tabela Verdade	17
3.3	Operações Lógicas	17
3.4	Exercícios	19
4	Álgebra de Boole	19
4.1	Operações Fundamentais	20
4.2	Representação na forma de Diagrama de Venn	20
5	Expressões, Axiomas e Simplificação	22
5.1	Versão OR (Soma lógica)	22
5.2	Versão AND (Produto Lógico)	23
6	Demonstração das Propriedades	23
6.1	Propriedade 2: Elemento Nulo	23
6.2	Propriedade 8: Distributiva	23
6.3	Propriedade 9: Absorção 1	24
6.4	Propriedade 11: Consensus	24
6.5	Exercícios	25
7	MINTERMOS E MAXTERMS	25
7.1	Mintermos	25
7.2	Maxtermo	26
8	Mapa de Karnaugh	27
8.1	Construção do Mapa de Karnaugh	28
8.1.1	Mapa com 2 variáveis	28
8.1.2	Mapa com 3 variáveis	28
8.1.3	Mapa com 4 variáveis	29
8.2	Preenchendo o mapa e simplificando as expressões	30
8.2.1	Preenchendo o mapa	30
8.2.2	Simplificando a expressão	30
8.3	Exercícios	31
9	Somador	32
9.1	Meio-somador	32
9.2	Somador completo	33
9.3	Somador de n bits	34
9.4	Exercícios	34
10	Subtrator	35
10.1	Meio-subtrator	35
10.2	Subtrator completo	36
10.3	Subtrator de n bits	37
10.4	Implementação alternativa	37
10.5	Exercícios	38

11 Multiplicador	38
11.1 Multiplicador por tabela-verdade	39
11.2 Multiplicador combinacional	40
11.3 Conclusão	41
11.4 Exercícios	42
12 Decodificadores e Codificadores	42
12.1 Decodificador	42
12.2 Codificador	44
12.3 Associação entre Codificadores e Decodificadores	45
12.4 Exercícios	45
13 Circuito Combinacional	45
14 Circuitos Sequenciais	46
15 Multiplexadores e Demultiplexadores	46
15.1 MUX	46
15.1.1 MUX 4x1	47
15.2 DEMUX	47
15.2.1 DEMUX 1x4	47
15.3 Multiplexador x Demultiplexador	48
16 Unidade Lógica Aritmética (ULA)	49
17 Formas de onda	51
17.1 Tipos de Onda	52
18 Flip-Flop	53
18.1 Flip-Flop SR	53
18.2 Flip-Flop D	53
18.3 Flip-Flop JK	53
19 Máquina de estados	53
19.1 Exemplo Simplificado	54
19.2 Máquina de Moore	58
19.2.1 Exemplo: Máquina de Estado de um Semáforo	58
19.3 Máquina de Mealy	61
19.3.1 Exemplo: Detector de Sequência	61
19.4 Revisão	64
19.5 Exercícios Propostos:	65
20 Referências	67

1 Números binários

1.1 Bytes

- Cada dígito de um número armazenado é chamado de **bit** (*binary digit*).
- Um conjunto de 4 bits é chamado de **nibble**.

Ele é útil quando manipulamos valores em hexadecimal.

- Um conjunto de 8 bits é chamado de **byte**.

Na maioria das CPUs, é a menor unidade utilizada para operações.

1.2 Bases numéricas

No cotidiano, utilizamos números compostos por algarismos de 0 a 9. Como isso equivale a um total de 10 algarismos, chamamos de **base decimal** (ou base 10).

- Você também conhece a base 2, também chamada de base binária, porque possui 2 algarismos válidos: 0 e 1.

Vale ressaltar que isso se aplica a **qualquer valor**. Logo, é possível utilizar um sistema numérico de base 7, por exemplo. Para isso, basta serem utilizados somente 7 algarismos.

Observação

Devido ao sistema numérico que utilizamos no dia a dia, a escolha mais intuitiva seria usar algarismos de 0 a 6 para representar essa base. Mas, é importante destacar que **quaisquer símbolos** poderiam ser utilizados para representar esses valores.

Bases numéricas mais utilizadas:

- $b_2 = \text{binária}$ (de 0 a 1)
- $b_4 = \text{nimble}$ (de 0 a 3)
- $b_8 = \text{octal}$ (de 0 a 7)
- $b_{16} = \text{hexadecimal}$ (de 0 a 15)

Para a representação dos valores de 10 a 15 na base hexadecimal, utilizamos letras de *A* a *F*. Dessa forma,

- $b_{16} \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

1.3 Sistema posicional

Considerando um número qualquer na base decimal, como 1325. Note que ele é composto da seguinte maneira:

$$1325 = 1000 + 300 + 20 + 5 = 1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$$

- Conclusão: a cada casa avançada para a esquerda, é **multiplicado por 10** o peso do algarismo na soma total.

Isso é o que chamamos de **sistema posicional**: cada algarismo está numa posição, que vai de 0 até n, da direita para a esquerda. Quanto mais à esquerda, **mais significativo** ele é, isto é, maior a sua influência sobre a soma total.

No caso do 1325: as posições vão de 0 a 3, da seguinte maneira: $\begin{smallmatrix} 1325 \\ 3210 \end{smallmatrix}$

- Algarismo mais significativo = 1
- Algarismo menos significativo = 5

Note que os valores das posições de cada algarismo coincidem com os expoentes de 10:

- $1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$

Isso também se aplica a números racionais. Considerando o número 124,36:

$$124,36 = 100 + 20 + 4 + 0,3 + 0,06$$

$$124,36 = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + 6 \times 10^{-2}$$

De maneira geral, isso se aplica a qualquer base numérica. O que varia entre as bases é o **valor que multiplica o peso** de cada algarismo, à medida que se avança para a esquerda.

- Esse valor sempre será equivalente ao número de algarismos da base numérica utilizada.

Exemplos:

$$- 362_7 = 3 \times 7^2 + 6 \times 7^1 + 2 \times 7^0$$

$$- 1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

1.4 Conversão de bases

1.4.1 Conversão de uma base qualquer para base 10

Agora que conhecemos o sistema posicional, temos um método direto para converter números de qualquer base para a base decimal ($b_x \rightarrow b_{10}$). Para isso, basta resolver o polinômio.

Exemplos:

- Convertendo 362 da base 7 para a base decimal ($b_7 \rightarrow b_{10}$):

$$362_7 = 3 \times 7^2 + 6 \times 7^1 + 2 \times 7^0$$

$$362_7 = 147 + 42 + 2$$

$$362_7 = 191_{10}$$

Resposta: 191.

- Convertendo 1010 da base binária para a base decimal ($b_2 \rightarrow b_{10}$):

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$1010_2 = 8 + 0 + 2 + 0$$

$$1010_2 = 10_{10}$$

Resposta: 10.

1.4.2 Conversão da base 10 para uma base qualquer

Como exemplo, vamos converter 29 para binário. Temos que:

$$29_{10} = 16 + 8 + 4 + 1$$

$$29_{10} = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$$

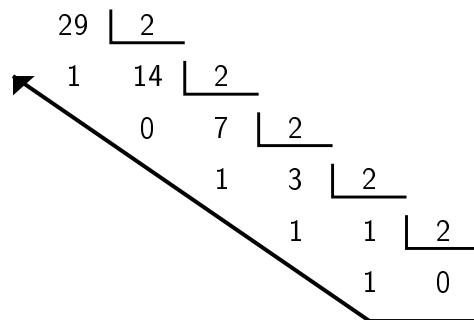
$$29_{10} = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$29_{10} = 11101_2$$

Ou seja, o raciocínio consiste em descobrir quais potências de 2 somadas resultam no valor desejado.

Na prática, a maneira mais fácil de descobri-las é dividindo o valor sucessivamente por 2, enquanto for possível, e unindo o quociente aos restos, nessa ordem.

- Resposta: $29_{10} = 011101_2$ ou 11101_2 .



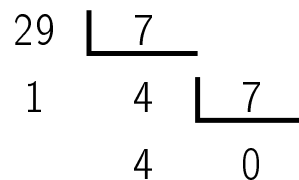
De maneira geral, esse método funciona para converter números da **base decimal para qualquer base**: para uma base x , divida-o por x enquanto for possível, e o resultado será a união do quociente aos restos das divisões.

Vamos converter o número 29 para a base 7:

$$29_{10} = 28 + 1$$

$$29_{10} = 4 \times 7^1 + 1 \times 7^0$$

$$29_{10} = 41_7$$



1.4.3 Conversão da base 10 para base 2, para números racionais

Voltemos à representação de um número racional, com base no sistema posicional:

$$20,375_{10} = 2 \times 10^1 + 0 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$$

Convertendo $20,375_{10}$ para binário:

$$20,375 = 20 + 0,375$$

Como visto anteriormente,

$$20 = 16 + 4 = 1 \times 2^4 + 1 \times 2^2$$

$$20 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$20_{10} = 10100_2$$

Agora, convertendo 0,375:

- Sendo F a parte fracionária de um racional, tal que $0 < F < 1$.

Nesse caso, teremos $F = 0,375$.

- A representação de F na base 2 será:

$$F = b_1 \times 2^{-1} + b_2 \times 2^{-2} + \dots + b_n \times 2^{-n}$$

Onde $b_1, b_2, \dots, b_n$ são os dígitos em binário que queremos encontrar.

- Se multiplicarmos ambos os lados dessa equação por 2, teremos:

$$2 \times F = 2 \times [b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + \dots + b_n \times 2^{-n}]$$

$$2 \times F = b_1 \times 2^0 + b_2 \times 2^{-1} + b_3 \times 2^{-2} + \dots + b_n \times 2^{-(n-1)}$$

$$2 \times F = b_1 + [b_2 \times 2^{-1} + b_3 \times 2^{-2} + \dots + b_n \times 2^{-(n-1)}]$$

$$2 \times F = b_1 + [b_2 \times \frac{1}{2} + b_3 \times \frac{1}{4} + \dots + b_n \times \frac{1}{2^{n-1}}]$$

Assim, como $b_x \in \{0, 1\}$, para todo x , a expressão $[b_2 \times \frac{1}{2} + b_3 \times \frac{1}{4} + \dots + b_n \times \frac{1}{2^{n-1}}]$ será estritamente menor que 1.

- Por isso, podemos concluir que $2 \times F$ equivale à soma entre uma parte inteira (nesse caso, b_1) e uma parte fracionária, e a parte inteira de $2 \times F$ será exatamente o dígito em binário que queremos encontrar.
- Logo, para converter um valor racional para binário, basta que esse processo seja repetido: multiplicar F por 2 e subtrair sua parte inteira, enquanto $F > 0$.
- Dado esse raciocínio, temos o seguinte algoritmo:

```
k := 1;
F := r10; (* parte racional em base 10 *)

do
    F := F * 2;
    dk := parte_inteira(F);
    F := F - dk;
    k := k + 1;
while (F > 0);
```

Exemplo de execução para $F = 0,375$:

```
F = 0,375
1ª iteração (k = 1): F * 2 = 0,75 -> d1 = 0, novo F = 0,75
2ª iteração (k = 2): F * 2 = 1,50 -> d2 = 1, novo F = 0,50
3ª iteração (k = 3): F * 2 = 1,00 -> d3 = 1, novo F = 0,00
FIM!
Resultado: 0,011
```

Por fim, temos:

$$20,375_{10} = 20 + 0,375$$

$$20,375_{10} = 10100_2 + 0,011_2$$

$$20,375_{10} = 10100,011_2$$

1.4.4 Conversão de base n para base potência de n

Quando uma base é potência de outra, existe uma relação direta entre grupos de dígitos na base menor e dígitos únicos na base maior. Utilizando b_2 e b_4 como exemplo:

Binário	Nimble
00 ₂	0 ₄
01 ₂	1 ₄
10 ₂	2 ₄
11 ₂	3 ₄

Ou seja, para converter valores de uma base a outra, basta dividir o número de **menor** base em grupos e analisar qual é o dígito correspondente na base maior.

- Exemplo 1: convertendo 110100111_2 para b_4

$$\begin{array}{ccccccccc} & 1 & & 2 & & 2 & & 1 & & 3 \\ & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} \\ 0 & 1 & & 1 & 0 & & 1 & 0 & & 1 & 1 \end{array}$$

$$\text{Logo, } 110100111_2 = 12213_4.$$

Aqui, como $4 = 2^2$, o expoente é 2 e o número de base menor (nesse caso, b_2) é dividido em grupos de 2 dígitos.

- Exemplo 2: convertendo 110100111_2 para b_8

$$\begin{array}{ccccccc} & 6 & & 4 & & 7 \\ & \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} \\ 1 & 1 & 0 & & 1 & 0 & 0 & & 1 & 1 & 1 \end{array}$$

$$\text{Logo, } 110100111_2 = 647_8.$$

Mesmo raciocínio: como $8 = 2^3$, o valor de base 2 é dividido em grupos de 3 dígitos.

Observação

Sempre comece o agrupamento da direita para a esquerda. Se faltarem dígitos para completar algum grupo, preencha com zeros à esquerda. Note que isso foi necessário para a resolução do exemplo 1.

1.4.5 Conversão de base potência de n para base n

Trata-se do processo inverso: basta selecionar cada dígito do número de **maior** base e substituí-lo pelo valor correspondente na base menor.

- Exemplo: convertendo 52543_8 para b_2

$$52543_8 = \begin{array}{ccccc} & 101 & 010 & 101 & 100 & 011 \\ & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} \\ & 5 & 2 & 5 & 4 & 3 \end{array} = 101010101100011_2$$

1.5 Exercícios

1. Converta:

- (a) 10110_2 para a base 10.
- (b) 213_4 para a base 10.
- (c) 57_8 para a base 10.
- (d) $1F_{16}$ para a base 10.
- (e) 402_5 para a base 10.
- (f) 45 para a base 2.
- (g) 89 para a base 5.
- (h) 120 para a base 8.
- (i) 200 para a base 16.
- (j) 33 para a base 3.
- (k) 0,25 para a base 2.
- (l) 0,625 para a base 2.
- (m) 2,5 para a base 2.
- (n) 7,75 para a base 2.
- (o) 0,375 para a base 2.
- (p) 101110_2 para a base 4.
- (q) 110101111_2 para a base 8.
- (r) 10110110_2 para a base 16.
- (s) 12201_3 para a base 9.
- (t) 100101_2 para a base 8.
- (u) 32_4 para a base 2.
- (v) 70_8 para a base 2.
- (w) $A5_{16}$ para a base 2.
- (x) 47_9 para a base 3.
- (y) C_{16} para a base 2.

1.6.4 Divisão

É idêntica à divisão na base 10.

- Exemplo:

$$\begin{array}{r}
 1110 \quad | \quad 10 \\
 \hline
 -10 \quad | \\
 \hline
 011 \quad | \\
 \hline
 -10 \quad | \\
 \hline
 010 \quad | \\
 \hline
 -10 \quad | \\
 \hline
 00
 \end{array}$$

1.7 Exercícios

1. Calcule:

- | | |
|------------------------|------------------------------|
| (a) $00011 + 00101$ | (m) 111×101 |
| (b) $1110 + 011$ | (n) 1110×011 |
| (c) $00101 + 10101$ | (o) 100100×10101 |
| (d) $111 + 111$ | (p) 1011×100 |
| (e) $1001 + 1010$ | (q) 11001×1010 |
| (f) $10101 + 1010110$ | (r) 1110101×1010110 |
| (g) $111 - 101$ | (s) $110 \div 10$ |
| (h) $1110 - 011$ | (t) $1111 \div 101$ |
| (i) $100100 - 10101$ | (u) $1000 \div 10$ |
| (j) $1011 - 100$ | (v) $10101 \div 111$ |
| (k) $11001 - 1010$ | (w) $11001 \div 101$ |
| (l) $110101 - 1010110$ | (x) $10010 \div 11$ |

1.8 Representação de números negativos em binário

Considerando somente valores positivos, o maior número armazenável em um computador de n bits é igual a $2^n - 1$.

Mas, aplicações práticas exigiam também a representação de valores negativos. A solução inicial foi reservar o bit mais significativo para indicar o sinal: positivo = 0 e negativo = 1. Isso é chamado de **bit de sinal** ou **sinal magnitude**.

Seguindo esse raciocínio, metade dessa faixa de 2^n elementos seria negativa e metade positiva.

Portanto, em um computador de 8 bits, considerando somente números positivos, teríamos $x \in [0, 255]$, enquanto, com negativos, teríamos $x \in [-127, +127]$.

1.8.1 Complemento de 1

Foi um método criado como uma tentativa de aplicação do *bit* de sinal. Não é utilizado atualmente, por possuir algumas inconsistências.

“Para representar um valor negativo, basta inverter todos os bits da sua representação positiva.”

Exemplo: $8_{10} = 00001000_2$ e $-8_{10} = 11110111_2$. O seu maior problema é o fato de que, seguindo essa lógica, existem representações para $+0$ e -0 , que correspondem a 00000000 e 11111111, respectivamente.

Isso exigiria uma maior complexidade por parte dos processadores e dos sistemas operacionais, para saber como gerenciar esses sinais (por exemplo: a comparação `if (x == 0)` deveria considerar os dois zeros).

Além disso, em operações aritméticas utilizando o complemento de 1, o *overflow* deve ser considerado, o que também aumentaria a complexidade dos sistemas.

Exemplo: $A - B = A + (-B) + \text{overflow}$

1.8.2 Complemento de 2

Nesse contexto, esse método foi criado com o intuito de resolver as inconsistências e complexidades do complemento de 1.

“Para representar um valor negativo, basta inverter todos os bits da sua representação positiva e somar 1.”

Por que isso resolve o problema?

Estudamos que, em complemento de 1, tínhamos $+0 = 00000000$ e $-0 = 11111111$. Agora, vamos corrigir a representação calculando $11111111 + 1$.

Tecnicamente, o resultado deveria ser 100000000, mas, por conta do *overflow*, o bit mais significativo é descartado, o que resulta em 00000000, equivalente a $+0$.

Assim, foi possível eliminar a representação do -0 .

Além disso, fazendo os devidos ajustes nos negativos, percebe-se que é possível a representação de um número a mais: 10000000.

Se estivéssemos trabalhando com 9 bits, 010000000 seria igual a +128. No entanto, como não é o caso, esse valor deve ser negativo, dado seu bit de sinal.

Considerando que -127 equivale a 10000001, temos:

$$-127 - 1 = -128$$

$$-127 + (-1) = -128$$

$$10000001 + 11111111 = 110000000$$

Por conta do overflow, obtemos 10000000 como resultado.

Sendo assim, em complemento de 2, temos $x \in [-128, +127]$. **Esse é o sistema utilizado por todos os computadores atualmente.**

Observação

Antes de converter um valor **negativo** para a base decimal, deve-se subtrair 1, corrigindo a soma que foi feita anteriormente. Depois, basta fazer a conversão normal, resolvendo o polinômio obtido com base no sistema posicional, como visto anteriormente.

Resumindo:

- Convertendo positivo para negativo: inverte os dígitos e soma 1;
- Convertendo negativo para positivo: subtrai 1 e inverte.

1.9 Exercícios

1. Calcule, em binário, considerando valores de 8 bits em complemento de 2:

(a) $9 + (-23)$

(f) $(-32) + (-45)$

(b) $17 + (-46)$

(g) $(-32) + 45$

(c) $64 + 72$

(h) $50 + (-78)$

(d) $64 + (-72)$

(i) $50 + 78$

(e) $(-127) + (-1)$

(j) $127 + (-1)$

1.10 Representação de números reais

Os sistemas de representação vistos até agora (inteiros com sinal, complemento de 2) são suficientes para números inteiros, mas e quando precisamos armazenar números muito grandes, muito pequenos ou com casas decimais?

Para isso, na base 10, usamos o que chamamos de **notação científica**.

Exemplos:

- $9.461.000.000.000.000 = 9,461 \times 10^{15}$
- $0,000000000053 = 5,3 \times 10^{-11}$

1.10.1 Notação científica × Ponto flutuante

Na base decimal, um número em notação científica é representado como $\pm M \times 10^{\pm E}$, onde:

- M = **mantissa** (ou significando), um valor entre 1 e 9;
- E = **expoente**, indica quantas casas a vírgula deve deslocar.

Na base binária, a representação funciona da mesma forma. A única diferença está na base do expoente, que é 2, em vez de 10: $\pm M \times 2^{\pm E}$.

Operações para as bases decimal e binária são feitas exatamente da mesma forma.

Exemplos:

- $2 \times 10^2 + 7 \times 10^1 = 2 \times 10^2 + 0,7 \times 10^2 = 2,7 \times 10^2$
- $1,101 \times 2^3 - 1,1 \times 2^2 = 1,101 \times 2^3 - 0,11 \times 2^3 = 0,111 \times 2^3 = 1,11 \times 2^2$
- $(2 \times 10^2) \times (3 \times 10^1) = (2 \times 3) \times 10^{2+1} = 6 \times 10^3$
- $(1,11 \times 2^3) \div (1,0 \times 2^1) = (1,11 \div 1,0) \times 2^{3-1} = 1,11 \times 2^2$

Mas, era necessária uma forma de representar a notação científica na memória de um computador. Diante disso, foi criada a representação que chamamos de **ponto flutuante**. Aqui, vamos estudar um dos vários sistemas de ponto flutuante que existem: **IEEE 754**, utilizado em grande parte dos processadores comerciais atualmente.

1.10.2 IEEE 754

Um número em ponto flutuante no padrão IEEE 754 é composto por três campos: sinal, expoente e mantissa.

Assim, temos:

- Sinal: 0 = positivo e 1 = negativo;
- Expoente: armazenado com viés (ou *bias*) para permitir expoentes negativos sem usar complemento de 2 (veremos sobre isso mais adiante).
- Mantissa: armazena apenas a parte fracionária, pois o bit inteiro é implícito (1,xxxx...).

Assim como temos a convenção de manter a parte inteira entre 1 e 9 no sistema de notação científica, em ponto flutuante, definimos a parte inteira como sendo sempre igual a 1. Dizemos que um número está **normalizado** quando segue essas convenções. Dessa forma, é possível simplificar o sistema, armazenando na mantissa somente a parte fracionária do valor. Chamamos a parte inteira de **bit implícito**.

1.10.3 Expoente com viés (*bias*)

O viés é o número que se localiza no meio da faixa de valores que podem ser representados no expoente (de 0 a $2^n - 1$, sendo n o número de *bits*).

Assim, o valor do expoente armazenado equivale ao resultado de **expoente + viés**.

Exemplo, considerando viés = 127:

- Expoente real = 7 $\rightarrow$ expoente armazenado = $7 + 127 = 134$.
- Expoente real = -3 $\rightarrow$ expoente armazenado = $-3 + 127 = 124$.

Utilizamos esse sistema porque ele permite a representação de valores negativos no expoente, sem utilizar complemento de 2, que aqui seria desvantajoso, uma vez que o processador precisaria de lógica extra para interpretar o sinal do expoente.

1.10.4 Precisão simples e dupla

Existem dois formatos de representação de flutuantes: precisão simples e precisão dupla. A seguir, observe essa tabela com o número de *bits* destinado a cada campo:

Formato	Sinal	Expoente	Mantissa	Total	Viés
Precisão simples (float)	1	8	23	32	127
Precisão dupla (double)	1	11	52	64	1023

Exemplo, em precisão simples:

$\overbrace{1}^{\text{sinal}} \overbrace{10000010}^{\text{expoente}} \overbrace{1010000000000000000000}^{\text{mantissa}}$

1.10.5 Casos especiais

Nem todo número binário no sistema IEEE 754 representa um número real comum. Isso acontece em alguns casos específicos:

- Underflow ou overflow: quando o valor é pequeno demais ou grande demais para ser armazenado de forma convencional;
- Quando a operação matemática é inválida.

Nesses casos, temos, em precisão simples:

Contexto	Valor representado	Expoente	Bit implícito	Mantissa
Normalizado	$1, \dots \times 2^{(E-127)}$	1 a 254	1	qualquer valor
Zero	0	0	0	0
Denormalizado	$0, \dots \times 2^{(-126)}$	0	0	$\neq 0$
Infinito	Infinito	255	-	0
NaN	Inválido	255	-	$\neq 0$

1.10.6 Exemplo de conversão para IEEE 754

Vamos converter $20,375_{10}$ para IEEE 754, com precisão simples.

1. Conversão para binário: como visto anteriormente, $20,375_{10} \rightarrow 10100,011_2$
2. Normalização: $10100,011_2 \rightarrow 1,0100011_2 \times 2^4$
3. Expoente: expoente real = 4 $\rightarrow$ expoente armazenado = $4 + 127 = 131_{10} = 10000011_2$
4. Mantissa: *bits* após a vírgula = 0100011. Completando com zeros à direita, até 23 bits, temos: 01000110000000000000000
5. Sinal: positivo = 0

Resultado:

$\begin{array}{ccccccc} \textit{sinal} & & \textit{expoente} & & & & \textit{mantissa} \\ \hline 0 & 10000011 & 01000110000000000000000 \end{array}$

Observação

Na mantissa, quando necessário, completamos o valor armazenado com zeros à direita, porque eles não influenciam no resultado. Pense na base decimal: $0,5_{10} = 0,500_{10}$, por exemplo. Lembre-se de que estamos tratando da **parte fracionária** do valor.

1.11 Exercícios

1. Converta para o formato IEEE 754 (em precisão simples):
 - (a) $18,75_{10}$
 - (b) $-5,25_{10}$
 - (c) $0,1_{10}$
 - (d) $-0,375_{10}$
 - (e) $42,25_{10}$
 - (f) $-0,1_{10}$
2. Converta os seguintes valores IEEE 754 (32 bits) para a base decimal:
 - (a) 0 10000011 101100000000000000000000
 - (b) 1 10000000 010000000000000000000000
 - (c) 0 01111111 000000000000000000000000
 - (d) 1 01111100 110000000000000000000000
 - (e) 0 10000010 110100000000000000000000
 - (f) 1 01111101 101000000000000000000000

2 Terminologia e Símbolos

Nesta seção, apresentamos os principais símbolos usados nas operações booleanas para garantir o entendimento dos conceitos apresentados no texto.

Símbolo	Operação / Descrição
$+$	OR (Disjunção Lógica)
$\cdot$	AND (Conjunção Lógica)
$\bar{X}$	NOT (Negação de X)
$\sum$	Soma de mintermos (forma canônica disjuntiva)
$\prod$	Produto de maxtermos (forma canônica conjuntiva)

3 Tabela Verdade

3.1 Introdução

A tabela verdade é um quadro que representa todas as possíveis combinações lógicas entre variáveis booleanas. Ela mostra como os valores de entrada (0 ou 1) influenciam a saída de um circuito.

Para construí-la, divide-se a tabela em duas partes: entradas (ou variáveis) e saídas (ou combinações resultantes). A cada nova variável adicionada, o número de combinações dobra, pois cada variável pode assumir dois valores: 0 ou 1.

Assim, para n variáveis, o número total de linhas na tabela será 2^n . Por exemplo, com 7 variáveis, temos $2^7 = 128$ combinações possíveis.

Observação: Neste material, utilizamos $\bar{A}$ para representar a negação da variável A .

3.2 Construção da Tabela Verdade

Na Tabela Verdade, as variáveis de entrada são colocadas do lado esquerdo, enquanto a função resultante que representa o circuito aparece à direita. Por exemplo, na função f definida a partir da variável A , colocamos na coluna da esquerda A e, na direita, o resultado, que depende do valor lógico de A . Assim, teremos:

Variável de entrada (A)	→	<table><tr><td>A</td><td>$\overline{A}$</td></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table>	A	$\overline{A}$	1	0	0	1	←	Saída ($\overline{A}$)
A	$\overline{A}$									
1	0									
0	1									
$f = \overline{A}$										

3.3 Operações Lógicas

- **Conjunção (AND) :** $A \cdot B$ - verdadeiro somente se A e B forem verdadeiros.
- **Disjunção (OR):** $A + B$ - falso somente se A e B forem falsos.
- **Negação (NOT):** $\bar{A}$ - inverte o valor lógico de A .
- **Implicação:** $A \rightarrow B$ - falso apenas se A for verdadeiro e B falso.
- **Bicondicional:** $A \leftrightarrow B$ - verdadeiro se A e B tiverem o mesmo valor lógico.

Exemplo de Negação (NOT):

$$f = \overline{A}$$

A	$\overline{A}$
1	0
0	1

$\overline{A}$ tem sempre o valor lógico oposto de A .

Exemplo de Conjunção (AND):

$$s = A \cdot B$$

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

A função só é verdadeira quando A e B são ambos 1.

Exemplo de Disjunção (OR):

$$g = A + B$$

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

A função só é verdadeira quando A ou B são 1.

Exemplo Composto: Disjunção, Conjunção e Negação:

Considere a função:

$$h = \overline{A \cdot B} \cdot C$$

Para construir a tabela verdade dessa função, primeiro calculamos $A \cdot B$, depois aplicamos a negação, e, por fim, combinamos esse resultado com C .

A	B	C	$A \cdot B$	$\overline{A \cdot B}$	$\overline{A \cdot B} \cdot C$
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	0	0

Note que a função h só é verdadeira quando C e $\overline{A \cdot B}$ são ambas verdadeiras.

Essa tabela permite acompanhar passo a passo o cálculo da função h para todas as combinações possíveis das variáveis A , B e C .

Observação: números de linhas da Tabela Verdade

A cada nova variável adicionada à tabela, o número de linhas dessa tabela verdade dobra. Dá para perceber isso comparando, por exemplo, os exemplos 1 e 2 - onde foram adicionadas 2 novas possibilidades - ou então do exemplo 3 para o 4, que aumentaram 4 possibilidades.

3.4 Exercícios

1. Construa a Tabela Verdade das funções abaixo e avalie o seu valor lógico:

1. $(A \cdot B) + (B \cdot A)$

2. $(\overline{A} + B) + (\overline{B} \cdot A)$

3. $(\overline{A + B}) \cdot (A + \overline{B})$

4. $(A \cdot (B \cdot C))$

5. $(A \cdot B) + (A \cdot C) + (B \cdot C)$

6. $(A + B) \cdot \overline{C}$

7. $(A + B) + \overline{A}$

8. $(A \cdot B) + C$

9. $(A \cdot B) + (B \cdot C)$

10. $(\overline{A + B}) \cdot A$

11. $(A \cdot B) + (A + B)$

12. $(A \cdot B) + (\overline{A} \cdot C) + (\overline{B} \cdot A)$

2. Mostre se as igualdades abaixo são verdadeiras:

1. $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

2. $(A \cdot B) + (A + C) = A \cdot (B + C)$

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

4 Álgebra de Boole

A Álgebra de Boole utiliza apenas dois valores:

- Verdadeiro: 1
- Falso: 0

4.1 Operações Fundamentais

As três operações fundamentais são:

1. **AND** (E lógico)

A saída da operação **AND** é verdadeira somente quando todas as entradas são verdadeiras, ou seja, basta que apenas uma entrada seja falsa (0) para tornar todo o circuito falso.

2. **OR** (OU lógico)

A saída é verdadeira se ao menos uma das entradas é verdadeira. Logo, a única situação em que o circuito tem seu valor lógico falso é quando todas as entradas são 0.

3. **NOT** (Negação)

A saída é o inverso da entrada. Se a entrada for verdadeira (1), a saída será falsa (0) e vice-versa.

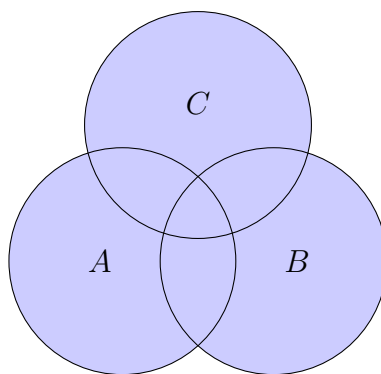
Observação: Relação com Diagrama de Venn

As operações booleanas também podem ser representadas por diagramas de Venn:

- **AND** – Interseção entre os conjuntos A e B
- **OR** – União dos conjuntos A e B
- **NOT** – Complemento do conjunto A

4.2 Representação na forma de Diagrama de Venn

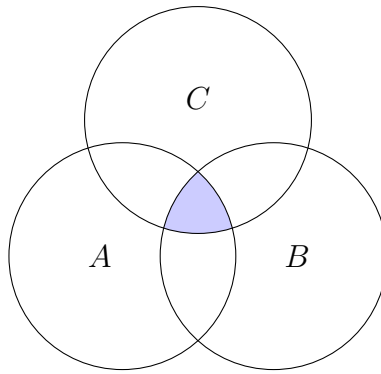
Diagrama de Venn da expressão $f = A + B + C$



Região destacada: $A \cup B \cup C$

Explicação: Nesse exemplo, para que a função f acima tenha valor lógico 1, é preciso que ao menos **UMA** das variáveis seja verdadeira.

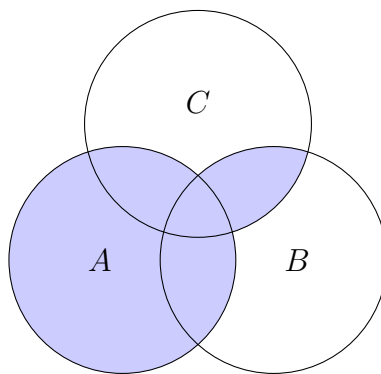
Diagrama de Venn da função: $f = A \cdot B \cdot C$



Região destacada: $A \cap B \cap C$

Explicação: Diferente do exemplo anterior, para que essa função f seja verdadeira, é preciso que **TODAS** as variáveis tenham valor lógico 1.

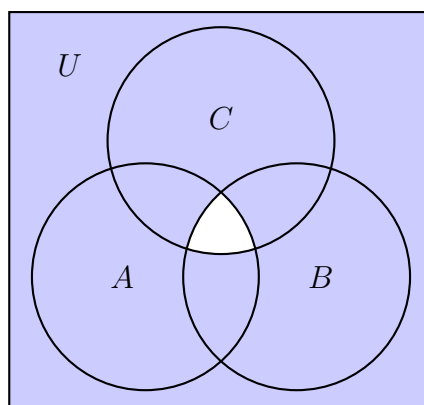
Diagrama de Venn da expressão: $f = A + (B \cdot C)$



Região destacada: $A \cup (B \cap C)$

Explicação:

Diagrama de Venn da expressão: $g = \overline{A \cdot B \cdot C}$



Região destacada: $\overline{A \cap B \cap C}$

Explicação: Esse exemplo é equivalente ao complemento da interseção de todas as variáveis. Por exemplo, se uma função f é determinada por essa interseção, e como visto anteriormente essa expressão só é verdadeira quando todas as variáveis são simultaneamente 1, temos que nossa função g é verdadeira sempre que f for falsa, ou seja, quando ao menos uma das variáveis não for 1.

5 Expressões, Axiomas e Simplificação

Sendo X, Y e Z variáveis que assumem os valores lógicos 0 e 1, e utilizando os símbolos:

- $+$ para disjunção (OU lógico)
- $\cdot$ para conjunção (E lógico)
- $\overline{X}$ para negação de X

As propriedades fundamentais da Álgebra de Boole serão representadas a seguir, separadas pelas operações OR e AND, para facilitar a compreensão.

5.1 Versão OR (Soma lógica)

Propriedade (OR)	Expressão
1 - Identidade	$X + 0 = X$
2 - Elemento Nulo	$X + 1 = 1$
3 - Idempotência	$X + X = X$
4 - Complemento	$X + \overline{X} = 1$
5 - Involução	$\overline{\overline{X}} = X$
6 - Comutativa	$X + Y = Y + X$
7 - Associativa	$(X + Y) + Z = X + (Y + Z)$
8 - Distributiva	$X + (Y \cdot Z) = (X + Y)(X + Z)$
9 - Absorção 1	$X + X \cdot Y = X$
10 - Absorção 2	$X + \overline{X} \cdot Y = X + Y$
11 - Consenso	$X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z$
12 - De Morgan	$\overline{X + Y} = \overline{X} \cdot \overline{Y}$

5.2 Versão AND (Produto Lógico)

Propriedade (AND)	Expressão
1 - Identidade	$X \cdot 1 = X$
2 - Elemento Nulo	$X \cdot 0 = 0$
3 - Idempotência	$X \cdot X = X$
4 - Complemento	$X \cdot \overline{X} = 0$
5 - Involução	$\overline{\overline{X}} = X$
6 - Comutativa	$X \cdot Y = Y \cdot X$
7 - Associativa	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$
8 - Distributiva	$X \cdot (Y + Z) = X \cdot Y + X \cdot Z$
9 - Absorção 1	$X \cdot (X + Y) = X$
10 - Absorção 2	$X \cdot (\overline{X} + Y) = X \cdot Y$
11 - Consenso	$(X + Y)(\overline{X} + Z)(Y + Z) = (X + Y)(\overline{X} + Z)$
12 - De Morgan	$\overline{X \cdot Y} = \overline{X} + \overline{Y}$

As propriedades 2, 8, 9 e 11 serão demonstradas com tabelas verdades. As outras ficam a cargo do leitor demonstrar.

6 Demonstração das Propriedades

6.1 Propriedade 2: Elemento Nulo

Versão OR: Essa propriedade diz que:

$$X + 1 = 1$$

Veja a tabela verdade a seguir que comprova a igualdade

X	$X + 1$
0	1
1	1

Assim, independente do valor lógico de X a função sempre 1.

Versão AND:

X	$X \cdot 0$
0	0
1	0

Dessa forma, fica perceptível que o valor lógico de X na função não altera o valor da função, que será sempre 0.

6.2 Propriedade 8: Distributiva

Versão OR:

X	Y	Z	$X + (Y \cdot Z)$	$(X + Y) \cdot (X + Z)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Versão AND:

X	Y	Z	$X \cdot (Y + Z)$	$(X \cdot Y) + (X \cdot Z)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

6.3 Propriedade 9: Absorção 1

Versão OR:

X	Y	$X + (X \cdot Y)$
0	0	0
0	1	0
1	0	1
1	1	1

Versão AND:

X	Y	$X \cdot (X + Y)$
0	0	0
0	1	0
1	0	1
1	1	1

6.4 Propriedade 11: Consensus

Versão OR:

X	Y	Z	$X \cdot Y + \bar{X} \cdot Z + Y \cdot Z$	$X \cdot Y + \bar{X} \cdot Z$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Versão AND:

X	Y	Z	$(X + Y)(\overline{X} + Z)(Y + Z)$	$(X + Y)(\overline{X} + Z)$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

6.5 Exercícios

Simplifique as expressões abaixo:

1. $(A + B) \cdot \overline{A}$
2. $A + (A \cdot B)$
3. $(\overline{A} \cdot B) + (X \cdot \overline{D}) + (B \cdot \overline{D}) + (A \cdot \overline{B})$
4. $(\overline{A} \cdot B \cdot \overline{X}) + (\overline{A} \cdot \overline{B} \cdot \overline{X}) + (A \cdot B \cdot \overline{X}) + (A \cdot \overline{B} \cdot \overline{X})$
5. $(\overline{A} \cdot B) + (X \cdot \overline{D}) + (B \cdot \overline{D}) + (A \cdot B)$
6. $(A + \overline{B} + (A \cdot B)) \cdot (A \cdot \overline{B}) \cdot (\overline{A} \cdot B) + D$
7. $(A + \overline{B} + (A \cdot \overline{B}) + D) \cdot ((A \cdot B) + (\overline{A} \cdot B) + (B \cdot D)) + (\overline{A} + B)$

7 MINTERMOS E MAXTERMOS

Mintermos e maxtermos são formas canônicas de representar funções booleanas. Toda expressão booleana pode ser obtida através da sua tabela verdade.

Observação: Dizemos que uma expressão está na *forma canônica* quando ela é representada por uma **soma de mintermos** (forma canônica disjuntiva) ou um **produto de maxtermos** (forma canônica conjuntiva). Em ambas, todas as variáveis da função aparecem em cada termo, direta ou negada.

7.1 Mintermos

Mintermi: é a forma canônica de representar uma linha da tabela verdade utilizando o operador AND (E lógico) entre todas as variáveis da função. Cada variável pode aparecer de duas formas: direta ou negada, dependendo do valor que ela assume na linha da tabela.

Notação:

- Valor 0: a variável aparece **negada** (exemplo: $A = 0 \Rightarrow \overline{A}$)
- Valor 1: a variável aparece **direta** (exemplo: $A = 1 \Rightarrow A$)

O mintermo da primeira linha da tabela verdade é o m_0 e o da última será m_{2^n-1} , sendo n o número de variáveis da função. Para construir um mintermo, pega-se todas as variáveis da linha e usa-se a operação lógica **AND** entre elas, respeitando o uso da forma direta ou negada de acordo com os valores. Assim, cada mintermo representa exatamente uma linha onde a saída da função é igual a 1.

Notação – Forma Canônica por Mintermos

A notação $f(A, B, C) = \sum m(x, y, z)$ indica que a função f é expressa como uma **soma de mintermos**, ou seja, uma disjunção (OU) dos termos onde a saída é igual a 1 na tabela verdade. As letras "x", "y" e "z" indicam as **linhas** que a função é **1**.

Construção do Mintermo:

A	B	C	Mintermo	
0	0	0	$\overline{A} \cdot \overline{B} \cdot \overline{C}$	m_0
0	0	1	$\overline{A} \cdot \overline{B} \cdot C$	m_1
0	1	0	$\overline{A} \cdot B \cdot \overline{C}$	m_2
0	1	1	$\overline{A} \cdot B \cdot C$	m_3
1	0	0	$A \cdot \overline{B} \cdot \overline{C}$	m_4
1	0	1	$A \cdot \overline{B} \cdot C$	m_5
1	1	0	$A \cdot B \cdot \overline{C}$	m_6
1	1	1	$A \cdot B \cdot C$	m_7

7.2 Maxtermo

Maxtermo: é a forma canônica de representar uma linha da tabela verdade utilizando o operador OR (OU lógico) entre todas as variáveis da função. A construção dos maxtermos é semelhante à dos mintermos, porém segue a lógica inversa:

Notação:

- Valor 0: a variável aparece **direta** (exemplo: $A = 0 \Rightarrow A$)
- Valor 1: a variável aparece **negada** (exemplo: $A = 1 \Rightarrow \overline{A}$)

Para cada linha da tabela verdade em que a saída é 0, pode-se formar um maxtermo. O maxtermo da primeira linha da tabela verdade é o M_0 , e o último será M_{2^n-1} , com n sendo o número de variáveis.

Notação – Forma Canônica por Maxtermos

A notação $f(A, B, C) = \prod M(x, y, z)$ indica que a função f é expressa como uma **produto de maxtermos**, ou seja, uma conjunção (AND) dos termos onde a saída é igual a 0 na tabela verdade. As letras "x", "y" e "z" indicam as **linhas** que a função é **zero**.

Construção do Maxtermo:

A	B	C	Mintermo	
0	0	0	$A + B + C$	M_0
0	0	1	$A + B + \overline{C}$	M_1
0	1	0	$A + \overline{B} + C$	M_2
0	1	1	$A + \overline{B} + \overline{C}$	M_3
1	0	0	$\overline{A} + B + C$	M_4
1	0	1	$\overline{A} + B + \overline{C}$	M_5
1	1	0	$\overline{A} + \overline{B} + C$	M_6
1	1	1	$\overline{A} + \overline{B} + \overline{C}$	M_7

Exemplo:

$$f = A + B$$

A	B	f	Mintermo	Maxtermo
0	0	0	$m_0 = \overline{A}\overline{B}$	$M_0 = A + B$
0	1	1	$m_1 = \overline{A}B$	$M_1 = A + \overline{B}$
1	0	1	$m_2 = A\overline{B}$	$M_2 = \overline{A} + B$
1	1	1	$m_3 = AB$	$M_3 = \overline{A} + \overline{B}$

- Os **mintermos** representam as linhas em que $f = 1$.
- Os **maxtermos** representam as linhas em que $f = 0$.
- A função $f = A + B$ tem como forma canônica:
 - **Soma de mintermos:** $f = m_1 + m_2 + m_3 = \overline{A}B + A\overline{B} + AB$
 - **Produto de maxtermos:** $f = M_0 = (A + B)$

Mintermos, Maxtermos e Complemento

Como cada mintermo é o complemento de seu maxtermo correspondente, podemos dizer que o complemento de uma função f é representado pela multiplicação (AND) dos maxtermos associados às linhas onde $f = 0$.

No exemplo acima, como $f = A + B$, temos que $f = 1$ nas linhas 1, 2 e 3, e $f = 0$ apenas na linha 0. Logo, o complemento de f é:

$$f' = M_0 = A + B \quad \Rightarrow \quad f = \overline{A + B}$$

Assim, mostramos que o maxtermo M_0 representa exatamente o valor de f' nesta função.

8 Mapa de Karnaugh

O **Mapa de Karnaugh** (ou **diagrama de Karnaugh**) é um método gráfico utilizado para simplificar expressões lógicas obtidas a partir de tabelas-verdade. Essa técnica permite realizar a simplificação de forma mais direta e visual, dispensando o uso extensivo das leis algébricas da Álgebra Booleana.

8.1 Construção do Mapa de Karnaugh

A forma e o tamanho do Mapa de Karnaugh dependem da quantidade de variáveis envolvidas na expressão lógica, pois o mapa deve representar todas as combinações possíveis entre estas, assim como em uma tabela-verdade.

O Mapa de Karnaugh pode ser entendido como uma *representação gráfica alternativa* da tabela, organizada para facilitar a identificação de padrões e agrupamentos de termos passíveis de simplificação.

Nesta apostila, abordaremos os mapas de 2, 3 e 4 variáveis, que são os casos mais comuns em aplicações de circuitos digitais.

8.1.1 Mapa com 2 variáveis

Em um mapa com duas variáveis, a primeira linha e a primeira coluna representam todas as possíveis combinações destas. Assim, o mapa é formado por quatro células, correspondentes aos quatro estados binários possíveis das duas variáveis.

Cada célula do mapa representa um *mintermo*, conforme explicado anteriormente.

Consideremos uma tabela-verdade com as variáveis A e B :

- Combinações possíveis da variável A : A e $\bar{A}$;
- Combinações possíveis da variável B : B e $\bar{B}$.

	$\bar{A}$	A
$\bar{B}$		
B		

Tabela 1: Mapa de Karnaugh para duas variáveis (A, B).

Ordem e Disposição das Variáveis

A ordem das variáveis no cabeçalho é essencial para a simplificação correta. As células são dispostas seguindo o **código Gray**, isto é, duas células adjacentes diferem em apenas um bit.

No mapa de 2 variáveis a sequência na linha/coluna é $\bar{X} \mid X$, sendo X uma variável arbitrária.

8.1.2 Mapa com 3 variáveis

Quando temos três variáveis, agrupamos duas delas para compor o eixo horizontal e deixamos a terceira no eixo vertical. Assim, representamos todas as combinações possíveis das três variáveis em um mapa de 8 células — o dobro do mapa de duas variáveis.

Para exemplificar, consideremos as variáveis A , B e C . Aqui agrupamos A e B no topo (eixo horizontal) e colocamos C na lateral esquerda (eixo vertical).

- Combinações possíveis de A e B (seguindo código Gray): $\bar{A}\bar{B}$, $\bar{A}B$, AB , $A\bar{B}$;

	$\overline{AB}$	$\overline{AB}$	AB	$A\overline{B}$
$\overline{C}$				
C				

Tabela 2: Mapa de Karnaugh para três variáveis (eixo horizontal: AB ; eixo vertical: C).

- Combinações possíveis de C : $\overline{C}$ e C .

Ordem das Variáveis (3 variáveis)

A disposição segue a propriedade do código Gray: células adjacentes diferem por apenas um bit. Assim, a linha superior tem a ordem $\overline{X}\overline{Y} \mid \overline{X}Y \mid XY \mid X\overline{Y}$, sendo X e Y variáveis arbitrárias. Para a coluna lateral, aplica-se a mesma sequência de variação de uma única variável apresentada no Mapa com 2 variáveis.

8.1.3 Mapa com 4 variáveis

Quando trabalhamos com quatro variáveis, o mapa é estruturado agrupando duas variáveis no eixo horizontal e as outras duas no eixo vertical. A escolha da ordem dos agrupamentos é fundamental para manter a sequência correta do **código Gray**, o que garante a variação de apenas um bit entre células adjacentes e, conseqüentemente, facilita o processo de simplificação.

Consideremos as variáveis A , B , C e D . Neste caso, agrupamos A e B no eixo horizontal e C e D no eixo vertical, conforme mostrado a seguir:

- Combinações possíveis de A e B (seguindo o código Gray): $\overline{A}\overline{B}$, $\overline{A}B$, AB , $A\overline{B}$;
- Combinações possíveis de C e D (seguindo o código Gray): $\overline{C}\overline{D}$, $\overline{C}D$, CD , $C\overline{D}$.

	$\overline{AB}$	$\overline{AB}$	AB	$A\overline{B}$
$\overline{C}\overline{D}$				
$\overline{C}D$				
CD				
$C\overline{D}$				

Tabela 3: Mapa de Karnaugh para quatro variáveis (eixo horizontal: AB ; eixo vertical: CD).

Ordem das Variáveis (4 variáveis)

A disposição das variáveis segue o mesmo princípio descrito no Mapa com 3 variáveis. Tanto o eixo horizontal quanto o vertical utilizam a sequência estabelecida pelo **código Gray**. Essa organização é essencial para que os agrupamentos no processo de simplificação sejam válidos e visualmente evidentes.

8.2 Preenchendo o mapa e simplificando as expressões

Uma vez construído o Mapa de Karnaugh, o próximo passo é preenchê-lo de acordo com a tabela-verdade e utilizá-lo para simplificar a expressão lógica correspondente.

8.2.1 Preenchendo o mapa

O preenchimento consiste em associar cada célula do mapa ao seu respectivo *mintermo*, marcando com o valor lógico **1** as posições onde a função de saída também assume o valor 1 na tabela-verdade.

Por exemplo, considere a função $F(A, B, C)$ descrita pela seguinte tabela-verdade:

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

Tabela 4: Tabela-verdade da função $F(A, B, C)$.

Os casos em que $F = 1$ correspondem aos mintermos 1, 3, 4, 5 e 6. Esses valores são transferidos para o mapa de Karnaugh nas células correspondentes, conforme mostrado na Figura:

	$\overline{A}\overline{B}$	$\overline{A}B$	$A\overline{B}$	AB
C	1	1	1	1
$\overline{C}$	1	0	0	1

Tabela 5: Mapa de Karnaugh da Tabela 1.

Dica

Cada célula do mapa representa uma combinação única das variáveis de entrada, organizada de acordo com o código Gray. A correspondência entre os mintermos da tabela e as posições do mapa é fundamental para que os agrupamentos sejam válidos.

8.2.2 Simplificando a expressão

Com o mapa preenchido, realizamos a simplificação agrupando as células com valor 1. Os grupos devem conter uma quantidade de células que seja potência de 2 (1, 2, 4, 8, ...), e devem ser formados apenas entre células adjacentes — incluindo conexões que “contornam” o mapa.

- Cada grupo representa um termo simplificado da expressão;
- As variáveis que mudam dentro de um grupo são eliminadas;

- O resultado final é obtido somando (OR) os termos de todos os grupos formados.

Para o exemplo anterior, temos os seguintes grupos (separados em 2 mapas por conta da intersecção entre diversos grupos):

	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
C	1	1	1	1
$\overline{C}$	1	0	0	1

Tabela 6: Conjunto 1 da Tabela 2.

	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
C	1	1	1	1
$\overline{C}$	1	0	0	1

Tabela 7: Conjunto 2 da Tabela 2.

Com isso, é possível notar que a função dada resulta em 2 conjuntos de 4 mintermos cada, sendo o conjunto em azul (2) um "contorno" do Mapa, como citado anteriormente. Assim, a função de saída que seria:

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

resume-se a:

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

Isso acontece a partir do passo-a-passo explicado, ou seja, no Conjunto 1, mantemos apenas a variável C do mintermo, tendo em vista que ambos A e B alteram em negação. Já no Conjunto 2, ambas as variáveis A e C alternam-se, enquanto a variável B permanece em negação em todo o conjunto.

Observação

Grupos maiores produzem expressões mais simplificadas. Quando possível, priorize agrupamentos que cubram o maior número de células adjacentes.

8.3 Exercícios

1. Construa o Mapa de Karnaugh para a função $F(A, B) = \overline{A} + B$. Simplifique a expressão.
2. Complete o mapa de 3 variáveis para a função $F(A, B, C)$ que vale 1 nos mintermos 1, 3, 5 e 7. Determine a expressão simplificada.
3. Explique, em suas próprias palavras, por que o código Gray é essencial na disposição das células.
4. (Desafio) Elabore um mapa de 4 variáveis que represente uma função de paridade (saída 1 quando o número de entradas 1 for ímpar).

9 Somador

O **somador** é um circuito lógico combinacional projetado para realizar a soma de números binários. Ele está presente em praticamente todos os sistemas eletrônicos modernos, servindo de base para operações aritméticas em processadores e calculadoras digitais. Apesar de sua aparente simplicidade, o somador é um dos blocos fundamentais da eletrônica digital.

9.1 Meio-somador

Para compreender o funcionamento de um somador completo, é interessante começar com uma versão mais simples: o **meio-somador**.

O meio-somador é um pequeno circuito formado por portas lógicas que recebe duas entradas — os bits A e B — e produz duas saídas:

- R : o resultado da soma entre A e B ;
- C : o *Carry*, ou transporte, gerado quando a soma “transborda”.

Carry

O *Carry* é o famoso “**vai um**” da adição. No sistema decimal, quando somamos $9 + 7 = 16$, o “1” é levado para a próxima casa. Em binário, o mesmo acontece: quando a soma de dois bits ultrapassa 1, o resultado volta a 0 e o *Carry* torna-se 1, sendo adicionado na próxima posição.

A Tabela 8 mostra as possíveis combinações:

A	B	C	R	Interpretação
0	0	0	0	$0 + 0 = 00$
0	1	0	1	$0 + 1 = 01$
1	0	0	1	$1 + 0 = 01$
1	1	1	0	$1 + 1 = 10$

Tabela 8: table
Tabela-verdade do meio-somador.

A partir da tabela, obtemos as expressões lógicas das saídas:

$$C = A \cdot B \quad \text{e} \quad R = A \oplus B$$

Essas expressões indicam que o **Carry** é produzido por uma porta *AND* e o **Resultado** por uma porta *XOR*, conforme ilustrado na Figura 1.

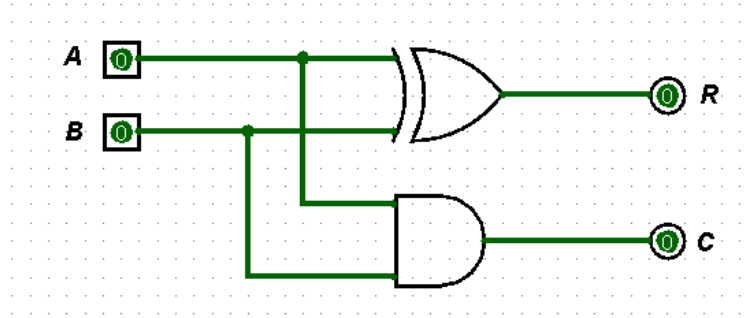


Figura 1: Circuito lógico do meio-somador.

9.2 Somador completo

O **somador completo** é uma evolução do meio-somador. Além das duas entradas principais (A e B), ele também recebe um **Carry-In** — o transporte gerado pela soma anterior. Sua saída, por sua vez, produz o **Carry-Out**, que pode ser levado à próxima posição.

C_{IN}	A	B	C_{OUT}	R	Interpretação
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

Tabela 9: Tabela-verdade do somador completo.

A partir dessa tabela, é possível obter as seguintes expressões simplificadas (via Mapa de Karnaugh):

$$C_{OUT} = (A \cdot B) + (A \cdot C_{IN}) + (B \cdot C_{IN})$$

$$R = (A \oplus B) \oplus C_{IN}$$

O circuito resultante é composto por dois meios-somadores e uma porta OR , conforme mostra a Figura 2.

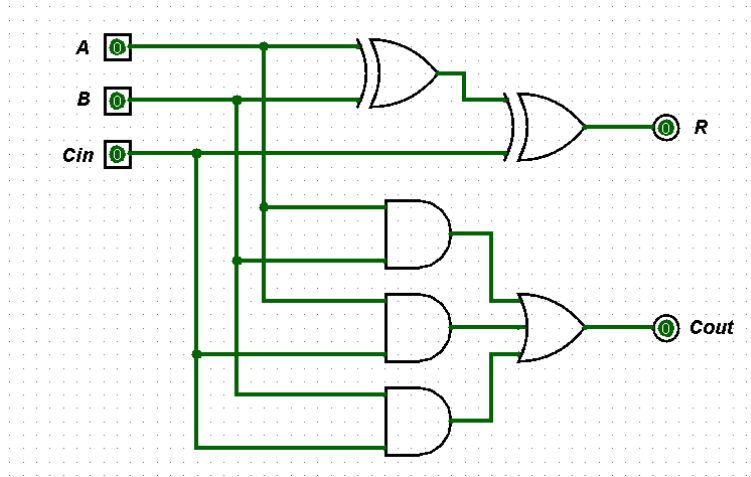
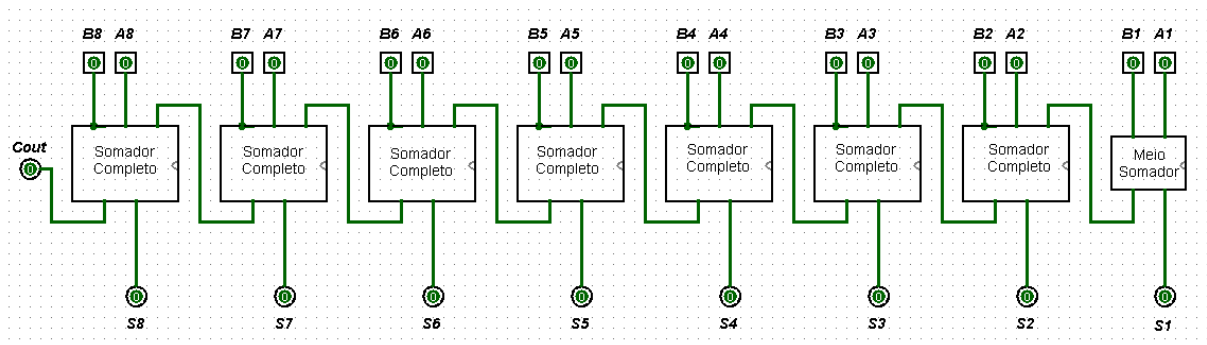


Figura 2: Circuito lógico do somador completo.

9.3 Somador de n bits

Por fim, ao conectar vários somadores completos em sequência, temos o **somador de n bits**. Nesse circuito, o transporte de saída (*Carry-Out*) de cada estágio torna-se o *Carry-In* do próximo, permitindo somar números binários com qualquer quantidade de bits.



Somador de 8 bits.

Cada saída representa um bit da soma final, e o último *Carry-Out* indica se ocorreu um **overflow** — ou seja, se o resultado excedeu o tamanho máximo representável.

Observação

A saída *Carry-Out* pode também servir como entrada para outro bloco somador, permitindo construir somadores ainda maiores. Nesse caso, apenas o primeiro bloco da sequência precisa ser um meio-somador; os demais devem ser somadores completos.

9.4 Exercícios

1. Monte a tabela-verdade de um meio-somador e verifique as expressões das saídas C e R .

2. A partir do circuito do somador completo, determine o valor da soma quando $A = 1$, $B = 1$ e $C_{in} = 1$.
3. Desenhe o esquema de um somador de 4 bits utilizando somadores completos em cascata.
4. (Desafio) Explique como o mesmo circuito poderia realizar subtrações utilizando complemento de dois.

10 Subtrator

O **subtrator** é um circuito lógico combinacional projetado para realizar a subtração entre números binários. Assim como o somador, ele é amplamente utilizado em processadores, calculadoras e sistemas digitais. Apesar de executar a operação inversa, sua estrutura lógica é bastante semelhante à do somador.

10.1 Meio-subtrator

Para compreender o funcionamento de um subtrator completo, é interessante começar com uma versão mais simples: o **meio-subtrator**.

Ele recebe duas entradas — A (minuendo) e B (subtraendo) — e gera duas saídas:

- R : o **bit de diferença**, resultado de $A - B$;
- B_o : o **bit de empréstimo** (*Borrow*), que indica quando é necessário “pedir emprestado” da próxima casa.

Borrow

O *Borrow* é o equivalente ao famoso "**empresta 1**" da subtração. No sistema decimal, fazemos isso ao calcular $307 - 128$, por exemplo: emprestamos 1 da casa vizinha. No sistema binário, ocorre o mesmo — sempre que o minuendo é 0 e o subtraendo é 1, o circuito precisa “pedir emprestado”.

A Tabela 10 mostra as possíveis combinações das entradas e saídas:

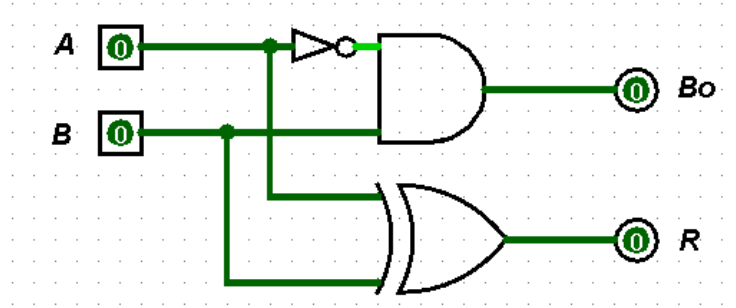
A	B	B_o	R	Interpretação
0	0	0	0	$0 - 0 = 0$
0	1	1	1	$0 - 1 = 1$ (com empréstimo)
1	0	0	1	$1 - 0 = 1$
1	1	0	0	$1 - 1 = 0$

Tabela 10: Tabela-verdade do meio-subtrator.

A partir da tabela, obtemos as expressões booleanas das saídas:

$$B_o = \overline{A} \cdot B \quad \text{e} \quad R = A \oplus B$$

Essas expressões mostram que o circuito é formado por uma porta *XOR* (para a diferença) e uma combinação *NOT-AND* (para o empréstimo).



Circuito lógico do meio-subtrator.

10.2 Subtrator completo

O **subtrator completo** é uma extensão do meio-subtrator que considera também o *empréstimo de entrada* (B_{in}), vindo da subtração anterior. Assim, ele possui três entradas — A , B e B_{in} — e duas saídas:

- R : diferença resultante;
- B_o : empréstimo de saída.

A	B	B_{in}	R	B_o	Interpretação
0	0	0	0	0	$0 - 0 - 0 = 0$
0	0	1	1	1	$0 - 0 - 1 = 1$ (com empréstimo)
0	1	0	1	1	$0 - 1 - 0 = 1$ (com empréstimo)
0	1	1	0	1	$0 - 1 - 1 = 0$ (com empréstimo)
1	0	0	1	0	$1 - 0 - 0 = 1$
1	0	1	0	0	$1 - 0 - 1 = 0$
1	1	0	0	0	$1 - 1 - 0 = 0$
1	1	1	1	1	$1 - 1 - 1 = 1$ (com empréstimo)

Tabela 11: Tabela-verdade do subtrator completo.

Com base na tabela e na simplificação via Mapa de Karnaugh, temos:

$$B_o = \overline{A}B + \overline{A}B_{in} + BB_{in} \quad \text{e} \quad R = A \oplus B \oplus B_{in}$$

O circuito pode ser construído conectando dois meios-subtratores e uma porta OR, como mostrado na Figura 3.

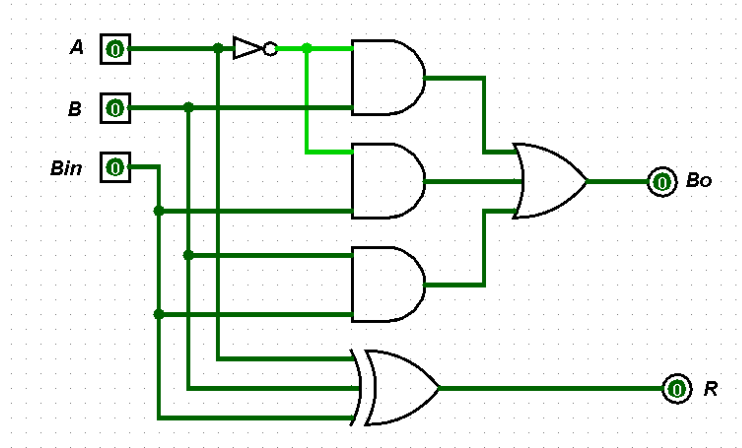


Figura 3: Circuito lógico do subtrator completo.

10.3 Subtrator de n bits

Da mesma forma que no somador, é possível criar subtratores com vários bits conectando múltiplos subtratores completos em sequência. O *Borrow-Out* de cada estágio torna-se o *Borrow-In* do estágio seguinte.

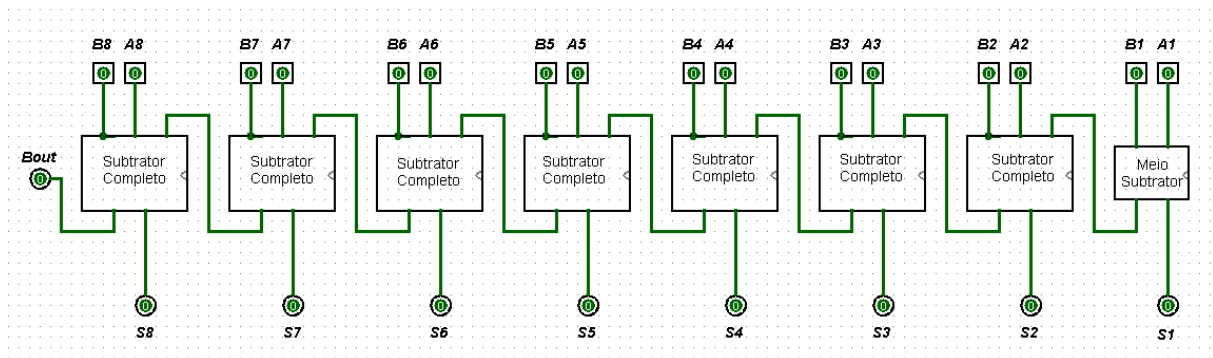


Figura 4: Circuito subtrator de 8 bits.

Assim, cada saída representa um bit da diferença final, e o último *Borrow-Out* indica se foi necessário um empréstimo além do tamanho representável (ou seja, se a subtração “estourou” o limite de bits).

10.4 Implementação alternativa

Também é possível implementar a subtração utilizando apenas o circuito somador. Isso porque, no mundo binário, subtrair um número é equivalente a somar o seu **complemento de dois**:

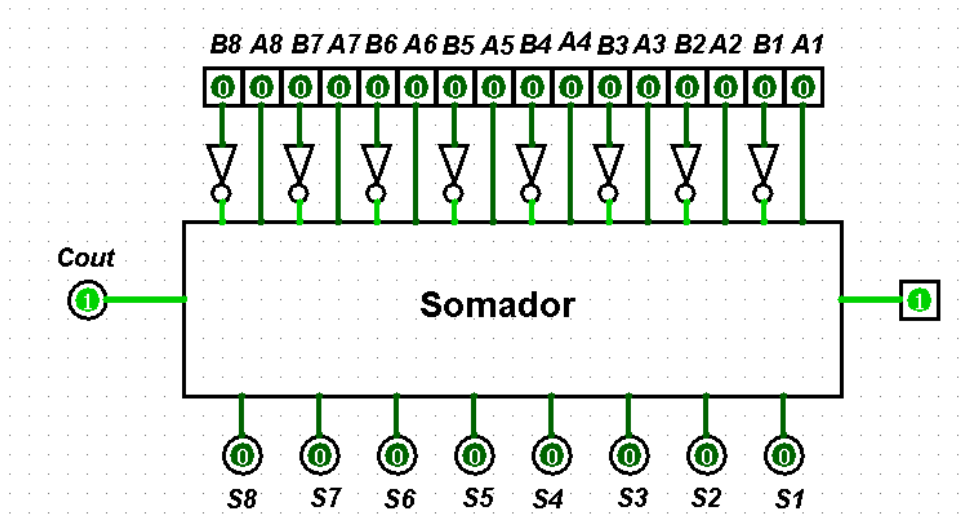
$$A - B = A + (-B) = A + (\overline{B} + 1)$$

Para isso:

- Inverte-se o subtraendo B com portas *NOT*;

- Define-se o primeiro *Carry-In* do somador como 1, representando a adição do “+1” do complemento de dois.

Essa abordagem é amplamente usada em processadores reais, já que elimina a necessidade de dois circuitos distintos para soma e subtração. Um mesmo somador pode realizar ambas as operações, dependendo apenas do controle do sinal que inverte os bits de B .



Circuito subtrator de 8 bits alternativo.

Observação

O uso do complemento de dois é o método mais prático e eficiente para realizar subtrações em sistemas digitais. Assim, o mesmo circuito de soma pode operar como subtrator apenas ajustando um único sinal lógico.

10.5 Exercícios

1. Elabore a tabela-verdade do meio-subtrator e derive as expressões para R e B_o .
2. Determine o resultado da operação binária $1010 - 0111$ utilizando um subtrator de 4 bits.
3. Simule mentalmente o comportamento do subtrator completo para $A = 0$, $B = 1$ e $B_{in} = 1$.
4. (Desafio) Explique como a subtração pode ser implementada com um circuito somador usando o complemento de dois.

11 Multiplicador

O **multiplicador** é um circuito combinacional formado por portas lógicas e caixas-pretas que realiza multiplicações entre números binários. Ele é um dos blocos mais importantes de uma *Unidade Lógica e Aritmética (ULA)*, sendo amplamente utilizado em processadores, DSPs e outros sistemas digitais.

Apesar de parecer complexo, o multiplicador nada mais é do que uma versão organizada do processo manual de multiplicação — aquele mesmo método em que repetimos somas deslocadas, linha por linha.

Existem duas formas principais de implementar um multiplicador:

- **Por tabela-verdade:** listando todas as combinações possíveis entre as entradas e suas saídas correspondentes;
- **Por lógica combinacional:** aplicando o raciocínio de multiplicações parciais e somas sucessivas, semelhante à multiplicação no papel.

A seguir, exploraremos ambas as abordagens.

11.1 Multiplicador por tabela-verdade

Um multiplicador baseado em tabela-verdade funciona da mesma forma que qualquer circuito combinacional: são listadas todas as possíveis combinações das entradas e as respectivas saídas do resultado da multiplicação.

Como exemplo, construiremos um multiplicador de 2×2 bits.

Lembre-se: número de bits da saída

A quantidade de bits do resultado é sempre a soma dos bits das entradas. Assim, um multiplicador de 2 bits por 2 bits gera uma saída de 4 bits.

A Tabela 12 mostra todas as combinações de entradas e suas saídas correspondentes:

A		B		Resultado				Interpretação
A_1	A_0	B_1	B_0	R_3	R_2	R_1	R_0	
0	0	0	0	0	0	0	0	$0 \times 0 = 0$
0	0	0	1	0	0	0	0	$0 \times 1 = 0$
0	0	1	0	0	0	0	0	$0 \times 2 = 0$
0	0	1	1	0	0	0	0	$0 \times 3 = 0$
0	1	0	0	0	0	0	0	$1 \times 0 = 0$
0	1	0	1	0	0	0	1	$1 \times 1 = 1$
0	1	1	0	0	0	1	0	$1 \times 2 = 2$
0	1	1	1	0	0	1	1	$1 \times 3 = 3$
1	0	0	0	0	0	0	0	$2 \times 0 = 0$
1	0	0	1	0	0	1	0	$2 \times 1 = 2$
1	0	1	0	0	1	0	0	$2 \times 2 = 4$
1	0	1	1	0	1	1	0	$2 \times 3 = 6$
1	1	0	0	0	0	0	0	$3 \times 0 = 0$
1	1	0	1	0	0	1	1	$3 \times 1 = 3$
1	1	1	0	0	1	1	0	$3 \times 2 = 6$
1	1	1	1	1	0	0	1	$3 \times 3 = 9$

Tabela 12: Tabela-verdade resumida para o multiplicador de 2×2 bits.

A partir da simplificação das saídas (via Mapa de Karnaugh ou observação direta), temos:

$$R_0 = A_0 \cdot B_0$$

$$R_1 = (A_1 \cdot A_0 \cdot \overline{B_1} \cdot B_0) + (\overline{A_1} \cdot A_0 \cdot B_1) + (A_0 \cdot B_1 \cdot \overline{B_0}) + (A_1 \cdot \overline{A_0} \cdot B_1)$$

$$R_2 = (A_1 \cdot \overline{A_0} \cdot B_1) + (A_1 \cdot B_1 \cdot \overline{B_0})$$

$$R_3 = A_1 \cdot A_0 \cdot B_1 \cdot B_0$$

Essas expressões podem ser traduzidas em um circuito combinacional, composto basicamente por portas AND (para gerar os produtos parciais) e portas OR (para somá-los).

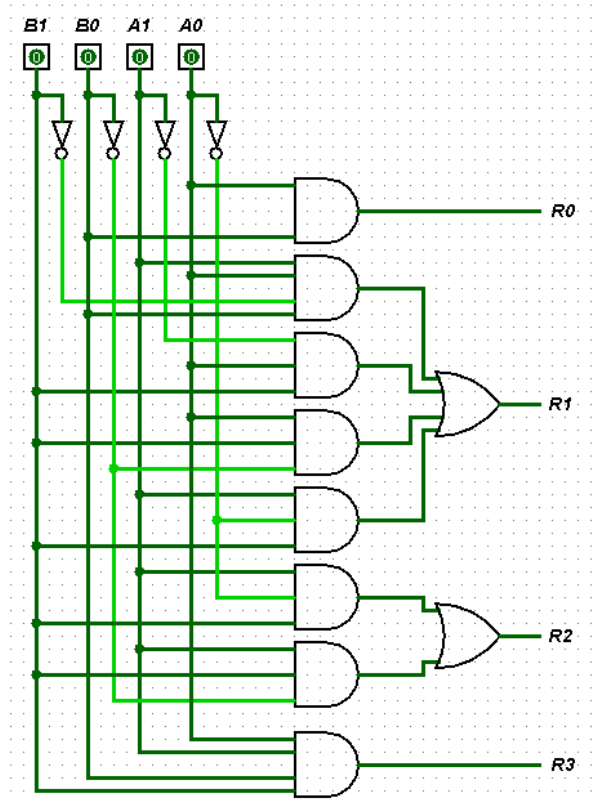


Figura 5: Circuito lógico de um multiplicador de 2×2 bits.

Generalização

O mesmo raciocínio pode ser aplicado a qualquer multiplicador de $n \times m$ bits. O circuito cresce proporcionalmente ao número de bits das entradas — quanto maior o tamanho dos operandos, maior o número de portas e somadores necessários.

11.2 Multiplicador combinacional

O **multiplicador combinacional** aplica diretamente a lógica da multiplicação no papel — só que feita com portas lógicas.

Cada bit do multiplicador (B) é conectado a todas as entradas do multiplicando (A) por meio de portas *AND*. Essas combinações geram os chamados **produtos parciais**, que depois são **somados** de forma deslocada, assim como fazemos manualmente ao multiplicar dois números decimais.

$$\begin{array}{r}
 \begin{array}{r}
 A_1 A_0 \\
 \times B_1 B_0 \\
 \hline
 A_1 B_0 A_0 B_0 \\
 A_1 B_1 A_0 B_1 + \\
 \hline
 R_3 R_2 R_1 R_0
 \end{array}
 \end{array}$$

O primeiro produto parcial é escrito normalmente; os seguintes são **deslocados uma posição à esquerda** — isso representa o aumento do valor posicional dos bits (de unidades para dezenas, centenas, etc., no sistema binário).

Esses produtos parciais são então somados por meio de somadores completos, formando uma estrutura de **array** (ou matriz de somadores). O resultado final tem $n + m$ bits.

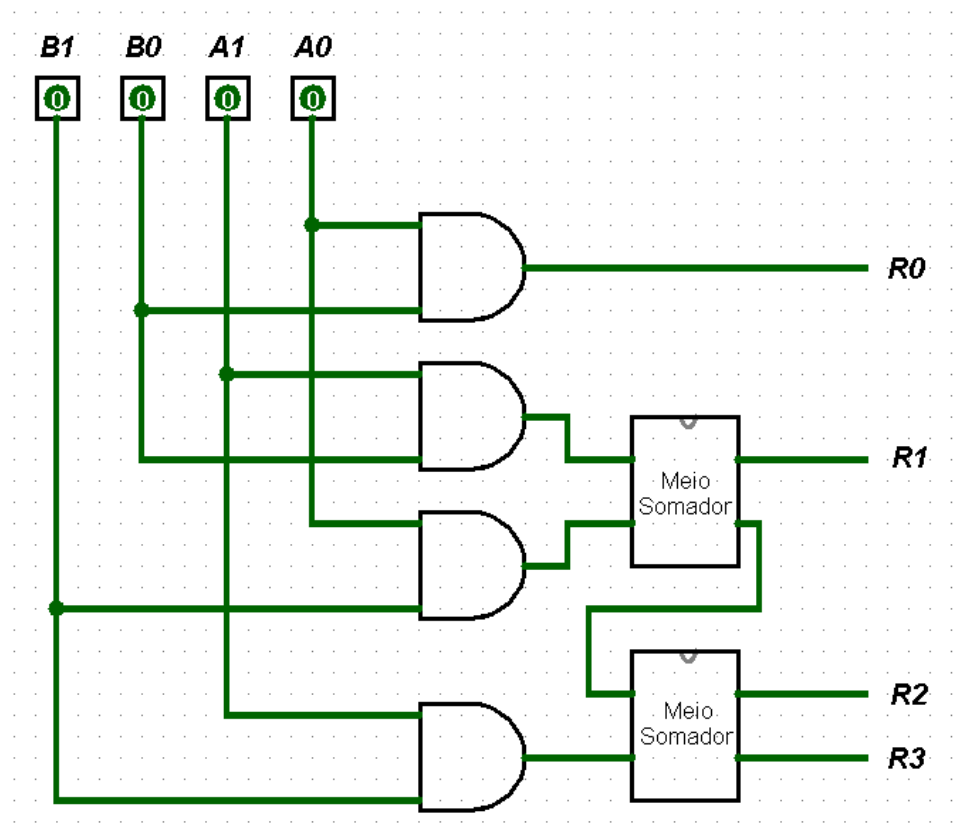


Figura 6: Estrutura de um multiplicador combinacional de 2×2 bits.

Multiplicadores maiores

A estrutura combinacional pode ser expandida para qualquer número de bits. Para operandos maiores, o circuito se organiza em camadas diagonais de somadores — formando um padrão conhecido como **array multiplier**.

11.3 Conclusão

O multiplicador é, essencialmente, uma combinação de **portas AND** (para gerar produtos parciais) e **somadores** (para somá-los). A versão por tabela-verdade é conceitualmente

útil para pequenos números de bits, mas se torna inviável em circuitos grandes. Já o método combinacional é o mais prático e escalável, sendo o preferido em sistemas reais.

Resumo do funcionamento

Cada linha do multiplicador representa uma soma condicional: se o bit do multiplicador for 1, somamos o multiplicando deslocado; se for 0, nada é somado. O circuito faz isso automaticamente, multiplicando por meio de repetições de soma e deslocamento.

11.4 Exercícios

1. Construa a tabela-verdade para um multiplicador de 1×2 bits.
2. Determine o resultado da multiplicação $10_2 \times 11_2$ utilizando o raciocínio dos produtos parciais.
3. Explique o papel das portas AND e dos somadores no funcionamento do multiplicador combinacional.
4. (Desafio) Proponha uma forma de otimizar um multiplicador de 4 bits, reduzindo o número de somadores necessários.

12 Decodificadores e Codificadores

Até este ponto, trabalhamos com circuitos que operam diretamente sobre valores binários. Entretanto, em sistemas digitais reais, é comum a interação com informações representadas de forma simbólica, como números decimais, caracteres e sinais provenientes de teclados ou *displays*.

Para que haja uma comunicação eficaz entre o mundo físico e o digital, é necessário realizar uma conversão entre representações simbólicas e códigos binários. Essa função é desempenhada por dois circuitos combinacionais fundamentais: os **decodificadores** e os **codificadores**.

- **Codificador:** converte um sinal simbólico em uma representação binária.
- **Decodificador:** realiza o processo inverso, convertendo um código binário em uma saída simbólica.

—

12.1 Decodificador

O **decodificador** é um circuito combinacional cuja função é identificar uma combinação binária de entrada e ativar a respectiva saída correspondente. Em termos práticos, ele “traduz” um código binário em uma única linha de saída ativa.

Por exemplo, um **decodificador $2 \rightarrow 4$** possui duas entradas binárias (E_1 e E_0) e quatro saídas (S_0, S_1, S_2, S_3). Cada combinação das entradas ativa exclusivamente uma das saídas, conforme apresentado na Tabela 13.

E_1	E_0	Saída ativa
0	0	S_0
0	1	S_1
1	0	S_2
1	1	S_3

Tabela 13: Tabela-verdade de um decodificador 2→4.

As expressões lógicas de cada saída são obtidas diretamente da tabela-verdade:

$$S_0 = \overline{E_1} \cdot \overline{E_0}, \quad S_1 = \overline{E_1} \cdot E_0, \quad S_2 = E_1 \cdot \overline{E_0}, \quad S_3 = E_1 \cdot E_0.$$

Essas expressões podem ser implementadas por meio de portas lógicas *AND* e *NOT*, resultando em um circuito de estrutura simples e direta.

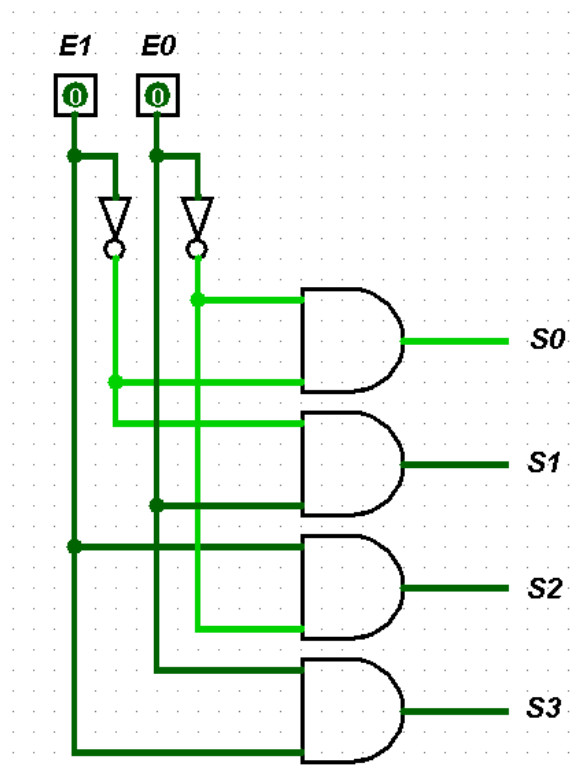


Figura 7: Circuito lógico de um decodificador 2→4.

Aplicações práticas

Os decodificadores são amplamente utilizados em:

- Seleção de endereços de memória (ativação de uma linha específica);
- Controle de *displays* de sete segmentos e painéis indicadores;
- Seleção de registradores e dispositivos em sistemas microprocessados.

12.2 Codificador

O **codificador** executa a operação inversa ao decodificador. Ele recebe múltiplas entradas — das quais apenas uma está ativa — e gera um código binário correspondente à linha de entrada ativada.

Por exemplo, um **codificador 4→2** possui quatro entradas (E_3, E_2, E_1, E_0) e duas saídas (S_1, S_0). A Tabela 14 apresenta o funcionamento deste circuito.

Entrada ativa	S_1	S_0
E_0	0	0
E_1	0	1
E_2	1	0
E_3	1	1

Tabela 14: Tabela-verdade de um codificador 4→2.

As expressões lógicas das saídas são:

$$S_0 = E_1 + E_3, \quad S_1 = E_2 + E_3.$$

Essas funções podem ser implementadas utilizando apenas portas *OR*, tornando o circuito compacto e eficiente.

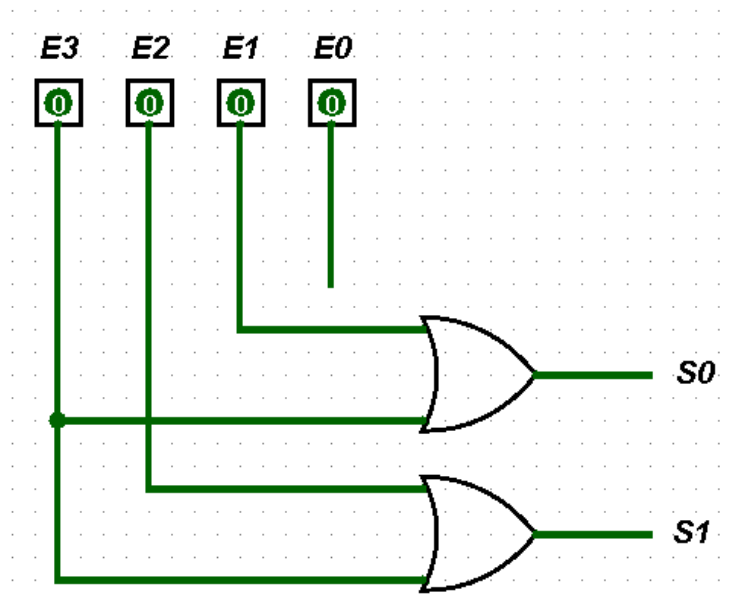


Figura 8: Circuito lógico de um codificador 4→2.

Codificador com prioridade

Na prática, pode ocorrer que mais de uma entrada esteja ativa simultaneamente. Nesses casos, utiliza-se o **codificador com prioridade**, que considera válida apenas a entrada de maior ordem (ou de maior peso binário), garantindo a consistência do resultado.

Aplicações práticas

Os codificadores são frequentemente empregados em:

- Teclados e painéis de controle, convertendo teclas em códigos binários;
- Sistemas de interrupção, determinando a prioridade entre múltiplas solicitações;
- Conversores de sinais provenientes de sensores e chaves seletoras.

12.3 Associação entre Codificadores e Decodificadores

Em diversos sistemas digitais, os codificadores e decodificadores atuam de forma complementar. Enquanto o codificador converte um sinal físico em código binário, o decodificador interpreta esse código e o converte novamente em uma forma simbólica ou de controle.

Um exemplo clássico ocorre em teclados e *displays* numéricos. Ao pressionar a tecla “2”, o circuito codificador gera o código binário 0010, o qual é então enviado ao decodificador. Este, por sua vez, ativa os segmentos correspondentes no *display*, exibindo o número “2”.

[Tecla “2” pressionada] $\Rightarrow$ (codificador) $\Rightarrow$ 0010 $\Rightarrow$ (decodificador) $\Rightarrow$ Exibição no *display*.

Resumo conceitual

- O **codificador** realiza a *compactação da informação*, transformando sinais físicos em representações binárias;
- O **decodificador** realiza a *expansão da informação*, convertendo códigos binários em ações ou sinais específicos;
- Juntos, constituem a base da comunicação entre o ambiente físico e o domínio lógico-digital.

12.4 Exercícios

1. Construa a tabela-verdade de um decodificador $3 \rightarrow 8$.
2. A partir de um codificador $8 \rightarrow 3$, determine a saída binária quando a entrada ativa for a linha 5.
3. Explique o funcionamento de um codificador com prioridade e cite uma aplicação real.
4. (Desafio) Desenhe um circuito em que um codificador e um decodificador trabalhem em conjunto para representar um teclado e um *display* numérico.

13 Circuito Combinacional

Circuitos combinacionais são aqueles que só dependem do estado atual da entrada, ou seja, não possuem **memória**. Um exemplo simples é um circuito onde a saída f é dada por $f = A \cdot B$, usando uma porta **AND** de duas entradas.

Esses circuitos incluem:

- Somadores
- Comparadores
- Codificadores
- Decodificadores
- Multiplexadores

14 Circuitos Sequenciais

Diferente dos circuitos combinacionais, os **circuitos sequenciais** dependem tanto das **entradas atuais** como do **estado anterior (memória)**.

Eles utilizam elementos de **armazenamento**, como flip-flops, para guardar o estado interno. Portanto, a saída é uma função das entradas e do estado armazenado.

São utilizados em:

- Registradores
- Máquinas de Estados
- Contadores

Os circuitos sequenciais podem ser **síncronos** (controlados por um sinal de clock) ou **assíncronos** (controlados por variações nas entradas)

15 Multiplexadores e Demultiplexadores

15.1 MUX

O Multiplexador (MUX) é um **circuito lógico combinacional** que seleciona uma entre X entradas de dados e a envia para uma única saída, conforme os Y sinais de controle.

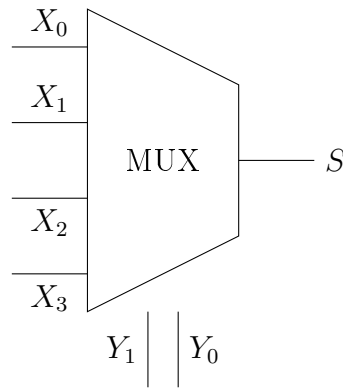
Normalmente usamos a relação $X = 2^Y$

Em outras palavras, ele atua como um "canal seletor":

Várias entradas $\rightarrow$ 1 saída escolhida.

Exemplo

Pensando em algo cotidiano: imagine um chuveiro elétrico com diferentes opções de temperatura, em que cada opção é uma entrada (X_0, X_1, X_2, X_3). O botão do chuveiro é a entrada de seleção e a água que sai é a saída. Ao escolher uma posição no botão, apenas uma entrada é conectada à saída. O MUX faz exatamente isso, só que de forma digital e automática, controlada por bits.



S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Tabela 15: Tabela verdade do multiplexador 4x1

15.1.1 MUX 4x1

Um MUX com 4 entradas binárias (X_0, X_1, X_2, X_3), 2 sinais de controle (Y_0, Y_1) e 1 saída (S), também denominado de multiplexador 4x1, é definido pela tabela verdade:

15.2 DEMUX

O Demultiplexador (DEMUX) é um **circuito lógico combinacional** que recebe **uma única entrada de dados** e a direciona para **uma entre X saídas**, conforme os Y sinais de controle. **Normalmente usamos a relação $X = 2^Y$**

Em outras palavras, ele atua como um distribuidor de dados:

1 entrada $\rightarrow$ várias saídas (apenas uma ativa por vez).

Exemplo

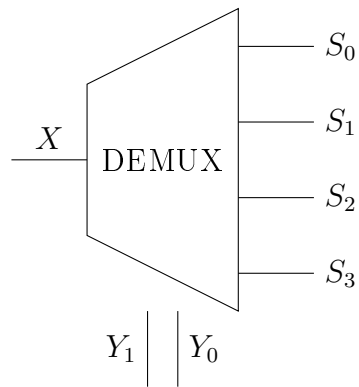
Imagine um centro de distribuição dos Correios.

A encomenda que chega representa a **entrada única** do DEMUX. As diferentes rotas de entrega (bairros ou cidades) representam as **saídas** (S_0, S_1, S_2, S_3). O CEP da encomenda funciona como o **sinal de controle**.

De acordo com esse código, a encomenda é enviada para **apenas uma rota específica**, enquanto as demais não recebem nada naquele momento. O DEMUX funciona exatamente assim: ele recebe um único dado e o direciona para uma única saída, conforme os bits de seleção.

15.2.1 DEMUX 1x4

Um DEMUX com 1 entrada binária (D), 2 sinais de controle (S_0, S_1) e 4 saídas (Y_0, Y_1, Y_2, Y_3), também denominado de demultiplexador 1x4, é definido pela tabela verdade:



Y_1	Y_0	Saída ativa
0	0	$S_0 = X$
0	1	$S_1 = X$
1	0	$S_2 = X$
1	1	$S_3 = X$

Tabela 16: Tabela verdade do demultiplexador 1x4

15.3 Multiplexador x Demultiplexador

Característica	MUX	DEMUX
Tipo de circuito	Combinacional	Combinacional
Entradas de dados	Várias	Uma
Entradas de seleção	m bits	m bits
Saídas	Uma	Várias
Função principal	Selecionar entrada	Direcionar saída
Fluxo de dados	Entradas $\rightarrow$ saída única	Entrada única $\rightarrow$ saídas

Tabela 17: Comparação entre MUX e DEMUX

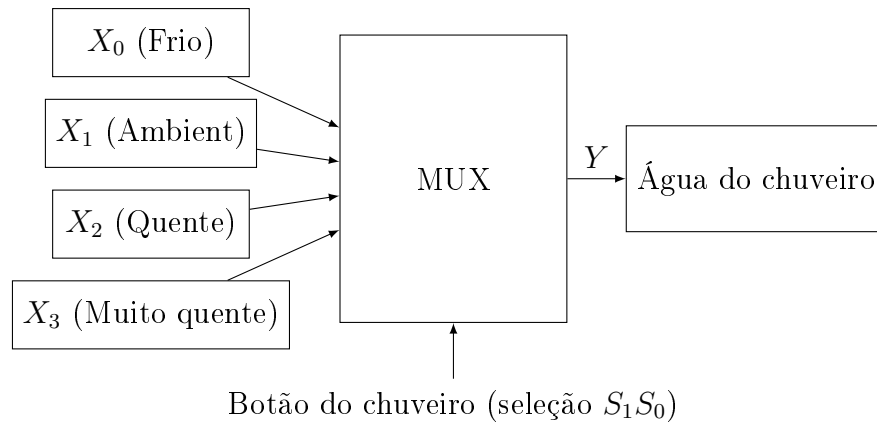


Figura 9: Exemplo de um MUX usando o seletor de temperatura do chuveiro

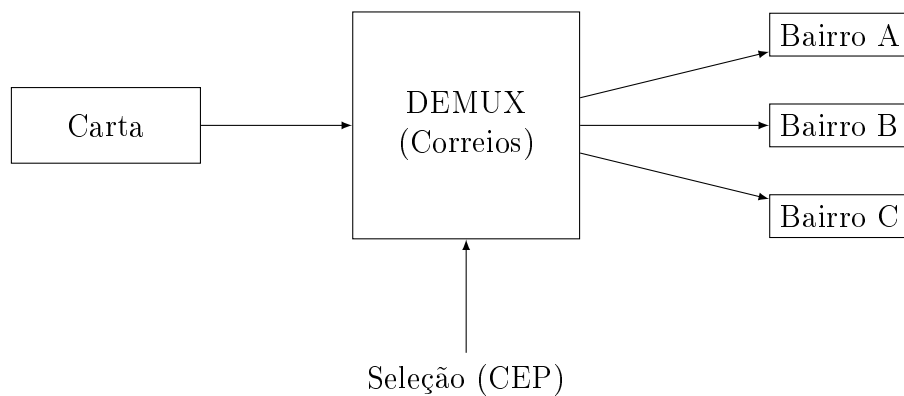


Figura 10: Exemplo de funcionamento de um DEMUX usando a analogia dos correios

16 Unidade Lógica Aritmética (ULA)

A Unidade Lógica Aritmética (ULA) tem como função realizar operações aritméticas e lógicas sobre dados binários, como soma, subtração, comparação e operações lógicas básicas.

A ULA recebe como entrada dois operandos binários, além de sinais de controle que definem qual operação deve ser executada. O resultado da operação é disponibilizado na saída e pode ser armazenado ou utilizado por outros blocos do sistema.

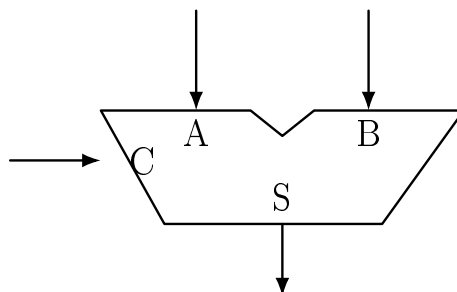


Figura 11: Unidade Lógica e Aritmética

Na imagem a operação realizada pela ULA que conta com as entradas A e B , é determinada por um sinal de controle C e o resultado é obtido na saída S .

As operações executadas pela ULA podem ser classificadas em dois grupos principais:

- operações aritméticas.
- operações lógicas.

Entre as operações aritméticas mais comuns estão a soma e a subtração. Já as operações lógicas incluem AND, OR, XOR e NOT. A seleção da operação é feita por sinais de controle, normalmente representados por bits.

Exemplo

Vamos realizar a soma entre os números 8 e 6.

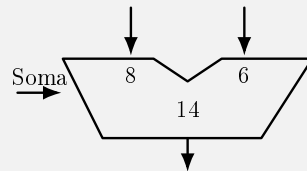
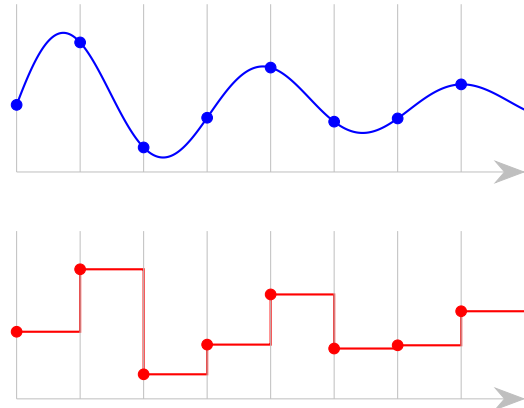


Figura 12: Soma na ULA: $8 + 6 = 14$

17 Formas de onda

Em um computador, toda a informação é processada com base nos **sinais elétricos** que passam pelos seus circuitos. Nesse contexto, existem dois tipos de interpretações de sinais elétricos: analógica e digital.

- **Sinais analógicos:** variam continuamente no tempo, podendo atingir qualquer valor dentro de um intervalo de tensão.
- **Sinais digitais:** variam de forma descontínua ou discreta, isto é, saltam entre valores específicos e pré-definidos. Quando os analisamos, não nos preocupamos com os valores intermediários a eles.

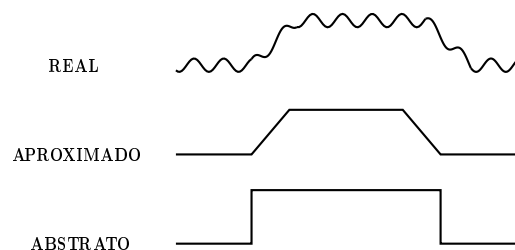


Para os computadores atuais, utilizamos sinais **digitais**, considerando apenas dois níveis possíveis: **alto** e **baixo**. A vantagem dessa abordagem está no fato de que ela simplifica a forma como um processador precisa ser projetado, de modo que consiga administrar esses dados, sem a necessidade de interpretar muitas variações de tensão. Com isso, o circuito precisa apenas fazer a distinção entre dois estados bem definidos:

- Nível alto (“ligado”, “verdadeiro”) → o circuito está conduzindo corrente elétrica;
- Nível baixo (“desligado”, “falso”) → o circuito não está conduzindo corrente significativa.

Esses estados podem ser facilmente representados por nível alto = 1 e nível baixo = 0.

Na prática, a transição de um nível para outro é gradual, devido às propriedades físicas dos componentes do computador. Mas, para simplificarmos o projeto do sistema, abstraímos, considerando que ela acontece de forma instantânea.



Essa abstração é possível porque são definidas faixas de tensão mínima e máxima para cada nível. Por exemplo:

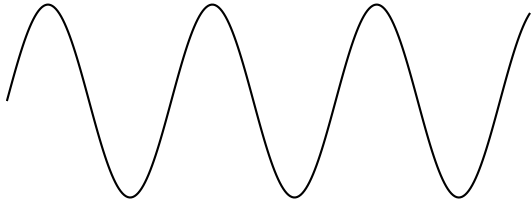
- Nível alto = entre 3V e 5V;
- Nível baixo = entre 0V e 2V;
- Entre 2V e 3V: região indefinida, não representa nenhum dos dois.

Dessa maneira, mesmo que fisicamente o sinal leve algum tempo para mudar de nível, do ponto de vista lógico ele é tratado como estando instantaneamente em 0 ou 1, desde que permaneça nas faixas definidas durante os momentos de leitura.

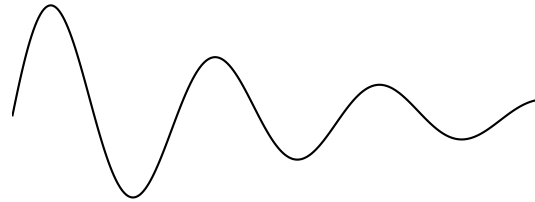
17.1 Tipos de Onda

Nesse sentido, representamos os sinais elétricos que circulam no *hardware* em forma de **ondas**, que podem ser **periódicas** ou **não periódicas**.

Periódicas: repetem-se em intervalos regulares;



Não periódicas: não seguem um padrão definido.



A fim de evitar inconsistências no sistema, é essencial que diferentes partes do processador interpretem sinais de forma coordenada, atualizando os seus estados ao mesmo tempo.

Para isso, é utilizado o que chamamos de **sinal de clock**: uma onda digital periódica, que atua como uma espécie de "metrônomo", para todas as operações.

Quanto mais rápido for o *clock* de um processador, mais rapidamente ocorrerá o fluxo dos dados (em *bits*). Consequentemente, teremos máquinas mais velozes.

Por outro lado, o *hardware* aquece mais, sendo necessária uma maior quantidade de energia destinada ao resfriamento do circuito.

Desse modo, o computador processa esses sinais elétricos, que representamos como ondas digitais, e os armazena em forma de números na memória, gerenciando todos os dados dessa maneira. Como simplificamos a interpretação deles em 0 e 1, a base numérica utilizada é a **base binária**.

18 Flip-Flop

Flip-flops são circuitos sequenciais de memória, responsáveis por armazenar 1 bit.

Existem diversos tipos de flip-flops e cada um conta com uma aplicação adequada.

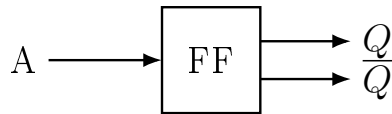


Figura 13: Estrutura básica de um Flip-flop.

18.1 Flip-Flop SR

Também chamado de latch SR (quando não possui clock), o Flip-flop SR tem duas entradas:

1. **S (Set)**: coloca a saída Q em 1;
2. **R (Reset)**: coloca a saída Q em 0.

18.2 Flip-Flop D

Esse flip-flop transfere o valor da entrada D para a saída Q quando o clock é ativado, então nele temos duas entradas:

1. **D**: entrada de dado (0 ou 1 a ser armazenado)
2. **Clock**: controla o funcionamento do FF.

18.3 Flip-Flop JK

J	K	CLK	Q
0	0	$\uparrow$	Q_0 (não muda)
1	0	$\uparrow$	1
0	1	$\uparrow$	0
1	1	$\uparrow$	$\overline{Q_0}$ (comuta)

Tabela 18: Tabela verdade flip-flop JK

19 Máquina de estados

Uma máquina de estados é um modelo que descreve sistemas cujo comportamento depende tanto das entradas atuais quanto do estado em que o sistema se encontra.

A partir de um estado atual, o sistema recebe entradas, produz saídas e pode transitar para um novo estado, repetindo esse processo ao longo do tempo. Conhecendo o estado atual e as entradas, é possível determinar completamente as saídas e o próximo estado.

Em circuitos digitais, as máquinas de estados são implementadas com Flip-Flops, e a combinação do valor armazenado em cada flip-flops determina o estado atual da máquina de estados.

As máquinas de estados são compostas por três elementos principais: entradas, estados e saída. Além disso, também existem funções para transições de estado e para definir saídas. Por exemplo:

1. Memória: conjunto de Flip-Flops que armazenam o estado atual do circuito.
2. Função de transição de estado (ou lógica de estado): determina o estado seguinte baseando-se no estado atual e nas entradas.
3. Função de saída (ou lógica de saída): define a saída atual a partir do estado atual e das entradas.

A combinação desses elementos produz uma máquina de estado. Para construir uma máquina de estado deve-se seguir os passos abaixo:

1. Separação e levantamento dos dados (ex: definir número de estados e de bits necessários, determinar uma combinação de bits para uma representação dos estados, das entradas e das saídas).
2. Representação em um diagrama estados: desenho auxiliar para descrição das transições de estado e das saídas.
3. Criação de uma tabela de estados: tabela verdade que combina os estados atuais e as entradas para definir os próximos estados e as saídas da máquina.
4. Mapa de karnaugh: usado para simplificação da tabela de estados quando possível.
5. Representação/Desenho do circuito lógico: implementação do circuito usando Flip-Flops e portas lógicas.

19.1 Exemplo Simplificado

Contador de 0 a 3 Podemos transformar um contador em uma máquina de estados, onde o estado seguinte depende do estado atual e das entradas. No entanto, neste caso não existem entradas, ou seja, é possível perceber que as entradas podem ou não serem usadas para uma transição de estado.

Separação ou levantamento de dados: Podemos pensar em algumas perguntas que devem ser respondidas para contemplar todos os dados e informações úteis, por exemplo: Quantos estados devem existir para ser possível representar valores de 0 a 3? Ou ainda quantos bits são necessários para essa representação?

Para contarmos de 0 a 3 temos um intervalo de 4 valores (0, 1, 2 e 3), ou seja, precisamos de ao menos 4 estados para isso. Além disso, quantos bits precisamos para representar 4 valores distintos? Como vimos anteriormente: $2^b \geq 4 = n$, isto é, precisamos de 2 bits para representar n valores com $n = 4$. Agora, precisamos definir quais estados devem ter quais valores em sua representação em binário. Para simplificação do exemplo, podemos tomar a representação binária de n como seu respectivo estado:

Número (n)	Representação Binária de n em 2 bits
0	00
1	01
2	10
3	11

Tabela 19: Representação binária de n

Nota: representação dos estados

Neste exemplo 19.1 tomamos os valores de n iguais as suas respectivas representações binárias para simplificação do exemplo proposto. No entanto, não é necessário, ou seja, poderíamos ter representado, por exemplo, $n = 0$ como 11 ou ainda $n' = 3$ como 01, já que estamos tratando apenas a **representação** dos estados e não a saída ou valor de fato.

Assim, após separar dos dados e determinar uma representação binária para cada estado determinamos os atributos iniciais para a construção da máquina de estados.

Observação: Portanto, como tomamos a representação binária de n como seu respectivo estado podemos substituir o nome "Representação Binária de n em 2 bits" em 19 por Estado Atual do contador.

Representação em um diagrama estados Após definirmos as informações iniciais podemos construir um diagrama de estados para auxiliar na definição da lógica de transição de estado. Para essa representação usamos círculos e setas que apontam para o estado seguinte baseando-se no estado atual. Neste exemplo de contador, sabemos que existe uma ordem, isto é, se estamos no 0 devemos ir para o 1, se estamos no 1 devemos transitar para o 2 e assim sucessivamente. Portanto a representação pode ser feita da seguinte forma:

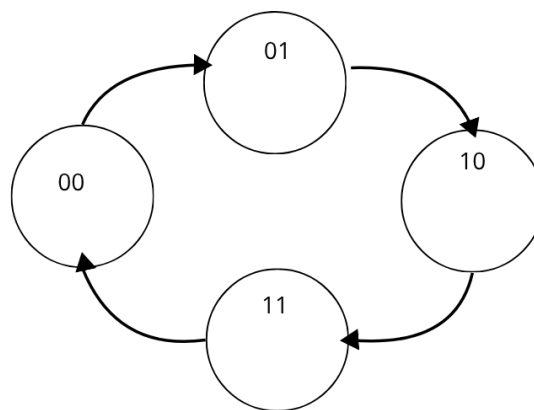


Figura 14: Máquina de Estados de Contador

As setas indicam que estando no estado atual (representado por qualquer círculo) irá para o próximo estado indicado pela seta posição da seta.

Nota: Caso tivéssemos entradas neste exemplo seus respectivos valores (representação binária) seriam representados em cima das setas indicando que a combinação daquele estado atual + valor da saída levam ao novo estado indicado pela ponta da seta.

Criação de uma tabela de estados: Para criação de uma tabela de estados devemos ter três colunas: estado atual, variáveis ou entradas e o estado seguinte. A coluna

do estado seguinte tem seus valores dependentes do estado atual e das variáveis. Para definirmos o estado seguinte, começamos em um estado atual e olhamos no diagrama de estados o local que a seta indica o próximo estado. Por exemplo: no estado 00 a seta indica para o estado 01. Assim, repetimos o processo sucessivamente até contemplarmos todos os estados e combinações possíveis. Abaixo contruímos a tabela de estados do contador 19.1 usando como apoio o diagrama 14, e como não possuímos entradas deixaremos X no valor delas.

Estado atual (E)	Entradas (X)	Estado Seguinte (D)
00	X	01
01	X	10
10	X	11
11	X	00

Tabela 20: Tabela de Estados

Estado atual (E)	Estado Seguinte (D)	Saídas $S1S0$
00	01	00
01	10	01
10	11	10
11	00	11

Tabela 21: Tabela de Estados Simplificada com saídas

Na segunda tabela 21, incluímos uma coluna que representa as saídas $S1S0$. É perceptível que como definimos a representação binária dos números como seus respectivos estados temos que as saídas de cada estado são iguais ao estado atual. No entanto, se definíssemos de outra maneira isso poderia não ocorrer. Além disso, excluímos a tabela de entradas, pois este exemplo não contempla nenhuma.

Mapa de Karnaugh Para simplificação das equação / funções de transição de estado podemos usar o mapa de karnaugh criando um mapa para cada bit do estado seguinte ($D1$ e $D0$) usando a combinação dos bits do estado atual, sendo o bit mais significativo do estado atual chamado de $E1$ e o menos significativo $E0$.

Para construir a expressão que representa o bit $D0$ é necessário pegar os valores nos quais esse bit admite o nível lógico 1, neste caso $D0 = \overline{E1}E0 + E1\overline{E0}$. E para o bit $D1$ seria análogo, ficando com $D1 = \overline{E1}E0 + E1\overline{E0}$.

	$\overline{E1}$	$E1$
$\overline{E0}$	1	1
$E0$	0	0

Tabela 22: Mapa de Karnaugh: bit $D0$

	$\overline{E1}$	$E1$
$\overline{E0}$	0	1
$E0$	1	0

Tabela 23: Mapa de Karnaugh: bit D1

No mapa do bit $D0$ 22, é possível simplificar a equação ficando com $D0 = \overline{E0}$. Porém no bit $D1$ 23 não existe uma simplificação, ou seja, a equação permanece sendo $D1 = \overline{E1}E0 + E1\overline{E0}$.

Nota: Representação da saída $S1$ e $S0$

Este processo de simplificação seria repetido para cada bit de saída caso a representação do número 'n' fosse diferente de sua representação em binário.

Representação/Desenho do circuito lógico: Após definirmos as funções de transição de estado e de saída podemos construir o desenho do circuito lógico. Abaixo está representado o circuito lógico do contador, onde as linhas azuis representam as saídas do Flip-Flop $D0$, as verdes do Flip-Flop $D1$, as laranjas as saídas $S1$ e $S0$ e por fim a vermelha o CLOCK.

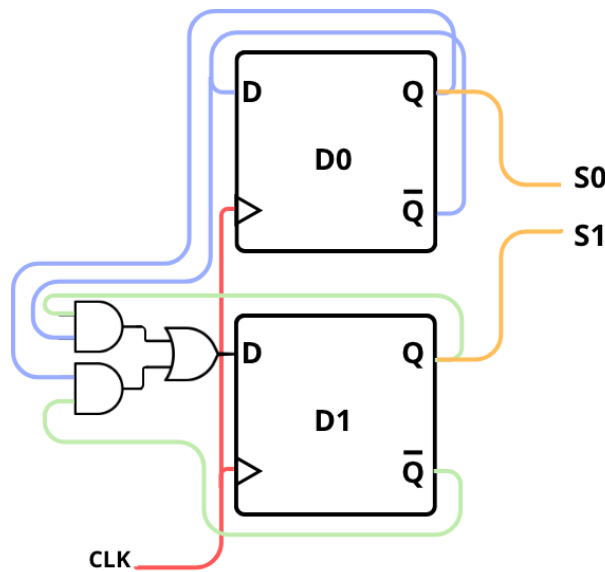


Figura 15: Circuito Lógico do Contador

Nota: Representação dos Estados e nomenclatura

Percebe-se que nesse circuito não escrevemos $E0$ e $E1$, os estados atuais da máquina, pois esses valores são os dados armazenados nos Flip-Flops $D1$ e $D0$. Isto significa que os valores de Estado Atual servem para organizarmos e construirmos as tabelas diferindo os estados atuais dos próximos estados, ou seja, os valores armazenados nos Flip-Flops Ds são os estados atuais, enquanto a entrada D em cada Flip-Flop é na verdade o próximo estado do circuito, que será alcançado quando houver um novo pulso de clock.

Resumindo: As máquinas de estados são circuitos sequenciais que transitam entre estados, ou seja, seguem uma sequência pré-definida de estados. Essa transição ocorre

através de um sinal de controle que é normalmente o clock. Além disso, os estados atuais são armazenados nos Flip-Flops (memória) e os estados futuros são determinados pela combinação do **estado atual** + **entradas**.

19.2 Máquina de Moore

A Máquina de Moore é um tipo de máquina de estados em que a **saída** depende exclusivamente do **estado atual** da máquina. Dessa forma, a saída permanece inalterada mesmo quando as entradas mudam. Isso torna a Máquina de Moore mais **simples** de projetar e de analisar seu comportamento, já que ela não é afetada imediatamente pelas alterações nas entradas. Por exemplo: o contador 15 é uma máquina de estados de **Moore**.

Novo Exemplo: Um semáforo em que a saída, ou seja, a luz exibida (verde amarelo ou vermelho), é gerada a partir dos estados da máquina.

19.2.1 Exemplo: Máquina de Estado de um Semáforo

Para construir o circuito lógico devemos seguir os passos descritos anteriormente:

Separação ou levantamento de dados: Podemos pensar num conjunto de perguntas que podem ser respondidas para definir as informações necessárias:

Como funciona um semáforo? Qual a sequência (ordem) que as cores são alternadas? Quantas cores (estados) tem num semáforo? Quantos bits são necessários para contemplar todos os estados possíveis? E ainda qual cor será ativa em qual estado?

- **Luz Vermelha:** Indica que o fluxo é proibido.
- **Luz Verde:** Permite o fluxo de veículos.
- **Luz Amarela:** Um estado intermediário de curta duração que sinaliza o encerramento do fluxo.

Ou seja, existem **3** estados possíveis: vermelho, verde ou amarelo, que são as cores atuais no semáforo. Dessa forma, como circuitos digitais usam binário para representação, precisamos utilizar Flip-Flops para armazenar o estado atual. A cada Flip-Flop adicionado ao circuito dobramos a quantidade de **estados possíveis**. Assim, como precisamos de 3 estados (**Luz Vermelha, Luz Amarela, Luz Verde**), precisamos utilizar 2 Flip-Flops: $2^1 < 3 < 2^2$. Por fim, é preciso definir qual luz será ativada em determinado estado. Uma combinação possível está representada na tabela abaixo:

Cor Atual	Estado Atual em Binário (E_1E_0)	Explicação
Vermelho	00	Ambos os Flip-Flops desligados
Verde	01	Apenas o Flip-Flop 0 (E_0) ligado
Amarelo	10	Apenas o Flip-Flop 1 (E_1) ligado

Tabela 24: Tabela representando os estados e seus respectivos códigos

Observação: o Estado 11 inutilizado

O estado 11 (ambos os Flip-Flops ligados) não é utilizado e é considerante irrelevante (**Don't Care**), pois só precisamos de 3 estados para o funcionamento correto do circuito. Isso permite simplificar as equações booleanas.

Representação em um diagrama estados: Após separarmos todas os dados necessários podemos representar a transição de estados em uma diagrama de estado. Neste caso, diferente do contador 19.1, as saídas não são a representação de um número em binário mas sim uma combinação que ligará um dos **LEDs** ou uma das **LUZES**.

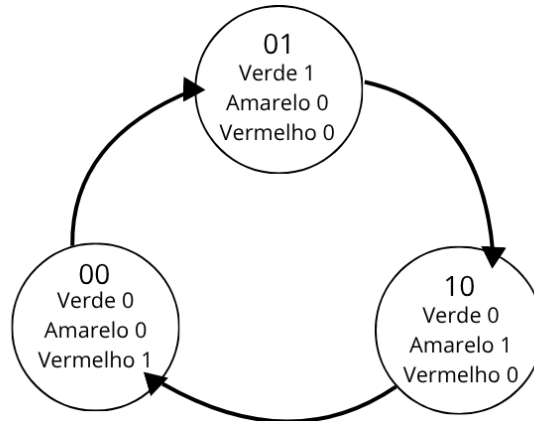


Figura 16: Máquina de Estados de um Semáforo

O diagrama da Figura 16 representa a máquina de estados que coordena a lógica de funcionamento de um semáforo. O sistema alterna entre três estados:

A Máquina inicia no **Estado de Luz Vermelha**, passa para o **Estado de Luz Verde**, segue para o **Estado de Luz Amarela**, e por fim retorna ao **Estado de Luz Vermelha** reiniciando o ciclo novamente. Além disso, percebe-se que dentro de cada círculo está indicado se cada uma das **três saídas**, as luzes, está ligada (1) ou desligada (0). No caso de um semáforo só temos uma luz ligada por vez, ou seja, uma saída ativa estará ativa enquanto as outras estarão inativas.

Criação de uma tabela de estados: Após definir os códigos binários para cada estado, definiremos a lógica de transição. Esta tabela é fundamental para determinar as equações booleanas que controlarão a entrada dos **Flip-flops** (E_1 e E_0) e o acionamento das **lâmpadas** (Vermelha, Amarela e Verde). Como não possuímos entradas nessa máquina de estados, usaremos uma tabela simplificada que não possui a coluna de entradas.

Estado Atual		Próximo Estado		Saídas (Luzes)		
E_1	E_0	D_1	D_0	Vermelha	Verde	Amarela
0	0	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	1

Tabela 25: Tabela da Transição de Estados e da Lógica da Saída

Mapa de karnaugh - Simplificação: Após a construção da Tabela de Transição, o passo seguinte será a simplificação usando o mapa de karnaugh para extrair as equações booleanas das **saídas** e **entradas dos Flip-Flops** simplificadas:

Equações dos Flip-Flops Para determinar qual será o estado seguinte para as entradas D dos Flip-Flops, olhamos para as colunas de "Próximo Estado":

	$\overline{E1}$	$E1$
$\overline{E0}$	1	0
$E0$	0	0

Tabela 26: Mapa de Karnaugh: bit D0

	$\overline{E1}$	$E1$
$\overline{E0}$	0	0
$E0$	1	0

Tabela 27: Mapa de Karnaugh: bit D1

Não existe simplificação para as equações das entradas dos Flip-Flops (Próximo Estado), ou seja, as equações permanecem sendo:

- $D_1 = \overline{E_1} \cdot E_0$
- $D_0 = \overline{E_1} \cdot \overline{E_0}$

Equações das Saídas - Lâmpadas

As lâmpadas dependem exclusivamente do estado atual (E_1 e E_0) e também não possuem simplificação:

- **Vermelha** = $\overline{E_1} \cdot \overline{E_0}$ (ativa apenas em 00)
- **Verde** = $\overline{E_1} \cdot E_0$ (ativa apenas em 01)
- **Amarela** = $E_1 \cdot \overline{E_0}$ (ativa apenas em 10)

Deixamos como exercício construir o mapa para cada saída para comprovar que não existe simplificação para elas.

Representação/Desenho do circuito lógico: Após definirmos as funções de transição e de saída podemos construir o circuito lógico. Abaixo está a representação de um circuito lógico deste exemplo:

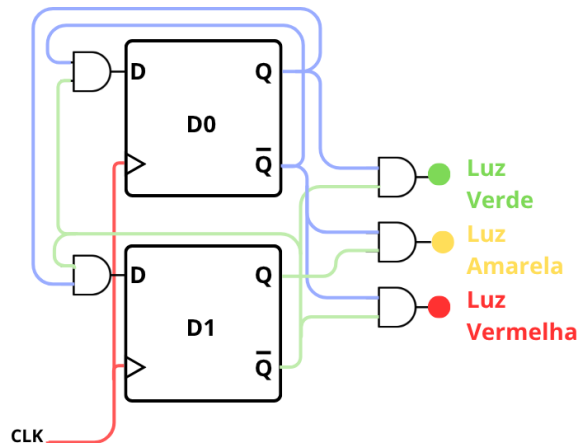


Figura 17: Circuito Lógico de um Semáforo

Na imagem acima 17, temos uma implementação de um circuito lógico de um semáforo. As linhas verdes são as saídas do Flip-Flop $D1$, as azuis do $D0$, a vermelha o $CLOCK$ e a direta temos as funções de saída de cada led, que neste caso é a combinação dos estados atuais da memória. Abaixo está uma outra implementação deste exemplo, porém utilizando o simulador Digital. Os fios **verdes claro** indicam que o fio está com nível lógico 1, ou seja, está ativo enquanto os fios **verde escuro** indicam inatividade:

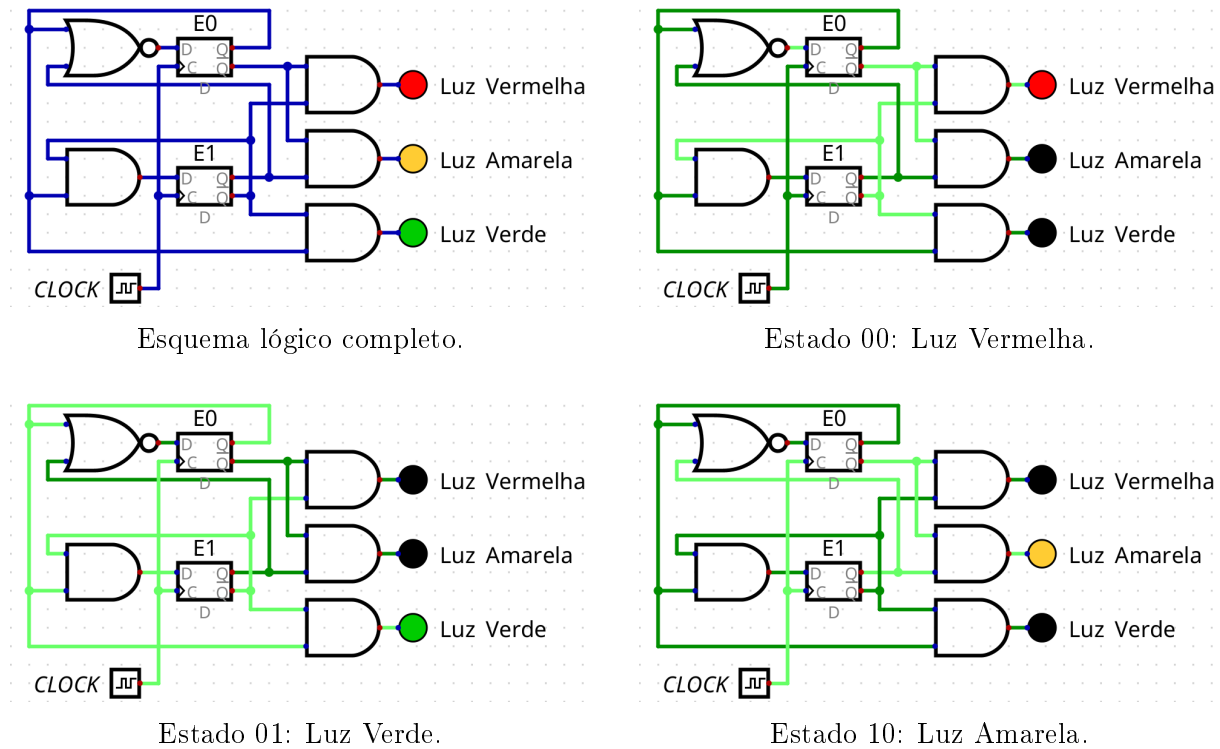


Figura 18: Simulação do circuito lógico em funcionamento, mostrando a transição de estados e o acionamento das lâmpadas. Feito no Digital.

Observa-se que o circuito opera de forma cíclica e síncrona, onde cada combinação dos Flip-flops define o acionamento de uma única luz por vez.

19.3 Máquina de Mealy

A Máquina de Mealy é outro tipo de máquina de estados que, diferente da Máquina de Moore, gera sua saída a partir da **combinação do estado atual com a entrada**. Isso significa que a saída pode ser alterada imediatamente sempre que houver uma mudança na entrada, sem que ocorra uma mudança de estado.

Isso geralmente resulta em uma máquina de estado com um **número de estados reduzido** para realizar a mesma função, no entanto, a lógica para sua implementação tende a ser mais complexa.

19.3.1 Exemplo: Detector de Sequência

Repetimos o processo de construção de uma máquina de estados seguindo os 5 passos vistos.

Separação e levantamento dos dados: Devemos definir a sequência desejada, a quantidade de bits dessa sequência e os valores de entrada. Por exemplo: detectar a

sequência 10, e os valores da entrada X assumem os valores 0/1. Agora definimos os estados necessários:

- **Inicial/Parado:** Indica que não houve nenhuma entrada e/ou nenhuma entrada correta.
- **Intermediário:** Indica que o primeiro número é correto.
- **Final:** Indica que a sequência é a desejada.

Seguindo a mesma lógica utilizada no exemplo anterior do semáforo 16, como o sistema possui 3 estados, são necessários 2 Flip-Flops para a implementação do circuito (2 bits). Uma possibilidade para binária para os estados é :

Estado	Estado representado em Binário (E_1E_0)
Inicial / Parado	00
Intermediário	01
Final / Sequência correta (10)	10

Tabela 28: Codificação de estados do detector de sequência.

Representação em um diagrama estados:

Agora construímos o diagrama desta máquina de estados considerando X como o valor da entrada ou do número digitado:

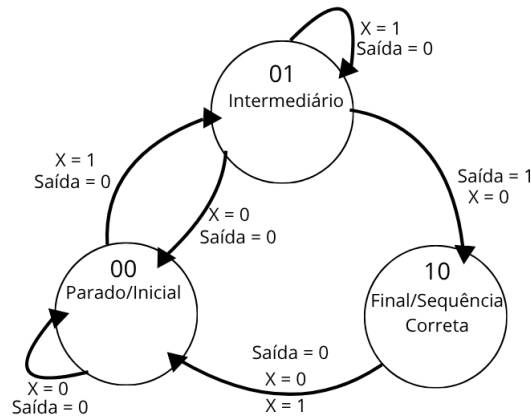


Figura 19: Diagrama de Estados de um Detector de Sequência

O diagrama da Figura 19 representa a máquina de estados que coordena a lógica de funcionamento de um detector de sequência.

Nota: Representação das saídas

Diferente das Máquina de Moore 19.2 que tinha as saídas dentro dos círculos, ou seja, as saídas só dependiam do **estado atual**, as saídas das máquinas de Mealy são representadas em cima das setas, visto que dependem de **estado atual + entradas**.

Criação de uma tabela de estados: Diferente da abordagem de Moore utilizada no semáforo, a saída (S) neste circuito depende simultaneamente da entrada atual (X) e

do estado atual. A tabela abaixo detalha como o sistema transita entre os estados e em que momento a sequência é validada:

Entrada (X)	Estado Atual		Próximo Estado		Saída (S)
X	E_1	E_0	D_1	D_0	S
0	0	0	0	0	0
1	0	0	0	1	0
0	0	1	1	0	1 ←
1	0	1	0	1	0
0	1	0	0	0	0
1	1	0	0	0	0

Sequência "10" detectada

Tabela 29: Tabela de Transição e Lógica de Saída de Mealy.

Mapa de Karnaugh - Simplificação: Deixamos como exercício para o leitor a realização desta simplificação.

Continuando, a partir da Tabela de Transição, construímos as funções lógicas (funções de transição de estado) para as entradas dos Flip-Flops (D_1 e D_0) e para a saída (S). Nota-se que, por ser uma Máquina de Mealy, a variável de entrada X aparece diretamente na equação de saída:

- $D_1 = \overline{E_1} \cdot E_0 \cdot \overline{X}$ (Acionado apenas quando o estado é "1" e chega o bit "0").
- $D_0 = \overline{E_1} \cdot \overline{E_0} \cdot X + \overline{E_1} \cdot E_0 \cdot X$ (Simplificado: o sistema permanece ou vai para o estado "1" sempre que a entrada for "1", desde que não esteja no estado final).
- **Saída $S = \overline{E_1} \cdot E_0 \cdot X$**

Observa-se que o detector de sequência apresenta uma resposta imediata assim que o bit "0" é detectado na entrada. Logo após isso a saída S assume nível lógico 1 sem aguardar a transição do próximo pulso de *clock* para o estado final.

Representação/Desenho do circuito lógico: Após extrairmos as funções de transição de estado e de saída podemos transformar num circuito lógico. Abaixo estão duas representações: um desenho e uma implementação possível no Digital.

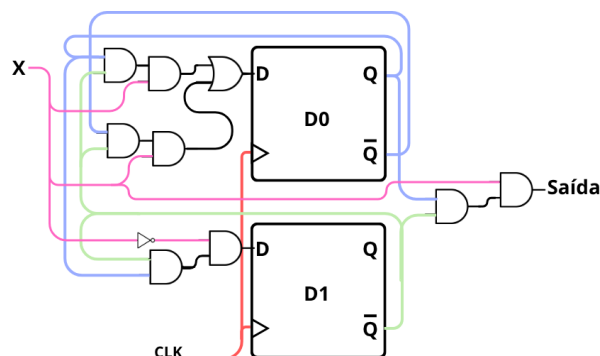


Figura 20: Desenho do Circuito Lógico de um Detector de Sequência

Nesta representação 20, as linhas azuis e verdes continuam representando as saídas dos Flip-Flops D_0 e D_1 , respectivamente, e o fio vermelho permanece indicando o CLOCK. Porém, neste exemplo, existe a entrada X , e por se tratar de uma máquina de mealy,

a saída depende tanto do valor do **estado atual** como do valor da **entrada**. Assim, percebemos que o fio rosa que representa a entrada sendo uma das variáveis que compoem a função de saída da variável S .

Implementação deste circuito no Digital:

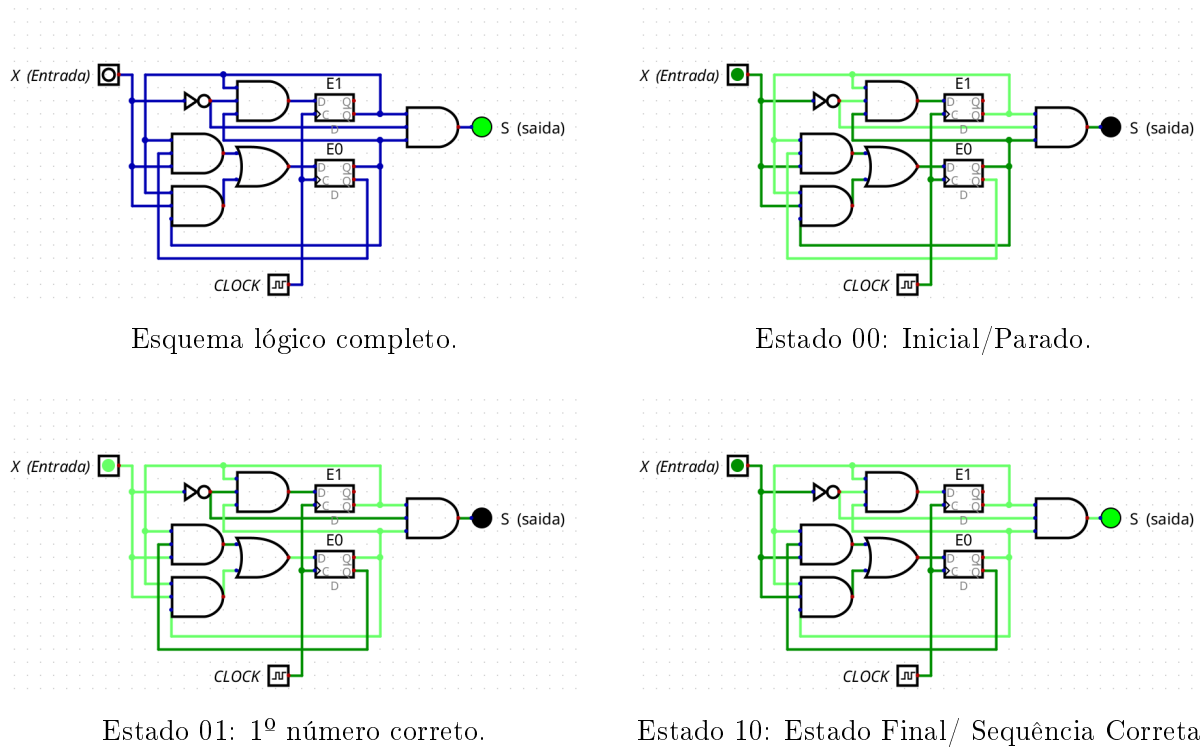


Figura 21: Simulação do detector de sequência em funcionamento, destacando a ativação da saída S no momento da detecção. Implemetação no Digital

As representações acima 20 e 21 comprovam que a Máquina de Mealy responde instantaneamente às variações da entrada. Diferente do semáforo, a saída deste detector ativa imediatamente assim que a sequência "10" é completada, sem depender de um próximo *clock*.

19.4 Revisão

Classificação: Moore ou Mealy

A diferença fundamental está na lógica de saída:

- Máquina de Moore: A saída depende exclusivamente do estado atual. A saída só é alterada em sincronia com o clock. No diagrama 16, a saída é representada dentro do círculo do estado.
- Máquina de Mealy: A saída depende do estado atual e das entradas. A saída responde instantaneamente às mudanças na entrada, sem a necessidade de esperar o próximo ciclo de clock. No diagrama 19, a saída é indicada nas setas de transição.

O Estado de Reset ou Inicial

Todas as máquina de estados precisam começar em um ponto de partida. O sinal de Reset (ou Clear dos Flip-Flops) garante que, ao ligar o circuito, a máquina inicie em

um estado conhecido e/ou inicial (geralmente o Estado 0). Caso não existe o Reset, o sistema pode iniciar em um estado desconhecido e imprevisível, podendo prejudicar o funcionamento correto.

19.5 Exercícios Propostos:

1. **Detector de Sequência:** Projete um detector de sequências cuja saída, y , será ‘1’ sempre que ocorrer a sequência de entradas 11.

Exemplo:

```
x:  0 1 1 0 0 1 1 1 0 1 1 0
y:  0 0 1 0 0 0 1 0 0 0 1 0
```

2. Projete um detector de sequências cuja saída, y , será ‘1’ sempre que ocorrer a sequência de entradas 1101. Agora sequências sobrepostas também devem ser detectadas.

Exemplo:

```
x:  0 1 0 1 1 1 0 1 1 0 1 0 1 1
y:  0 0 0 0 0 0 0 1 0 0 1 0 0 0
```

3. **Sequência de valores:** Projete uma máquina de estados que seja capaz de gerar a sequência 001, 010, 100 e 110 de forma cíclica.
4. **Máquina de refrigerante:** Em máquinas automáticas de compra de refrigerantes e afins, um mecanismo de contagem de moedas é utilizado para se controlar tanto a liberação do produto quanto a liberação de troco. Devido a sua operação, este mecanismo pode ser modelado como uma máquina de estados finitos que receba uma sequência de moedas depositadas para então controlar a liberação do produto e a liberação de troco para o montante depositado.

Uma arquitetura possível para o mecanismo citado pode ser como a apresentada na figura abaixo, em que uma unidade de controle receba dois sinais de entrada (M_2 e M_1) provenientes de um detector de moedas e então controle, por meio de seus sinais de saída (P e T), as unidades de liberação do produto e de troco, respectivamente.

Considerando que os sinais de liberação sejam ativos em nível lógico “1” e que o detector de moedas gere sinais como os apresentados na tabela abaixo, projete uma máquina de estado baseada no modelo de **Moore** para atuar como unidade de controle do sistema exemplificado. De uma forma geral, esta unidade de controle deve ser tal que à medida que sejam depositados valores (e tais valores gerem saídas no detector de moedas), a soma total seja computada e o sinal de liberação do produto seja ativado tão logo a soma chegue e/ou ultrapasse a quantia de 2 unidades monetárias. Caso a quantia depositada ultrapasse o limite citado, o sinal de liberação de troco deve ser levado a nível lógico “1”. Assuma que a cada depósito efetuado, um sinal de pulso é enviado à máquina de estado e considere, ainda, que assim que houver a liberação do produto, o próximo depósito deverá reiniciar a soma.

Moeda	M_2M_1
-	00
0.25	01
0.50	10
1.00	11

- 5. Operação de um Robô:** Projete uma máquina de estados para controlar a operação de um robô que se movimenta e desvie de objetos a sua frente. Assuma que este robô seja composto basicamente por um detector de objetos e duas rodas que o possibilitem virar tanto à direita quanto à esquerda.

C_2C_1	Ação
00	Seguir em frente
01	Virar à direita
10	Virar à esquerda
11	Não operar

Este detector de objetos é posicionado no nariz do robô e sua saída é conectada à entrada de sua unidade de controle por meio de um sinal x , tal que enquanto $x = '1'$, um objeto é detectado à sua frente. Já as saídas desta unidade de controle, C_1C_2 , devem controlar o movimento a ser efetuado pelo robô tal qual a codificação apresentada acima.

Um exemplo de estados que possam ser utilizados no projeto é apresentado na tabela abaixo.

Estado	Descrição
A	Obstáculo não detectado, última virada à esquerda
B	Obstáculo detectado e virando à direita
C	Obstáculo não detectado, última virada à direita
D	Obstáculo detectado e virando à esquerda

- 6. Catraca de Ônibus:** Projete a máquina de estados para controlar a catraca de um ônibus.

- **Entradas:** pago ok (p), empurrar (e).
- **Estados:** aberta (A), fechada (F).
- **Regras:**
 - (a) A catraca deve permanecer fechada enquanto a passagem não for paga.
 - (b) Quando a passagem for paga, a catraca libera a passagem e pode ser empurrada.
 - (c) Depois de empurrada, a catraca deve voltar a ficar fechada.
 - (d) Pagamentos feitos com a catraca aberta implicam na perda do dinheiro e a catraca continua aberta.

- 7. Contador:** Faça uma máquina de estados e seu circuito para um contador de 2 bits.
- 8. Fechadura Digital (Sequência Fixa):** Faça uma máquina de estados e seu circuito para uma fechadura digital de 4 dígitos binários com o seguinte funcionamento:
- (a) Aperte *start* para começar.
 - (b) Digite 0 → Digite 1 → Digite 1 → Digite 0.
 - (c) Saída: 1 (abre).
 - (d) Caso a sequência digitada seja diferente de 0110, deve retornar ao início.
- 9. Fechadura Digital (Variação):** Idem ao anterior, mas entre o *start* e a sequência 0110 pode ter qualquer dígito.
- 10. Máquina de Venda:** Faça uma máquina de estados e seu circuito para uma máquina de venda de refrigerante automática.
- O refrigerante custa R\$ 4,00.
 - A máquina aceita moedas de R\$ 1,00 e R\$ 2,00.
 - A máquina não fornece troco. Quando o valor de R\$ 4,00 for atingido, libera o refrigerante.
- 11. Máquina de Venda (com troco):** Idem ao anterior, mas a máquina aceita notas de R\$ 5,00 e deve fornecer troco.
- 12. Máquina com mais refrigerante:** Idem ao anterior, mas a máquina tem 2 refri que o cliente pode escolher.

20 Referências

Referências

- [1] Luiz Carlos Pessoa Albini. Circuitos digitais: Anotações de aula.
- [2] Paulo Ricardo Lisboa de Almeida. Circuitos digitais, 2021.
- [3] Marco A. Zanata Alves. Circuitos lógicos: Flip-flops - slides.
- [4] Thomas L. Floyd. *Sistemas Digitais: Fundamentos e Aplicações*. 2009.
- [5] Daniel A. Furtado. Sistemas digitais: Introdução aos circuitos codificadores e decodificadores.
- [6] Abel Guilhermino. Máquinas de estados finitos.
- [7] José L. Guntzel and Francisco A. Nascimento. *Introdução aos Sistemas Digitais*. 2001.
- [8] Rodrigo Hausen. Circuitos digitais, 2013.
- [9] David A. Patterson and John L. Hennessy. *Arquitetura e Organização de Computadores: A Interface Hardware/Software*. 2014.
- [10] Marcia A. G. Ruggiero and Vera L. R. Lopes. *Cálculo Numérico: Aspectos Teóricos e Computacionais*. 1996.
- [11] Ronald J. Tocci, Gregory L. Moss, and Neal S. Widmer. *Sistemas Digitais*. 10 edition, 2017.